

# C++ Reference V1.0

Only For ACM

Spiderd From <http://www.cplusplus.com/reference/>

By RogerRo

2016/11/2

Because the Reference is **ONLY FOR ACM**, here I've only spidered, I think, the essential parts, including (sorted) :

<algorithm>	<bitset>	<cctype>
<cmath>	<cstdio>	<cstdlib>
<cstring>	<deque>	<iostream>
<iostream>	<map>	<queue>
<set>	<string>	<unordered_map>
<unordered_set>	<vector>	

Please contact me at Github:RogerRordo if there are any mistakes.

RogerRo

2016/11/02 at SYSU

## Content

```
/algorithm  
/algorithm/adjacent_find  
/algorithm/all_of  
/algorithm/any_of  
/algorithm/binary_search  
/algorithm/copy  
/algorithm/copy_backward  
/algorithm/copy_if  
/algorithm/copy_n  
/algorithm/count  
/algorithm/count_if  
/algorithm/equal  
/algorithm/equal_range  
/algorithm/fill  
/algorithm/fill_n  
/algorithm/find  
/algorithm/find_end  
/algorithm/find_first_of  
/algorithm/find_if  
/algorithm/find_if_not  
/algorithm/for_each  
/algorithm/generate  
/algorithm/generate_n  
/algorithm/includes  
/algorithm/inplace_merge  
/algorithm/is_heap  
/algorithm/is_heap_until  
/algorithm/is_partitioned  
/algorithm/is_permutation  
/algorithm/is_sorted  
/algorithm/is_sorted_until  
/algorithm/iter_swap  
/algorithm/lexicographical_compare  
/algorithm/lower_bound  
/algorithm/make_heap  
/algorithm/max  
/algorithm/max_element  
/algorithm/merge  
/algorithm/min  
/algorithm/min_element  
/algorithm/minmax  
/algorithm/minmax_element  
/algorithm/mismatch  
/algorithm/move  
/algorithm/move_backward  
/algorithm/next_permutation  
/algorithm/none_of
```

```
/algorithm/nth_element
/algorithm/partial_sort
/algorithm/partial_sort_copy
/algorithm/partition
/algorithm/partition_copy
/algorithm/partition_point
/algorithm/pop_heap
/algorithm/prev_permutation
/algorithm/push_heap
/algorithm/random_shuffle
/algorithm/remove
/algorithm/remove_copy
/algorithm/remove_copy_if
/algorithm/remove_if
/algorithm/replace
/algorithm/replace_copy
/algorithm/replace_copy_if
/algorithm/replace_if
/algorithm/reverse
/algorithm/reverse_copy
/algorithm/rotate
/algorithm/rotate_copy
/algorithm/search
/algorithm/search_n
/algorithm/set_difference
/algorithm/set_intersection
/algorithm/set_symmetric_difference
/algorithm/set_union
/algorithm/shuffle
/algorithm/sort
/algorithm/sort_heap
/algorithm/stable_partition
/algorithm/stable_sort
/algorithm/swap
/algorithm/swap_ranges
/algorithm/transform
/algorithm/unique
/algorithm/unique_copy
/algorithm/upper_bound

/bitset
/bitset/bitset
/bitset/bitset/all
/bitset/bitset/any
/bitset/bitset/bitset
/bitset/bitset/count
/bitset/bitset/flip
/bitset/bitset/hash
/bitset/bitset/none
/bitset/bitset/operator[]
/bitset/bitset/operators
/bitset/bitset/reference
```

```
/bitset/bitset/reset
/bitset/bitset/set
/bitset/bitset/size
/bitset/bitset/test
/bitset/bitset/to_string
/bitset/bitset/to_ullong
/bitset/bitset/to_ulong

/cctype
/cctype/isalnum
/cctype/isalpha
/cctype/isblank
/cctype/iscntrl
/cctype/isdigit
/cctype/isgraph
/cctype/islower
/cctype/isprint
/cctype/ispunct
/cctype/ispace
/cctype/isupper
/cctype/isxdigit
/cctype/tolower
/cctype/toupper

/cmath
/cmath/abs
/cmath/acos
/cmath/acosh
/cmath/asin
/cmath/asinh
/cmath/atan
/cmath/atan2
/cmath/atanh
/cmath/cbrt
/cmath/ceil
/cmath/copysign
/cmath/cos
/cmath/cosh
/cmath/double_t
/cmath/erf
/cmath/erfc
/cmath/exp
/cmath/exp2
/cmath/expm1
/cmath/fabs
/cmath/fdim
/cmath/float_t
/cmath/floor
/cmath/fma
/cmath/fmax
/cmath/fmin
/cmath/fmod
/cmath/fpclassify
```

```
/cmath/frexp
/math/HUGE_VAL
/math/HUGE_VALF
/math/HUGE_VALL
/math/hypot
/math/ilogb
/math/INFINITY
/math/isfinite
/math/isgreater
/math/isgreaterequal
/math/isinf
/math/isless
/math/islessequal
/math/islessgreater
/math/isnan
/math/isnormal
/math/isunordered
/math/ldexp
/math/lgamma
/math/llrint
/math/llround
/math/log
/math/log10
/math/log1p
/math/log2
/math/logb
/math/lrint
/math/lround
/math/math_errhandling
/math/modf
/math/NAN
/math/nanf
/math/nan-function
/math/nanl
/math/nearbyint
/math/nextafter
/math/nexttoward
/math/pow
/math/remainder
/math/remquo
/math/rint
/math/round
/math/scalbln
/math/scalbn
/math/signbit
/math/sin
/math/sinh
/math/sqrt
/math/tan
/math/tanh
/math/tgamma
```

```
/cmath/trunc  
  
/cstdio  
  
/cstdio/BUFSIZ  
/cstdio/clearerr  
/cstdio/EOF  
/cstdio/fclose  
/cstdio/feof  
/cstdio/ferror  
/cstdio/fflush  
/cstdio/fgetc  
/cstdio/fgetpos  
/cstdio/fgets  
/cstdio/FILE  
/cstdio/FILENAME_MAX  
/cstdio/fopen  
  
/cstdio/FOPEN_MAX  
/cstdio/fpos_t  
/cstdio/fprintf  
/cstdio/fputc  
/cstdio/fputs  
/cstdio/fread  
/cstdio/freopen  
/cstdio/fscanf  
/cstdio/fseek  
/cstdio/fsetpos  
/cstdio/ftell  
/cstdio/fwrite  
/cstdio/getc  
/cstdio/getchar  
/cstdio/gets  
/cstdio/_tmpnam  
/cstdio/NULL  
/cstdio/perror  
/cstdio/printf  
/cstdio/putc  
/cstdio/putchar  
/cstdio/puts  
/cstdio/remove  
/cstdio/rename  
/cstdio/rewind  
/cstdio/scanf  
/cstdio/setbuf  
/cstdio/setvbuf  
/cstdio/size_t  
/cstdio/snprintf  
/cstdio/sprintf  
/cstdio/sscanf  
/cstdio/stderr  
/cstdio/stdin  
/cstdio/stdout  
/cstdio/tmpfile
```

```
/cstdio/TMP_MAX
/cstdio/tmpnam
/cstdio/ungetc
/cstdio/vfprintf
/cstdio/vfscanf
/cstdio/vprintf
/cstdio/vscanf
/cstdio/vsnprintf
/cstdio/vsprintf
/cstdio/vsscanf

/cstdlib
/cstdlib/abort
/cstdlib/abs
/cstdlib/atexit
/cstdlib/atof
/cstdlib/atoi
/cstdlib/atol
/cstdlib/atoll
/cstdlib/at_quick_exit
/cstdlib/bsearch
/cstdlib/calloc
/cstdlib/div
/cstdlib/div_t
/cstdlib/exit
/cstdlib/_Exit
/cstdlib/EXIT_FAILURE
/cstdlib/EXIT_SUCCESS
/cstdlib/free
/cstdlib/getenv
/cstdlib/itoa
/cstdlib/labs
/cstdlib/ldiv
/cstdlib/ldiv_t
/cstdlib/llabs
/cstdlib/lldiv
/cstdlib/lldiv_t
/cstdlib/malloc
/cstdlib/MB_CUR_MAX
/cstdlib/mblen
/cstdlib/mbstowcs
/cstdlib/mbtowc
/cstdlib/NULL
/cstdlib/qsort
/cstdlib/quick_exit
/cstdlib/rand
/cstdlib/RAND_MAX
/cstdlib/realloc
/cstdlib/size_t
/cstdlib/srand
/cstdlib strtod
/cstdlib strtos
```

```
/cstdlib/strtol
/cstdlib/strtold
/cstdlib/strtoll
/cstdlib/strtoul
/cstdlib/strtoull
/cstdlib/system
/cstdlib/wcstombs
/cstdlib/wcrtomb

/cstring
/cstring/memchr
/cstring/memcmp
/cstring/memcpy
/cstring/memmove
/cstring/memset
/cstring/NULL
/cstring/size_t
/cstring/strcat
/cstring/strchr
/cstring/strcmp
/cstring/strcoll
/cstring/strcpy
/cstring/strcspn
/cstring/strerror
/cstring/strlen
/cstring/strncat
/cstring/strncmp
/cstring/strncpy
/cstring/strpbrk
/cstring/strrchr
/cstring/strspn
/cstring/strstr
/cstring/strtok
/cstring/strxfrm

/deque
/deque/deque
/deque/deque/assign
/deque/deque/at
/deque/deque/back
/deque/deque/begin
/deque/deque/cbegin
/deque/deque/cend
/deque/deque/clear
/deque/deque/crbegin
/deque/deque/crend
/deque/deque/deque
/deque/deque/~deque
/deque/deque/emplace
/deque/deque/emplace_back
/deque/deque/emplace_front
/deque/deque/empty
/deque/deque/end
```

```
/deque/deque/erase
/deque/deque/front
/deque/deque/get_allocator
/deque/deque/insert
/deque/deque/max_size
/deque/deque/operator=
/deque/deque/operator[]
/deque/deque/operators
/deque/deque/pop_back
/deque/deque/pop_front
/deque/deque/push_back
/deque/deque/push_front
/deque/deque/rbegin
/deque/deque/rend
/deque/deque/resize
/deque/deque/shrink_to_fit
/deque/deque/size
/deque/deque/swap
/deque/deque/swap-free

/ios

/ios/basic_ios
/ios/basic_ios/bad
/ios/basic_ios/~basic_ios
/ios/basic_ios/basic_ios
/ios/basic_ios/clear
/ios/basic_ios/copyfmt
/ios/basic_ios/eof
/ios/basic_ios/exceptions
/ios/basic_ios/fail
/ios/basic_ios/fill
/ios/basic_ios/good
/ios/basic_ios/imbue
/ios/basic_ios/init
/ios/basic_ios/move
/ios/basic_ios/narrow
/ios/basic_ios/operator_bool
/ios/basic_ios/operator_not
/ios/basic_ios/rdbuf
/ios/basic_ios/rdstate
/ios/basic_ios/set_rdbuf
/ios/basic_ios/setstate
/ios/basic_ios/swap
/ios/basic_ios/tie
/ios/basic_ios/widen

/ios/boolalpha
/ios/dec
/ios/defaultfloat
/ios/fixed
/ios/fpos
/ios/hex
/ios/hexfloat
```

```
/ios/internal
/ios/io_errc
  /ios/io_errc/is_error_code_enum
  /ios/io_errc/make_error_code
  /ios/io_errc/make_error_condition
/ios/ios
  /ios/ios/bad
/ios/ios_base
  /ios/ios_base/event
  /ios/ios_base/event_callback
  /ios/ios_base/failure
  /ios/ios_base/flags
  /ios/ios_base/fmtflags
  /ios/ios_base/getloc
  /ios/ios_base/imbue
  /ios/ios_base/Init
  /ios/ios_base/~ios_base
  /ios/ios_base/ios_base
  /ios/ios_base/iostate
  /ios/ios_base/iword
  /ios/ios_base/openmode
  /ios/ios_base/precision
  /ios/ios_base/pword
  /ios/ios_base/register_callback
  /ios/ios_base/seekdir
  /ios/ios_base/setf
  /ios/ios_base/sync_with_stdio
  /ios/ios_base/unsetf
  /ios/ios_base/width
  /ios/ios_base/xalloc
/ios/ios/clear
/ios/ios/copyfmt
/ios/ios/eof
/ios/ios/exceptions
/ios/ios/fail
/ios/ios/fill
/ios/ios/good
/ios/ios/imbue
/ios/ios/init
/ios/ios/ios
/ios/ios/~ios
/ios/ios/move
/ios/ios/narrow
/ios/ios/operator_bool
/ios/ios/operator_not
/ios/ios/rdbuf
/ios/ios/rdstate
/ios/ios/set_rdbuf
/ios/ios/setstate
/ios/ios/swap
/ios/ios/tie
```

```
/ios/iostream_category
/ios/ios/widen
/ios/left
/ios/noboolalpha
/ios/noshowbase
/ios/noshowpoint
/ios/noshowpos
/ios/noskipws
/ios/nounitbuf
/ios/nouppercase
/ios/oct
/ios/right
/ios/scientific
/ios/showbase
/ios/showpoint
/ios/showpos
/ios/skipws
/ios/streamoff
/ios/streampos
/ios/streamsize
/ios/unitbuf
/ios/uppercase
/ios/wios
/ios/wstreampos

/istream
/istream/basic_iostream
/istream/basic_iostream/~basic_iostream
/istream/basic_iostream/basic_iostream
/istream/basic_iostream/operator=
/istream/basic_iostream/swap
/istream/basic_istream
/istream/basic_istream/~basic_istream
/istream/basic_istream/basic_istream
/istream/basic_istream/gcount
/istream/basic_istream/get
/istream/basic_istream/getline
/istream/basic_istream/ignore
/istream/basic_istream/operator=
/istream/basic_istream/operator>>
/istream/basic_istream/operator-free
/istream/basic_istream/peek
/istream/basic_istream/putback
/istream/basic_istream/read
/istream/basic_istream/readsome
/istream/basic_istream/seekg
/istream/basic_istream/sentry
/istream/basic_istream/swap
/istream/basic_istream/sync
/istream/basic_istream/tellg
/istream/basic_istream/unget
/istream/iostream
```

```
/istream/iostream/iostream
/istream/iostream/~iostream
/istream/iostream/operator=
/istream/iostream/swap

/istream/istream
/istream/istream/gcount
/istream/istream/get
/istream/istream/getline
/istream/istream/ignore
/istream/istream/istream
/istream/istream/~istream
/istream/istream/operator=
/istream/istream/operator>>
/istream/istream/operator-free
/istream/istream/peek
/istream/istream/putback
/istream/istream/read
/istream/istream/readsome
/istream/istream/seekg
/istream/istream/sentry
/istream/istream/swap
/istream/istream"sync
/istream/istream/tellg
/istream/istream/unget

/istream/wiostream
/istream/wistream
/istream/ws

/map
/map/map
/map/map/at
/map/map/begin
/map/map/cbegin
/map/map/cend
/map/map/clear
/map/map/count
/map/map/crbegin
/map/map/crend
/map/map/emplace
/map/map/emplace_hint
/map/map/empty
/map/map/end
/map/map/equal_range
/map/map/erase
/map/map/find
/map/map/get_allocator
/map/map/insert
/map/map/key_comp
/map/map/lower_bound
/map/map/map
/map/map/~map
/map/map/max_size
```

```
/map/map/operator=
/map/map/operator[]
/map/map/operators
/map/map/rbegin
/map/map/rend
/map/map/size
/map/map/swap
/map/map/swap-free
/map/map/upper_bound
/map/map/value_comp

/map/multimap
/map/multimap/begin
/map/multimap/cbegin
/map/multimap/cend
/map/multimap/clear
/map/multimap/count
/map/multimap/crbegin
/map/multimap/crend
/map/multimap/emplace
/map/multimap/emplace_hint
/map/multimap/empty
/map/multimap/end
/map/multimap/equal_range
/map/multimap/erase
/map/multimap/find
/map/multimap/get_allocator
/map/multimap/insert
/map/multimap/key_comp
/map/multimap/lower_bound
/map/multimap/max_size
/map/multimap/multimap
/map/multimap/~multimap
/map/multimap/operator=
/map/multimap/operators
/map/multimap/rbegin
/map/multimap/rend
/map/multimap/size
/map/multimap/swap
/map/multimap/swap-free
/map/multimap/upper_bound
/map/multimap/value_comp

/queue
/queue/priority_queue
/queue/priority_queue/emplace
/queue/priority_queue/empty
/queue/priority_queue/pop
/queue/priority_queue/priority_queue
/queue/priority_queue/push
/queue/priority_queue/size
/queue/priority_queue/swap
/queue/priority_queue/swap-free
```

```
/queue/priority_queue/top
/queue/priority_queue/uses_allocator

/queue/queue
/queue/queue/back
/queue/queue/emplace
/queue/queue/empty
/queue/queue/front
/queue/queue/operators
/queue/queue/pop
/queue/queue/push
/queue/queue/queue
/queue/queue/size
/queue/queue/swap
/queue/queue/swap-free
/queue/queue/uses_allocator

/set
/set/multiset
/set/multiset/begin
/set/multiset/cbegin
/set/multiset/cend
/set/multiset/clear
/set/multiset/count
/set/multiset/crbegin
/set/multiset/crend
/set/multiset/emplace
/set/multiset/emplace_hint
/set/multiset/empty
/set/multiset/end
/set/multiset/equal_range
/set/multiset/erase
/set/multiset/find
/set/multiset/get_allocator
/set/multiset/insert
/set/multiset/key_comp
/set/multiset/lower_bound
/set/multiset/max_size
/set/multiset/multiset
/set/multiset/~multiset
/set/multiset/operator=
/set/multiset/operators
/set/multiset/rbegin
/set/multiset/rend
/set/multiset/size
/set/multiset/swap
/set/multiset/swap-free
/set/multiset/upper_bound
/set/multiset/value_comp

/set/set
/set/set/begin
/set/set/cbegin
/set/set/cend
```

```
/set/set/clear
/set/set/count
/set/set/crbegin
/set/set/crend
/set/set/emplace
/set/set/emplace_hint
/set/set/empty
/set/set/end
/set/set/equal_range
/set/set/erase
/set/set/find
/set/set/get_allocator
/set/set/insert
/set/set/key_comp
/set/set/lower_bound
/set/set/max_size
/set/set/operator=
/set/set/operators
/set/set/rbegin
/set/set/rend
/set/set/set
/set/set/~set
/set/set/size
/set/set/swap
/set/set/swap-free
/set/set/upper_bound
/set/set/value_comp

/string
/string/basic_string
/string/basic_string/append
/string/basic_string/assign
/string/basic_string/at
/string/basic_string/back
/string/basic_string/~basic_string
/string/basic_string/basic_string
/string/basic_string/begin
/string/basic_string/capacity
/string/basic_string/cbegin
/string/basic_string/cend
/string/basic_string/clear
/string/basic_string/compare
/string/basic_string/copy
/string/basic_string/crbegin
/string/basic_string/crend
/string/basic_string/c_str
/string/basic_string/data
/string/basic_string/empty
/string/basic_string/end
/string/basic_string/erase
/string/basic_string/find
/string/basic_string/find_first_not_of
```

```
/string/basic_string/find_first_of
/string/basic_string/find_last_not_of
/string/basic_string/find_last_of
/string/basic_string/front
/string/basic_string/get_allocator
/string/basic_string/getline
/string/basic_string/insert
/string/basic_string/length
/string/basic_string/max_size
/string/basic_stringnpos
/string/basic_string/operator<<
/string/basic_string/operator=
/string/basic_string/operator>>
/string/basic_string/operator[]
/string/basic_string/operator+
/string/basic_string/operator+=
/string/basic_string/operators
/string/basic_string/pop_back
/string/basic_string/push_back
/string/basic_string/rbegin
/string/basic_string/rend
/string/basic_string/replace
/string/basic_string/reserve
/string/basic_string/resize
/string/basic_string/rfind
/string/basic_string/shrink_to_fit
/string/basic_string/size
/string/basic_string/substr
/string/basic_string/swap
/string/basic_string/swap-free

/string/char_traits
/string/char_traits/assign
/string/char_traits/compare
/string/char_traits/copy
/string/char_traits/eof
/string/char_traits/eq
/string/char_traits/eq_int_type
/string/char_traits/find
/string/char_traits/length
/string/char_traits/lt
/string/char_traits/move
/string/char_traits/not_eof
/string/char_traits/to_char_type
/string/char_traits/to_int_type

/string/stod
/string/stof
/string/stoi
/string/stol
/string/stold
/string/stoll
/string/stoul
```

```
/string/stoull
/string/string
/string/string/append
/string/string/assign
/string/string/at
/string/string/back
/string/string/begin
/string/string/capacity
/string/string/cbegin
/string/string/cend
/string/string/clear
/string/string/compare
/string/string/copy
/string/string/crbegin
/string/string/crend
/string/string/c_str
/string/string/data
/string/string/empty
/string/string/end
/string/string/erase
/string/string/find
/string/string/find_first_not_of
/string/string/find_first_of
/string/string/find_last_not_of
/string/string/find_last_of
/string/string/front
/string/string/get_allocator
/string/string/getline
/string/string/insert
/string/string/length
/string/string/max_size
/string/string/npos
/string/string/operator<<
/string/string/operator=
/string/string/operator>>
/string/string/operator[]
/string/string/operator+
/string/string/operator+=
/string/string/operators
/string/string/pop_back
/string/string/push_back
/string/string/rbegin
/string/string/rend
/string/string/replace
/string/string/reserve
/string/string/resize
/string/string/rfind
/string/string/shrink_to_fit
/string/string/size
/string/string/string
/string/string/~string
```

```
/string/string/substr
/string/string/swap
/string/string/swap-free

/string/to_string
/string/to_wstring
/string/u16string
/string/u32string
/string/wstring

/unordered_map

/unordered_map/unordered_map
/unordered_map/unordered_map/at
/unordered_map/unordered_map/begin
/unordered_map/unordered_map/bucket
/unordered_map/unordered_map/bucket_count
/unordered_map/unordered_map/bucket_size
/unordered_map/unordered_map/cbegin
/unordered_map/unordered_map/cend
/unordered_map/unordered_map/clear
/unordered_map/unordered_map/count
/unordered_map/unordered_map/emplace
/unordered_map/unordered_map/emplace_hint
/unordered_map/unordered_map/empty
/unordered_map/unordered_map/end
/unordered_map/unordered_map/equal_range
/unordered_map/unordered_map/erase
/unordered_map/unordered_map/find
/unordered_map/unordered_map/get_allocator
/unordered_map/unordered_map/hash_function
/unordered_map/unordered_map/insert
/unordered_map/unordered_map/key_eq
/unordered_map/unordered_map/load_factor
/unordered_map/unordered_map/max_bucket_count
/unordered_map/unordered_map/max_load_factor
/unordered_map/unordered_map/max_size
/unordered_map/unordered_map/operator=
/unordered_map/unordered_map/operator[]
/unordered_map/unordered_map/operators
/unordered_map/unordered_map/rehash
/unordered_map/unordered_map/reserve
/unordered_map/unordered_map/size
/unordered_map/unordered_map/swap
/unordered_map/unordered_map/swap(global)
/unordered_map/unordered_map/~unordered_map
/unordered_map/unordered_map/unordered_map

/unordered_map/unordered_multimap

/unordered_map/unordered_multimap/begin
/unordered_map/unordered_multimap/bucket
/unordered_map/unordered_multimap/bucket_count
/unordered_map/unordered_multimap/bucket_size
/unordered_map/unordered_multimap/cbegin
/unordered_map/unordered_multimap/cend
```

```
/unordered_map/unordered_multimap/clear
/unordered_map/unordered_multimap/count
/unordered_map/unordered_multimap/emplace
/unordered_map/unordered_multimap/emplace_hint
/unordered_map/unordered_multimap/empty
/unordered_map/unordered_multimap/end
/unordered_map/unordered_multimap/equal_range
/unordered_map/unordered_multimap/erase
/unordered_map/unordered_multimap/find
/unordered_map/unordered_multimap/get_allocator
/unordered_map/unordered_multimap/hash_function
/unordered_map/unordered_multimap/insert
/unordered_map/unordered_multimap/key_eq
/unordered_map/unordered_multimap/load_factor
/unordered_map/unordered_multimap/max_bucket_count
/unordered_map/unordered_multimap/max_load_factor
/unordered_map/unordered_multimap/max_size
/unordered_map/unordered_multimap/operator=
/unordered_map/unordered_multimap/operators
/unordered_map/unordered_multimap/rehash
/unordered_map/unordered_multimap/reserve
/unordered_map/unordered_multimap/size
/unordered_map/unordered_multimap/swap
/unordered_map/unordered_multimap/swap(global)
/unordered_map/unordered_multimap/~unordered_multimap
/unordered_map/unordered_multimap/unordered_multimap

/unordered_set
/unordered_set/unordered_multiset
/unordered_set/unordered_multiset/begin
/unordered_set/unordered_multiset/bucket
/unordered_set/unordered_multiset/bucket_count
/unordered_set/unordered_multiset/bucket_size
/unordered_set/unordered_multiset/cbegin
/unordered_set/unordered_multiset/cend
/unordered_set/unordered_multiset/clear
/unordered_set/unordered_multiset/count
/unordered_set/unordered_multiset/emplace
/unordered_set/unordered_multiset/emplace_hint
/unordered_set/unordered_multiset/empty
/unordered_set/unordered_multiset/end
/unordered_set/unordered_multiset/equal_range
/unordered_set/unordered_multiset/erase
/unordered_set/unordered_multiset/find
/unordered_set/unordered_multiset/get_allocator
/unordered_set/unordered_multiset/hash_function
/unordered_set/unordered_multiset/insert
/unordered_set/unordered_multiset/key_eq
/unordered_set/unordered_multiset/load_factor
/unordered_set/unordered_multiset/max_bucket_count
/unordered_set/unordered_multiset/max_load_factor
/unordered_set/unordered_multiset/max_size
```

```
/unordered_set/unordered_multiset/operator=
/unordered_set/unordered_multiset/operators
/unordered_set/unordered_multiset/rehash
/unordered_set/unordered_multiset/reserve
/unordered_set/unordered_multiset/size
/unordered_set/unordered_multiset/swap
/unordered_set/unordered_multiset/swap(global)
/unordered_set/unordered_multiset/~unordered_multiset
/unordered_set/unordered_multiset/unordered_multiset

/unordered_set/unordered_set
    /unordered_set/unordered_set/begin
    /unordered_set/unordered_set/bucket
    /unordered_set/unordered_set/bucket_count
    /unordered_set/unordered_set/bucket_size
    /unordered_set/unordered_set/cbegin
    /unordered_set/unordered_set/cend
    /unordered_set/unordered_set/clear
    /unordered_set/unordered_set/count
    /unordered_set/unordered_set/emplace
    /unordered_set/unordered_set/emplace_hint
    /unordered_set/unordered_set/empty
    /unordered_set/unordered_set/end
    /unordered_set/unordered_set/equal_range
    /unordered_set/unordered_set/erase
    /unordered_set/unordered_set/find
    /unordered_set/unordered_set/get_allocator
    /unordered_set/unordered_set/hash_function
    /unordered_set/unordered_set/insert
    /unordered_set/unordered_set/key_eq
    /unordered_set/unordered_set/load_factor
    /unordered_set/unordered_set/max_bucket_count
    /unordered_set/unordered_set/max_load_factor
    /unordered_set/unordered_set/max_size
    /unordered_set/unordered_set/operator=
    /unordered_set/unordered_set/operators
    /unordered_set/unordered_set/rehash
    /unordered_set/unordered_set/reserve
    /unordered_set/unordered_set/size
    /unordered_set/unordered_set/swap
    /unordered_set/unordered_set/swap(global)
    /unordered_set/unordered_set/~unordered_set
    /unordered_set/unordered_set/unordered_set

/vector
    /vector/vector
        /vector/vector/assign
        /vector/vector/at
        /vector/vector/back
        /vector/vector/begin

/vector/vector
    /vector/vector/flip
    /vector/vector/hash
```

/vector/vector-bool/reference  
/vector/vector-bool/swap  
/vector/vector/capacity  
/vector/vector/cbegin  
/vector/vector/cend  
/vector/vector/clear  
/vector/vector/crbegin  
/vector/vector/crend  
/vector/vector/data  
/vector/vector/emplace  
/vector/vector/emplace\_back  
/vector/vector/empty  
/vector/vector/end  
/vector/vector/erase  
/vector/vector/front  
/vector/vector/get\_allocator  
/vector/vector/insert  
/vector/vector/max\_size  
/vector/vector/operator=  
/vector/vector/operator[]  
/vector/vector/operators  
/vector/vector/pop\_back  
/vector/vector/push\_back  
/vector/vector/rbegin  
/vector/vector/rend  
/vector/vector/reserve  
/vector/vector/resize  
/vector/vector/shrink\_to\_fit  
/vector/vector/size  
/vector/vector/swap  
/vector/vector/swap-free  
/vector/vector/vector  
/vector/vector/~vector

# /algorithm

library

## <algorithm>

<algorithm>

### Standard Template Library: Algorithms

The header <algorithm> defines a collection of functions especially designed to be used on ranges of elements.

A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the [STL containers](#). Notice though, that algorithms operate through iterators directly on the values, not affecting in any way the structure of any possible container (it never affects the size or storage allocation of the container).

### Functions in <algorithm>

#### Non-modifying sequence operations:

<a href="#">all_of</a>	Test condition on all elements in range (function template )
<a href="#">any_of</a>	Test if any element in range fulfills condition (function template )
<a href="#">none_of</a>	Test if no elements fulfill condition (function template )
<a href="#">for_each</a>	Apply function to range (function template )
<a href="#">find</a>	Find value in range (function template )
<a href="#">find_if</a>	Find element in range (function template )
<a href="#">find_if_not</a>	Find element in range (negative condition) (function template )
<a href="#">find_end</a>	Find last subsequence in range (function template )
<a href="#">find_first_of</a>	Find element from set in range (function template )
<a href="#">adjacent_find</a>	Find equal adjacent elements in range (function template )
<a href="#">count</a>	Count appearances of value in range (function template )
<a href="#">count_if</a>	Return number of elements in range satisfying condition (function template )
<a href="#">mismatch</a>	Return first position where two ranges differ (function template )
<a href="#">equal</a>	Test whether the elements in two ranges are equal (function template )
<a href="#">is_permutation</a>	Test whether range is permutation of another (function template )
<a href="#">search</a>	Search range for subsequence (function template )
<a href="#">search_n</a>	Search range for elements (function template )

#### Modifying sequence operations:

<a href="#">copy</a>	Copy range of elements (function template )
<a href="#">copy_n</a>	Copy elements (function template )
<a href="#">copy_if</a>	Copy certain elements of range (function template )
<a href="#">copy_backward</a>	Copy range of elements backward (function template )
<a href="#">move</a>	Move range of elements (function template )
<a href="#">move_backward</a>	Move range of elements backward (function template )
<a href="#">swap</a>	Exchange values of two objects (function template )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )
<a href="#">iter_swap</a>	Exchange values of objects pointed to by two iterators (function template )
<a href="#">transform</a>	Transform range (function template )
<a href="#">replace</a>	Replace value in range (function template )
<a href="#">replace_if</a>	Replace values in range (function template )
<a href="#">replace_copy</a>	Copy range replacing value (function template )
<a href="#">replace_copy_if</a>	Copy range replacing value (function template )
<a href="#">fill</a>	Fill range with value (function template )
<a href="#">fill_n</a>	Fill sequence with value (function template )
<a href="#">generate</a>	Generate values for range with function (function template )
<a href="#">generate_n</a>	Generate values for sequence with function (function template )
<a href="#">remove</a>	Remove value from range (function template )
<a href="#">remove_if</a>	Remove elements from range (function template )
<a href="#">remove_copy</a>	Copy range removing value (function template )
<a href="#">remove_copy_if</a>	Copy range removing values (function template )
<a href="#">unique</a>	Remove consecutive duplicates in range (function template )
<a href="#">unique_copy</a>	Copy range removing duplicates (function template )
<a href="#">reverse</a>	Reverse range (function template )
<a href="#">reverse_copy</a>	Copy range reversed (function template )
<a href="#">rotate</a>	Rotate left the elements in range (function template )
<a href="#">rotate_copy</a>	Copy range rotated left (function template )
<a href="#">random_shuffle</a>	Randomly rearrange elements in range (function template )
<a href="#">shuffle</a>	Randomly rearrange elements in range using generator (function template )

#### Partitions:

<a href="#">is_partitioned</a>	Test whether range is partitioned (function template )
<a href="#">partition</a>	Partition range in two (function template )

<b>stable_partition</b>	Partition range in two - stable ordering (function template )
<b>partition_copy</b>	Partition range into two (function template )
<b>partition_point</b>	Get partition point (function template )

#### Sorting:

<b>sort</b>	Sort elements in range (function template )
<b>stable_sort</b>	Sort elements preserving order of equivalents (function template )
<b>partial_sort</b>	Partially sort elements in range (function template )
<b>partial_sort_copy</b>	Copy and partially sort range (function template )
<b>is_sorted</b>	Check whether range is sorted (function template )
<b>is_sorted_until</b>	Find first unsorted element in range (function template )
<b>nth_element</b>	Sort element in range (function template )

#### Binary search (operating on partitioned/sorted ranges):

<b>lower_bound</b>	Return iterator to lower bound (function template )
<b>upper_bound</b>	Return iterator to upper bound (function template )
<b>equal_range</b>	Get subrange of equal elements (function template )
<b>binary_search</b>	Test if value exists in sorted sequence (function template )

#### Merge (operating on sorted ranges):

<b>merge</b>	Merge sorted ranges (function template )
<b>inplace_merge</b>	Merge consecutive sorted ranges (function template )
<b>includes</b>	Test whether sorted range includes another sorted range (function template )
<b>set_union</b>	Union of two sorted ranges (function template )
<b>set_intersection</b>	Intersection of two sorted ranges (function template )
<b>set_difference</b>	Difference of two sorted ranges (function template )
<b>set_symmetric_difference</b>	Symmetric difference of two sorted ranges (function template )

#### Heap:

<b>push_heap</b>	Push element into heap range (function template )
<b>pop_heap</b>	Pop element from heap range (function template )
<b>make_heap</b>	Make heap from range (function template )
<b>sort_heap</b>	Sort elements of heap (function template )
<b>is_heap</b>	Test if range is heap (function template )
<b>is_heap_until</b>	Find first element not in heap order (function template )

#### Min/max:

<b>min</b>	Return the smallest (function template )
<b>max</b>	Return the largest (function template )
<b>minmax</b>	Return smallest and largest elements (function template )
<b>min_element</b>	Return smallest element in range (function template )
<b>max_element</b>	Return largest element in range (function template )
<b>minmax_element</b>	Return smallest and largest elements in range (function template )

#### Other:

## /algorithm/adjacent\_find

function template

**std::adjacent\_find**

<algorithm>

```

equality (1) template <class ForwardIterator>
              ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class BinaryPredicate>
predicate (2)   ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last,
                                              BinaryPredicate pred);

```

#### Find equal adjacent elements in range

Searches the range [first, last) for the first occurrence of two consecutive elements that match, and returns an iterator to the first of these two elements, or last if no such pair is found.

Two elements match if they compare equal using operator== (or using pred, in version (2)).

The behavior of this function template is equivalent to:

```

1 template <class ForwardIterator>
2   ForwardIterator adjacent_find (ForwardIterator first, ForwardIterator last)
3 {
4   if (first != last)
5   {
6     ForwardIterator next=first; ++next;
7     while (next != last) {
8       if (*first == *next)    // or: if (pred(*first,*next)), for version (2)
9         return first;
10        ++first; ++next;
11    }
12  }

```

```
13 |     return last;
14 }
```

## Parameters

first, last

Forward iterators to the initial and final positions of the searched sequence. The range used is [first, last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

pred

Binary function that accepts two elements as arguments, and returns a value convertible to bool. The returned value indicates whether the elements are considered to match in the context of this function.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

## Return value

An iterator to the first element of the first pair of matching consecutive elements in the range [first, last).  
If no such pair is found, the function returns last.

## Example

```
1 // adjacent_find example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::adjacent_find
4 #include <vector>             // std::vector
5
6 bool myfunction (int i, int j) {
7     return (i==j);
8 }
9
10 int main () {
11     int myints[] = {5,20,5,30,30,20,10,10,20};
12     std::vector<int> myvector (myints,myints+8);
13     std::vector<int>::iterator it;
14
15     // using default comparison:
16     it = std::adjacent_find (myvector.begin(), myvector.end());
17
18     if (it!=myvector.end())
19         std::cout << "the first pair of repeated elements are: " << *it << '\n';
20
21     //using predicate comparison:
22     it = std::adjacent_find (++it, myvector.end(), myfunction);
23
24     if (it!=myvector.end())
25         std::cout << "the second pair of repeated elements are: " << *it << '\n';
26
27     return 0;
28 }
```

Output:

```
the first pair of repeated elements are: 30
the second pair of repeated elements are: 10
```

## Complexity

Up to linear in the distance between first and last: Compares elements until a match is found.

## Data races

Some (or all) of the objects in the range [first, last) are accessed (once at most).

## Exceptions

Throws if any element comparison (or pred) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause undefined behavior.

## See also

<a href="#">find</a>	Find value in range ( <a href="#">function template</a> )
<a href="#">find_if</a>	Find element in range ( <a href="#">function template</a> )
<a href="#">unique</a>	Remove consecutive duplicates in range ( <a href="#">function template</a> )

## /algorithm/all\_of

function template

**std::all\_of**

<algorithm>

```
template <class InputIterator, class UnaryPredicate>
    bool all_of (InputIterator first, InputIterator last, UnaryPredicate pred);
```

### Test condition on all elements in range

Returns true if pred returns true for all the elements in the range [first, last) or if the range is empty, and false otherwise.

The behavior of this function template is equivalent to:

```

1 template<class InputIterator, class UnaryPredicate>
2     bool all_of (InputIterator first, InputIterator last, UnaryPredicate pred)
3 {
4     while (first!=last) {
5         if (!pred(*first)) return false;
6         ++first;
7     }
8     return true;
9 }
```

## Parameters

`first, last`

Input iterators to the initial and final positions in a sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`pred`

Unary function that accepts an element in the range as argument and returns a value convertible to `bool`. The value returned indicates whether the element fulfills the condition checked by this function.

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

`true` if `pred` returns `true` for all the elements in the range or if the range is empty, and `false` otherwise.

## Example

```

1 // all_of example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::all_of
4 #include <array>              // std::array
5
6 int main () {
7     std::array<int,8> foo = {3,5,7,11,13,17,19,23};
8
9     if ( std::all_of(foo.begin(), foo.end(), [](int i){return i%2;}) )
10    std::cout << "All the elements are odd numbers.\n";
11
12    return 0;
13 }
```

Output:

All the elements are odd numbers.

## Complexity

Up to linear in the distance between `first` and `last`: Calls `pred` for each element until a mismatch is found.

## Data races

Some (or all) of the objects in the range `[first, last)` are accessed (once at most).

## Exceptions

Throws if either `pred` or an operation on an iterator throws.

Note that invalid parameters cause *undefined behavior*.

## See also

<a href="#">any_of</a>	Test if any element in range fulfills condition (function template )
<a href="#">none_of</a>	Test if no elements fulfill condition (function template )
<a href="#">find_if</a>	Find element in range (function template )

## /algorithm/any\_of

function template

**std::any\_of**

<algorithm>

```

template <class InputIterator, class UnaryPredicate>
    bool any_of (InputIterator first, InputIterator last, UnaryPredicate pred);
```

### Test if any element in range fulfills condition

Returns `true` if `pred` returns `true` for any of the elements in the range `[first, last)`, and `false` otherwise.

If `[first, last)` is an empty range, the function returns `false`.

The behavior of this function template is equivalent to:

```

1 template<class InputIterator, class UnaryPredicate>
2     bool any_of (InputIterator first, InputIterator last, UnaryPredicate pred)
3 {
4     while (first!=last) {
5         if (pred(*first)) return true;
6         ++first;
7     }
}
```

```
8 |     return false;
9 | }
```

## Parameters

`first, last`

`Input iterators` to the initial and final positions in a sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`pred`

Unary function that accepts an element in the range as argument and returns a value convertible to `bool`. The value returned indicates whether the element fulfills the condition checked by this function.

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

true if `pred` returns true for any of the elements in the range `[first, last)`, and false otherwise.

If `[first, last)` is an empty range, the function returns false.

## Example

```
1 // any_of example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::any_of
4 #include <array>              // std::array
5
6 int main () {
7     std::array<int,7> foo = {0,1,-1,3,-3,5,-5};
8
9     if ( std::any_of(foo.begin(), foo.end(), [](int i){return i<0;}) )
10    std::cout << "There are negative elements in the range.\n";
11
12    return 0;
13 }
```

Output:

```
There are negative elements in the range.
```

## Complexity

Up to linear in the `distance` between `first` and `last`: Calls `pred` for each element until a match is found.

## Data races

Some (or all) of the objects in the range `[first, last)` are accessed (once at most).

## Exceptions

Throws if either `pred` or an operation on an iterator throws.

Note that invalid parameters cause *undefined behavior*.

## See also

<code>all_of</code>	Test condition on all elements in range (function template )
<code>none_of</code>	Test if no elements fulfill condition (function template )
<code>find_if</code>	Find element in range (function template )

# /algorithm/binary\_search

function template

## std::binary\_search

<algorithm>

```
default (1)  template <class ForwardIterator, class T>
              bool binary_search (ForwardIterator first, ForwardIterator last,
                                  const T& val);
custom (2)   template <class ForwardIterator, class T, class Compare>
              bool binary_search (ForwardIterator first, ForwardIterator last,
                                  const T& val, Compare comp);
```

### Test if value exists in sorted sequence

Returns true if any element in the range `[first, last)` is equivalent to `val`, and false otherwise.

The elements are compared using `operator<` for the first version, and `comp` for the second. Two elements, `a` and `b` are considered equivalent if `(!(a<b) && !(b<a))` or if `(!comp(a,b) && !comp(b,a))`.

The elements in the range shall already be `sorted` according to this same criterion (`operator<` or `comp`), or at least `partitioned` with respect to `val`.

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for `random-access iterators`.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class T>
2     bool binary_search (ForwardIterator first, ForwardIterator last, const T& val)
```

```

3 {
4     first = std::lower_bound(first, last, val);
5     return (first!=last && !(val<*first));
6 }

```

## Parameters

`first, last`

Forward iterators to the initial and final positions of a sorted (or properly partitioned) sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`val`

Value to search for in the range.

For (1), `T` shall be a type supporting being compared with elements of the range `[first, last)` as either operand of `operator<`.

`comp`

Binary function that accepts two arguments of the type pointed by `ForwardIterator` (and of type `T`), and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

true if an element equivalent to `val` is found, and false otherwise.

## Example

```

1 // binary_search example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::binary_search, std::sort
4 #include <vector>             // std::vector
5
6 bool myfunction (int i,int j) { return (i<j); }
7
8 int main () {
9     int myints[] = {1,2,3,4,5,4,3,2,1};
10    std::vector<int> v(myints,myints+9);                                // 1 2 3 4 5 4 3 2 1
11
12    // using default comparison:
13    std::sort (v.begin(), v.end());
14
15    std::cout << "looking for a 3... ";
16    if (std::binary_search (v.begin(), v.end(), 3))
17        std::cout << "found!\n"; else std::cout << "not found.\n";
18
19    // using myfunction as comp:
20    std::sort (v.begin(), v.end(), myfunction);
21
22    std::cout << "looking for a 6... ";
23    if (std::binary_search (v.begin(), v.end(), 6, myfunction))
24        std::cout << "found!\n"; else std::cout << "not found.\n";
25
26    return 0;
27 }

```

Output:

```

looking for a 3... found!
looking for a 6... not found.

```

## Complexity

On average, logarithmic in the distance between `first` and `last`: Performs approximately  $\log_2(N)+2$  element comparisons (where  $N$  is this distance).

On non-random-access iterators, the iterator `advances` produce themselves an additional linear complexity in  $N$  on average.

## Data races

The objects in the range `[first, last)` are accessed.

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">find</a>	Find value in range (function template )
<a href="#">lower_bound</a>	Return iterator to lower bound (function template )
<a href="#">upper_bound</a>	Return iterator to upper bound (function template )
<a href="#">equal_range</a>	Get subrange of equal elements (function template )

## /algorithm/copy

function template

`std::copy`

<algorithm>

```
template <class InputIterator, class OutputIterator>
    OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result);
```

### Copy range of elements

Copies the elements in the range  $[first, last)$  into the range beginning at  $result$ .

The function returns an iterator to the end of the destination range (which points to the element following the last element copied).

The ranges shall not overlap in such a way that  $result$  points to an element in the range  $[first, last)$ . For such cases, see [copy\\_backward](#).

The behavior of this function template is equivalent to:

```
1 template<class InputIterator, class OutputIterator>
2     OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)
3 {
4     while (first!=last) {
5         *result = *first;
6         ++result; ++first;
7     }
8     return result;
9 }
```

### Parameters

`first, last`

[Input iterators](#) to the initial and final positions in a sequence to be copied. The range used is  $[first, last)$ , which contains all the elements between  $first$  and  $last$ , including the element pointed by  $first$  but not the element pointed by  $last$ .

`result`

[Output iterator](#) to the initial position in the destination sequence.  
This shall not point to any element in the range  $[first, last)$ .

### Return value

An iterator to the end of the destination range where elements have been copied.

### Example

```
1 // copy algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::copy
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[]={10,20,30,40,50,60,70};
8     std::vector<int> myvector (7);
9
10    std::copy ( myints, myints+7, myvector.begin() );
11
12    std::cout << "myvector contains:" ;
13    for (std::vector<int>::iterator it = myvector.begin(); it!=myvector.end(); ++it)
14        std::cout << ' ' << *it;
15
16    std::cout << '\n';
17
18    return 0;
19 }
```

Output:

```
myvector contains: 10 20 30 40 50 60 70
```

### Complexity

Linear in the [distance](#) between  $first$  and  $last$ : Performs an assignment operation for each element in the range.

### Data races

The objects in the range  $[first, last)$  are accessed (each object is accessed exactly once).

The objects in the range between  $result$  and the returned value are modified (each object is modified exactly once).

### Exceptions

Throws if either an element assignment or an operation on iterators throws.

Note that invalid arguments cause *undefined behavior*.

### See also

<a href="#">copy_backward</a>	Copy range of elements backward ( <a href="#">function template</a> )
<a href="#">fill</a>	Fill range with value ( <a href="#">function template</a> )
<a href="#">replace</a>	Replace value in range ( <a href="#">function template</a> )

## /algorithm/copy\_backward

function template

**std::copy\_backward**

<algorithm>

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward (BidirectionalIterator1 first,
                                    BidirectionalIterator1 last,
                                    BidirectionalIterator2 result);
```

### Copy range of elements backward

Copies the elements in the range `[first, last)` starting from the end into the range terminating at `result`.

The function returns an iterator to the first element in the destination range.

The resulting range has the elements in the exact same order as `[first, last)`. To reverse their order, see [reverse\\_copy](#).

The function begins by copying `*(last-1)` into `*(result-1)`, and then follows backward by the elements preceding these, until `first` is reached (and including it).

The ranges shall not overlap in such a way that `result` (which is the *past-the-end element* in the destination range) points to an element in the range `(first, last]`. For such cases, see [copy](#).

The behavior of this function template is equivalent to:

```
1 template<class BidirectionalIterator1, class BidirectionalIterator2>
2     BidirectionalIterator2 copy_backward ( BidirectionalIterator1 first,
3                                         BidirectionalIterator1 last,
4                                         BidirectionalIterator2 result )
5 {
6     while (last!=first) *(--result) = *(--last);
7     return result;
8 }
```

## Parameters

`first, last`

Bidirectional iterators to the initial and final positions in a sequence to be copied. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`result`

Bidirectional iterator to the *past-the-end* position in the destination sequence.

This shall not point to any element in the range `(first, last]`.

## Return value

An iterator to the first element of the destination sequence where elements have been copied.

## Example

```
1 // copy_backward example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::copy_backward
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> myvector;
8
9     // set some values:
10    for (int i=1; i<=5; i++)
11        myvector.push_back(i*10);           // myvector: 10 20 30 40 50
12
13    myvector.resize(myvector.size()+3);   // allocate space for 3 more elements
14
15    std::copy_backward ( myvector.begin(), myvector.begin()+5, myvector.end() );
16
17    std::cout << "myvector contains:";
18    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Output:

```
myvector contains: 10 20 30 10 20 30 40 50
```

## Complexity

Linear in the *distance* between `first` and `last`: Performs an assignment operation for each element in the range.

## Data races

The objects in the range `[first, last)` are accessed (each object is accessed exactly once).

The objects in the range between the returned value and `result` are modified (each object is modified exactly once).

## Exceptions

Throws if either an element assignment or an operation on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">reverse_copy</a>	Copy range reversed (function template )
<a href="#">copy</a>	Copy range of elements (function template )

<b>fill</b>	Fill range with value (function template )
<b>replace</b>	Replace value in range (function template )

## /algorithm/copy\_if

function template

### std::copy\_if

<algorithm>

```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator copy_if (InputIterator first, InputIterator last,
                      OutputIterator result, UnaryPredicate pred);
```

#### Copy certain elements of range

Copies the elements in the range `[first, last)` for which `pred` returns true to the range beginning at `result`.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class OutputIterator, class UnaryPredicate>
2     OutputIterator copy_if (InputIterator first, InputIterator last,
3                             OutputIterator result, UnaryPredicate pred)
4 {
5     while (first!=last) {
6         if (pred(*first)) {
7             *result = *first;
8             ++result;
9         }
10        ++first;
11    }
12    return result;
13 }
```

### Parameters

`first, last`

Input iterators to the initial and final positions in a sequence. The range copied is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`InputIterator` shall point to a type **assignable** to the elements pointed by `OutputIterator`.

`result`

`Output iterator` to the initial position of the range where the resulting sequence is stored. The range includes as many elements as `[first, last)`.

`pred`

Unary function that accepts an element in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether the element is to be copied (if `true`, it is copied).

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

The ranges shall not overlap.

### Return value

An iterator pointing to the element that follows the last element written in the result sequence.

### Example

```
1 // copy_if example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::copy_if, std::distance
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> foo = {25,15,5,-5,-15};
8     std::vector<int> bar (foo.size());
9
10 // copy only positive numbers:
11 auto it = std::copy_if (foo.begin(), foo.end(), bar.begin(), [](int i){return !(i<0);} );
12 bar.resize(std::distance(bar.begin(),it)); // shrink container to new size
13
14 std::cout << "bar contains:";
15 for (int& x: bar) std::cout << ' ' << x;
16 std::cout << '\n';
17
18 return 0;
19 }
```

Output:

bar contains: 25 15 5

### Complexity

Linear in the `distance` between `first` and `last`: Applies `pred` to each element in the range and performs at most that many assignments.

### Data races

The objects in the range `[first, last)` are accessed.

The objects in the range between `result` and the returned value are modified.

## Exceptions

Throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">copy</a>	Copy range of elements (function template )
<a href="#">replace_copy_if</a>	Copy range replacing value (function template )
<a href="#">remove_copy_if</a>	Copy range removing values (function template )

## /algorithm/copy\_n

function template

### std::copy\_n

<algorithm>

```
template <class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n (InputIterator first, Size n, OutputIterator result);
```

#### Copy elements

Copies the first *n* elements from the range beginning at *first* into the range beginning at *result*.

The function returns an iterator to the end of the destination range (which points to one past the last element copied).

If *n* is negative, the function does nothing.

If the ranges overlap, some of the elements in the range pointed by *result* may have undefined but valid values.

The behavior of this function template is equivalent to:

```
1 template<class InputIterator, class Size, class OutputIterator>
2     OutputIterator copy_n (InputIterator first, Size n, OutputIterator result)
3 {
4     while (n>0) {
5         *result = *first;
6         ++result; ++first;
7         --n;
8     }
9     return result;
10 }
```

## Parameters

first

Input iterators to the initial position in a sequence of at least *n* elements to be copied.  
InputIterator shall point to a type assignable to the elements pointed by OutputIterator.

n

Number of elements to copy.  
If this value is negative, the function does nothing.  
Size shall be (convertible to) an integral type.

result

Output iterator to the initial position in the destination sequence of at least *n* elements.  
This shall not point to any element in the range [first, last).

## Return value

An iterator to the end of the destination range where elements have been copied.

## Example

```
1 // copy_n algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::copy
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[]={10,20,30,40,50,60,70};
8     std::vector<int> myvector;
9
10    myvector.resize(7);    // allocate space for 7 elements
11
12    std::copy_n ( myints, 7, myvector.begin() );
13
14    std::cout << "myvector contains:";
15    for (std::vector<int>::iterator it = myvector.begin(); it!=myvector.end(); ++it)
16        std::cout << ' ' << *it;
17
18    std::cout << '\n';
19
20    return 0;
21 }
```

Output:

```
myvector contains: 10 20 30 40 50 60 70
```

## Complexity

Linear in the `distance` between `first` and `last`: Performs an assignment operation for each element in the range.

## Data races

The objects in the range of  $n$  elements pointed by `first` are accessed (each object is accessed exactly once).  
The objects in the range between `result` and the returned value are modified (each object is modified exactly once).

## Exceptions

Throws if either an element assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">copy_backward</a>	Copy range of elements backward (function template )
<a href="#">fill</a>	Fill range with value (function template )
<a href="#">replace</a>	Replace value in range (function template )

## /algorithm/count

function template

**std::count**

<algorithm>

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

### Count appearances of value in range

Returns the number of elements in the range `[first, last)` that compare equal to `val`.

The function uses operator`==` to compare the individual elements to `val`.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class T>
2     typename iterator_traits<InputIterator>::difference_type
3         count (InputIterator first, InputIterator last, const T& val)
4 {
5     typename iterator_traits<InputIterator>::difference_type ret = 0;
6     while (first!=last) {
7         if (*first == val) ++ret;
8         ++first;
9     }
10    return ret;
11 }
```

## Parameters

`first, last`

Input iterators to the initial and final positions of the sequence of elements. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`val`

Value to match.

`T` shall be a type supporting comparisons with the elements pointed by `InputIterator` using operator`==` (with the elements as left-hand side operands, and `val` as right-hand side).

## Return value

The number of elements in the range `[first, last)` that compare equal to `val`.

The return type (`iterator_traits<InputIterator>::difference_type`) is a signed integral type.

## Example

```
1 // count algorithm example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::count
4 #include <vector>        // std::vector
5
6 int main () {
7     // counting elements in array:
8     int myints[] = {10,20,30,30,20,10,20};    // 8 elements
9     int mycount = std::count (myints, myints+8, 10);
10    std::cout << "10 appears " << mycount << " times.\n";
11
12    // counting elements in container:
13    std::vector<int> myvector (myints, myints+8);
14    mycount = std::count (myvector.begin(), myvector.end(), 20);
15    std::cout << "20 appears " << mycount << " times.\n";
16
17    return 0;
18 }
```

Output:

```
10 appears 3 times.
20 appears 3 times.
```

## Complexity

Linear in the `distance` between `first` and `last`: Compares once each element.

## Data races

The objects in the range `[first, last)` are accessed (each object is accessed exactly once).

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<code>for_each</code>	Apply function to range (function template )
<code>count_if</code>	Return number of elements in range satisfying condition (function template )
<code>find</code>	Find value in range (function template )
<code>replace</code>	Replace value in range (function template )

## /algorithm/count\_if

function template

### std::count\_if

<algorithm>

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

#### Return number of elements in range satisfying condition

Returns the number of elements in the range `[first, last)` for which `pred` is true.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class UnaryPredicate>
2   typename iterator_traits<InputIterator>::difference_type
3     count_if (InputIterator first, InputIterator last, UnaryPredicate pred)
4   {
5     typename iterator_traits<InputIterator>::difference_type ret = 0;
6     while (first!=last) {
7       if (pred(*first)) ++ret;
8       ++first;
9     }
10    return ret;
11 }
```

## Parameters

`first, last`

Input iterators to the initial and final positions of the sequence of elements. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`pred`

Unary function that accepts an element in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether the element is counted by this function.

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

The number of elements in the range `[first, last)` for which `pred` does not return `false`.

The return type (`iterator_traits<InputIterator>::difference_type`) is a signed integral type.

## Example

```
1 // count_if example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::count_if
4 #include <vector>             // std::vector
5
6 bool IsOdd (int i) { return ((i%2)==1); }
7
8 int main () {
9   std::vector<int> myvector;      // myvector: 1 2 3 4 5 6 7 8 9
10  for (int i=1; i<10; i++) myvector.push_back(i);
11
12  int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
13  std::cout << "myvector contains " << mycount << " odd values.\n";
14
15  return 0;
16 }
```

Output:

```
myvector contains 5 odd values.
```

## Complexity

Linear in the distance between *first* and *last*: Calls *pred* once for each element.

## Data races

The objects in the range [first, last) are accessed (each object is accessed exactly once).

## Exceptions

Throws if *pred* throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

#### **See also**

<b>count</b>	Count appearances of value in range (function template )
<b>for_each</b>	Apply function to range (function template )
<b>find</b>	Find value in range (function template )

## /algorithm/equal

## function template

## `std::equal`

<algorithm>

```
template <class InputIterator1, class InputIterator2>
equality (1)  bool equal (InputIterator1 first1, InputIterator1 last1,
                      InputIterator2 first2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
predicate (2)  bool equal (InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, BinaryPredicate pred);
```

### **Test whether the elements in two ranges are equal**

Compares the elements in the range `[first1, last1)` with those in the range beginning at `first2`, and returns `true` if all of the elements in both ranges match.

The elements are compared using operator== (or *pred*, in version (2)).

The behavior of this function template is equivalent to:

```
1 template <class InputIterator1, class InputIterator2>
2     bool equal ( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2 )
3 {
4     while (first1!=last1) {
5         if (!(*first1 == *first2)) // or: if (!pred(*first1,*first2)), for version 2
6             return false;
7         ++first1; ++first2;
8     }
9     return true;
10 }
```

## Parameters

`first1, last1`

**Input iterators** to the initial and final positions of the first sequence. The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

first2

`Input iterator` to the initial position of the second sequence. The comparison includes up to as many elements of this sequence as those in the range `[first1, last1]`.

pred

Binary function that accepts two elements as argument (one of each of the two sequences, in the same order), and returns a value convertible to `bool`. The value returned indicates whether the elements are considered to match in the context of this function. The function shall not modify any of its arguments. This can either be a function pointer or a function object.

## **Return value**

true if all the elements in the range [first1, last1) compare equal to those of the range starting at first2, and false otherwise.

## **Example**

```

21 // using predicate comparison:
22 if ( std::equal (myvector.begin(), myvector.end(), myints, mypredicate) )
23   std::cout << "The contents of both sequences are equal.\n";
24 else
25   std::cout << "The contents of both sequences differ.\n";
26
27 return 0;
28 }
```

#### Output:

The contents of both sequences are equal.  
The contents of both sequence differ.

### Complexity

Up to linear in the distance between *first1* and *last1*: Compares elements until a mismatch is found.

### Data races

Some (or all) of the objects in both ranges are accessed (once at most).

### Exceptions

Throws if any of the element comparisons (or *pred*) throws, or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

### See also

<a href="#">mismatch</a>	Return first position where two ranges differ (function template )
<a href="#">find_first_of</a>	Find element from set in range (function template )
<a href="#">find_end</a>	Find last subsequence in range (function template )
<a href="#">search</a>	Search range for subsequence (function template )

## /algorithm/equal\_range

function template

### std::equal\_range

<algorithm>

```

template <class ForwardIterator, class T>
default (1) pair<ForwardIterator,ForwardIterator>
           equal_range (ForwardIterator first, ForwardIterator last, const T& val);
template <class ForwardIterator, class T, class Compare>
custom (2)  pair<Forwarditerator,Forwarditerator>
            equal_range (ForwardIterator first, ForwardIterator last, const T& val,
                          Compare comp);
```

#### Get subrange of equal elements

Returns the bounds of the subrange that includes all the elements of the range [*first*,*last*] with values equivalent to *val*.

The elements are compared using operator< for the first version, and *comp* for the second. Two elements, *a* and *b* are considered equivalent if *(!(a<b) && !(b<a))* or if *(!comp(a,b) && !comp(b,a))*.

The elements in the range shall already be *sorted* according to this same criterion (operator< or *comp*), or at least *partitioned* with respect to *val*.

If *val* is not equivalent to any value in the range, the subrange returned has a length of zero, with both iterators pointing to the nearest value greater than *val*, if any, or to *last*, if *val* compares greater than all the elements in the range.

The behavior of this function template is equivalent to:

```

1 template <class ForwardIterator, class T>
2   pair<ForwardIterator,ForwardIterator>
3     equal_range (ForwardIterator first, ForwardIterator last, const T& val)
4 {
5   ForwardIterator it = std::lower_bound (first,last,val);
6   return std::make_pair ( it, std::upper_bound(it,last,val) );
7 }
```

### Parameters

*first*, *last*

Forward iterators to the initial and final positions of a *sorted* (or properly *partitioned*) sequence. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*val*

Value of the subrange to search for in the range.

For (1), *T* shall be a type supporting being compared with elements of the range [*first*,*last*] as either operand of operator<.

*comp*

Binary function that accepts two arguments of the type pointed by *ForwardIterator* (and of type *T*), and returns a value convertible to *bool*. The value returned indicates whether the first argument is considered to go before the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

### Return value

A `pair` object, whose member `pair::first` is an iterator to the lower bound of the subrange of equivalent values, and `pair::second` its upper bound. The values are the same as those that would be returned by functions `lower_bound` and `upper_bound` respectively.

## Example

```
1 // equal_range example
2 // equal_range example
3 #include <iostream>           // std::cout
4 #include <algorithm>          // std::equal_range, std::sort
5 #include <vector>             // std::vector
6
7 bool mygreater (int i,int j) { return (i>j); }
8
9 int main () {
10    int myints[] = {10,20,30,30,20,10,10,20};
11    std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
12    std::pair<std::vector<int>::iterator, std::vector<int>::iterator> bounds;
13
14    // using default comparison:
15    std::sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30
16    bounds=std::equal_range (v.begin(), v.end(), 20); //           ^   ^
17
18    // using "mygreater" as comp:
19    std::sort (v.begin(), v.end(), mygreater);       // 30 30 20 20 20 10 10 10
20    bounds=std::equal_range (v.begin(), v.end(), 20, mygreater); //           ^   ^
21
22    std::cout << "bounds at positions " << (bounds.first - v.begin());
23    std::cout << " and " << (bounds.second - v.begin()) << '\n';
24
25    return 0;
26 }
```

Output:

```
bounds at positions 2 and 5
```

## Complexity

On average, up to twice logarithmic in the `distance` between `first` and `last`: Performs approximately  $2*\log_2(N)+1$  element comparisons (where  $N$  is this distance). On *non-random-access iterators*, the iterator `advances` produce themselves an additional up to twice linear complexity in  $N$  on average.

## Data races

The objects in the range `[first, last)` are accessed.

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<code>lower_bound</code>	Return iterator to lower bound (function template )
<code>upper_bound</code>	Return iterator to upper bound (function template )
<code>binary_search</code>	Test if value exists in sorted sequence (function template )

# /algorithm/fill

function template

## std::fill

<algorithm>

```
template <class ForwardIterator, class T>
void fill (ForwardIterator first, ForwardIterator last, const T& val);
```

### Fill range with value

Assigns `val` to all the elements in the range `[first, last)`.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class T>
2 void fill (ForwardIterator first, ForwardIterator last, const T& val)
3 {
4     while (first != last) {
5         *first = val;
6         ++first;
7     }
8 }
```

## Parameters

`first, last`

Forward iterators to the initial and final positions in a sequence of elements that support being assigned a value of type `T`. The range filled is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`val`

Value to assign to the elements in the filled range.

## Return value

none

## Example

```
1 // fill algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::fill
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> myvector (8);           // myvector: 0 0 0 0 0 0 0 0
8
9     std::fill (myvector.begin(),myvector.begin()+4,5);    // myvector: 5 5 5 5 0 0 0 0
10    std::fill (myvector.begin()+3,myvector.end()-2,8);   // myvector: 5 5 5 8 8 8 0 0
11
12    std::cout << "myvector contains:";
13    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
14        std::cout << ' ' << *it;
15    std::cout << '\n';
16
17    return 0;
18 }
```

Output:

```
myvector contains: 5 5 5 8 8 8 0 0
```

## Complexity

Linear in the *distance* between *first* and *last*: Assigns a value to each element.

## Data races

The objects in the range [*first*,*last*) are modified (each object is accessed exactly once).

## Exceptions

Throws if either an element assignment or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">fill_n</a>	Fill sequence with value (function template )
<a href="#">generate</a>	Generate values for range with function (function template )
<a href="#">replace</a>	Replace value in range (function template )
<a href="#">for_each</a>	Apply function to range (function template )

## /algorithm/fill\_n

function template

### std::fill\_n

<algorithm>

```
template <class OutputIterator, class Size, class T>
void fill_n (OutputIterator first, Size n, const T& val);

template <class OutputIterator, class Size, class T>
OutputIterator fill_n (OutputIterator first, Size n, const T& val);
```

#### Fill sequence with value

Assigns *val* to the first *n* elements of the sequence pointed by *first*.

The behavior of this function template is equivalent to:

```
1 template <class OutputIterator, class Size, class T>
2     OutputIterator fill_n (OutputIterator first, Size n, const T& val)
3 {
4     while (n>0) {
5         *first = val;
6         ++first; --n;
7     }
8     return first;    // since C++11
9 }
```

## Parameters

*first*

Output iterators to the initial position in a sequence of at least *n* elements that support being assigned a value of type *T*.

*n*

Number of elements to fill.

This value shall not be negative.

If negative, the function does nothing.

*size* shall be (convertible to) an integral type.

*val*

Value to be used to fill the range.

## Return value

none

An iterator pointing to the element that follows the last element filled.

## Example

```
1 // fill_n example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::fill_n
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> myvector (8,10);           // myvector: 10 10 10 10 10 10 10 10
8
9     std::fill_n (myvector.begin(),4,20);         // myvector: 20 20 20 20 10 10 10 10
10    std::fill_n (myvector.begin() + 3,3,33);      // myvector: 20 20 20 33 33 33 10 10
11
12    std::cout << "myvector contains:";
13    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
14        std::cout << ' ' << *it;
15    std::cout << '\n';
16
17    return 0;
18 }
```

Output:

```
myvector contains: 20 20 20 33 33 33 10 10
```

## Complexity

Linear in  $n$ : Assigns a value to each element.

## Data races

The  $n$  first objects at the range pointed by *first* are modified (each object is modified exactly once).

## Exceptions

Throws if either an element assignment or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">fill</a>	Fill range with value ( <a href="#">function template</a> )
<a href="#">generate_n</a>	Generate values for sequence with function ( <a href="#">function template</a> )
<a href="#">replace</a>	Replace value in range ( <a href="#">function template</a> )
<a href="#">for_each</a>	Apply function to range ( <a href="#">function template</a> )

# /algorithm/find

function template

## std::find

<algorithm>

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

### Find value in range

Returns an iterator to the first element in the range `[first, last)` that compares equal to *val*. If no such element is found, the function returns *last*.

The function uses operator`==` to compare the individual elements to *val*.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class T>
2     InputIterator find (InputIterator first, InputIterator last, const T& val)
3 {
4     while (first!=last) {
5         if (*first==val) return first;
6         ++first;
7     }
8     return last;
9 }
```

## Parameters

*first*, *last*

Input iterators to the initial and final positions in a sequence. The range searched is `[first, last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*val*

Value to search for in the range.

*T* shall be a type supporting comparisons with the elements pointed by `InputIterator` using operator`==` (with the elements as left-hand side operands, and *val* as right-hand side).

## Return value

An iterator to the first element in the range that compares equal to *val*.  
If no elements match, the function returns *last*.

## Example

```
1 // find example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::find
4 #include <vector>        // std::vector
5
6 int main () {
7     // using std::find with array and pointer:
8     int myints[] = { 10, 20, 30, 40 };
9     int * p;
10
11    p = std::find (myints, myints+4, 30);
12    if (p != myints+4)
13        std::cout << "Element found in myints: " << *p << '\n';
14    else
15        std::cout << "Element not found in myints\n";
16
17    // using std::find with vector and iterator:
18    std::vector<int> myvector (myints,myints+4);
19    std::vector<int>::iterator it;
20
21    it = find (myvector.begin(), myvector.end(), 30);
22    if (it != myvector.end())
23        std::cout << "Element found in myvector: " << *it << '\n';
24    else
25        std::cout << "Element not found in myvector\n";
26
27    return 0;
28 }
```

Output:

```
Element found in myints: 30
Element found in myvector: 30
```

## Complexity

Up to linear in the *distance* between *first* and *last*: Compares elements until a match is found.

## Data races

Some (or all) of the objects in the range [*first*,*last*] are accessed (once at most).

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">search</a>	Search range for subsequence (function template )
<a href="#">binary_search</a>	Test if value exists in sorted sequence (function template )
<a href="#">for_each</a>	Apply function to range (function template )

# /algorithm/find\_end

function template

## std::find\_end

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
equality (1)    ForwardIterator1 find_end (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2);
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
predicate (2)   ForwardIterator1 find_end (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, ForwardIterator2 last2,
                           BinaryPredicate pred);
```

### Find last subsequence in range

Searches the range [*first1*,*last1*) for the last occurrence of the sequence defined by [*first2*,*last2*), and returns an iterator to its first element, or *last1* if no occurrences are found.

The elements in both ranges are compared sequentially using operator== (or *pred*, in version (2)): A subsequence of [*first1*,*last1*) is considered a match only when this is true for **all** the elements of [*first2*,*last2*).

This function returns the last of such occurrences. For an algorithm that returns the first instead, see [search](#).

The behavior of this function template is equivalent to:

```
1 template<class ForwardIterator1, class ForwardIterator2>
2     ForwardIterator1 find_end (ForwardIterator1 first1, ForwardIterator1 last1,
3                               ForwardIterator2 first2, ForwardIterator2 last2)
4 {
5     if (first2==last2) return last1; // specified in C++11
```

```

6     ForwardIterator1 ret = last1;
7
8     while (first1!=last1)
9     {
10         ForwardIterator1 it1 = first1;
11         ForwardIterator2 it2 = first2;
12         while (*it1==*it2) {      // or: while (pred(*it1,*it2)) for version (2)
13             ++it1; ++it2;
14             if (it2==last2) { ret=first1; break; }
15             if (it1==last1) return ret;
16         }
17         ++first1;
18     }
19     return ret;
20 }
21 }
```

## Parameters

`first1, last1`

Forward iterators to the initial and final positions of the searched sequence. The range used is [`first1, last1`), which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2, last2`

Forward iterators to the initial and final positions of the sequence to be searched for. The range used is [`first2, last2`).

For (1), the elements in both ranges shall be of types comparable using operator`==` (with the elements of the first range as left-hand side operands, and those of the second as right-hand side operands).

`pred`

Binary function that accepts two elements as arguments (one of each of the two sequences, in the same order), and returns a value convertible to `bool`. The returned value indicates whether the elements are considered to match in the context of this function.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to the first element of the last occurrence of [`first2, last2`] in [`first1, last1`].

If the sequence is not found, the function returns `last1`.

If [`first2, last2`] is an empty range, the result is unspecified.

If [`first2, last2`] is an empty range, the function returns `last1`.

## Example

```

1 // find_end example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::find_end
4 #include <vector>              // std::vector
5
6 bool myfunction (int i, int j) {
7     return (i==j);
8 }
9
10 int main () {
11     int myints[] = {1,2,3,4,5,1,2,3,4,5};
12     std::vector<int> haystack (myints,myints+10);
13
14     int needle1[] = {1,2,3};
15
16     // using default comparison:
17     std::vector<int>::iterator it;
18     it = std::find_end (haystack.begin(), haystack.end(), needle1, needle1+3);
19
20     if (it!=haystack.end())
21         std::cout << "needle1 last found at position " << (it-haystack.begin()) << '\n';
22
23     int needle2[] = {4,5,1};
24
25     // using predicate comparison:
26     it = std::find_end (haystack.begin(), haystack.end(), needle2, needle2+3, myfunction);
27
28     if (it!=haystack.end())
29         std::cout << "needle2 last found at position " << (it-haystack.begin()) << '\n';
30
31     return 0;
32 }
```

Output:

```
needle1 found at position 5
needle2 found at position 3
```

## Complexity

Up to linear in  $\text{count}_2 * (1 + \text{count}_1 - \text{count}_2)$ , where  $\text{count}_X$  is the distance between `firstX` and `lastX`: Compares elements until the last matching subsequence is found.

## Data races

Some (or all) of the objects in both ranges are accessed (possibly more than once).

## Exceptions

Throws if any element comparison (or call to *pred*) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">search</a>	Search range for subsequence (function template )
<a href="#">find</a>	Find value in range (function template )
<a href="#">find_if</a>	Find element in range (function template )

## /algorithm/find\_first\_of

function template

### std::find\_first\_of

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
equality (1)    ForwardIterator1 find_first_of (ForwardIterator1 first1, ForwardIterator1 last1,
                                                ForwardIterator2 first2, ForwardIterator2 last2);
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
predicate (2)   ForwardIterator1 find_first_of (ForwardIterator1 first1, ForwardIterator1 last1,
                                                ForwardIterator2 first2, ForwardIterator2 last2,
                                                BinaryPredicate pred);

template <class InputIterator, class ForwardIterator>
equality (1)    InputIterator find_first_of (InputIterator first1, InputIterator last1,
                                              ForwardIterator first2, ForwardIterator last2);
template <class InputIterator, class ForwardIterator, class BinaryPredicate>
predicate (2)   InputIterator find_first_of (InputIterator first1, InputIterator last1,
                                              ForwardIterator first2, ForwardIterator last2,
                                              BinaryPredicate pred);
```

#### Find element from set in range

Returns an iterator to the first element in the range `[first1, last1]` that matches any of the elements in `[first2, last2]`. If no such element is found, the function returns `last1`.

The elements in `[first1, last1]` are sequentially compared to each of the values in `[first2, last2]` using operator`==` (or `pred`, in version (2)), until a pair matches.

The behavior of this function template is equivalent to:

```
1 template<class InputIterator, class ForwardIterator>
2     InputIterator find_first_of ( InputIterator first1, InputIterator last1,
3                                     ForwardIterator first2, ForwardIterator last2)
4 {
5     while (first1!=last1) {
6         for (ForwardIterator it=first2; it!=last2; ++it) {
7             if (*it==*first1)           // or: if (pred(*it,*first)) for version (2)
8                 return first1;
9         }
10        ++first1;
11    }
12    return last1;
13 }
```

## Parameters

`first1, last1`

Forward iterators to the initial and final positions of the searched sequence. The range used is `[first1, last1]`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

Input iterators to the initial and final positions of the searched sequence. The range used is `[first1, last1]`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2, last2`

Forward iterators to the initial and final positions of the element values to be searched for. The range used is `[first2, last2]`.

For (1), the elements in both ranges shall be of types comparable using operator`==` (with the elements of the first range as left-hand side operands, and those of the second as right-hand side operands).

`pred`

Binary function that accepts two elements as arguments (one of each of the two sequences, in the same order), and returns a value convertible to `bool`. The value returned indicates whether the elements are considered to match in the context of this function.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to the first element in `[first1, last1]` that is part of `[first2, last2]`.

If no matches are found, the function returns `last1`.

If `[first2, last2]` is an empty range, the result is unspecified.

If `[first2, last2]` is an empty range, the function returns `last1`.

## Example

```
1 // find_first_of example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::find_first_of
4 #include <vector>             // std::vector
5 #include <cctype>              // std::tolower
6
7 bool comp_case_insensitive (char c1, char c2) {
```

```

8 |     return (std::tolower(c1)==std::tolower(c2));
9 |
10|
11 int main () {
12     int mychars[] = {'a','b','c','A','B','C'};
13     std::vector<char> haystack (mychars,mychars+6);
14     std::vector<char>::iterator it;
15
16     int needle[] = {'A','B','C'};
17
18     // using default comparison:
19     it = find_first_of (haystack.begin(), haystack.end(), needle, needle+3);
20
21     if (it!=haystack.end())
22         std::cout << "The first match is: " << *it << '\n';
23
24     // using predicate comparison:
25     it = find_first_of (haystack.begin(), haystack.end(),
26                         needle, needle+3, comp_case_insensitive);
27
28     if (it!=haystack.end())
29         std::cout << "The first match is: " << *it << '\n';
30
31     return 0;
32 }

```

Output:

```
The first match is: A
The first match is: a
```

## Complexity

Up to linear in  $\text{count}_1 \times \text{count}_2$  (where  $\text{count}_X$  is the [distance](#) between  $\text{first}_X$  and  $\text{last}_X$ ): Compares elements until a match is found.

## Data races

Some (or all) of the objects in both ranges are accessed (once at most in the case of  $[\text{first}_1, \text{last}_1]$ , and possibly more than once in  $[\text{first}_2, \text{last}_2]$ ).

## Exceptions

Throws if any element comparison (or  $\text{pred}$ ) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">search</a>	Search range for subsequence (function template )
<a href="#">find</a>	Find value in range (function template )
<a href="#">find_if</a>	Find element in range (function template )

## /algorithm/find\_if

function template

### std::find\_if

<algorithm>

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

#### Find element in range

Returns an iterator to the first element in the range  $(\text{first}, \text{last})$  for which  $\text{pred}$  returns true. If no such element is found, the function returns  $\text{last}$ .

The behavior of this function template is equivalent to:

```

1 template<class InputIterator, class UnaryPredicate>
2     InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
3 {
4     while (first!=last) {
5         if (pred(*first)) return first;
6         ++first;
7     }
8     return last;
9 }
```

## Parameters

`first, last`

[Input iterators](#) to the initial and final positions in a sequence. The range used is  $[\text{first}, \text{last})$ , which contains all the elements between  $\text{first}$  and  $\text{last}$ , including the element pointed by  $\text{first}$  but not the element pointed by  $\text{last}$ .

`pred`

Unary function that accepts an element in the range as argument and returns a value convertible to `bool`. The value returned indicates whether the element is considered a match in the context of this function.

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

An iterator to the first element in the range for which  $\text{pred}$  does not return `false`.  
If  $\text{pred}$  is `false` for all elements, the function returns  $\text{last}$ .

## Example

```
1 // find_if example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::find_if
4 #include <vector>        // std::vector
5
6 bool IsOdd (int i) {
7     return ((i%2)==1);
8 }
9
10 int main () {
11     std::vector<int> myvector;
12
13     myvector.push_back(10);
14     myvector.push_back(25);
15     myvector.push_back(40);
16     myvector.push_back(55);
17
18     std::vector<int>::iterator it = std::find_if (myvector.begin(), myvector.end(), IsOdd);
19     std::cout << "The first odd value is " << *it << '\n';
20
21     return 0;
22 }
```

Output:

```
The first odd value is 25
```

## Complexity

Up to linear in the *distance* between *first* and *last*: Calls *pred* for each element until a match is found.

## Data races

Some (or all) of the objects in the range [*first*,*last*] are accessed (once at most).

## Exceptions

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

## See also

<a href="#">find</a>	Find value in range (function template )
<a href="#">for_each</a>	Apply function to range (function template )

## /algorithm/find\_if\_not

function template

### std::find\_if\_not

<algorithm>

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if_not (InputIterator first, InputIterator last, UnaryPredicate pred);
```

#### Find element in range (negative condition)

Returns an iterator to the first element in the range [*first*,*last*] for which *pred* returns *false*. If no such element is found, the function returns *last*.

The behavior of this function template is equivalent to:

```
1 template<class InputIterator, class UnaryPredicate>
2     InputIterator find_if_not (InputIterator first, InputIterator last, UnaryPredicate pred)
3 {
4     while (first!=last) {
5         if (!pred(*first)) return first;
6         ++first;
7     }
8     return last;
9 }
```

## Parameters

*first*, *last*

Input iterators to the initial and final positions in a sequence. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*pred*

Unary function that accepts an element in the range as argument and returns a value convertible to *bool*. The value returned indicates whether the element is considered a match in the context of this function.  
The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

An iterator to the first element in the range for which *pred* returns *false*.

If *pred* is true for all elements, the function returns *last*.

## Example

```
1 // find_if_not example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::find_if_not
4 #include <array>              // std::array
5
6 int main () {
7     std::array<int,5> foo = {1,2,3,4,5};
8
9     std::array<int,5>::iterator it =
10    std::find_if_not (foo.begin(), foo.end(), [](int i){return i%2;});
11    std::cout << "The first even value is " << *it << '\n';
12
13    return 0;
14 }
```

Output:

```
The first even value is 2
```

## Complexity

Up to linear in the distance between *first* and *last*: Calls *pred* for each element until a mismatch is found.

## Data races

Some (or all) of the objects in the range [*first*,*last*] are accessed (once at most).

## Exceptions

Throws if either *pred* or an operation on an iterator throws.

Note that invalid parameters cause *undefined behavior*.

## See also

<a href="#">find_if</a>	Find element in range (function template )
<a href="#">mismatch</a>	Return first position where two ranges differ (function template )
<a href="#">none_of</a>	Test if no elements fulfill condition (function template )

# /algorithm/for\_each

function template

## std::for\_each

<algorithm>

```
template <class InputIterator, class Function>
Function for_each (InputIterator first, InputIterator last, Function fn);
```

### Apply function to range

Applies function *fn* to each of the elements in the range [*first*,*last*].

The behavior of this template function is equivalent to:

```
1 template<class InputIterator, class Function>
2     Function for_each(InputIterator first, InputIterator last, Function fn)
3 {
4     while (first!=last) {
5         fn (*first);
6         ++first;
7     }
8     return fn;      // or, since C++11: return move(fn);
9 }
```

## Parameters

*first*, *last*

Input iterators to the initial and final positions in a sequence. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*fn*

Unary function that accepts an element in the range as argument.

This can either be a function pointer or a [move](#) constructible function object.

Its return value, if any, is ignored.

## Return value

Returns *fn*.

Returns *fn*, as if calling [std::move](#)(*fn*).

## Example

```
1 // for_each example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::for_each
4 #include <vector>              // std::vector
5
6 void myfunction (int i) { // function:
```

```

7   std::cout << ' ' << i;
8 }
9
10 struct myclass {           // function object type:
11     void operator() (int i) {std::cout << ' ' << i;}
12 } myobject;
13
14 int main () {
15     std::vector<int> myvector;
16     myvector.push_back(10);
17     myvector.push_back(20);
18     myvector.push_back(30);
19
20     std::cout << "myvector contains:";
21     for_each (myvector.begin(), myvector.end(), myfunction);
22     std::cout << '\n';
23
24     // or:
25     std::cout << "myvector contains:";
26     for_each (myvector.begin(), myvector.end(), myobject);
27     std::cout << '\n';
28
29     return 0;
30 }

```

Output:

```

myvector contains: 10 20 30
myvector contains: 10 20 30

```

## Complexity

Linear in the *distance* between *first* and *last*: Applies *fn* to each element.

## Data races

The objects in the range [*first*,*last*) are accessed (each object is accessed exactly once). These objects may be modified if *InputIterator* is a *mutable iterator* type and *fn* is not a constant function.

## Exceptions

Throws if *fn* throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">transform</a>	Transform range (function template )
<a href="#">find</a>	Find value in range (function template )
<a href="#">search</a>	Search range for subsequence (function template )

# /algorithm/generate

function template

## std::generate

<algorithm>

```

template <class ForwardIterator, class Generator>
void generate (ForwardIterator first, ForwardIterator last, Generator gen);

```

### Generate values for range with function

Assigns the value returned by successive calls to *gen* to the elements in the range [*first*,*last*).

The behavior of this function template is equivalent to:

```

1 template <class ForwardIterator, class Generator>
2     void generate ( ForwardIterator first, ForwardIterator last, Generator gen )
3 {
4     while (first != last) {
5         *first = gen();
6         ++first;
7     }
8 }

```

## Parameters

*first*, *last*

Forward iterators to the initial and final positions in a sequence. The range affected is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*gen*

Generator function that is called with no arguments and returns some value of a type convertible to those pointed by the iterators.  
This can either be a function pointer or a function object.

## Return value

none

## Example

```

1 // generate algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::generate
4 #include <vector>             // std::vector
5 #include <ctime>              // std::time
6 #include <cstdlib>            // std::rand, std::srand
7
8 // function generator:
9 int RandomNumber () { return (std::rand()%100); }
10
11 // class generator:
12 struct c_unique {
13     int current;
14     c_unique() {current=0;}
15     int operator()() {return ++current;}
16 } UniqueNumber;
17
18 int main () {
19     std::srand ( unsigned ( std::time(0) ) );
20
21     std::vector<int> myvector (8);
22
23     std::generate (myvector.begin(), myvector.end(), RandomNumber);
24
25     std::cout << "myvector contains:";
26     for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
27         std::cout << ' ' << *it;
28     std::cout << '\n';
29
30     std::generate (myvector.begin(), myvector.end(), UniqueNumber);
31
32     std::cout << "myvector contains:";
33     for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
34         std::cout << ' ' << *it;
35     std::cout << '\n';
36
37     return 0;
38 }
```

A possible output:

```
myvector contains: 57 87 76 66 85 54 17 15
myvector contains: 1 2 3 4 5 6 7 8
```

## Complexity

Linear in the *distance* between *first* and *last*: Calls *gen* and performs an assignment for each element.

## Data races

The objects in the range [*first*,*last*) are modified (each object is accessed exactly once).

## Exceptions

Throws if any of *gen*, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">generate_n</a>	Generate values for sequence with function ( <a href="#">function template</a> )
<a href="#">fill</a>	Fill range with value ( <a href="#">function template</a> )
<a href="#">for_each</a>	Apply function to range ( <a href="#">function template</a> )

## /algorithm/generate\_n

function template

### std::generate\_n

<algorithm>

```

template <class OutputIterator, class Size, class Generator>
void generate_n (OutputIterator first, Size n, Generator gen);

template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n (OutputIterator first, Size n, Generator gen);
```

#### Generate values for sequence with function

Assigns the value returned by successive calls to *gen* to the first *n* elements of the sequence pointed by *first*.

The behavior of this function template is equivalent to:

```

1 template <class OutputIterator, class Size, class Generator>
2 void generate_n ( OutputIterator first, Size n, Generator gen )
3 {
4     while (n>0) {
5         *first = gen();
6         ++first; --n;
7     }
8 }
```

## Parameters

*first*

Output iterators to the initial positions in a sequence of at least  $n$  elements that support being assigned a value of the type returned by  $gen$ .

**n**

Number of values to generate.

This value shall not be negative.

If negative, the function does nothing.

Size shall be (convertible to) an integral type.

**gen**

Generator function that is called with no arguments and returns some value of a type convertible to those pointed by the iterators.  
This can either be a function pointer or a function object.

## Return value

**none**

An iterator pointing to the element that follows the last element whose value has been generated.

## Example

```
1 // generate_n example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::generate_n
4
5 int current = 0;
6 int UniqueNumber () { return ++current; }
7
8 int main () {
9     int myarray[9];
10
11    std::generate_n (myarray, 9, UniqueNumber);
12
13    std::cout << "myarray contains:";
14    for (int i=0; i<9; ++i)
15        std::cout << ' ' << myarray[i];
16    std::cout << '\n';
17
18    return 0;
19 }
```

A possible output:

```
myarray contains: 1 2 3 4 5 6 7 8 9
```

## Complexity

Linear in  $n$ : Calls  $gen$  and performs an assignment for each element.

## Data races

The  $n$  first objects at the range pointed by  $first$  are modified (each object is modified exactly once).

## Exceptions

Throws if any of  $gen$ , the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<b>generate</b>	Generate values for range with function (function template )
<b>fill_n</b>	Fill sequence with value (function template )
<b>for_each</b>	Apply function to range (function template )

# /algorithm/includes

function template

## std::includes

<algorithm>

```
template <class InputIterator1, class InputIterator2>
bool includes ( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2 );

template <class InputIterator1, class InputIterator2, class Compare>
bool includes ( InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2, Compare comp );
```

### Test whether sorted range includes another sorted range

Returns true if the sorted range  $[first1, last1]$  contains all the elements in the sorted range  $[first2, last2]$ .

The elements are compared using  $\operatorname{operator}<$  for the first version, and  $comp$  for the second. Two elements,  $a$  and  $b$  are considered equivalent if  $(!(a < b) \&& !(b < a))$  or if  $(!comp(a,b) \&& !comp(b,a))$ .

The elements in the range shall already be ordered according to this same criterion ( $\operatorname{operator}<$  or  $comp$ ).

The behavior of this function template is equivalent to:

```
1 template <class InputIterator1, class InputIterator2>
2     bool includes (InputIterator1 first1, InputIterator1 last1,
3                     InputIterator2 first2, InputIterator2 last2)
```

```

4 {
5     while (first2!=last2) {
6         if ( (first1==last1) || (*first2<*first1) ) return false;
7         if (!(*first1<*first2)) ++first2;
8         ++first1;
9     }
10    return true;
11 }

```

## Parameters

`first1, last1`

`Input iterators` to the initial and final positions of the first sorted sequence (which is tested on whether it contains the second sequence). The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2, last2`

`Input iterators` to the initial and final positions of the second sorted sequence (which is tested on whether it is contained in the first sequence). The range used is `[first2, last2)`.

`comp`

Binary function that accepts two elements as arguments (one from each of the two sequences, in the same order), and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

`true` if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`, `false` otherwise.

If `[first2, last2)` is an empty range, the result is unspecified.

If `[first2, last2)` is an empty range, the function returns `true`.

## Example

```

1 // includes algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::includes, std::sort
4
5 bool myfunction (int i, int j) { return i<j; }
6
7 int main () {
8     int container[] = {5,10,15,20,25,30,35,40,45,50};
9     int continent[] = {40,30,20,10};
10
11    std::sort (container,container+10);
12    std::sort (continent,continent+4);
13
14 // using default comparison:
15    if ( std::includes(container,container+10,continent,continent+4) )
16        std::cout << "container includes continent!\n";
17
18 // using myfunction as comp:
19    if ( std::includes(container,container+10,continent,continent+4, myfunction) )
20        std::cout << "container includes continent!\n";
21
22    return 0;
23 }

```

Output:

```

container includes continent!
container includes continent!

```

## Complexity

Up to linear in twice the `distances` in both ranges: Performs up to  $2*(\text{count}_1+\text{count}_2)-1$  comparisons (where  $\text{count}_X$  is the `distance` between `firstX` and `lastX`).

## Data races

Some (or all) of the objects in both ranges are accessed (twice each at most).

## Exceptions

Throws if any element comparison (or call to `comp`) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<code>search</code>	Search range for subsequence (function template )
<code>find_end</code>	Find last subsequence in range (function template )
<code>equal</code>	Test whether the elements in two ranges are equal (function template )

## /algorithm/inplace\_merge

function template

`std::inplace_merge`

<algorithm>

---

```

template <class BidirectionalIterator>
default (1) void inplace_merge (BidirectionalIterator first, BidirectionalIterator middle,
                           BidirectionalIterator last);
template <class BidirectionalIterator, class Compare>
custom (2) void inplace_merge (BidirectionalIterator first, BidirectionalIterator middle,
                           BidirectionalIterator last, Compare comp);

```

### Merge consecutive sorted ranges

Merges two consecutive sorted ranges: `[first, middle)` and `[middle, last)`, putting the result into the combined sorted range `[first, last)`.

The elements are compared using `operator<` for the first version, and `comp` for the second. The elements in both ranges shall already be ordered according to this same criterion (`operator<` or `comp`). The resulting range is also sorted according to this.

The function preserves the relative order of elements with equivalent values, with the elements in the first range preceding those equivalent in the second.

### Parameters

`first`  
Bidirectional iterator to the initial position in the first sorted sequence to merge. This is also the initial position where the resulting merged range is stored.

`middle`  
Bidirectional iterator to the initial position of the second sorted sequence, which because both sequences must be consecutive, matches the *past-the-end* position of the first sequence.

`last`  
Bidirectional iterator to the *past-the-end* position of the second sorted sequence. This is also the *past-the-end* position of the range where the resulting merged range is stored.

`comp`  
Binary function that accepts two arguments of the types pointed by the iterators, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

BidirectionalIterator shall point to a type for which `swap` is properly defined and which is both *move-constructible* and *move-assignable*.

### Return value

none

### Example

```

1 // inplace_merge example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::inplace_merge, std::sort, std::copy
4 #include <vector>             // std::vector
5
6 int main () {
7     int first[] = {5,10,15,20,25};
8     int second[] = {50,40,30,20,10};
9     std::vector<int> v(10);
10    std::vector<int>::iterator it;
11
12    std::sort (first,first+5);
13    std::sort (second,second+5);
14
15    it=std::copy (first, first+5, v.begin());
16    std::copy (second,second+5,it);
17
18    std::inplace_merge (v.begin(),v.begin()+5,v.end());
19
20    std::cout << "The resulting vector contains:";
21    for (it=v.begin(); it!=v.end(); ++it)
22        std::cout << ' ' << *it;
23    std::cout << '\n';
24
25    return 0;
26 }

```

Output:

```
The resulting vector contains: 5 10 10 15 20 20 25 30 40 50
```

### Complexity

If enough extra memory is available, linear in the *distance* between `first` and `last`: Performs  $N-1$  comparisons and up to twice that many element moves. Otherwise, up to linearithmic: Performs up to  $N \log(N)$  element comparisons (where  $N$  is the distance above), and up to that many element swaps.

### Data races

The objects in the range `[first, last)` are modified.

### Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

### See also

<a href="#">merge</a>	Merge sorted ranges (function template )
<a href="#">partition</a>	Partition range in two (function template )

## /algorithm/is\_heap

function template

**std::is\_heap**

&lt;algorithm&gt;

```
default (1) template <class RandomAccessIterator>
    bool is_heap (RandomAccessIterator first, RandomAccessIterator last);
custom (2)  template <class RandomAccessIterator, class Compare>
    bool is_heap (RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
```

### Test if range is heap

Returns `true` if the range `[first, last)` forms a *heap*, as if constructed with `make_heap`.

The elements are compared using `operator<` for the first version, and `comp` for the second.

### Parameters

`first, last`

`RandomAccess` iterators to the initial and final positions of the sequence. The range checked is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as `first` argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

### Return value

`true` if the range `[first, last)` is a *heap* (as if constructed with `make_heap`), `false` otherwise.

If the range `[first, last)` contains less than two elements, the function always returns `true`.

### Example

```
1 // is_heap example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::is_heap, std::make_heap, std::pop_heap
4 #include <vector>        // std::vector
5
6 int main () {
7     std::vector<int> foo {9,5,2,6,4,1,3,8,7};
8
9     if (!std::is_heap(foo.begin(),foo.end()))
10     std::make_heap(foo.begin(),foo.end());
11
12     std::cout << "Popping out elements:";
13     while (!foo.empty()) {
14         std::pop_heap(foo.begin(),foo.end()); // moves largest element to back
15         std::cout << ' ' << foo.back(); // prints back
16         foo.pop_back(); // pops element out of container
17     }
18     std::cout << '\n';
19
20     return 0;
21 }
```

Output:

```
Popping out elements: 9 8 7 6 5 4 3 2 1
```

### Complexity

Up to linear in one less than the `distance` between `first` and `last`: Compares pairs of elements until a mismatch is found.

### Data races

The objects in the range `[first, last)` are accessed.

### Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

### See also

<a href="#">is_heap_until</a>	Find first element not in heap order (function template )
<a href="#">make_heap</a>	Make heap from range (function template )
<a href="#">sort_heap</a>	Sort elements of heap (function template )

## /algorithm/is\_heap\_until

function template

## std::is\_heap\_until

<algorithm>

```
template <class RandomAccessIterator>
default (1)     RandomAccessIterator is_heap_until (RandomAccessIterator first,
                                                    RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
custom (2)      RandomAccessIterator is_heap_until (RandomAccessIterator first,
                                                    RandomAccessIterator last
                                                    Compare comp);
```

### Find first element not in heap order

Returns an iterator to the first element in the range `[first, last)` which is not in a valid position if the range is considered a heap (as if constructed with [make\\_heap](#)).

The range between `first` and the iterator returned **is a heap**.

If the entire range is a valid heap, the function returns `last`.

The elements are compared using `operator<` for the first version, and `comp` for the second.

### Parameters

`first, last`

Random-access iterators to the initial and final positions in a sequence. The range checked is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

### Return value

An iterator to the first element in the range which is not in a valid position for the range to be a heap, or `last` if all elements are validly positioned or if the range contains less than two elements.

### Example

```
1 // is_heap example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::is_heap_until, std::sort, std::reverse
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> foo {2,6,9,3,8,4,5,1,7};
8
9     std::sort(foo.begin(),foo.end());
10    std::reverse(foo.begin(),foo.end());
11
12    auto last = std::is_heap_until (foo.begin(),foo.end());
13
14    std::cout << "The " << (last-foo.begin()) << " first elements are a valid heap:";
15    for (auto it=foo.begin(); it!=last; ++it)
16        std::cout << ' ' << *it;
17    std::cout << '\n';
18
19    return 0;
20 }
```

Most implementations consider a range sorted in reverse order a valid heap:

Possible output:

```
The 9 first elements are a valid heap: 9 8 7 6 5 4 3 2 1
```

### Complexity

Up to linear in the `distance` between `first` and `last`: Compares elements until a mismatch is found.

### Data races

The objects in the range `[first, last)` are accessed.

### Exceptions

Throws if either `comp` or an operation on an iterator throws.

Note that invalid parameters cause *undefined behavior*.

### See also

<a href="#">is_heap</a>	Test if range is heap (function template )
<a href="#">make_heap</a>	Make heap from range (function template )
<a href="#">is_sorted_until</a>	Find first unsorted element in range (function template )

function template

## std::is\_partitioned

<algorithm>

```
template <class InputIterator, class UnaryPredicate>
    bool is_partitioned (InputIterator first, InputIterator last, UnaryPredicate pred);
```

### Test whether range is partitioned

Returns true if all the elements in the range [first, last) for which pred returns true precede those for which it returns false.

If the range is empty, the function returns true.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class UnaryPredicate>
2     bool is_partitioned (InputIterator first, InputIterator last, UnaryPredicate pred)
3 {
4     while (first!=last && pred(*first)) {
5         ++first;
6     }
7     while (first!=last) {
8         if (pred(*first)) return false;
9         ++first;
10    }
11    return true;
12 }
```

## Parameters

first, last

Input iterators to the initial and final positions of the sequence. The range used is [first, last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to bool. The value returned indicates whether the element belongs to the first group (if true, the element is expected before all the elements for which it returns false).

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

true if all the elements in the range [first, last) for which pred returns true precede those for which it returns false.  
Otherwise it returns false.

If the range is empty, the function returns true.

## Example

```
1 // is_partitioned example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::is_partitioned
4 #include <array>              // std::array
5
6 bool IsOdd (int i) { return (i%2)==1; }
7
8 int main () {
9     std::array<int,7> foo {1,2,3,4,5,6,7};
10
11    // print contents:
12    std::cout << "foo:"; for (int& x:foo) std::cout << ' ' << x;
13    if ( std::is_partitioned(foo.begin(),foo.end(),IsOdd) )
14        std::cout << " (partitioned)\n";
15    else
16        std::cout << " (not partitioned)\n";
17
18    // partition array:
19    std::partition (foo.begin(),foo.end(),IsOdd);
20
21    // print contents again:
22    std::cout << "foo:"; for (int& x:foo) std::cout << ' ' << x;
23    if ( std::is_partitioned(foo.begin(),foo.end(),IsOdd) )
24        std::cout << " (partitioned)\n";
25    else
26        std::cout << " (not partitioned)\n";
27
28    return 0;
29 }
```

Possible output:

```
foo: 1 2 3 4 5 6 7 (not partitioned)
foo: 1 7 3 5 4 6 2 (partitioned)
```

## Complexity

Up to linear in the distance between first and last: Calls pred for each element until a mismatch is found.

## Data races

Some (or all) of the objects in the range [first, last) are accessed (once at most).

## Exceptions

Throws if either *pred* or an operation on an iterator throws.  
Note that invalid parameters cause *undefined behavior*.

## See also

<a href="#">partition</a>	Partition range in two (function template )
<a href="#">partition_point</a>	Get partition point (function template )

## /algorithm/is\_permutation

function template

### std::is\_permutation

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
equality (1)    bool is_permutation (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2);
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
predicate (2)   bool is_permutation (ForwardIterator1 first1, ForwardIterator1 last1,
                           ForwardIterator2 first2, BinaryPredicate pred);
```

#### Test whether range is permutation of another

Compares the elements in the range `[first1, last1]` with those in the range beginning at `first2`, and returns `true` if all of the elements in both ranges match, even in a different order.

The elements are compared using `operator==` (or *pred*, in version (2)).

The behavior of this function template is equivalent to:

```
1 template <class InputIterator1, class InputIterator2>
2     bool is_permutation (InputIterator1 first1, InputIterator1 last1,
3                           InputIterator2 first2)
4 {
5     std::tie (first1,first2) = std::mismatch (first1,last1,first2);
6     if (first1==last1) return true;
7     InputIterator2 last2 = first2; std::advance (last2,std::distance(first1,last1));
8     for (InputIterator1 it1=first1; it1!=last1; ++it1) {
9         if (std::find(first1,it1,*it1)==it1) {
10             auto n = std::count (first2,last2,*it1);
11             if (n==0 || std::count (it1,last1,*it1)!=n) return false;
12         }
13     }
14     return true;
15 }
```

## Parameters

`first1, last1`

Input iterators to the initial and final positions of the first sequence. The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2`

Input iterator to the initial position of the second sequence.

The function considers as many elements of this sequence as those in the range `[first1, last1)`.

If this sequence is shorter, it causes *undefined behavior*.

`pred`

Binary function that accepts two elements as argument (one of each of the two sequences, in the same order), and returns a value convertible to `bool`. The value returned indicates whether the elements are considered to match in the context of this function.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

`InputIterator1` and `InputIterator2` shall point to the same type.

## Return value

`true` if all the elements in the range `[first1, last1)` compare equal to those of the range starting at `first2` in any order, and `false` otherwise.

## Example

```
1 // is_permutation example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::is_permutation
4 #include <array>              // std::array
5
6 int main () {
7     std::array<int,5> foo = {1,2,3,4,5};
8     std::array<int,5> bar = {3,1,4,5,2};
9
10    if ( std::is_permutation (foo.begin(), foo.end(), bar.begin()) )
11        std::cout << "foo and bar contain the same elements.\n";
12
13    return 0;
14 }
```

Output:

```
foo and bar contain the same elements.
```

## Complexity

If both sequence are **equal** (with the elements in the same order), linear in the **distance** between *first1* and *last1*. Otherwise, up to quadratic: Performs at most  $n^2$  element comparisons until the result is determined (where *N* is the **distance** between *first1* and *last1*).

## Data races

Some (or all) of the objects in both ranges are accessed (possibly multiple times each).

## Exceptions

Throws if any of the element comparisons (or *pred*) throws, or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

## See also

<b>equal</b>	Test whether the elements in two ranges are equal ( <a href="#">function template</a> )
<b>mismatch</b>	Return first position where two ranges differ ( <a href="#">function template</a> )
<b>next_permutation</b>	Transform range to next permutation ( <a href="#">function template</a> )
<b>prev_permutation</b>	Transform range to previous permutation ( <a href="#">function template</a> )

# /algorithm/is\_sorted

function template

## std::is\_sorted

<algorithm>

```
default (1) template <class ForwardIterator>
    bool is_sorted (ForwardIterator first, ForwardIterator last);
custom (2) template <class ForwardIterator, class Compare>
    bool is_sorted (ForwardIterator first, ForwardIterator last, Compare comp);
```

### Check whether range is sorted

Returns **true** if the range [*first*,*last*] is sorted into ascending order.

The elements are compared using operator< for the first version, and *comp* for the second.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator>
2     bool is_sorted (ForwardIterator first, ForwardIterator last)
3 {
4     if (first==last) return true;
5     ForwardIterator next = first;
6     while (++next!=last) {
7         if (*next<*first)      // or, if (comp(*next,*first)) for version (2)
8             return false;
9         ++first;
10    }
11    return true;
12 }
```

## Parameters

*first*, *last*

**Forward iterators** to the initial and final positions of the sequence. The range checked is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*comp*

Binary function that accepts two elements in the range as arguments, and returns a value convertible to **bool**. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

**true** if the range [*first*,*last*] is sorted into ascending order, **false** otherwise.

If the range [*first*,*last*] contains less than two elements, the function always returns **true**.

## Example

```
1 // is_sorted example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::is_sorted, std::prev_permutation
4 #include <array>              // std::array
5
6 int main () {
7     std::array<int,4> foo {2,4,1,3};
8
9     do {
10        // try a new permutation:
11        std::prev_permutation(foo.begin(),foo.end());
12
13        // print range:
14        std::cout << "foo:";
15        for (int& x:foo) std::cout << ' ' << x;
16        std::cout << '\n';
17
18    } while (!std::is_sorted(foo.begin(),foo.end()));
19 }
```

```

20     std::cout << "the range is sorted!\n";
21
22     return 0;
23 }
```

## Output:

```

foo: 2 3 4 1
foo: 2 3 1 4
foo: 2 1 4 3
foo: 2 1 3 4
foo: 1 4 3 2
foo: 1 4 2 3
foo: 1 3 4 2
foo: 1 3 2 4
foo: 1 2 4 3
foo: 1 2 3 4
the range is sorted!
```

## Complexity

Up to linear in one less than the [distance](#) between *first* and *last*: Compares pairs of elements until a mismatch is found.

## Data races

The objects in the range `[first, last)` are accessed.

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">is_sorted_until</a>	Find first unsorted element in range ( <a href="#">function template</a> )
<a href="#">sort</a>	Sort elements in range ( <a href="#">function template</a> )
<a href="#">partial_sort</a>	Partially sort elements in range ( <a href="#">function template</a> )

## /algorithm/is\_sorted\_until

function template

### std::is\_sorted\_until

<algorithm>

```

default (1) template <class ForwardIterator>
            ForwardIterator is_sorted_until (ForwardIterator first, ForwardIterator last);
custom (2)   template <class ForwardIterator, class Compare>
            ForwardIterator is_sorted_until (ForwardIterator first, ForwardIterator last,
                                              Compare comp);
```

#### Find first unsorted element in range

Returns an iterator to the first element in the range `[first, last)` which does not follow an ascending order.

The range between *first* and the iterator returned [is sorted](#).

If the entire range is sorted, the function returns *last*.

The elements are compared using operator< for the first version, and *comp* for the second.

The behavior of this function template is equivalent to:

```

1 template <class ForwardIterator>
2   ForwardIterator is_sorted_until (ForwardIterator first, ForwardIterator last)
3 {
4     if (first==last) return first;
5     ForwardIterator next = first;
6     while (++next!=last) {
7       if (*next<*first) return next;
8       ++first;
9     }
10    return last;
11 }
```

## Parameters

*first, last*

Forward iterators to the initial and final positions in a sequence. The range checked is `[first, last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*comp*

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to the first element in the range which does not follow an ascending order, or *last* if all elements are sorted or if the range contains less than two elements.

## Example

```
1 // is_sorted_until example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::is_sorted_until, std::prev_permutation
4 #include <array>              // std::array
5
6 int main () {
7     std::array<int,4> foo {2,4,1,3};
8     std::array<int,4>::iterator it;
9
10    do {
11        // try a new permutation:
12        std::prev_permutation(foo.begin(),foo.end());
13
14        // print range:
15        std::cout << "foo:";
16        for (int& x:foo) std::cout << ' ' << x;
17        it=std::is_sorted_until(foo.begin(),foo.end());
18        std::cout << " (" << (it-foo.begin()) << " elements sorted)\n";
19
20    } while (it!=foo.end());
21
22    std::cout << "the range is sorted!\n";
23
24    return 0;
25 }
```

Output:

```
foo: 2 3 4 1 (3 elements sorted)
foo: 2 3 1 4 (2 elements sorted)
foo: 2 1 4 3 (1 elements sorted)
foo: 2 1 3 4 (1 elements sorted)
foo: 1 4 3 2 (2 elements sorted)
foo: 1 4 2 3 (2 elements sorted)
foo: 1 3 4 2 (3 elements sorted)
foo: 1 3 2 4 (2 elements sorted)
foo: 1 2 4 3 (3 elements sorted)
foo: 1 2 3 4 (4 elements sorted)
the range is sorted!
```

## Complexity

Up to linear in the *distance* between *first* and *last*: Calls *comp* for each element until a mismatch is found.

## Data races

Some (or all) of the objects in the range [*first*,*last*] are accessed (once at most).

## Exceptions

Throws if either *comp* or an operation on an iterator throws.

Note that invalid parameters cause *undefined behavior*.

## See also

<a href="#">is_sorted</a>	Check whether range is sorted ( <a href="#">function template</a> )
<a href="#">sort</a>	Sort elements in range ( <a href="#">function template</a> )
<a href="#">find_if</a>	Find element in range ( <a href="#">function template</a> )

# /algorithm/iter\_swap

function template

## std::iter\_swap

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap (ForwardIterator1 a, ForwardIterator2 b);
```

### Exchange values of objects pointed to by two iterators

Swaps the elements pointed to by *a* and *b*.

The function calls *swap* (unqualified) to exchange the elements.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator1, class ForwardIterator2>
2     void iter_swap (ForwardIterator1 a, ForwardIterator2 b)
3 {
4     swap (*a, *b);
5 }
```

## Parameters

*a*, *b*

Forward iterators to the objects to swap.  
swap shall be defined to exchange values of the type pointed to by the iterators.

## Return value

none

## Example

```
1 // iter_swap example          // std::cout
2 #include <iostream>           // std::algorithm
3 #include <algorithm>          // std::iter_swap
4 #include <vector>             // std::vector
5
6 int main () {
7
8     int myints[] = {10,20,30,40,50};           //    myints:  10  20  30  40  50
9     std::vector<int> myvector (4,99);          //  myvector:  99  99  99  99
10
11    std::iter_swap(myints,myvector.begin());    //    myints: [99] 20  30  40  50
12    //  myvector: [10] 99  99  99
13
14    std::iter_swap(myints+3,myvector.begin()+2); //    myints:  99  20  30 [99] 50
15    //  myvector:  10  99 [40] 99
16
17    std::cout << "myvector contains:";
18    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Output:

```
myvector contains: 10 99 40 99
```

## Complexity

Constant: Calls `swap` once.

## Data races

The objects pointed to by both iterators are modified.

## Exceptions

Throws if the call to `swap` throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<code>swap</code>	Exchange values of two objects (function template )
<code>copy</code>	Copy range of elements (function template )
<code>replace</code>	Replace value in range (function template )

# /algorithm/lexicographical\_compare

function template

## std::lexicographical\_compare

<algorithm>

```
template <class InputIterator1, class InputIterator2>
default (1)  bool lexicographical_compare (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2);
template <class InputIterator1, class InputIterator2, class Compare>
custom (2)   bool lexicographical_compare (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2,
                                         Compare comp);
```

### Lexicographical less-than comparison

Returns `true` if the range `[first1,last1)` compares *lexicographically less* than the range `[first2,last2)`.

A *lexicographical comparison* is the kind of comparison generally used to sort words alphabetically in dictionaries; It involves comparing sequentially the elements that have the same position in both ranges against each other until one element is not equivalent to the other. The result of comparing these first non-matching elements is the result of the lexicographical comparison.

If both sequences compare equal until one of them ends, the shorter sequence is *lexicographically less* than the longer one.

The elements are compared using operator`<` for the first version, and `comp` for the second. Two elements, `a` and `b` are considered equivalent if `(!(a<b) && !(b<a))` or if `(!comp(a,b) && !comp(b,a))`.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator1, class InputIterator2>
2     bool lexicographical_compare (InputIterator1 first1, InputIterator1 last1,
3                                     InputIterator2 first2, InputIterator2 last2)
4 {
5     while (first1!=last1)
6     {
```

```

7     if (first2==last2 || *first2<*first1) return false;
8     else if (*first1<*first2) return true;
9     ++first1; ++first2;
10    }
11    return (first2!=last2);
12 }

```

## Parameters

`first1, last1`

Input iterators to the initial and final positions of the first sequence. The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2, last2`

Input iterators to the initial and final positions of the second sequence. The range used is `[first2, last2)`.

`comp`

Binary function that accepts two arguments of the types pointed by the iterators, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

true if the first range compares *lexicographically less* than than the second.

false otherwise (including when all the elements of both ranges are equivalent).

## Example

```

1 // lexicographical_compare example
2 #include <iostream>           // std::cout, std::boolalpha
3 #include <algorithm>          // std::lexicographical_compare
4 #include <cctype>             // std::tolower
5
6 // a case-insensitive comparison function:
7 bool mycomp (char c1, char c2)
8 { return std::tolower(c1)<std::tolower(c2); }
9
10 int main () {
11     char foo[]="Apple";
12     char bar[]="apartment";
13
14     std::cout << std::boolalpha;
15
16     std::cout << "Comparing foo and bar lexicographically (foo<bar):\n";
17
18     std::cout << "Using default comparison (operator<): ";
19     std::cout << std::lexicographical_compare(foo,foo+5,bar,bar+9);
20     std::cout << '\n';
21
22     std::cout << "Using mycomp as comparison object: ";
23     std::cout << std::lexicographical_compare(foo,foo+5,bar,bar+9,mycomp);
24     std::cout << '\n';
25
26     return 0;
27 }

```

The default comparison compares plain ASCII character codes, where 'A' (65) compares less than 'a' (97).

Our `mycomp` function transforms the letters to lowercase before comparing them, so here the first letter not matching is the third ('a' vs 'p').

Output:

```

Comparing foo and bar lexicographically (foo<bar):
Using default comparison (operator<): true
Using mycomp as comparison object: false

```

## Complexity

At most, performs  $2 \times \min(\text{count}_1, \text{count}_2)$  comparisons or applications of `comp` (where `count $X$`  is the distance between `first $X$`  and `last $X$` ).

## Complexity

Up to linear in  $2 \times \min(\text{count}_1, \text{count}_2)$  (where `count $X$`  is the distance between `first $X$`  and `last $X$` ): Compares elements symmetrically until a mismatch is found.

## Data races

The objects in the ranges `[first1, last1)` and `[first2, last2)` are accessed.

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">mismatch</a>	Return first position where two ranges differ ( <a href="#">function template</a> )
<a href="#">equal</a>	Test whether the elements in two ranges are equal ( <a href="#">function template</a> )
<a href="#">search</a>	Search range for subsequence ( <a href="#">function template</a> )

<b>find_end</b>	Find last subsequence in range (function template )
<b>includes</b>	Test whether sorted range includes another sorted range (function template )

## /algorithm/lower\_bound

function template

### std::lower\_bound

<algorithm>

```
template <class ForwardIterator, class T>
default (1) ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last,
                                         const T& val);

template <class ForwardIterator, class T, class Compare>
custom (2)   ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last,
                                         const T& val, Compare comp);
```

#### Return iterator to lower bound

Returns an iterator pointing to the first element in the range `[first, last)` which does not compare less than `val`.

The elements are compared using operator`<` for the first version, and `comp` for the second. The elements in the range shall already be `sorted` according to this same criterion (`operator<` or `comp`), or at least `partitioned` with respect to `val`.

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for `random-access iterators`.

Unlike `upper_bound`, the value pointed by the iterator returned by this function may also be equivalent to `val`, and not only greater.

The behavior of this function template is equivalent to:

```
template <class ForwardIterator, class T>
ForwardIterator lower_bound (ForwardIterator first, ForwardIterator last, const T& val)
{
    ForwardIterator it;
    iterator_traits<ForwardIterator>::difference_type count, step;
    count = distance(first, last);
    while (count>0)
    {
        it = first; step=count/2; advance (it,step);
        if (*it<val) {                                // or: if (comp(*it, val)), for version (2)
            first=++it;
            count-=step+1;
        }
        else count=step;
    }
    return first;
}
```

## Parameters

`first, last`

Forward iterators to the initial and final positions of a `sorted` (or properly `partitioned`) sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`val`

Value of the lower bound to search for in the range.

For (1), `T` shall be a type supporting being compared with elements of the range `[first, last)` as the right-hand side operand of `operator<`.

`comp`

Binary function that accepts two arguments (the first of the type pointed by `ForwardIterator`, and the second, always `val`), and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to the lower bound of `val` in the range.

If all the element in the range compare less than `val`, the function returns `last`.

## Example

```
// lower_bound/upper_bound example
#include <iostream>           // std::cout
#include <algorithm>          // std::lower_bound, std::upper_bound, std::sort
#include <vector>              // std::vector

int main () {
    int myints[] = {10,20,30,30,20,10,10,20};
    std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
    std::sort (v.begin(), v.end());                 // 10 10 10 20 20 20 30 30
    std::vector<int>::iterator low,up;
    low=std::lower_bound (v.begin(), v.end(), 20); // ^
    up= std::upper_bound (v.begin(), v.end(), 20); // ^
    std::cout << "lower_bound at position " << (low- v.begin()) << '\n';
    std::cout << "upper_bound at position " << (up - v.begin()) << '\n';
    return 0;
}
```

Output:

```
lower_bound at position 3
upper_bound at position 6
```

## Complexity

On average, logarithmic in the *distance* between *first* and *last*: Performs approximately  $\log_2(N)+1$  element comparisons (where *N* is this distance).

On *non-random-access iterators*, the iterator *advances* produce themselves an additional linear complexity in *N* on average.

## Data races

The objects in the range [*first*,*last*) are accessed.

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">upper_bound</a>	Return iterator to upper bound (function template )
<a href="#">equal_range</a>	Get subrange of equal elements (function template )
<a href="#">binary_search</a>	Test if value exists in sorted sequence (function template )
<a href="#">min_element</a>	Return smallest element in range (function template )

## /algorithm/make\_heap

function template

### std::make\_heap

<algorithm>

```
default (1) template <class RandomAccessIterator>
    void make_heap (RandomAccessIterator first, RandomAccessIterator last);
custom (2) template <class RandomAccessIterator, class Compare>
    void make_heap (RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp );
```

#### Make heap from range

Rearranges the elements in the range [*first*,*last*) in such a way that they form a *heap*.

A *heap* is a way to organize the elements of a range that allows for fast retrieval of the element with the highest value at any moment (with [pop\\_heap](#)), even repeatedly, while allowing for fast insertion of new elements (with [push\\_heap](#)).

The element with the highest value is always pointed by *first*. The order of the other elements depends on the particular implementation, but it is consistent throughout all heap-related functions of this header.

The elements are compared using operator< (for the first version), or *comp* (for the second): The element with the highest value is an element for which this would return *false* when compared to every other element in the range.

The standard container adaptor [priority\\_queue](#) calls [make\\_heap](#), [push\\_heap](#) and [pop\\_heap](#) automatically to maintain *heap properties* for a container.

## Parameters

*first*, *last*

*Random-access iterators* to the initial and final positions of the sequence to be transformed into a heap. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
*RandomAccessIterator* shall point to a type for which [swap](#) is properly defined and which is both *move-constructible* and *move-assignable*.

*comp*

Binary function that accepts two elements in the range as arguments, and returns a value convertible to *bool*. The value returned indicates whether the element passed as first argument is considered to be less than the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

## Return value

none

## Example

```
1 // range heap example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,5,15};
8     std::vector<int> v(myints,myints+5);
9
10    std::make_heap (v.begin(),v.end());
11    std::cout << "initial max heap : " << v.front() << '\n';
12
13    std::pop_heap (v.begin(),v.end()); v.pop_back();
14    std::cout << "max heap after pop : " << v.front() << '\n';
15
16    v.push_back(99); std::push_heap (v.begin(),v.end());
17    std::cout << "max heap after push: " << v.front() << '\n';
18
```

```

19     std::sort_heap (v.begin(),v.end());
20
21     std::cout << "final sorted range :";
22     for (unsigned i=0; i<v.size(); i++)
23         std::cout << ' ' << v[i];
24
25     std::cout << '\n';
26
27     return 0;
28 }
```

Output:

```

initial max heap  : 30
max heap after pop : 20
max heap after push: 99
final sorted range : 5 10 15 20 99
```

## Complexity

Up to linear in three times the *distance* between *first* and *last*: Compares elements and potentially swaps (or moves) them until rearranged as a heap.

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">push_heap</a>	Push element into heap range (function template )
<a href="#">pop_heap</a>	Pop element from heap range (function template )
<a href="#">sort_heap</a>	Sort elements of heap (function template )
<a href="#">reverse</a>	Reverse range (function template )

# /algorithm/max

function template

**std::max**

<algorithm>

```

default (1) template <class T> const T& max (const T& a, const T& b);
custom (2) template <class T, class Compare>
            const T& max (const T& a, const T& b, Compare comp);

default (1) template <class T> const T& max (const T& a, const T& b);
custom (2) template <class T, class Compare>
            const T& max (const T& a, const T& b, Compare comp);
            template <class T> T max (initializer_list<T> il);
initializer list (3) template <class T, class Compare>
                     T max (initializer_list<T> il, Compare comp);

default (1) template <class T> constexpr const T& max (const T& a, const T& b);
custom (2) template <class T, class Compare>
            constexpr const T& max (const T& a, const T& b, Compare comp);
            template <class T> constexpr T max (initializer_list<T> il);
initializer list (3) template <class T, class Compare>
                     constexpr T max (initializer_list<T> il, Compare comp);
```

## Return the largest

Returns the largest of *a* and *b*. If both are equivalent, *a* is returned.

The versions for *initializer lists* (3) return the largest of all the elements in the list. Returning the first of them if these are more than one.

The function uses operator< (or *comp*, if provided) to compare the values.

The behavior of this function template (C++98) is equivalent to:

```

1 template <class T> const T& max (const T& a, const T& b) {
2     return (a<b)?b:a;      // or: return comp(a,b)?b:a; for version (2)
3 }
```

## Parameters

*a*, *b*

Values to compare.

*comp*

Binary function that accepts two values of type *T* as arguments, and returns a value convertible to *bool*. The value returned indicates whether the element passed as first argument is considered less than the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

*il*

An *initializer\_list* object.

These objects are automatically constructed from *initializer list* declarators.

$T$  shall support being compared with operator`<`.

$T$  shall be *copy constructible*.

For (3),  $T$  shall be *copy constructible*.

## Return value

The largest of the values passed as arguments.

## Example

```
1 // max example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::max
4
5 int main () {
6     std::cout << "max(1,2)==" << std::max(1,2) << '\n';
7     std::cout << "max(2,1)==" << std::max(2,1) << '\n';
8     std::cout << "max('a','z')== " << std::max('a', 'z') << '\n';
9     std::cout << "max(3.14,2.73)== " << std::max(3.14,2.73) << '\n';
10    return 0;
11 }
```

Output:

```
max(1,2)==2
max(2,1)==2
max('a','z')==z
max(3.14,2.73)==3.14
```

## Complexity

Linear in one less than the number of elements compared (constant for (1) and (2)).

## Exceptions

Throws if any comparison throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">min</a>	Return the smallest (function template )
<a href="#">max_element</a>	Return largest element in range (function template )

# /algorithm/max\_element

function template

## std::max\_element

<algorithm>

```
default (1) template <class ForwardIterator>
            ForwardIterator max_element (ForwardIterator first, ForwardIterator last);
custom (2)  template <class ForwardIterator, class Compare>
            ForwardIterator max_element (ForwardIterator first, ForwardIterator last,
                                         Compare comp);
```

### Return largest element in range

Returns an iterator pointing to the element with the largest value in the range `[first, last]`.

The comparisons are performed using either operator`<` for the first version, or `comp` for the second; An element is the largest if no other element does not compare less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator>
2     ForwardIterator max_element ( ForwardIterator first, ForwardIterator last )
3 {
4     if (first==last) return last;
5     ForwardIterator largest = first;
6
7     while ( ++first!=last)
8         if (*largest<*first)    // or: if (comp(*largest,*first)) for version (2)
9             largest=first;
10    return largest;
11 }
```

## Parameters

`first, last`

Input iterators to the initial and final positions of the sequence to compare. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered less than the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to largest value in the range, or *last* if the range is empty.

## Example

```
1 // min_element/max_element example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::min_element, std::max_element
4
5 bool myfn(int i, int j) { return i<j; }
6
7 struct myclass {
8     bool operator() (int i,int j) { return i<j; }
9 } myobj;
10
11 int main () {
12     int myints[] = {3,7,2,5,6,4,9};
13
14     // using default comparison:
15     std::cout << "The smallest element is " << *std::min_element(myints,myints+7) << '\n';
16     std::cout << "The largest element is " << *std::max_element(myints,myints+7) << '\n';
17
18     // using function myfn as comp:
19     std::cout << "The smallest element is " << *std::min_element(myints,myints+7,myfn) << '\n';
20     std::cout << "The largest element is " << *std::max_element(myints,myints+7,myfn) << '\n';
21
22     // using object myobj as comp:
23     std::cout << "The smallest element is " << *std::min_element(myints,myints+7,myobj) << '\n';
24     std::cout << "The largest element is " << *std::max_element(myints,myints+7,myobj) << '\n';
25
26     return 0;
27 }
```

Output:

```
The smallest element is 2
The largest element is 9
The smallest element is 2
The largest element is 9
The smallest element is 2
The largest element is 9
```

## Complexity

Linear in one less than the number of elements compared.

## Data races

The objects in the range [*first*,*last*) are accessed.

## Exceptions

Throws if any comparison throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">min_element</a>	Return smallest element in range (function template )
<a href="#">upper_bound</a>	Return iterator to upper bound (function template )
<a href="#">max</a>	Return the largest (function template )

# /algorithm/merge

function template

## std::merge

<algorithm>

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
default (1)     OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2,
                                         OutputIterator result);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
custom (2)      OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2,
                                         OutputIterator result, Compare comp);
```

### Merge sorted ranges

Combines the elements in the sorted ranges [*first*1,*last*1) and [*first*2,*last*2), into a new range beginning at *result* with all its elements sorted.

The elements are compared using *operator<* for the first version, and *comp* for the second. The elements in both ranges shall already be ordered according to this same criterion (*operator<* or *comp*). The resulting range is also sorted according to this.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator1, class InputIterator2, class OutputIterator>
2     OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
3                             InputIterator2 first2, InputIterator2 last2,
4                             OutputIterator result)
5 {
6     while (true) {
7         if (first1==last1) return std::copy(first2,last2,result);
```

```

8     if (first2==last2) return std::copy(first1,last1,result);
9     *result++ = (*first2<*first1)? *first2++ : *first1++;
10    }
11 }

```

## Parameters

`first1, last1`

`Input iterators` to the initial and final positions of the first sorted sequence. The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2, last2`

`Input iterators` to the initial and final positions of the second sorted sequence. The range used is `[first2, last2)`.

`result`

`Output iterator` to the initial position of the range where the resulting combined range is stored. Its size is equal to the sum of both ranges above.

`comp`

Binary function that accepts two arguments of the types pointed by the iterators, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

The ranges shall not overlap.

The elements in both input ranges should be `assignable` to the elements in the range pointed by `result`. They should also be comparable (with `operator<` for (1), and with `comp` for (2)).

## Return value

An iterator pointing to the *past-the-end* element in the resulting sequence.

## Example

```

1 // merge algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::merge, std::sort
4 #include <vector>             // std::vector
5
6 int main () {
7     int first[] = {5,10,15,20,25};
8     int second[] = {50,40,30,20,10};
9     std::vector<int> v(10);
10
11    std::sort (first,first+5);
12    std::sort (second,second+5);
13    std::merge (first,first+5,second,second+5,v.begin());
14
15    std::cout << "The resulting vector contains:";
16    for (std::vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
17        std::cout << ' ' << *it;
18    std::cout << '\n';
19
20    return 0;
21 }

```

Output:

The resulting vector contains: 5 10 10 15 20 20 25 30 40 50

## Complexity

Up to linear in  $(1+\text{count}_1-\text{count}_2)$ , where  $\text{count}_X$  is the *distance* between `firstX` and `lastX`: Compares and assigns all elements.

## Data races

The objects in the ranges `[first1, last1)` and `[first2, last2)` are accessed.

The objects in the range between `result` and the returned value are modified.

## Exceptions

Throws if any of element comparisons, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<code>inplace_merge</code>	Merge consecutive sorted ranges (function template )
<code>set_union</code>	Union of two sorted ranges (function template )
<code>copy</code>	Copy range of elements (function template )

## /algorithm/min

function template

`std::min`

<algorithm>

```
default (1) template <class T> const T& min (const T& a, const T& b);
```

```

custom (2) template <class T, class Compare>
    const T& min (const T& a, const T& b, Compare comp);

default (1) template <class T> const T& min (const T& a, const T& b);
custom (2) template <class T, class Compare>
    const T& min (const T& a, const T& b, Compare comp);
    template <class T> T min (initializer_list<T> il);
initializer list (3) template <class T, class Compare>
    T min (initializer_list<T> il, Compare comp);

default (1) template <class T> constexpr const T& min (const T& a, const T& b);
custom (2) template <class T, class Compare>
    constexpr const T& min (const T& a, const T& b, Compare comp);
    template <class T> constexpr T min (initializer_list<T> il);
initializer list (3) template <class T, class Compare>
    constexpr T min (initializer_list<T> il, Compare comp);

```

## Return the smallest

Returns the smallest of *a* and *b*. If both are equivalent, *a* is returned.

The versions for *initializer lists* (3) return the smallest of all the elements in the list. Returning the first of them if these are more than one.

The function uses *operator<* (or *comp*, if provided) to compare the values.

The behavior of this function template (C++98) is equivalent to:

```

1 template <class T> const T& min (const T& a, const T& b) {
2     return !(b<a)?a:b;      // or: return !comp(b,a)?a:b; for version (2)
3 }

```

## Parameters

<i>a, b</i>	Values to compare.
<i>comp</i>	Binary function that accepts two values of type <i>T</i> as arguments, and returns a value convertible to <i>bool</i> . The value returned indicates whether the element passed as first argument is considered less than the second. The function shall not modify any of its arguments. This can either be a function pointer or a function object.
<i>il</i>	An <i>initializer_list</i> object. These objects are automatically constructed from <i>initializer list</i> declarators.

*T* shall support being compared with *operator<*.

*T* shall be *copy constructible*.

For (3), *T* shall be *copy constructible*.

## Return value

The lesser of the values passed as arguments.

## Example

```

1 // min example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::min
4
5 int main () {
6     std::cout << "min(1,2)=="
7     std::cout << std::min(1,2) << '\n';
8     std::cout << "min(2,1)=="
9     std::cout << std::min(2,1) << '\n';
10    std::cout << "min('a','z')=="
11    std::cout << std::min('a','z') << '\n';
12    std::cout << "min(3.14,2.72)=="
13    std::cout << std::min(3.14,2.72) << '\n';
14    return 0;
15 }

```

Output:

```

min(1,2)==1
min(2,1)==1
min('a','z')==a
min(3.14,2.72)==2.72

```

## Complexity

Linear in one less than the number of elements compared (constant for (1) and (2)).

## Exceptions

Throws if any comparison throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">max</a>	Return the largest ( <a href="#">function template</a> )
<a href="#">min_element</a>	Return smallest element in range ( <a href="#">function template</a> )

# /algorithm/min\_element

function template

## std::min\_element

<algorithm>

```
default (1) template <class ForwardIterator>
    ForwardIterator min_element (ForwardIterator first, ForwardIterator last);
    template <class ForwardIterator, class Compare>
custom (2)     ForwardIterator min_element (ForwardIterator first, ForwardIterator last,
                                            Compare comp);
```

### Return smallest element in range

Returns an iterator pointing to the element with the smallest value in the range [first, last).

The comparisons are performed using either operator< for the first version, or *comp* for the second; An element is the smallest if no other element compares less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator>
2     ForwardIterator min_element ( ForwardIterator first, ForwardIterator last )
3 {
4     if (first==last) return last;
5     ForwardIterator smallest = first;
6
7     while ( ++first!=last )
8         if (*first<*smallest) // or: if (comp(*first,*smallest)) for version (2)
9             smallest=first;
10    return smallest;
11 }
```

## Parameters

first, last

Input iterators to the initial and final positions of the sequence to compare. The range used is [first, last), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

comp

Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned indicates whether the element passed as first argument is considered less than the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to smallest value in the range, or *last* if the range is empty.

## Example

```
1 // min_element/max_element example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::min_element, std::max_element
4
5 bool myfn(int i, int j) { return i<j; }
6
7 struct myclass {
8     bool operator() (int i,int j) { return i<j; }
9 } myobj;
10
11 int main () {
12     int myints[] = {3,7,2,5,6,4,9};
13
14     // using default comparison:
15     std::cout << "The smallest element is " << *std::min_element(myints,myints+7) << '\n';
16     std::cout << "The largest element is " << *std::max_element(myints,myints+7) << '\n';
17
18     // using function myfn as comp:
19     std::cout << "The smallest element is " << *std::min_element(myints,myints+7,myfn) << '\n';
20     std::cout << "The largest element is " << *std::max_element(myints,myints+7,myfn) << '\n';
21
22     // using object myobj as comp:
23     std::cout << "The smallest element is " << *std::min_element(myints,myints+7,myobj) << '\n';
24     std::cout << "The largest element is " << *std::max_element(myints,myints+7,myobj) << '\n';
25
26     return 0;
27 }
```

Output:

```
The smallest element is 2
The largest element is 9
The smallest element is 2
The largest element is 9
The smallest element is 2
The largest element is 9
```

## Complexity

Linear in one less than the number of elements compared.

## Data races

The objects in the range [first, last) are accessed.

## Exceptions

Throws if any comparison throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">max_element</a>	Return largest element in range (function template )
<a href="#">lower_bound</a>	Return iterator to lower bound (function template )
<a href="#">min</a>	Return the smallest (function template )

# /algorithm/minmax

function template

## std::minmax

<algorithm>

```
default (1)  template <class T>
              pair<const T&, const T&> minmax (const T& a, const T& b);
custom (2)   template <class T, class Compare>
              pair<const T&, const T&> minmax (const T& a, const T& b, Compare comp);
template <class T>
initializer list (3)  pair<T,T> minmax (initializer_list<T> il);
template <class T, class Compare>
                      pair<T,T> minmax (initializer_list<T> il, Compare comp);

default (1)  template <class T>
              constexpr pair<const T&, const T&> minmax (const T& a, const T& b);
custom (2)   template <class T, class Compare>
              constexpr pair<const T&, const T&> minmax (const T& a, const T& b, Compare comp);
template <class T>
initializer list (3)  constexpr pair<T,T> minmax (initializer_list<T> il);
template <class T, class Compare>
                      constexpr pair<T,T> minmax (initializer_list<T> il, Compare comp);
```

### Return smallest and largest elements

Returns a [pair](#) with the smallest of *a* and *b* as first element, and the largest as second. If both are equivalent, the function returns [make\\_pair\(a,b\)](#).

The versions for *initializer lists* (3) return a [pair](#) with the smallest of all the elements in the list as first element (the first of them, if there are more than one), and the largest as second (the last of them, if there are more than one).

The function uses `operator<` (or *comp*, if provided) to compare the values.

The behavior of this function template (version (1)) is equivalent to:

```
1 template <class T> pair<const T&, const T&> minmax (const T& a, const T& b) {
2     return (b < a) ? std::make_pair(b,a) : std::make_pair(a,b);
3 }
```

## Parameters

*a, b*

Values to compare.

*comp*

Binary function that accepts two values of type *T* as arguments, and returns a value convertible to *bool*. The value returned indicates whether the element passed as first argument is considered less than the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

*il*

An [initializer\\_list](#) object.

These objects are automatically constructed from *initializer list* declarators.

*T* shall support being compared with `operator<`.

For (3), *T* shall be *copy constructible*.

## Return value

The lesser of the values passed as arguments.

## Example

```
1 // minmax example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::minmax
4
5 int main () {
6     auto result = std::minmax({1,2,3,4,5});
7
8     std::cout << "minmax({1,2,3,4,5}): ";
9     std::cout << result.first << ' ' << result.second << '\n';
10    return 0;
11 }
```

Output:

```
minmax({1,2,3,4,5}): 1 5
```

## Complexity

Up to linear in one and half times the number of elements compared (constant for (1) and (2)).

## Exceptions

Throws if any comparison throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">minmax_element</a>	Return smallest and largest elements in range (function template )
<a href="#">min</a>	Return the smallest (function template )
<a href="#">max</a>	Return the largest (function template )

## /algorithm/minmax\_element

function template

### std::minmax\_element

<algorithm>

```
template <class ForwardIterator>
default (1) pair<ForwardIterator,ForwardIterator>
    minmax_element (ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class Compare>
custom (2) pair<ForwardIterator,ForwardIterator>
    minmax_element (ForwardIterator first, ForwardIterator last, Compare comp);
```

#### Return smallest and largest elements in range

Returns a [pair](#) with an iterator pointing to the element with the smallest value in the range `[first, last)` as first element, and the largest as second.

The comparisons are performed using either `operator<` for the first version, or `comp` for the second.

If more than one equivalent element has the smallest value, the first iterator points to the first of such elements.

If more than one equivalent element has the largest value, the second iterator points to the last of such elements.

## Parameters

`first, last`

Input iterators to the initial and final positions of the sequence to compare. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered less than the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

A [pair](#) with an iterator pointing to the element with the smallest value in the range `[first, last)` as first element, and the largest as second.

`pair` is a class template defined in [<utility>](#).

## Example

```
1 // minmax_element
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::minmax_element
4 #include <array>         // std::array
5
6 int main () {
7     std::array<int,7> foo {3,7,2,9,5,8,6};
8
9     auto result = std::minmax_element (foo.begin(),foo.end());
10
11    // print result:
12    std::cout << "min is " << *result.first;
13    std::cout << ", at position " << (result.first-foo.begin()) << '\n';
14    std::cout << "max is " << *result.second;
15    std::cout << ", at position " << (result.second-foo.begin()) << '\n';
16
17    return 0;
18 }
19
```

Output:

```
min is 2, at position 2
max is 9, at position 3
```

## Complexity

Up to linear in 1.5 times one less than the number of elements compared.

## Data races

The objects in the range `[first, last)` are accessed.

## Exceptions

Throws if any comparison throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">minmax</a>	Return smallest and largest elements (function template )
<a href="#">min_element</a>	Return smallest element in range (function template )
<a href="#">max_element</a>	Return largest element in range (function template )
<a href="#">lower_bound</a>	Return iterator to lower bound (function template )
<a href="#">upper_bound</a>	Return iterator to upper bound (function template )

# /algorithm/mismatch

function template

## std::mismatch

<algorithm>

```
template <class InputIterator1, class InputIterator2>
equality (1) pair<InputIterator1, InputIterator2>
    mismatch (InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2);
template <class InputIterator1, class InputIterator2, class BinaryPredicate>
predicate (2) pair<InputIterator1, InputIterator2>
    mismatch (InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, BinaryPredicate pred);
```

### Return first position where two ranges differ

Compares the elements in the range `[first1, last1)` with those in the range beginning at `first2`, and returns the first element of both sequences that does not match.

The elements are compared using operator`==` (or `pred`, in version (2)).

The function returns a `pair` of iterators to the first element in each range that does not match.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator1, class InputIterator2>
2     pair<InputIterator1, InputIterator2>
3         mismatch (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2 )
4 {
5     while ( (first1!=last1) && (*first1==*first2) ) // or: pred(*first1,*first2), for version 2
6     { ++first1; ++first2; }
7     return std::make_pair(first1,first2);
8 }
```

## Parameters

`first1, last1`

Input iterators to the initial and final positions of the first sequence. The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2`

Input iterator to the initial position of the second sequence. Up to as many elements as in the range `[first1, last1)` can be accessed by the function.

`pred`

Binary function that accepts two elements as argument (one of each of the two sequences, in the same order), and returns a value convertible to `bool`. The value returned indicates whether the elements are considered to match in the context of this function.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

A `pair`, where its members `first` and `second` point to the first element in both sequences that did not compare equal to each other.

If the elements compared in both sequences have all matched, the function returns a `pair` with `first` set to `last1` and `second` set to the element in that same relative position in the second sequence.

If none matched, it returns `make_pair(first1,first2)`.

## Example

```
1 // mismatch algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::mismatch
4 #include <vector>             // std::vector
5 #include <utility>            // std::pair
6
7 bool mypredicate (int i, int j) {
8     return (i==j);
9 }
10
11 int main () {
12     std::vector<int> myvector;
13     for (int i=1; i<6; i++) myvector.push_back (i*10); // myvector: 10 20 30 40 50
```

```

14 int myints[] = {10,20,80,320,1024};           //  myints: 10 20 80 320 1024
15
16 std::pair<std::vector<int>::iterator,int*> mypair;
17
18 // using default comparison:
19 mypair = std::mismatch (myvector.begin(), myvector.end(), myints);
20 std::cout << "First mismatching elements: " << *mypair.first;
21 std::cout << " and " << *mypair.second << '\n';
22
23 ++mypair.first; ++mypair.second;
24
25 // using predicate comparison:
26 mypair = std::mismatch (mypair.first, myvector.end(), mypair.second, mypredicate);
27 std::cout << "Second mismatching elements: " << *mypair.first;
28 std::cout << " and " << *mypair.second << '\n';
29
30
31 return 0;
32 }
```

Output:

```
First mismatching elements: 30 and 80
Second mismatching elements: 40 and 320
```

## Complexity

Up to linear in the [distance](#) between *first1* and *last1*: Compares elements until a mismatch is found.

## Data races

Some (or all) of the objects in both ranges are accessed (once at most).

## Exceptions

Throws if any element comparison (or *pred*) throws or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

## See also

<a href="#">find_first_of</a>	Find element from set in range ( <a href="#">function template</a> )
<a href="#">find_end</a>	Find last subsequence in range ( <a href="#">function template</a> )
<a href="#">search</a>	Search range for subsequence ( <a href="#">function template</a> )
<a href="#">equal</a>	Test whether the elements in two ranges are equal ( <a href="#">function template</a> )

# /algorithm/move

function template

## std::move

<algorithm>

```
template <class InputIterator, class OutputIterator>
    OutputIterator move (InputIterator first, InputIterator last, OutputIterator result);
```

### Move range of elements

Moves the elements in the range [*first*,*last*] into the range beginning at *result*.

The value of the elements in the [*first*,*last*] is transferred to the elements pointed by *result*. After the call, the elements in the range [*first*,*last*] are left in an unspecified but valid state.

The ranges shall not overlap in such a way that *result* points to an element in the range [*first*,*last*]. For such cases, see [move\\_backward](#).

The behavior of this function template is equivalent to:

```

1 template<class InputIterator, class OutputIterator>
2     OutputIterator move (InputIterator first, InputIterator last, OutputIterator result)
3 {
4     while (first!=last) {
5         *result = std::move(*first);
6         ++result; ++first;
7     }
8     return result;
9 }
```

## Parameters

*first*, *last*

[Input iterators](#) to the initial and final positions in a sequence to be moved. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*result*

[Output iterator](#) to the initial position in the destination sequence.  
This shall not point to any element in the range [*first*,*last*].

## Return value

An iterator to the end of the destination range where elements have been moved.

## **Example**

```
1 // move algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::move (ranges)
4 #include <utility>            // std::move (objects)
5 #include <vector>              // std::vector
6 #include <string>              // std::string
7
8 int main () {
9     std::vector<std::string> foo = {"air", "water", "fire", "earth"};
10    std::vector<std::string> bar (4);
11
12    // moving ranges:
13    std::cout << "Moving ranges...\n";
14    std::move (foo.begin(), foo.begin() + 4, bar.begin());
15
16    std::cout << "foo contains " << foo.size() << " elements:";
17    std::cout << " (each in an unspecified but valid state)";
18    std::cout << '\n';
19
20    std::cout << "bar contains " << bar.size() << " elements:";
21    for (std::string& x: bar) std::cout << " [" << x << "]";
22    std::cout << '\n';
23
24    // moving container:
25    std::cout << "Moving container...\n";
26    foo = std::move (bar);
27
28    std::cout << "foo contains " << foo.size() << " elements:";
29    for (std::string& x: foo) std::cout << " [" << x << "]";
30    std::cout << '\n';
31
32    std::cout << "bar is in an unspecified but valid state";
33    std::cout << '\n';
34
35    return 0;
36 }
```

## Possible output:

```
Moving ranges...
foo contains 4 elements: (each in an unspecified but valid state)
bar contains 4 elements: [air] [water] [fire] [earth]
Moving container...
foo contains 4 elements: [air] [water] [fire] [earth]
bar is in an unspecified but valid state
```

## Complexity

Linear in the **distance** between *first* and *last*: Performs a move-assignment for each element in the range.

## Data races

The objects in both ranges are modified.

## Exceptions

Throws if either an element move-assignment or an operation on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

#### **See also**

<b>move_backward</b>	Move range of elements backward (function template )
<b>copy</b>	Copy range of elements (function template )

## /algorithm/move\_backward

## function template

## **std::move\_backward**

## <algorithm>

## **Move range of elements backward**

Moves the elements in the range [first, last) starting from the end into the range terminating at result.

The function returns an iterator to the first element in the destination range.

The resulting range has the elements in the exact same order as `[first, last)`. To reverse their order, see `reverse`.

The function begins by moving  $*$ (last-1) into  $*$ (result-1), and then follows backward by the elements preceding these, until *first* is reached (and including it).

The ranges shall not overlap in such a way that *result* (which is the *past-the-end element* in the destination range) points to an element in the range `(first, last]`. For such cases, see [move](#).

The behavior of this function template is equivalent to:

```

4 |             BidirectionalIterator2 result )
5 | {
6 |     while (last!=first) *(--result) = std::move(*(--last));
7 |     return result;
8 | }

```

## Parameters

`first, last`

Bidirectional iterators to the initial and final positions in a sequence to be moved. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`result`

Bidirectional iterator to the *past-the-end* position in the destination sequence.

This shall not point to any element in the range `(first, last]`.

## Return value

An iterator to the first element of the destination sequence where elements have been moved.

## Example

```

1 // move_backward example
2 #include <iostream>      // std::cout
3 #include <algorithm>    // std::move_backward
4 #include <string>       // std::string
5
6 int main () {
7     std::string elems[10] = {"air", "water", "fire", "earth"};
8
9     // insert new element at the beginning:
10    std::move_backward (elems,elems+4,elems+5);
11    elems[0] = "ether";
12
13    std::cout << "elems contains:" ;
14    for (int i=0; i<10; ++i)
15        std::cout << " [" << elems[i] << "]";
16    std::cout << '\n';
17
18    return 0;
19 }

```

Output:

```
elems contains: [ether] [air] [water] [fire] [earth] [] [] [] [] []
```

## Complexity

Linear in the `distance` between `first` and `last`: Performs a move-assignment for each element in the range.

## Data races

The objects in both ranges are modified.

## Exceptions

Throws if either an element move-assignment or an operation on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<code>move</code>	Move range of elements (function template )
<code>copy_backward</code>	Copy range of elements backward (function template )

## /algorithm/next\_permutation

function template

**std::next\_permutation**

<algorithm>

```

        template <class BidirectionalIterator>
default (1)  bool next_permutation (BidirectionalIterator first,
                                    BidirectionalIterator last);
custom (2)   template <class BidirectionalIterator, class Compare>
              bool next_permutation (BidirectionalIterator first,
                                     BidirectionalIterator last, Compare comp);

```

### Transform range to next permutation

Rearranges the elements in the range `[first, last)` into the next *lexicographically* greater permutation.

A *permutation* is each one of the  $N!$  possible arrangements the elements can take (where  $N$  is the number of elements in the range). Different permutations can be ordered according to how they compare *lexicographically* to each other; The first such-sorted possible permutation (the one that would compare *lexicographically smaller* to all other permutations) is the one which has all its elements sorted in ascending order, and the largest has all its elements sorted in descending order.

The comparisons of individual elements are performed using either operator`<` for the first version, or `comp` for the second.

If the function can determine the next higher permutation, it rearranges the elements as such and returns `true`. If that was not possible (because it is already at the largest possible permutation), it rearranges the elements according to the first permutation (sorted in ascending order) and returns `false`.

## Parameters

<code>first, last</code>	<code>Bidirectional iterators</code> to the initial and final positions of the sequence. The range used is <code>[first, last)</code> , which contains all the elements between <code>first</code> and <code>last</code> , including the element pointed by <code>first</code> but not the element pointed by <code>last</code> .
<code>comp</code>	<code>BidirectionalIterator</code> shall point to a type for which <code>swap</code> is properly defined.
	Binary function that accepts two arguments of the type pointed by <code>BidirectionalIterator</code> , and returns a value convertible to <code>bool</code> . The value returned indicates whether the first argument is considered to go before the second in the specific <i>strict weak ordering</i> it defines. The function shall not modify any of its arguments. This can either be a function pointer or a function object.

## Return value

`true` if the function could rearrange the object as a lexicographicaly greater permutation.  
Otherwise, the function returns `false` to indicate that the arrangement is not greater than the previous, but the lowest possible (sorted in ascending order).

## Example

```
1 // next_permutation example
2 #include <iostream>           // std::cout
3 #include <algorithm>         // std::next_permutation, std::sort
4
5 int main () {
6     int myints[] = {1,2,3};
7
8     std::sort (myints,myints+3);
9
10    std::cout << "The 3! possible permutations with 3 elements:\n";
11    do {
12        std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
13    } while ( std::next_permutation(myints,myints+3) );
14
15    std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
16
17    return 0;
18 }
```

Output:

```
The 3! possible permutations with 3 elements:
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
After loop: 1 2 3
```

## Complexity

Up to linear in half the `distance` between `first` and `last` (in terms of actual swaps).

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any element `swap` throws or if any operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<code>prev_permutation</code>	Transform range to previous permutation (function template )
<code>lexicographical_compare</code>	Lexicographical less-than comparison (function template )

# /algorithm/none\_of

function template

## std::none\_of

<algorithm>

```
template <class InputIterator, class UnaryPredicate>
    bool none_of (InputIterator first, InputIterator last, UnaryPredicate pred);
```

### Test if no elements fulfill condition

Returns `true` if `pred` returns `false` for all the elements in the range `[first, last)` or if the range is empty, and `false` otherwise.

The behavior of this function template is equivalent to:

```
1 template<class InputIterator, class UnaryPredicate>
2     bool none_of (InputIterator first, InputIterator last, UnaryPredicate pred)
3 {
4     while (first!=last) {
5         if (pred(*first)) return false;
```

```

6     ++first;
7 }
8 return true;
9 }
```

## Parameters

`first, last`

`Input iterators` to the initial and final positions in a sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`pred`

Unary function that accepts an element in the range as argument and returns a value convertible to `bool`. The value returned indicates whether the element fulfills the condition checked by this function.

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

true if `pred` returns `false` for all the elements in the range `[first, last)` or if the range is empty, and `false` otherwise.

## Example

```

1 // none_of example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::none_of
4 #include <array>         // std::array
5
6 int main () {
7     std::array<int,8> foo = {1,2,4,8,16,32,64,128};
8
9     if ( std::none_of(foo.begin(), foo.end(), [](int i){return i<0;}) )
10    std::cout << "There are no negative elements in the range.\n";
11
12    return 0;
13 }
```

Output:

There are no negative elements in the range.

## Complexity

Up to linear in the `distance` between `first` and `last`: Calls `pred` for each element until a match is found.

## Data races

Some (or all) of the objects in the range `[first, last)` are accessed (once at most).

## Exceptions

Throws if either `pred` or an operation on an iterator throws.

Note that invalid parameters cause *undefined behavior*.

## See also

<code>all_of</code>	Test condition on all elements in range (function template )
<code>any_of</code>	Test if any element in range fulfills condition (function template )
<code>find_if_not</code>	Find element in range (negative condition) (function template )

## /algorithm/nth\_element

function template

### std::nth\_element

<algorithm>

```

        template <class RandomAccessIterator>
default (1) void nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                           RandomAccessIterator last);
        template <class RandomAccessIterator, class Compare>
custom (2)  void nth_element (RandomAccessIterator first, RandomAccessIterator nth,
                           RandomAccessIterator last, Compare comp);
```

#### Sort element in range

Rearranges the elements in the range `[first, last)`, in such a way that the element at the `nth` position is the element that would be in that position in a sorted sequence.

The other elements are left without any specific order, except that none of the elements preceding `nth` are greater than it, and none of the elements following it are less.

The elements are compared using `operator<` for the first version, and `comp` for the second.

## Parameters

`first, last`

Random-access iterators to the initial and final positions of the sequence to be used. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
Notice that in this function, these are not consecutive parameters, but the first and the **third**.

`nth`

Random-access iterator pointing to the location within the range `[first, last)` that will contain the sorted element.  
Notice that the value of the element pointed by `nth` before the call is not relevant.

Random-access iterator pointing to the location within the range `[first, last]` that will contain the sorted element.  
If this points to `last`, the function call has no effect.  
Notice that the value of the element pointed by `nth` before the call is not relevant.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

## Return value

none

## Example

```
1 // nth_element example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::nth_element, std::random_shuffle
4 #include <vector>             // std::vector
5
6 bool myfunction (int i,int j) { return (i<j); }
7
8 int main () {
9     std::vector<int> myvector;
10
11    // set some values:
12    for (int i=1; i<10; i++) myvector.push_back(i);    // 1 2 3 4 5 6 7 8 9
13
14    std::random_shuffle (myvector.begin(), myvector.end());
15
16    // using default comparison (operator <):
17    std::nth_element (myvector.begin(), myvector.begin()+5, myvector.end());
18
19    // using function as comp
20    std::nth_element (myvector.begin(), myvector.begin()+5, myvector.end(),myfunction);
21
22    // print out content:
23    std::cout << "myvector contains:";
24    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
25        std::cout << ' ' << *it;
26    std::cout << '\n';
27
28    return 0;
29 }
```

Possible output:

```
myvector contains: 3 1 4 2 5 6 9 7 8
```

## Complexity

On average, linear in the `distance` between `first` and `last`: Compares elements, and possibly swaps (or moves) them, until the elements are properly rearranged.

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<code>sort</code>	Sort elements in range ( <a href="#">function template</a> )
<code>partial_sort</code>	Partially sort elements in range ( <a href="#">function template</a> )
<code>partition</code>	Partition range in two ( <a href="#">function template</a> )
<code>find_if</code>	Find element in range ( <a href="#">function template</a> )

## /algorithm/partial\_sort

function template

### `std::partial_sort`

`<algorithm>`

```
template <class RandomAccessIterator>
default (1) void partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                           RandomAccessIterator last);
template <class RandomAccessIterator, class Compare>
custom (2) void partial_sort (RandomAccessIterator first, RandomAccessIterator middle,
                           RandomAccessIterator last, Compare comp);
```

## Partially sort elements in range

Rearranges the elements in the range `[first, last)`, in such a way that the elements before `middle` are the smallest elements in the entire range and are sorted in ascending order, while the remaining elements are left without any specific order.

The elements are compared using operator`<` for the first version, and `comp` for the second.

## Parameters

`first, last`

Random-access iterators to the initial and final positions of the sequence to be partially sorted. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

Notice that in this function these are not consecutive parameters, but the first and the `third`.

`middle`

Random-access iterator pointing to the element within the range `[first, last)` that is used as the upper boundary of the elements that are fully sorted.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

`RandomAccessIterator` shall point to a type for which `swap` is properly defined and which is both *move-constructible* and *move-assignable*.

## Return value

`none`

## Example

```
1 // partial_sort example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::partial_sort
4 #include <vector>             // std::vector
5
6 bool myfunction (int i,int j) { return (i<j); }
7
8 int main () {
9     int myints[] = {9,8,7,6,5,4,3,2,1};
10    std::vector<int> myvector (myints, myints+9);
11
12    // using default comparison (operator <):
13    std::partial_sort (myvector.begin(), myvector.begin()+5, myvector.end());
14
15    // using function as comp
16    std::partial_sort (myvector.begin(), myvector.begin()+5, myvector.end(),myfunction);
17
18    // print out content:
19    std::cout << "myvector contains:";
20    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
21        std::cout << ' ' << *it;
22    std::cout << '\n';
23
24    return 0;
25 }
```

Possible output:

```
myvector contains: 1 2 3 4 5 9 8 7 6
```

## Complexity

On average, less than linearithmic in the `distance` between `first` and `last`: Performs approximately  $N \log(M)$  comparisons of elements (where  $N$  is this distance, and  $M$  is the `distance` between `first` and `middle`). It also performs up to that many element swaps (or moves).

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<code>sort</code>	Sort elements in range ( <a href="#">function template</a> )
<code>stable_sort</code>	Sort elements preserving order of equivalents ( <a href="#">function template</a> )
<code>search</code>	Search range for subsequence ( <a href="#">function template</a> )
<code>reverse</code>	Reverse range ( <a href="#">function template</a> )

## /algorithm/partial\_sort\_copy

function template

`std::partial_sort_copy`

`<algorithm>`

```

template <class InputIterator, class RandomAccessIterator>
default (1) RandomAccessIterator
    partial_sort_copy (InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template <class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator
custom (2) partial_sort_copy (InputIterator first, InputIterator last,
                           RandomAccessIterator result_first,
                           RandomAccessIterator result_last, Compare comp);

```

### Copy and partially sort range

Copies the smallest elements in the range `[first, last)` to `[result_first, result_last)`, sorting the elements copied. The number of elements copied is the same as the [distance](#) between `result_first` and `result_last` (unless this is more than the amount of elements in `[first, last)`).

The range `[first, last)` is not modified.

The elements are compared using `operator<` for the first version, and `comp` for the second.

### Parameters

`first, last`  
`Input iterators` to the initial and final positions of the sequence to copy from. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
`InputIterator` shall point to a type [assignable](#) to the elements pointed by `RandomAccessIterator`.

`result_first, result_last`  
`Random-access iterators` to the initial and final positions of the destination sequence. The range used is `[result_first, result_last)`.  
`RandomAccessIterator` shall point to a type for which `swap` is properly defined and which is both [move-constructible](#) and [move-assignable](#).

`comp`  
Binary function that accepts two elements in the result range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific [strict weak ordering](#) it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

### Return value

An iterator pointing to the element that follows the last element written in the result sequence.

### Example

```

1 // partial_sort_copy example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::partial_sort_copy
4 #include <vector>             // std::vector
5
6 bool myfunction (int i,int j) { return (i<j); }
7
8 int main () {
9     int myints[] = {9,8,7,6,5,4,3,2,1};
10    std::vector<int> myvector (5);
11
12    // using default comparison (operator <):
13    std::partial_sort_copy (myints, myints+9, myvector.begin(), myvector.end());
14
15    // using function as comp
16    std::partial_sort_copy (myints, myints+9, myvector.begin(), myvector.end(), myfunction);
17
18    // print out content:
19    std::cout << "myvector contains:";
20    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
21        std::cout << ' ' << *it;
22    std::cout << '\n';
23
24    return 0;
25 }
```

Output:

```
myvector contains: 1 2 3 4 5
```

### Complexity

On average, less than linearithmic in the [distance](#) between `first` and `last`: Performs approximately  $N \log(\min(N, M))$  comparisons of elements (where  $N$  is this distance, and  $M$  is the [distance](#) between `result_first` and `result_last`). It also performs up to that many element swaps (or moves) and  $\min(N, M)$  assignments between ranges.

### Data races

The objects in the range `[first, last)` are modified.

### Exceptions

Throws if any of the element comparisons, the element assignments, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

### See also

<a href="#">partial_sort</a>	Partially sort elements in range ( <a href="#">function template</a> )
<a href="#">sort</a>	Sort elements in range ( <a href="#">function template</a> )
<a href="#">copy</a>	Copy range of elements ( <a href="#">function template</a> )

## /algorithm/partition

function template

**std::partition**

<algorithm>

```
template <class BidirectionalIterator, class UnaryPredicate>
BidirectionalIterator partition (BidirectionalIterator first,
                               BidirectionalIterator last, UnaryPredicate pred);

template <class ForwardIterator, class UnaryPredicate>
ForwardIterator partition (ForwardIterator first,
                           ForwardIterator last, UnaryPredicate pred);
```

### Partition range in two

Rearranges the elements from the range `[first, last]`, in such a way that all the elements for which `pred` returns `true` precede all those for which it returns `false`. The iterator returned points to the first element of the second group.

The relative ordering within each group is not necessarily the same as before the call. See `stable_partition` for a function with a similar behavior but with stable ordering within each group.

The behavior of this function template (C++98) is equivalent to:

```
1 template <class BidirectionalIterator, class UnaryPredicate>
2   BidirectionalIterator partition (BidirectionalIterator first,
3                                   BidirectionalIterator last, UnaryPredicate pred)
4 {
5   while (first!=last) {
6     while (pred(*first)) {
7       ++first;
8       if (first==last) return first;
9     }
10    do {
11      --last;
12      if (first==last) return first;
13    } while (!pred(*last));
14    swap (*first,*last);
15    ++first;
16  }
17  return first;
18 }
```

### Parameters

`first, last`

Bidirectional iterators to the initial and final positions of the sequence to partition. The range used is `[first, last]`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

Forward iterators to the initial and final positions of the sequence to partition. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

ForwardIterator shall point to a type for which `swap` is defined and swaps the value of its arguments.

`pred`

Unary function that accepts an element in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether the element is to be placed before (if `true`, it is placed before all the elements for which it returns `false`).

The function shall not modify its argument.

This can either be a function pointer or a function object.

### Return value

An iterator that points to the first element of the second group of elements (those for which `pred` returns `false`), or `last` if this group is empty.

### Example

```
1 // partition algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::partition
4 #include <vector>             // std::vector
5
6 bool IsOdd (int i) { return (i%2)==1; }
7
8 int main () {
9   std::vector<int> myvector;
10
11  // set some values:
12  for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
13
14  std::vector<int>::iterator bound;
15  bound = std::partition (myvector.begin(), myvector.end(), IsOdd);
16
17  // print out content:
18  std::cout << "odd elements:";
19  for (std::vector<int>::iterator it=myvector.begin(); it!=bound; ++it)
20    std::cout << ' ' << *it;
21  std::cout << '\n';
22
23  std::cout << "even elements:";
24  for (std::vector<int>::iterator it=bound; it!=myvector.end(); ++it)
25    std::cout << ' ' << *it;
26  std::cout << '\n';
27
```

```
28 |     return 0;
29 }
```

Possible output:

```
odd elements: 1 9 3 7 5
even elements: 6 4 8 2
```

## Complexity

Linear in the *distance* between *first* and *last*: Applies *pred* to each element, and possibly swaps some of them (if the iterator type is a *bidirectional*, at most half that many swaps, otherwise at most that many).

## Data races

The objects in the range [*first*,*last*) are modified.

## Exceptions

Throws if either an element swap or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">reverse</a>	Reverse range (function template )
<a href="#">rotate</a>	Rotate left the elements in range (function template )
<a href="#">find_if</a>	Find element in range (function template )
<a href="#">swap</a>	Exchange values of two objects (function template )

## /algorithm/partition\_copy

function template

### std::partition\_copy

<algorithm>

```
template <class InputIterator, class OutputIterator1,
          class OutputIterator2, class UnaryPredicate pred>
pair<OutputIterator1,OutputIterator2>
partition_copy (InputIterator first, InputIterator last,
                OutputIterator1 result_true, OutputIterator2 result_false,
                UnaryPredicate pred);
```

#### Partition range into two

Copies the elements in the range [*first*,*last*) for which *pred* returns *true* into the range pointed by *result\_true*, and those for which it does not into the range pointed by *result\_false*.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class OutputIterator1,
2           class OutputIterator2, class UnaryPredicate pred>
3 pair<OutputIterator1,OutputIterator2>
4 partition_copy (InputIterator first, InputIterator last,
5                 OutputIterator1 result_true, OutputIterator2 result_false,
6                 UnaryPredicate pred)
7 {
8     while (first!=last) {
9         if (pred(*first)) {
10             *result_true = *first;
11             ++result_true;
12         }
13         else {
14             *result_false = *first;
15             ++result_false;
16         }
17         ++first;
18     }
19     return std::make_pair (result_true,result_false);
20 }
```

## Parameters

*first*, *last*

*Input iterators* to the initial and final positions of the range to be copy-partitioned. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*result\_true*

*Output iterator* to the initial position of the range where the elements for which *pred* returns *true* are stored.

*result\_false*

*Output iterator* to the initial position of the range where the elements for which *pred* returns *false* are stored.

*pred*

Unary function that accepts an element pointed by *InputIterator* as argument, and returns a value convertible to *bool*. The value returned indicates on which result range the element is copied.  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

The ranges shall not overlap.

The type pointed by the *output iterator* types shall support being assigned the value of an element in the range [*first*,*last*).

## Return value

A `pair` of iterators with the end of the generated sequences pointed by `result_true` and `result_false`, respectively.

Its member `first` points to the element that follows the last element copied to the sequence of elements for which `pred` returned `true`. Its member `second` points to the element that follows the last element copied to the sequence of elements for which `pred` returned `false`.

## Example

```
1 // partition_copy example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::partition_copy, std::count_if
4 #include <vector>             // std::vector
5
6 bool IsOdd (int i) { return (i%2)==1; }
7
8 int main () {
9     std::vector<int> foo {1,2,3,4,5,6,7,8,9};
10    std::vector<int> odd, even;
11
12    // resize vectors to proper size:
13    unsigned n = std::count_if (foo.begin(), foo.end(), IsOdd);
14    odd.resize(n); even.resize(foo.size()-n);
15
16    // partition:
17    std::partition_copy (foo.begin(), foo.end(), odd.begin(), even.begin(), IsOdd);
18
19    // print contents:
20    std::cout << "odd: "; for (int& x:odd) std::cout << ' ' << x; std::cout << '\n';
21    std::cout << "even: "; for (int& x:even) std::cout << ' ' << x; std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
odd: 1 3 5 7 9
even: 2 4 6 8
```

## Complexity

Linear in the `distance` between `first` and `last`: Calls `pred` and performs an assignment once for each element.

## Data races

The objects in the range `[first, last)` are accessed (each exactly once).

The objects in the ranges pointed by `result_true` and `result_false` up to the elements pointed by the iterators returned are modified (each exactly once).

## Exceptions

Throws if any of `pred`, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<code>partition</code>	Partition range in two ( <a href="#">function template</a> )
<code>stable_partition</code>	Partition range in two - stable ordering ( <a href="#">function template</a> )
<code>is_partitioned</code>	Test whether range is partitioned ( <a href="#">function template</a> )
<code>partition_point</code>	Get partition point ( <a href="#">function template</a> )

## /algorithm/partition\_point

function template

### std::partition\_point

<algorithm>

```
template <class ForwardIterator, class UnaryPredicate>
ForwardIterator partition_point (ForwardIterator first, ForwardIterator last,
                                UnaryPredicate pred);
```

#### Get partition point

Returns an iterator to the first element in the partitioned range `[first, last)` for which `pred` is not `true`, indicating its partition point.

The elements in the range shall already be partitioned, as if `partition` had been called with the same arguments.

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for `random-access iterators`.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class UnaryPredicate>
2     ForwardIterator partition_point (ForwardIterator first, ForwardIterator last,
3                                     UnaryPredicate pred)
4 {
5     auto n = distance(first, last);
6     while (n>0)
7     {
8         ForwardIterator it = first;
9         auto step = n/2;
10        std::advance (it,step);
11        if (pred(*it)) { first=++it; n-=step+1; }
12        else n=step;
```

```
13 }
14 return first;
15 }
```

## Parameters

first, last

Forward iterators to the initial and final positions of the partitioned sequence. The range checked is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

pred

Unary function that accepts an element in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether the element goes before the partition point (if `true`, it goes before; if `false` goes at or after it).

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

An iterator to the first element in the partitioned range `[first, last)` for which `pred` is not `true`, or `last` if it is not `true` for any element.

## Example

```
1 // partition_point example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::partition, std::partition_point
4 #include <vector>             // std::vector
5
6 bool IsOdd (int i) { return (i%2)==1; }
7
8 int main () {
9     std::vector<int> foo {1,2,3,4,5,6,7,8,9};
10    std::vector<int> odd;
11
12    std::partition (foo.begin(),foo.end(),IsOdd);
13
14    auto it = std::partition_point(foo.begin(),foo.end(),IsOdd);
15    odd.assign (foo.begin(),it);
16
17    // print contents of odd:
18    std::cout << "odd:";
19    for (int& x:odd) std::cout << ' ' << x;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Output:

```
odd: 1 3 5 7 9
```

## Complexity

On average, logarithmic in the distance between `first` and `last`: Performs approximately  $\log_2(N)+2$  element comparisons (where  $N$  is this distance).

On non-random-access iterators, the iterator advances produce themselves an additional linear complexity in  $N$  on average.

## Data races

Some of the objects in the range `[first, last)` are accessed.

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">partition</a>	Partition range in two (function template )
<a href="#">find_if_not</a>	Find element in range (negative condition) (function template )
<a href="#">binary_search</a>	Test if value exists in sorted sequence (function template )

## /algorithm/pop\_heap

function template

**std::pop\_heap**

<algorithm>

```
default (1) template <class RandomAccessIterator>
              void pop_heap (RandomAccessIterator first, RandomAccessIterator last);
              template <class RandomAccessIterator, class Compare>
custom (2)   void pop_heap (RandomAccessIterator first, RandomAccessIterator last,
                           Compare comp);
```

### Pop element from heap range

Rearranges the elements in the heap range `[first, last)` in such a way that the part considered a heap is shortened by one: The element with the highest value is moved to `(last-1)`.

While the element with the highest value is moved from `first` to `(last-1)` (which now is out of the heap), the other elements are reorganized in such a way that

the range `[first, last-1]` preserves the properties of a heap.

A range can be organized into a heap by calling `make_heap`. After that, its heap properties are preserved if elements are added and removed from it using `push_heap` and `pop_heap`, respectively.

## Parameters

`first, last`

Random-access iterators to the initial and final positions of the heap to be shrank by one. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
This shall not be an empty range.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to be less than the second in the specific *strict weak ordering* it defines.  
Unless `[first, last)` is a one-element heap, this argument shall be the same as used to construct the heap.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

## Return value

none

## Example

```
1 // range heap example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
4 #include <vector>              // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,5,15};
8     std::vector<int> v(myints,myints+5);
9
10    std::make_heap (v.begin(),v.end());
11    std::cout << "initial max heap : " << v.front() << '\n';
12
13    std::pop_heap (v.begin(),v.end()); v.pop_back();
14    std::cout << "max heap after pop : " << v.front() << '\n';
15
16    v.push_back(99); std::push_heap (v.begin(),v.end());
17    std::cout << "max heap after push: " << v.front() << '\n';
18
19    std::sort_heap (v.begin(),v.end());
20
21    std::cout << "final sorted range :";
22    for (unsigned i=0; i<v.size(); i++)
23        std::cout << ' ' << v[i];
24
25    std::cout << '\n';
26
27    return 0;
28 }
```

Output:

```
initial max heap : 30
max heap after pop : 20
max heap after push: 99
final sorted range : 5 10 15 20 99
```

## Complexity

Up to twice logarithmic in the distance between `first` and `last`: Compares elements and potentially swaps (or moves) them until rearranged as a shorter heap.

## Data races

Some (or all) of the objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">make_heap</a>	Make heap from range ( <a href="#">function template</a> )
<a href="#">push_heap</a>	Push element into heap range ( <a href="#">function template</a> )
<a href="#">sort_heap</a>	Sort elements of heap ( <a href="#">function template</a> )
<a href="#">reverse</a>	Reverse range ( <a href="#">function template</a> )

## /algorithm/prev\_permutation

function template

**std::prev\_permutation**

<algorithm>

```

default (1) template <class BidirectionalIterator>
    bool prev_permutation (BidirectionalIterator first,
                           BidirectionalIterator last );
custom (2)  bool prev_permutation (BidirectionalIterator first,
                           BidirectionalIterator last, Compare comp);

```

### Transform range to previous permutation

Rearranges the elements in the range `[first, last)` into the previous *lexicographically*-ordered permutation.

A *permutation* is each one of the  $N!$  possible arrangements the elements can take (where  $N$  is the number of elements in the range). Different permutations can be ordered according to how they compare *lexicographically* to each other; The first such-sorted possible permutation (the one that would compare *lexicographically smaller* to all other permutations) is the one which has all its elements sorted in ascending order, and the largest has all its elements sorted in descending order.

The comparisons of individual elements are performed using either operator`<` for the first version, or `comp` for the second.

If the function can determine the previous permutation, it rearranges the elements as such and returns `true`. If that was not possible (because it is already at the lowest possible permutation), it rearranges the elements according to the last permutation (sorted in descending order) and returns `false`.

### Parameters

`first, last`

*Bidirectional iterators* to the initial and final positions of the sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`BidirectionalIterator` shall point to a type for which `swap` is properly defined.

`comp`

Binary function that accepts two arguments of the type pointed by `BidirectionalIterator`, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

### Return value

`true` if the function could rearrange the object as a lexicographically smaller permutation.

Otherwise, the function returns `false` to indicate that the arrangement is not less than the previous, but the largest possible (sorted in descending order).

### Example

```

1 // next_permutation example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::next_permutation, std::sort, std::reverse
4
5 int main () {
6     int myints[] = {1,2,3};
7
8     std::sort (myints,myints+3);
9     std::reverse (myints,myints+3);
10
11    std::cout << "The 3! possible permutations with 3 elements:\n";
12    do {
13        std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
14    } while ( std::prev_permutation(myints,myints+3) );
15
16    std::cout << "After loop: " << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
17
18    return 0;
19 }

```

Output:

```

3 2 1
3 1 2
2 3 1
2 1 3
1 3 2
1 2 3
After loop: 3 2 1

```

### Complexity

Up to linear in half the *distance* between `first` and `last` (in terms of actual swaps).

### Data races

The objects in the range `[first, last)` are modified.

### Exceptions

Throws if any element `swap` throws or if any operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

### See also

<a href="#">next_permutation</a>	Transform range to next permutation (function template )
----------------------------------	--

<a href="#">lexicographical_compare</a>	Lexicographical less-than comparison (function template )
---	---

# /algorithm/push\_heap

function template

## std::push\_heap

<algorithm>

```
default (1) template <class RandomAccessIterator>
    void push_heap (RandomAccessIterator first, RandomAccessIterator last);
template <class RandomAccessIterator, class Compare>
custom (2) void push_heap (RandomAccessIterator first, RandomAccessIterator last,
                           Compare comp);
```

### Push element into heap range

Given a heap in the range `[first, last-1]`, this function extends the range considered a heap to `[first, last)` by placing the value in `(last-1)` into its corresponding location within it.

A range can be organized into a heap by calling [make\\_heap](#). After that, its heap properties are preserved if elements are added and removed from it using [push\\_heap](#) and [pop\\_heap](#), respectively.

### Parameters

`first, last`

Random-access iterators to the initial and final positions of the new heap range, including the pushed element. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to be less than the second in the specific *strict weak ordering* it defines.

Unless `[first, last)` is an empty or one-element heap, this argument shall be the same as used to construct the heap.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

### Return value

none

### Example

```
1 // range heap example
2 #include <iostream>           // std::cout
3 #include <algorithm>         // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
4 #include <vector>            // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,5,15};
8     std::vector<int> v(myints,myints+5);
9
10    std::make_heap (v.begin(),v.end());
11    std::cout << "initial max heap : " << v.front() << '\n';
12
13    std::pop_heap (v.begin(),v.end()); v.pop_back();
14    std::cout << "max heap after pop : " << v.front() << '\n';
15
16    v.push_back(99); std::push_heap (v.begin(),v.end());
17    std::cout << "max heap after push: " << v.front() << '\n';
18
19    std::sort_heap (v.begin(),v.end());
20
21    std::cout << "final sorted range : ";
22    for (unsigned i=0; i<v.size(); i++)
23        std::cout << ' ' << v[i];
24
25    std::cout << '\n';
26
27    return 0;
28 }
```

Output:

```
initial max heap : 30
max heap after pop : 20
max heap after push: 99
final sorted range : 5 10 15 20 99
```

### Complexity

Up to logarithmic in the [distance](#) between `first` and `last`: Compares elements and potentially swaps (or moves) them until rearranged as a longer heap.

### Data races

Some (or all) of the objects in the range `[first, last)` are modified.

### Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

### See also

[make\\_heap](#)

Make heap from range ([function template](#))

<b>pop_heap</b>	Pop element from heap range ( <a href="#">function template</a> )
<b>sort_heap</b>	Sort elements of heap ( <a href="#">function template</a> )
<b>reverse</b>	Reverse range ( <a href="#">function template</a> )

## /algorithm/random\_shuffle

function template

### std::random\_shuffle

<algorithm>

```
generator by default (1) template <class RandomAccessIterator>
                           void random_shuffle (RandomAccessIterator first, RandomAccessIterator last);
                           template <class RandomAccessIterator, class RandomNumberGenerator>
                           void random_shuffle (RandomAccessIterator first, RandomAccessIterator last,
                                                RandomNumberGenerator& gen);

specific generator (2)   void random_shuffle (RandomAccessIterator first, RandomAccessIterator last,
                                                RandomNumberGenerator& gen);

generator by default (1) template <class RandomAccessIterator>
                           void random_shuffle (RandomAccessIterator first, RandomAccessIterator last);
                           template <class RandomAccessIterator, class RandomNumberGenerator>
                           void random_shuffle (RandomAccessIterator first, RandomAccessIterator last,
                                                RandomNumberGenerator&& gen);
```

#### Randomly rearrange elements in range

Rearranges the elements in the range `[first, last)` randomly.

The function swaps the value of each element with that of some other randomly picked element. When provided, the function `gen` determines which element is picked in every case. Otherwise, the function uses some unspecified source of randomness.

To specify a *uniform random generator* as those defined in `<random>`, see `shuffle`.

The behavior of this function template (2) is equivalent to:

```
1 template <class RandomAccessIterator, class RandomNumberGenerator>
2   void random_shuffle (RandomAccessIterator first, RandomAccessIterator last,
3                       RandomNumberGenerator& gen)
4 {
5   iterator_traits<RandomAccessIterator>::difference_type i, n;
6   n = (last-first);
7   for (i=n-1; i>0; --i) {
8     swap (first[i],first[gen(i+1)]);
9   }
10 }
```

#### Parameters

`first, last`

Random-access iterators to the initial and final positions of the sequence to be shuffled. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`gen`

Unary function taking one argument and returning a value, both convertible to/from the appropriate `difference type` used by the iterators. The function shall return a non-negative value less than its argument.  
This can either be a function pointer or a function object.

`RandomAccessIterator` shall point to a type for which `swap` is defined and swaps the value of its arguments.

#### Return value

none

#### Example

```
1 // random_shuffle example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::random_shuffle
4 #include <vector>             // std::vector
5 #include <ctime>              // std::time
6 #include <cstdlib>            // std::rand, std::srand
7
8 // random generator function:
9 int myrandom (int i) { return std::rand()%i;}
10
11 int main () {
12   std::srand ( unsigned ( std::time(0) ) );
13   std::vector<int> myvector;
14
15   // set some values:
16   for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
17
18   // using built-in random generator:
19   std::random_shuffle ( myvector.begin(), myvector.end() );
20
21   // using myrandom:
22   std::random_shuffle ( myvector.begin(), myvector.end(), myrandom );
23
24   // print out content:
25   std::cout << "myvector contains:";
26   for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
27     std::cout << ' ' << *it;
28
29   std::cout << '\n';
30 }
```

```
31 |     return 0;
32 }
```

Possible output:

```
myvector contains: 3 4 1 6 8 9 2 7 5
```

## Complexity

Linear in the *distance* between *first* and *last* minus one: Obtains random values and swaps elements.

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the random number generations, the element swaps or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<b>rotate</b>	Rotate left the elements in range (function template )
<b>reverse</b>	Reverse range (function template )
<b>generate</b>	Generate values for range with function (function template )
<b>swap</b>	Exchange values of two objects (function template )

# /algorithm/remove

function template

## std::remove

<algorithm>

```
template <class ForwardIterator, class T>
ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& val);
```

### Remove value from range

[Note: This is the reference for algorithm `remove`. See `remove` for `<cstdio>`'s `remove`.]

Transforms the range `[first, last)` into a range with all the elements that compare equal to *val* removed, and returns an iterator to the new end of that range.

The function cannot alter the properties of the object containing the range of elements (i.e., it cannot alter the size of an array or a container): The removal is done by replacing the elements that compare equal to *val* by the next element that does not, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new *past-the-end* element.

The relative order of the elements not removed is preserved, while the elements between the returned iterator and *last* are left in a valid but unspecified state.

The function uses operator`==` to compare the individual elements to *val*.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class T>
2     ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& val)
3 {
4     ForwardIterator result = first;
5     while (first!=last) {
6         if (!(*first == val)) {
7             *result = *first;
8             ++result;
9         }
10        ++first;
11    }
12    return result;
13 }
```

The elements are replaced by *move-assigning* them their new values.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class T>
2     ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& val)
3 {
4     ForwardIterator result = first;
5     while (first!=last) {
6         if (!(*first == val)) {
7             *result = move(*first);
8             ++result;
9         }
10        ++first;
11    }
12    return result;
13 }
```

## Parameters

*first*, *last*

**Forward iterators** to the initial and final positions in a sequence of **move-assignable** elements supporting being compared to a value of type *T*. The range used is `[first, last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*val*

Value to be removed.

## Return value

An iterator to the element that follows the last element not removed.

The range between *first* and this iterator includes all the elements in the sequence that do not compare equal to *val*.

## Example

```
1 // remove algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::remove
4
5 int main () {
6     int myints[] = {10,20,30,30,20,10,10,20};      // 10 20 30 30 20 10 10 20
7
8     // bounds of range:
9     int* pbegin = myints;                          // ^
10    int* pend = myints+sizeof(myints)/sizeof(int); // ^
11
12    pend = std::remove (pbegin, pend, 20);         // 10 30 30 10 10 ? ? ?
13
14    std::cout << "range contains:";                  // ^
15    for (int* p=pbegin; p!=pend; ++p)
16        std::cout << ' ' << *p;
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
range contains: 10 30 30 10 10
```

## Complexity

Linear in the *distance* between *first* and *last*: Compares each element, and possibly performs assignments on some of them.

## Data races

The objects in the range [*first*,*last*] are accessed and potentially modified.

## Exceptions

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">remove_if</a>	Remove elements from range (function template )
<a href="#">remove_copy</a>	Copy range removing value (function template )
<a href="#">replace</a>	Replace value in range (function template )
<a href="#">count</a>	Count appearances of value in range (function template )
<a href="#">find</a>	Find value in range (function template )

## /algorithm/remove\_copy

function template

### std::remove\_copy

<algorithm>

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy (InputIterator first, InputIterator last,
                           OutputIterator result, const T& val);
```

#### Copy range removing value

Copies the elements in the range [*first*,*last*] to the range beginning at *result*, except those elements that compare equal to *val*.

The resulting range is shorter than [*first*,*last*] by as many elements as matches in the sequence, which are "removed".

The function uses operator== to compare the individual elements to *val*.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class OutputIterator, class T>
2     OutputIterator remove_copy (InputIterator first, InputIterator last,
3                                 OutputIterator result, const T& val)
4 {
5     while (first!=last) {
6         if (!(*first == val)) {
7             *result = *first;
8             ++result;
9         }
10        ++first;
11    }
12    return result;
13 }
```

## Parameters

`first, last`  
Forward iterators to the initial and final positions in a sequence of elements supporting being compared to a value of type `T`. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`result`  
Output iterator to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element in the range `[first, last)`.

`val`  
Value to be removed.

The ranges shall not overlap.

## Return value

An iterator pointing to the end of the copied range, which includes all the elements in `[first, last)` except those that compare equal to `val`.

## Example

```
1 // remove_copy example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::remove_copy
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,30,20,10,10,20};           // 10 20 30 30 20 10 10 20
8     std::vector<int> myvector (8);
9
10    std::remove_copy (myints,myints+8,myvector.begin(),20); // 10 30 30 10 10 0 0 0
11
12    std::cout << "myvector contains:";
13    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
14        std::cout << ' ' << *it;
15    std::cout << '\n';
16
17    return 0;
18 }
```

Output:

```
myvector contains: 10 30 30 10 10 0 0 0
```

## Complexity

Linear in the distance between `first` and `last`: Compares each element, and performs an assignment operation for those not removed.

## Data races

The objects in the range `[first, last)` are accessed.

The objects in the range between `result` and the returned value are modified.

## Exceptions

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">remove</a>	Remove value from range (function template )
<a href="#">remove_copy_if</a>	Copy range removing values (function template )
<a href="#">replace_copy</a>	Copy range replacing value (function template )
<a href="#">count</a>	Count appearances of value in range (function template )
<a href="#">find</a>	Find value in range (function template )

## /algorithm/remove\_copy\_if

function template

### std::remove\_copy\_if

<algorithm>

```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator remove_copy_if (InputIterator first, InputIterator last,
                             OutputIterator result, UnaryPredicate pred);
```

#### Copy range removing values

Copies the elements in the range `[first, last)` to the range beginning at `result`, except those elements for which `pred` returns true.

The resulting range is shorter than `[first, last)` by as many elements as matches, which are "removed".

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class OutputIterator, class UnaryPredicate>
2     OutputIterator remove_copy_if (InputIterator first, InputIterator last,
3                                     OutputIterator result, UnaryPredicate pred)
4 {
5     while (first!=last) {
6         if (!pred(*first)) {
7             *result = *first;
8         }
9         first++;
10    }
11 }
```

```

8     ++result;
9 }
10}
11}
12 return result;
13}

```

## Parameters

`first, last`

Forward iterators to the initial and final positions in a sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`result`

Output iterator to the initial position of the range where the resulting sequence is stored.

The pointed type shall support being assigned the value of an element in the range `[first, last)`.

`pred`

Unary function that accepts an element in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether the element is to be removed from the copy (if `true`, it is not copied).

The function shall not modify its argument.

This can either be a function pointer or a function object.

The ranges shall not overlap.

## Return value

An iterator pointing to the end of the copied range, which includes all the elements in `[first, last)` except those for which `pred` returns `true`.

## Example

```

1 // remove_copy_if example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::remove_copy_if
4 #include <vector>             // std::vector
5
6 bool IsOdd (int i) { return ((i%2)==1); }
7
8 int main () {
9     int myints[] = {1,2,3,4,5,6,7,8,9};
10    std::vector<int> myvector (9);
11
12    std::remove_copy_if (myints,myints+9,myvector.begin(),IsOdd);
13
14    std::cout << "myvector contains:";
15    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
16        std::cout << ' ' << *it;
17    std::cout << '\n';
18
19    return 0;
20}

```

Output:

```
myvector contains: 2 4 6 8 0 0 0 0 0
```

## Complexity

Linear in the `distance` between `first` and `last`: Applies `pred` to each element, and performs an assignment operation for those not removed.

## Data races

The objects in the range `[first, last)` are accessed.

The objects in the range between `result` and the returned value are modified.

## Exceptions

Throws if any of `pred`, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">remove</a>	Remove value from range (function template )
<a href="#">remove_copy</a>	Copy range removing value (function template )
<a href="#">replace_copy_if</a>	Copy range replacing value (function template )
<a href="#">count</a>	Count appearances of value in range (function template )
<a href="#">copy</a>	Copy range of elements (function template )

## /algorithm/remove\_if

function template

**std::remove\_if**

<algorithm>

```

template <class ForwardIterator, class UnaryPredicate>
ForwardIterator remove_if (ForwardIterator first, ForwardIterator last,
                          UnaryPredicate pred);

```

## Remove elements from range

Transforms the range `[first, last)` into a range with all the elements for which `pred` returns `true` removed, and returns an iterator to the new end of that range.

The function cannot alter the properties of the object containing the range of elements (i.e., it cannot alter the size of an array or a container): The removal is done by replacing the elements for which `pred` returns `true` by the next element for which it does not, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new *past-the-end* element.

The relative order of the elements not removed is preserved, while the elements between the returned iterator and `last` are left in a valid but unspecified state.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class UnaryPredicate>
2   ForwardIterator remove_if (ForwardIterator first, ForwardIterator last,
3                             UnaryPredicate pred)
4 {
5   ForwardIterator result = first;
6   while (first!=last) {
7     if (!pred(*first)) {
8       *result = *first;
9       ++result;
10    }
11    ++first;
12  }
13  return result;
14 }
```

The elements are replaced by *move-assigning* them their new values.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class UnaryPredicate>
2   ForwardIterator remove_if (ForwardIterator first, ForwardIterator last,
3                             UnaryPredicate pred)
4 {
5   ForwardIterator result = first;
6   while (first!=last) {
7     if (!pred(*first)) {
8       *result = std::move(*first);
9       ++result;
10    }
11    ++first;
12  }
13  return result;
14 }
```

## Parameters

`first, last`

Forward iterators to the initial and final positions in a sequence of *move-assignable* elements. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`pred`

Unary function that accepts an element in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether the element is to be removed (if `true`, it is removed).

The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

An iterator to the element that follows the last element not removed.

The range between `first` and this iterator includes all the elements in the sequence for which `pred` does not return `true`.

## Example

```
1 // remove_if example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::remove_if
4
5 bool IsOdd (int i) { return ((i%2)==1); }
6
7 int main () {
8   int myints[] = {1,2,3,4,5,6,7,8,9};           // 1 2 3 4 5 6 7 8 9
9
10  // bounds of range:
11  int* pbegin = myints;                         // ^
12  int* pend = myints+sizeof(myints)/sizeof(int); // ^           ^
13
14  pend = std::remove_if (pbegin, pend, IsOdd);   // 2 4 6 8 ? ? ? ? ?
15  // ^           ^
16
17  std::cout << "the range contains:";
18  for (int* p=pbegin; p!=pend; ++p)
19    std::cout << ' ' << *p;
20  std::cout << '\n';
21
22  return 0;
23 }
```

Output:

```
the range contains: 2 4 6 8
```

## Complexity

Linear in the `distance` between `first` and `last`: Applies `pred` to each element, and possibly performs assignments on some of them.

## Data races

The objects in the range `[first, last)` are accessed and potentially modified.

## Exceptions

throws if any of `pred`, the element assignments or the operations throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<code>remove</code>	Remove value from range (function template )
<code>remove_copy</code>	Copy range removing value (function template )
<code>replace_if</code>	Replace values in range (function template )
<code>transform</code>	Transform range (function template )
<code>find_if</code>	Find element in range (function template )

## /algorithm/replace

function template

### std::replace

<algorithm>

```
template <class ForwardIterator, class T>
void replace (ForwardIterator first, ForwardIterator last,
              const T& old_value, const T& new_value);
```

#### Replace value in range

Assigns `new_value` to all the elements in the range `[first, last)` that compare equal to `old_value`.

The function uses operator`==` to compare the individual elements to `old_value`.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class T>
2 void replace (ForwardIterator first, ForwardIterator last,
3               const T& old_value, const T& new_value)
4 {
5     while (first!=last) {
6         if (*first == old_value) *first=new_value;
7         ++first;
8     }
9 }
```

## Parameters

`first, last`

Forward iterators to the initial and final positions in a sequence of elements that support being compared and assigned a value of type `T`. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`old_value`

Value to be replaced.

`new_value`

Replacement value.

## Return value

none

## Example

```
1 // replace algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::replace
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
8     std::vector<int> myvector (myints, myints+8);           // 10 20 30 30 20 10 10 20
9
10    std::replace (myvector.begin(), myvector.end(), 20, 99); // 10 99 30 30 99 10 10 99
11
12    std::cout << "myvector contains:";
13    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
14        std::cout << ' ' << *it;
15    std::cout << '\n';
16
17    return 0;
18 }
```

Output:

```
myvector contains: 10 99 30 30 99 10 10 99
```

## Complexity

Linear in the `distance` between `first` and `last`: Compares each element and assigns to those matching.

## Data races

The objects in the range `[first, last)` are accessed and potentially modified.

## Exceptions

Throws if any of the element comparisons, element assignments or operations on iterators throw.  
Note that invalid arguments cause *undefined behavior*.

## See also

<code>replace_if</code>	Replace values in range (function template )
<code>replace_copy</code>	Copy range replacing value (function template )
<code>remove</code>	Remove value from range (function template )
<code>count</code>	Count appearances of value in range (function template )
<code>find</code>	Find value in range (function template )

## /algorithm/replace\_copy

function template

### `std::replace_copy`

<algorithm>

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy (InputIterator first, InputIterator last,
                           OutputIterator result,
                           const T& old_value, const T& new_value);
```

#### Copy range replacing value

Copies the elements in the range `[first, last)` to the range beginning at `result`, replacing the appearances of `old_value` by `new_value`.

The function uses `operator==` to compare the individual elements to `old_value`.

The ranges shall not overlap in such a way that `result` points to an element in the range `[first, last)`.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class OutputIterator, class T>
2   OutputIterator replace_copy (InputIterator first, InputIterator last,
3                               OutputIterator result, const T& old_value, const T& new_value)
4 {
5   while (first!=last) {
6     *result = (*first==old_value)? new_value: *first;
7     ++first; ++result;
8   }
9   return result;
10 }
```

## Parameters

`first, last`

`Input iterator` to the initial and final positions in a sequence. The range copied is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`result`

`Output iterator` to the initial position of the range where the resulting sequence is stored. The range includes as many elements as `[first, last)`. The pointed type shall support being assigned a value of type `T`.

`old_value`

Value to be replaced.

`new_value`

Replacement value.

The ranges shall not overlap.

## Return value

An iterator pointing to the element that follows the last element written in the result sequence.

## Example

```
1 // replace_copy example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::replace_copy
4 #include <vector>             // std::vector
5
6 int main () {
7   int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
8
9   std::vector<int> myvector (8);
10  std::replace_copy (myints, myints+8, myvector.begin(), 20, 99);
11
12 std::cout << "myvector contains:";
13 for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
14   std::cout << ' ' << *it;
```

```

15     std::cout << '\n';
16
17     return 0;
18 }
```

Output:

```
myvector contains: 10 99 30 30 99 10 10 99
```

## Complexity

Linear in the *distance* between *first* and *last*: Performs a comparison and an assignment for each element.

## Data races

The objects in the range *[first, last)* are accessed.

The objects in the range between *result* and the returned value are modified.

## Exceptions

Throws if any of the element comparisons, element assignments or operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">remove_copy</a>	Copy range removing value (function template )
<a href="#">copy</a>	Copy range of elements (function template )
<a href="#">replace</a>	Replace value in range (function template )
<a href="#">replace_copy_if</a>	Copy range replacing value (function template )

## /algorithm/replace\_copy\_if

function template

### std::replace\_copy\_if

<algorithm>

```
template <class InputIterator, class OutputIterator, class UnaryPredicate, class T>
OutputIterator replace_copy_if (InputIterator first, InputIterator last,
                               OutputIterator result, UnaryPredicate pred,
                               const T& new_value);
```

#### Copy range replacing value

Copies the elements in the range *[first, last)* to the range beginning at *result*, replacing those for which *pred* returns *true* by *new\_value*.

The behavior of this function template is equivalent to:

```

1 template <class InputIterator, class OutputIterator, class UnaryPredicate, class T>
2     OutputIterator replace_copy_if (InputIterator first, InputIterator last,
3                                     OutputIterator result, UnaryPredicate pred,
4                                     const T& new_value)
5 {
6     while (first!=last) {
7         *result = (pred(*first))? new_value: *first;
8         ++first; ++result;
9     }
10    return result;
11 }
```

## Parameters

*first, last*

Input iterators to the initial and final positions in a sequence. The range copied is *[first, last)*, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*result*

Output iterator to the initial position of the range where the resulting sequence is stored. The range includes as many elements as *[first, last)*. The pointed type shall support being assigned a value of type *T*.

*pred*

Unary function that accepts an element in the range as argument, and returns a value convertible to *bool*. The value returned indicates whether the element is to be replaced in the copy (if *true*, it is replaced).

The function shall not modify its argument.

This can either be a function pointer or a function object.

*new\_value*

Value to assign to replaced values.

The ranges shall not overlap.

## Return value

An iterator pointing to the element that follows the last element written in the result sequence.

## Example

```

1 // replace_copy_if example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::replace_copy_if
```

```

4 #include <vector>           // std::vector
5
6 bool IsOdd (int i) { return ((i%2)==1); }
7
8 int main () {
9     std::vector<int> foo,bar;
10
11    // set some values:
12    for (int i=1; i<10; i++) foo.push_back(i);           // 1 2 3 4 5 6 7 8 9
13
14    bar.resize(foo.size());   // allocate space
15    std::replace_copy_if (foo.begin(), foo.end(), bar.begin(), IsOdd, 0);
16                                // 0 2 0 4 0 6 0 8 0
17
18    std::cout << "bar contains:" ;
19    for (std::vector<int>::iterator it=bar.begin(); it!=bar.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
second contains: 0 2 0 4 0 6 0 8 0
```

## Complexity

Linear in the *distance* between *first* and *last*: Applies *pred* and performs an assignment for each element.

## Data races

The objects in the range *[first, last)* are accessed.

The objects in the range between *result* and the returned value are modified.

## Exceptions

Throws if any of *pred*, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">remove_copy_if</a>	Copy range removing values (function template )
<a href="#">replace</a>	Replace value in range (function template )
<a href="#">replace_copy</a>	Copy range replacing value (function template )
<a href="#">transform</a>	Transform range (function template )

## /algorithm/replace\_if

function template

### std::replace\_if

<algorithm>

```
template <class ForwardIterator, class UnaryPredicate, class T>
void replace_if (ForwardIterator first, ForwardIterator last,
                 UnaryPredicate pred, const T& new_value );
```

#### Replace values in range

Assigns *new\_value* to all the elements in the range *[first, last)* for which *pred* returns *true*.

The behavior of this function template is equivalent to:

```

1 template < class ForwardIterator, class UnaryPredicate, class T >
2     void replace_if (ForwardIterator first, ForwardIterator last,
3                         UnaryPredicate pred, const T& new_value)
4 {
5     while (first!=last) {
6         if (pred(*first)) *first=new_value;
7         ++first;
8     }
9 }
```

## Parameters

*first, last*

Forward iterators to the initial and final positions in a sequence of elements that support being assigned a value of type *T*. The range used is *[first, last)*, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*pred*

Unary function that accepts an element in the range as argument, and returns a value convertible to *bool*. The value returned indicates whether the element is to be replaced (if *true*, it is replaced).  
The function shall not modify its argument.  
This can either be a function pointer or a function object.

*new\_value*

Value to assign to replaced elements.

## Return value

none

## Example

```
1 // replace_if example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::replace_if
4 #include <vector>             // std::vector
5
6 bool IsOdd (int i) { return ((i%2)==1); }
7
8 int main () {
9     std::vector<int> myvector;
10
11    // set some values:
12    for (int i=1; i<10; i++) myvector.push_back(i);           // 1 2 3 4 5 6 7 8 9
13
14    std::replace_if (myvector.begin(), myvector.end(), IsOdd, 0); // 0 2 0 4 0 6 0 8 0
15
16    std::cout << "myvector contains:";
17    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
18        std::cout << ' ' << *it;
19    std::cout << '\n';
20
21    return 0;
22 }
```

Output:

```
myvector contains: 0 2 0 4 0 6 0 8 0
```

## Complexity

Linear in the *distance* between *first* and *last*: Applies *pred* to each element and assigns to those matching.

## Data races

The objects in the range [*first*,*last*] are accessed and potentially modified.

## Exceptions

Throws if any of *pred*, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">replace</a>	Replace value in range (function template )
<a href="#">remove_if</a>	Remove elements from range (function template )
<a href="#">transform</a>	Transform range (function template )
<a href="#">for_each</a>	Apply function to range (function template )
<a href="#">find_if</a>	Find element in range (function template )

# /algorithm/reverse

function template

## std::reverse

<algorithm>

```
template <class BidirectionalIterator>
void reverse (BidirectionalIterator first, BidirectionalIterator last);
```

### Reverse range

Reverses the order of the elements in the range [*first*,*last*].

The function calls *iter\_swap* to swap the elements to their new locations.

The behavior of this function template is equivalent to:

```
1 template <class BidirectionalIterator>
2     void reverse (BidirectionalIterator first, BidirectionalIterator last)
3 {
4     while ((first!=last)&&(first!--last)) {
5         std::iter_swap (first,last);
6         ++first;
7     }
8 }
```

## Parameters

*first*, *last*

*Bidirectional iterators* to the initial and final positions of the sequence to be reversed. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
*BidirectionalIterator* shall point to a type for which *swap* is properly defined.

## Return value

none

## Example

```
1 // reverse algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::reverse
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> myvector;
8
9     // set some values:
10    for (int i=1; i<10; ++i) myvector.push_back(i);    // 1 2 3 4 5 6 7 8 9
11
12    std::reverse(myvector.begin(),myvector.end());      // 9 8 7 6 5 4 3 2 1
13
14    // print out content:
15    std::cout << "myvector contains:";
16    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
17        std::cout << ' ' << *it;
18    std::cout << '\n';
19
20    return 0;
21 }
```

Output:

```
myvector contains: 9 8 7 6 5 4 3 2 1
```

## Complexity

Linear in half the distance between *first* and *last*: Swaps elements.

## Data races

The objects in the range [*first*,*last*) are modified.

## Exceptions

Throws if either an element swap or an operation on an iterator throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">reverse_copy</a>	Copy range reversed (function template )
<a href="#">rotate</a>	Rotate left the elements in range (function template )
<a href="#">random_shuffle</a>	Randomly rearrange elements in range (function template )
<a href="#">swap</a>	Exchange values of two objects (function template )

## /algorithm/reverse\_copy

function template

### std::reverse\_copy

<algorithm>

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy (BidirectionalIterator first,
                            BidirectionalIterator last, OutputIterator result);
```

#### Copy range reversed

Copies the elements in the range [*first*,*last*) to the range beginning at *result*, but in reverse order.

The behavior of this function template is equivalent to:

```
1 template <class BidirectionalIterator, class OutputIterator>
2     OutputIterator reverse_copy (BidirectionalIterator first,
3                                 BidirectionalIterator last, OutputIterator result)
4 {
5     while (first!=last) {
6         --last;
7         *result = *last;
8         ++result;
9     }
10    return result;
11 }
```

## Parameters

*first*, *last*

Bidirectional iterators to the initial and final positions of the sequence to be copied. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*result*

Output iterator to the initial position of the range where the reversed range is stored.

The pointed type shall support being assigned the value of an element in the range [*first*,*last*).

The ranges shall not overlap.

## Return value

An output iterator pointing to the end of the copied range, which contains the same elements in reverse order.

## Example

```
1 // reverse_copy example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::reverse_copy
4 #include <vector>        // std::vector
5
6 int main () {
7     int myints[] = {1,2,3,4,5,6,7,8,9};
8     std::vector<int> myvector;
9
10    myvector.resize(9);    // allocate space
11
12    std::reverse_copy (myints, myints+9, myvector.begin());
13
14    // print out content:
15    std::cout << "myvector contains:";
16    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
17        std::cout << ' ' << *it;
18
19    std::cout << '\n';
20
21    return 0;
22 }
```

Output:

```
myvector contains: 9 8 7 6 5 4 3 2 1
```

## Complexity

Linear in the *distance* between *first* and *last*: Performs an assignment for each element.

## Data races

The objects in the range [*first*,*last*) are accessed.

The objects in the range between *result* and the returned value are modified.

Each object is accessed exactly once.

## Exceptions

Throws if either an element assignment or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">reverse</a>	Reverse range ( <a href="#">function template</a> )
<a href="#">rotate_copy</a>	Copy range rotated left ( <a href="#">function template</a> )
<a href="#">copy</a>	Copy range of elements ( <a href="#">function template</a> )
<a href="#">copy_backward</a>	Copy range of elements backward ( <a href="#">function template</a> )
<a href="#">swap</a>	Exchange values of two objects ( <a href="#">function template</a> )

# /algorithm/rotate

function template

## std::rotate

<algorithm>

```
template <class ForwardIterator>
void rotate (ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);

template <class ForwardIterator>
ForwardIterator rotate (ForwardIterator first, ForwardIterator middle,
                       ForwardIterator last);
```

### Rotate left the elements in range

Rotates the order of the elements in the range [*first*,*last*), in such a way that the element pointed by *middle* becomes the new first element.

The behavior of this function template (C++98) is equivalent to:

```
1 template <class ForwardIterator>
2 void rotate (ForwardIterator first, ForwardIterator middle,
3              ForwardIterator last)
4 {
5     ForwardIterator next = middle;
6     while (first!=next)
7     {
8         swap (*first++,*next++);
9         if (next==last) next=middle;
10        else if (first==middle) middle=next;
11    }
12 }
```

## Parameters

`first, last`  
Forward iterators to the initial and final positions of the sequence to be rotated left. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
Notice that in this function these are not consecutive parameters, but the first and the **third**.

`middle`  
Forward iterator pointing to the element within the range `[first, last)` that is moved to the first position in the range.

`ForwardIterator` shall point to a type for which `swap` is properly defined and which is both *move-constructible* and *move-assignable*.

## Return value

none

An iterator pointing to the element that now contains the value previously pointed by `first`.

## Example

```
1 // rotate algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::rotate
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> myvector;
8
9     // set some values:
10    for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
11
12    std::rotate(myvector.begin(),myvector.begin()+3,myvector.end());           // 4 5 6 7 8 9 1 2 3
13
14    // print out content:
15    std::cout << "myvector contains:";
16    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
17        std::cout << ' ' << *it;
18    std::cout << '\n';
19
20    return 0;
21 }
```

Output:

```
myvector contains: 4 5 6 7 8 9 1 2 3
```

## Complexity

Up to linear in the `distance` between `first` and `last`: Swaps (or moves) elements until all elements have been relocated.

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any element swap (or move) throws or if any operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">rotate_copy</a>	Copy range rotated left (function template )
<a href="#">reverse</a>	Reverse range (function template )
<a href="#">random_shuffle</a>	Randomly rearrange elements in range (function template )
<a href="#">swap</a>	Exchange values of two objects (function template )

## /algorithm/rotate\_copy

function template

### std::rotate\_copy

<algorithm>

```
template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy (ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

#### Copy range rotated left

Copies the elements in the range `[first, last)` to the range beginning at `result`, but rotating the order of the elements in such a way that the element pointed by `middle` becomes the first element in the resulting range.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class OutputIterator>
2     OutputIterator rotate_copy (ForwardIterator first, ForwardIterator middle,
3                                 ForwardIterator last, OutputIterator result)
4 {
5     result=std::copy (middle,last,result);
6     return std::copy (first,middle,result);
7 }
```

## Parameters

---

**first, last**  
Forward iterators to the initial and final positions of the range to be copy-rotated. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
Notice that in this function, these are not consecutive parameters, but the first and **third** ones.

**middle**  
Forward iterator pointing to the element within the range `[first, last)` that is copied as the first element in the resulting range.

**result**  
Output iterator to the initial position of the range where the reversed range is stored.  
The pointed type shall support being assigned the value of an element in the range `[first, last)`.

The ranges shall not overlap.

## Return value

---

An output iterator pointing to the end of the copied range.

## Example

---

```
1 // rotate_copy algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::rotate_copy
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,40,50,60,70};
8
9     std::vector<int> myvector (7);
10
11    std::rotate_copy(myints,myints+3,myints+7,myvector.begin());
12
13    // print out content:
14    std::cout << "myvector contains:";
15    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
16        std::cout << ' ' << *it;
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
myvector contains: 40 50 60 70 10 20 30
```

## Complexity

---

Linear in the `distance` between `first` and `last`: Performs an assignment for each element.

## Data races

---

The objects in the range `[first, last)` are accessed.

The objects in the range between `result` and the returned value are modified.

Each object is accessed exactly once.

## Exceptions

---

Throws if either an element assignment or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

---

<a href="#">rotate</a>	Rotate left the elements in range (function template )
<a href="#">reverse_copy</a>	Copy range reversed (function template )
<a href="#">random_shuffle</a>	Randomly rearrange elements in range (function template )
<a href="#">copy</a>	Copy range of elements (function template )

# /algorithm/search

function template

**std::search**

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
equality (1)   ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1,
                                         ForwardIterator2 first2, ForwardIterator2 last2);
template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
predicate (2)  ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1,
                                         ForwardIterator2 first2, ForwardIterator2 last2,
                                         BinaryPredicate pred);
```

## Search range for subsequence

Searches the range `[first1, last1)` for the first occurrence of the sequence defined by `[first2, last2)`, and returns an iterator to its first element, or `last1` if no occurrences are found.

The elements in both ranges are compared sequentially using `operator==` (or `pred`, in version (2)): A subsequence of `[first1, last1)` is considered a match only when this is true for **all** the elements of `[first2, last2)`.

This function returns the first of such occurrences. For an algorithm that returns the last instead, see [find\\_end](#).

The behavior of this function template is equivalent to:

```
1 template<class ForwardIterator1, class ForwardIterator2>
2     ForwardIterator1 search ( ForwardIterator1 first1, ForwardIterator1 last1,
3                               ForwardIterator2 first2, ForwardIterator2 last2)
4 {
5     if (first2==last2) return first1; // specified in C++11
6
7     while (first1!=last1)
8     {
9         ForwardIterator1 it1 = first1;
10        ForwardIterator2 it2 = first2;
11        while (*it1==*it2) { // or: while (pred(*it1,*it2)) for version 2
12            ++it1; ++it2;
13            if (it2==last2) return first1;
14            if (it1==last1) return last1;
15        }
16        ++first1;
17    }
18    return last1;
19 }
```

## Parameters

*first1*, *last1*

Forward iterators to the initial and final positions of the searched sequence. The range used is [*first1*,*last1*), which contains all the elements between *first1* and *last1*, including the element pointed by *first1* but not the element pointed by *last1*.

*first2*, *last2*

Forward iterators to the initial and final positions of the sequence to be searched for. The range used is [*first2*,*last2*).

For (1), the elements in both ranges shall be of types comparable using operator== (with the elements of the first range as left-hand side operands, and those of the second as right-hand side operands).

*pred*

Binary function that accepts two elements as arguments (one of each of the two sequences, in the same order), and returns a value convertible to `bool`.

The returned value indicates whether the elements are considered to match in the context of this function.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to the first element of the first occurrence of [*first2*,*last2*) in [*first1*,*last1*).

If the sequence is not found, the function returns *last1*.

If [*first2*,*last2*) is an empty range, the result is unspecified.

If [*first2*,*last2*) is an empty range, the function returns *first1*.

## Example

```
1 // search algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::search
4 #include <vector>             // std::vector
5
6 bool mypredicate (int i, int j) {
7     return (i==j);
8 }
9
10 int main () {
11     std::vector<int> haystack;
12
13     // set some values:      haystack: 10 20 30 40 50 60 70 80 90
14     for (int i=1; i<10; i++) haystack.push_back(i*10);
15
16     // using default comparison:
17     int needle1[] = {40,50,60,70};
18     std::vector<int>::iterator it;
19     it = std::search (haystack.begin(), haystack.end(), needle1, needle1+4);
20
21     if (it!=haystack.end())
22         std::cout << "needle1 found at position " << (it-haystack.begin()) << '\n';
23     else
24         std::cout << "needle1 not found\n";
25
26     // using predicate comparison:
27     int needle2[] = {20,30,50};
28     it = std::search (haystack.begin(), haystack.end(), needle2, needle2+3, mypredicate);
29
30     if (it!=haystack.end())
31         std::cout << "needle2 found at position " << (it-haystack.begin()) << '\n';
32     else
33         std::cout << "needle2 not found\n";
34
35     return 0;
36 }
```

Output:

```
needle1 found at position 3
needle2 not found
```

## Complexity

Up to linear in  $\text{count1} * \text{count2}$  (where  $\text{countX}$  is the [distance](#) between  $\text{firstX}$  and  $\text{lastX}$ ): Compares elements until a matching subsequence is found.

## Data races

Some (or all) of the objects in both ranges are accessed (possibly more than once).

## Exceptions

Throws if any of the element comparisons (or  $\text{pred}$ ) throws or if any of the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">find_end</a>	Find last subsequence in range ( <a href="#">function template</a> )
<a href="#">search_n</a>	Search range for elements ( <a href="#">function template</a> )
<a href="#">equal</a>	Test whether the elements in two ranges are equal ( <a href="#">function template</a> )
<a href="#">mismatch</a>	Return first position where two ranges differ ( <a href="#">function template</a> )
<a href="#">find</a>	Find value in range ( <a href="#">function template</a> )

## /algorithm/search\_n

function template

### std::search\_n

<algorithm>

```
template <class ForwardIterator, class Size, class T>
equality (1)    ForwardIterator search_n (ForwardIterator first, ForwardIterator last,
                                         Size count, const T& val);
template <class ForwardIterator, class Size, class T, class BinaryPredicate>
predicate (2)   ForwardIterator search_n ( ForwardIterator first, ForwardIterator last,
                                         Size count, const T& val, BinaryPredicate pred );
```

#### Search range for elements

Searches the range  $[\text{first}, \text{last})$  for a sequence of  $\text{count}$  elements, each comparing equal to  $\text{val}$  (or for which  $\text{pred}$  returns `true`).

The function returns an iterator to the first of such elements, or  $\text{last}$  if no such sequence is found.

The behavior of this function template is equivalent to:

```
1 template<class ForwardIterator, class Size, class T>
2     ForwardIterator search_n (ForwardIterator first, ForwardIterator last,
3                               Size count, const T& val)
4 {
5     ForwardIterator it, limit;
6     Size i;
7
8     limit=first; std::advance(limit,std::distance(first,last)-count);
9
10    while (first!=limit)
11    {
12        it = first; i=0;
13        while (*it==val)      // or: while (pred(*it,val)) for the pred version
14            { ++it; if (++i==count) return first; }
15        ++first;
16    }
17    return last;
18 }
```

## Parameters

`first, last`

Forward iterators to the initial and final positions of the searched sequence. The range used is  $[\text{first}, \text{last})$ , which contains all the elements between  $\text{first}$  and  $\text{last}$ , including the element pointed by  $\text{first}$  but not the element pointed by  $\text{last}$ .

`count`

Minimum number of successive elements to match.  
`Size` shall be (convertible to) an integral type.

`val`

Individual value to be compared, or to be used as argument for  $\text{pred}$  (in the second version).  
for the first version, `T` shall be a type supporting comparisons with the elements pointed by `InputIterator` using `operator==` (with the elements as left-hand size operands, and `val` as right-hand side).

`pred`

Binary function that accepts two arguments (one element from the sequence as first, and `val` as second), and returns a value convertible to `bool`. The value returned indicates whether the element is considered a match in the context of this function.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

## Return value

An iterator to the first element of the sequence.

If no such sequence is found, the function returns `last`.

## Example

```
1 // search_n example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::search_n
```

```

4 #include <vector>           // std::vector
5
6 bool mypredicate (int i, int j) {
7     return (i==j);
8 }
9
10 int main () {
11     int myints[]={10,20,30,30,20,10,10,20};
12     std::vector<int> myvector (myints,myints+8);
13
14     std::vector<int>::iterator it;
15
16     // using default comparison:
17     it = std::search_n (myvector.begin(), myvector.end(), 2, 30);
18
19     if (it!=myvector.end())
20         std::cout << "two 30s found at position " << (it-myvector.begin()) << '\n';
21     else
22         std::cout << "match not found\n";
23
24     // using predicate comparison:
25     it = std::search_n (myvector.begin(), myvector.end(), 2, 10, mypredicate);
26
27     if (it!=myvector.end())
28         std::cout << "two 10s found at position " << int(it-myvector.begin()) << '\n';
29     else
30         std::cout << "match not found\n";
31
32     return 0;
33 }
```

Output:

```
Two 30s found at position 2
Two 10s found at position 5
```

## Complexity

Up to linear in the *distance* between *first* and *last*: Compares elements until a matching subsequence is found.

## Data races

Some (or all) of the objects in the range [*first*,*last*) are accessed (once at most).

## Exceptions

Throws if any of the element comparisons (or *pred*) throws or if any of the operations on iterators throws.  
Note that invalid parameters cause *undefined behavior*.

## See also

<a href="#">search</a>	Search range for subsequence (function template )
<a href="#">find_first_of</a>	Find element from set in range (function template )
<a href="#">find</a>	Find value in range (function template )
<a href="#">equal</a>	Test whether the elements in two ranges are equal (function template )

## /algorithm/set\_difference

function template

### std::set\_difference

<algorithm>

```

        template <class InputIterator1, class InputIterator2, class OutputIterator>
    default (1)     OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
                                                InputIterator2 first2, InputIterator2 last2,
                                                OutputIterator result);

        template <class InputIterator1, class InputIterator2,
                  class OutputIterator, class Compare>
    custom (2)      OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
                                                InputIterator2 first2, InputIterator2 last2,
                                                OutputIterator result, Compare comp);
```

#### Difference of two sorted ranges

Constructs a sorted range beginning in the location pointed by *result* with the *set difference* of the sorted range [*first1*,*last1*] with respect to the sorted range [*first2*,*last2*].

The *difference* of two sets is formed by the elements that are present in the first set, but not in the second one. The elements copied by the function come always from the first range, in the same order.

For containers supporting multiple occurrences of a value, the *difference* includes as many occurrences of a given value as in the first range, minus the amount of matching elements in the second, preserving order.

Notice that this is a directional operation - for a symmetrical equivalent, see [set\\_symmetric\\_difference](#).

The elements are compared using *operator<* for the first version, and *comp* for the second. Two elements, *a* and *b* are considered equivalent if *(!(a<b) && !(b<a))* or if *(!comp(a,b) && !comp(b,a))*.

The elements in the ranges shall already be ordered according to this same criterion (*operator<* or *comp*). The resulting range is also sorted according to this.

The behavior of this function template is equivalent to:

```

1 template <class InputIterator1, class InputIterator2, class OutputIterator>
2 OutputIterator set_difference (InputIterator1 first1, InputIterator1 last1,
3                               InputIterator2 first2, InputIterator2 last2,
4                               OutputIterator result)
5 {
6     while (first1!=last1 && first2!=last2)
7     {
8         if (*first1<*first2) { *result = *first1; ++result; ++first1; }
9         else if (*first2<*first1) ++first2;
10        else { ++first1; ++first2; }
11    }
12    return std::copy(first1,last1,result);
13 }

```

## Parameters

`first1, last1`

Input iterators to the initial and final positions of the first sorted sequence. The range used is  $[first1, last1]$ , which contains all the elements between  $first1$  and  $last1$ , including the element pointed by  $first1$  but not the element pointed by  $last1$ .

`first2, last2`

Input iterators to the initial and final positions of the second sorted sequence. The range used is  $[first2, last2]$ .

`result`

Output iterator to the initial position of the range where the resulting sequence is stored.

The pointed type shall support being assigned the value of an element from the first range.

`comp`

Binary function that accepts two arguments of the types pointed by the input iterators, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

The ranges shall not overlap.

## Return value

An iterator to the end of the constructed range.

## Example

```

1 // set_difference example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::set_difference, std::sort
4 #include <vector>             // std::vector
5
6 int main () {
7     int first[] = {5,10,15,20,25};
8     int second[] = {50,40,30,20,10};
9     std::vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
10    std::vector<int>::iterator it;
11
12    std::sort (first,first+5);      // 5 10 15 20 25
13    std::sort (second,second+5);   // 10 20 30 40 50
14
15    it=std::set_difference (first, first+5, second, second+5, v.begin());
16                           // 5 15 25 0 0 0 0 0 0 0
17    v.resize(it-v.begin());       // 5 15 25
18
19    std::cout << "The difference has " << (v.size()) << " elements:\n";
20    for (it=v.begin(); it!=v.end(); ++it)
21        std::cout << ' ' << *it;
22    std::cout << '\n';
23
24    return 0;
25 }

```

Output:

```

The difference has 3 elements:
5 15 25

```

## Complexity

Up to linear in  $2 * (\text{count}_1 + \text{count}_2) - 1$  (where  $\text{count}_X$  is the distance between  $first_X$  and  $last_X$ ): Compares and assigns elements.

## Data races

The objects in the ranges  $[first1, last1]$  and  $[first2, last2]$  are accessed.  
The objects in the range between  $result$  and the returned value are modified.

## Exceptions

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">set_union</a>	Union of two sorted ranges (function template )
---------------------------	---

<a href="#">set_intersection</a>	Intersection of two sorted ranges (function template )
----------------------------------	--

<a href="#">set_symmetric_difference</a>	Symmetric difference of two sorted ranges (function template )
--	--

## /algorithm/set\_intersection

function template

**std::set\_intersection**

&lt;algorithm&gt;

```

template <class InputIterator1, class InputIterator2, class OutputIterator>
default (1) OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
                                             InputIterator2 first2, InputIterator2 last2,
                                             OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
custom (2) OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
                                             InputIterator2 first2, InputIterator2 last2,
                                             OutputIterator result, Compare comp);

```

### Intersection of two sorted ranges

Constructs a sorted range beginning in the location pointed by *result* with the *set intersection* of the two sorted ranges [*first1*,*last1*] and [*first2*,*last2*].

The *intersection* of two sets is formed only by the elements that are present in both sets. The elements copied by the function come always from the first range, in the same order.

The elements are compared using operator< for the first version, and *comp* for the second. Two elements, *a* and *b* are considered equivalent if !(*a*<*b*) && !(*b*<*a*) or if !(*comp*(*a*,*b*) && !(*comp*(*b*,*a*)).

The elements in the ranges shall already be ordered according to this same criterion (operator< or *comp*). The resulting range is also sorted according to this.

The behavior of this function template is equivalent to:

```

1 template <class InputIterator1, class InputIterator2, class OutputIterator>
2     OutputIterator set_intersection (InputIterator1 first1, InputIterator1 last1,
3                                     InputIterator2 first2, InputIterator2 last2,
4                                     OutputIterator result)
5 {
6     while (first1!=last1 && first2!=last2)
7     {
8         if (*first1<*first2) ++first1;
9         else if (*first2<*first1) ++first2;
10        else {
11            *result = *first1;
12            ++result; ++first1; ++first2;
13        }
14    }
15    return result;
16 }

```

### Parameters

*first1*, *last1*

**Input iterators** to the initial and final positions of the first sorted sequence. The range used is [*first1*,*last1*), which contains all the elements between *first1* and *last1*, including the element pointed by *first1* but not the element pointed by *last1*.

*first2*, *last2*

**Input iterators** to the initial and final positions of the second sorted sequence. The range used is [*first2*,*last2*).

*result*

**Output iterator** to the initial position of the range where the resulting sequence is stored.

The pointed type shall support being assigned the value of an element from the first range.

*comp*

Binary function that accepts two arguments of the types pointed by the input iterators, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

The ranges shall not overlap.

### Return value

An iterator to the end of the constructed range.

### Example

```

1 // set_intersection example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::set_intersection, std::sort
4 #include <vector>             // std::vector
5
6 int main () {
7     int first[] = {5,10,15,20,25};
8     int second[] = {50,40,30,20,10};
9     std::vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
10    std::vector<int>::iterator it;
11
12    std::sort (first,first+5);      // 5 10 15 20 25
13    std::sort (second,second+5);    // 10 20 30 40 50
14
15    it=std::set_intersection (first, first+5, second, second+5, v.begin());
16                                // 10 20 0 0 0 0 0 0 0 0
17    v.resize(it-v.begin());       // 10 20
18

```

```

19 std::cout << "The intersection has " << (v.size()) << " elements:\n";
20 for (it=v.begin(); it!=v.end(); ++it)
21     std::cout << *it;
22     std::cout << '\n';
23
24 return 0;
25 }

```

Output:

```
The intersection has 2 elements:
10 20
```

## Complexity

Up to linear in  $2 * (\text{count}_1 + \text{count}_2) - 1$  (where  $\text{count}_X$  is the distance between  $\text{first}_X$  and  $\text{last}_X$ ): Compares and assigns elements.

## Data races

The objects in the ranges  $[\text{first}_1, \text{last}_1]$  and  $[\text{first}_2, \text{last}_2]$  are accessed.  
The objects in the range between  $\text{result}$  and the returned value are modified.

## Exceptions

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">set_union</a>	Union of two sorted ranges (function template )
<a href="#">set_difference</a>	Difference of two sorted ranges (function template )
<a href="#">set_symmetric_difference</a>	Symmetric difference of two sorted ranges (function template )
<a href="#">merge</a>	Merge sorted ranges (function template )

## /algorithm/set\_symmetric\_difference

function template

### std::set\_symmetric\_difference

<algorithm>

```

template <class InputIterator1, class InputIterator2, class OutputIterator>
default (1)    OutputIterator set_symmetric_difference (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2,
                                         OutputIterator result);

template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
custom (2)    OutputIterator set_symmetric_difference (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2,
                                         OutputIterator result, Compare comp);

```

#### Symmetric difference of two sorted ranges

Constructs a sorted range beginning in the location pointed by  $\text{result}$  with the *set symmetric difference* of the two sorted ranges  $[\text{first}_1, \text{last}_1]$  and  $[\text{first}_2, \text{last}_2]$ .

The *symmetric difference* of two sets is formed by the elements that are present in one of the sets, but not in the other. Among the equivalent elements in each range, those discarded are those that appear before in the existent order before the call. The existing order is also preserved for the copied elements.

The elements are compared using operator< for the first version, and  $\text{comp}$  for the second. Two elements,  $a$  and  $b$  are considered equivalent if  $(!(a < b) \&& !(b < a))$  or if  $(!\text{comp}(a, b) \&& !\text{comp}(b, a))$ .

The elements in the ranges shall already be ordered according to this same criterion (operator< or  $\text{comp}$ ). The resulting range is also sorted according to this.

The behavior of this function template is equivalent to:

```

1 template <class InputIterator1, class InputIterator2, class OutputIterator>
2     OutputIterator set_symmetric_difference (InputIterator1 first1, InputIterator1 last1,
3                                         InputIterator2 first2, InputIterator2 last2,
4                                         OutputIterator result)
5 {
6     while (true)
7     {
8         if (first1==last1) return std::copy(first2, last2, result);
9         if (first2==last2) return std::copy(first1, last1, result);
10
11        if (*first1<*first2) { *result=*first1; ++result; ++first1; }
12        else if (*first2<*first1) { *result = *first2; ++result; ++first2; }
13        else { ++first1; ++first2; }
14    }
15 }

```

## Parameters

$\text{first}_1, \text{last}_1$

Input iterators to the initial and final positions of the first sorted sequence. The range used is  $[\text{first}_1, \text{last}_1]$ , which contains all the elements between  $\text{first}_1$  and  $\text{last}_1$ , including the element pointed by  $\text{first}_1$  but not the element pointed by  $\text{last}_1$ .

$\text{first}_2, \text{last}_2$

Input iterators to the initial and final positions of the second sorted sequence. The range used is  $[\text{first}_2, \text{last}_2]$ .

$\text{result}$

[Output iterator](#) to the initial position of the range where the resulting sequence is stored.  
The pointed type shall support being assigned the value of an element from the other ranges.

#### comp

Binary function that accepts two arguments of the types pointed by the input iterators, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

The ranges shall not overlap.

### Return value

An iterator to the end of the constructed range.

### Example

```
1 // set_symmetric_difference example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::set_symmetric_difference, std::sort
4 #include <vector>             // std::vector
5
6 int main () {
7     int first[] = {5,10,15,20,25};
8     int second[] = {50,40,30,20,10};
9     std::vector<int> v(10);           // 0 0 0 0 0 0 0 0 0 0
10    std::vector<int>::iterator it;
11
12    std::sort (first,first+5);      // 5 10 15 20 25
13    std::sort (second,second+5);   // 10 20 30 40 50
14
15    it=std::set_symmetric_difference (first, first+5, second, second+5, v.begin()); // 5 15 25 30 40 50 0 0 0 0
16
17    v.resize(it-v.begin());       // 5 15 25 30 40 50
18
19    std::cout << "The symmetric difference has " << (v.size()) << " elements:\n";
20    for (it=v.begin(); it!=v.end(); ++it)
21        std::cout << ' ' << *it;
22    std::cout << '\n';
23
24    return 0;
25 }
```

Output:

```
The symmetric difference has 6 elements:
5 15 25 30 40 50
```

### Complexity

Up to linear in  $2*(\text{count}_1+\text{count}_2)-1$  (where  $\text{count}_X$  is the [distance](#) between  $\text{first}_X$  and  $\text{last}_X$ ): Compares and assigns elements.

### Data races

The objects in the ranges  $[\text{first}_1, \text{last}_1]$  and  $[\text{first}_2, \text{last}_2]$  are accessed.  
The objects in the range between  $\text{result}$  and the returned value are modified.

### Exceptions

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

### See also

<a href="#">set_union</a>	Union of two sorted ranges ( <a href="#">function template</a> )
<a href="#">set_intersection</a>	Intersection of two sorted ranges ( <a href="#">function template</a> )
<a href="#">set_difference</a>	Difference of two sorted ranges ( <a href="#">function template</a> )
<a href="#">merge</a>	Merge sorted ranges ( <a href="#">function template</a> )

## /algorithm/set\_union

function template

### std::set\_union

<algorithm>

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
default (1)     OutputIterator set_union (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2,
                                         OutputIterator result);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class Compare>
custom (2)      OutputIterator set_union (InputIterator1 first1, InputIterator1 last1,
                                         InputIterator2 first2, InputIterator2 last2,
                                         OutputIterator result, Compare comp);
```

#### Union of two sorted ranges

Constructs a sorted range beginning in the location pointed by  $\text{result}$  with the *set union* of the two sorted ranges  $[\text{first}_1, \text{last}_1]$  and  $[\text{first}_2, \text{last}_2]$ .

The *union* of two sets is formed by the elements that are present in either one of the sets, or in both. Elements from the second range that have an equivalent

element in the first range are not copied to the resulting range.

The elements are compared using `operator<` for the first version, and `comp` for the second. Two elements, `a` and `b` are considered equivalent if `(!(a < b) && !(b < a))` or if `(!comp(a, b) && !comp(b, a))`.

The elements in the ranges shall already be ordered according to this same criterion (`operator<` or `comp`). The resulting range is also sorted according to this.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator1, class InputIterator2, class OutputIterator>
2     OutputIterator set_union (InputIterator1 first1, InputIterator1 last1,
3                               InputIterator2 first2, InputIterator2 last2,
4                               OutputIterator result)
5 {
6     while (true)
7     {
8         if (first1==last1) return std::copy(first2,last2,result);
9         if (first2==last2) return std::copy(first1,last1,result);
10
11        if (*first1<*first2) { *result = *first1; ++first1; }
12        else if (*first2<*first1) { *result = *first2; ++first2; }
13        else { *result = *first1; ++first1; ++first2; }
14        ++result;
15    }
16 }
```

## Parameters

`first1, last1`

Input iterators to the initial and final positions of the first sorted sequence. The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2, last2`

Input iterators to the initial and final positions of the second sorted sequence. The range used is `[first2, last2)`.

`result`

Output iterator to the initial position of the range where the resulting sequence is stored.

The pointed type shall support being assigned the value of an element from the other ranges.

`comp`

Binary function that accepts two arguments of the types pointed by the input iterators, and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second in the specific *strict weak ordering* it defines.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

The ranges shall not overlap.

## Return value

An iterator to the end of the constructed range.

## Example

```
1 // set_union example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::set_union, std::sort
4 #include <vector>             // std::vector
5
6 int main () {
7     int first[] = {5,10,15,20,25};           // 0 0 0 0 0 0 0 0 0
8     int second[] = {50,40,30,20,10};         // 50 40 30 20 10
9     std::vector<int> v(10);                 // 0 0 0 0 0 0 0 0 0 0
10    std::vector<int>::iterator it;
11
12    std::sort (first,first+5);      // 5 10 15 20 25
13    std::sort (second,second+5);    // 10 20 30 40 50
14
15    it=std::set_union (first, first+5, second, second+5, v.begin()); // 5 10 15 20 25 30 40 50 0 0
16    v.resize(it-v.begin());       // 5 10 15 20 25 30 40 50
17
18    std::cout << "The union has " << (v.size()) << " elements:\n";
19    for (it=v.begin(); it!=v.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23
24    return 0;
25 }
```

Output:

```
The union has 8 elements:
5 10 15 20 25 30 40 50
```

## Complexity

Up to linear in  $2 * (\text{count}_1 + \text{count}_2) - 1$  (where `countX` is the *distance* between `firstX` and `lastX`): Compares and assigns elements.

## Data races

The objects in the ranges `[first1, last1)` and `[first2, last2)` are accessed.  
The objects in the range between `result` and the returned value are modified.

## Exceptions

Throws if any of the element comparisons, the element assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">set_intersection</a>	Intersection of two sorted ranges (function template )
<a href="#">set_difference</a>	Difference of two sorted ranges (function template )
<a href="#">set_symmetric_difference</a>	Symmetric difference of two sorted ranges (function template )
<a href="#">merge</a>	Merge sorted ranges (function template )

## /algorithm/shuffle

function template

### std::shuffle

<algorithm>

```
template <class RandomAccessIterator, class URNG>
void shuffle (RandomAccessIterator first, RandomAccessIterator last, URNG&& g);
```

#### Randomly rearrange elements in range using generator

Rearranges the elements in the range `[first, last)` randomly, using `g` as *uniform random number generator*.

The function swaps the value of each element with that of some other randomly picked element. The function determines the element picked by calling `g()`.

This function works with standard generators as those defined in [<random>](#). To shuffle the elements of the range without such a generator, see [random\\_shuffle](#) instead.

The behavior of this function template is equivalent to:

```
1 template <class RandomAccessIterator, class URNG>
2 void shuffle (RandomAccessIterator first, RandomAccessIterator last, URNG&& g)
3 {
4     for (auto i=(last-first)-1; i>0; --i) {
5         std::uniform_int_distribution<decltype(i)> d(0,i);
6         swap (first[i], first[d(g)]);
7     }
8 }
```

## Parameters

`first, last`

Forward iterators to the initial and final positions of the sequence to be shuffled. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
ForwardIterator shall point to a type for which `swap` is defined and swaps the value of its arguments.

`g`

A uniform random number generator, used as the source of randomness.  
URNG shall be a *uniform random number generator*, such as one of the standard generator classes (see [<random>](#) for more information).

## Return value

none

## Example

```
1 // shuffle algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::shuffle
4 #include <array>              // std::array
5 #include <random>             // std::default_random_engine
6 #include <chrono>             // std::chrono::system_clock
7
8 int main () {
9     std::array<int,5> foo {1,2,3,4,5};
10
11    // obtain a time-based seed:
12    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
13
14    shuffle (foo.begin(), foo.end(), std::default_random_engine(seed));
15
16    std::cout << "shuffled elements:" ;
17    for (int& x: foo) std::cout << ' ' << x;
18    std::cout << '\n';
19
20    return 0;
21 }
```

Possible output:

```
shuffled elements: 3 1 4 2 5
```

## Complexity

Linear in the `distance` between `first` and `last` minus one: Obtains random values and swaps elements.

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the random number generations, the element swaps or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">random_shuffle</a>	Randomly rearrange elements in range (function template )
<a href="#">default_random_engine</a>	Default random engine (class )
<a href="#">swap</a>	Exchange values of two objects (function template )

# /algorithm/sort

function template

**std::sort**

<algorithm>

```
default (1) template <class RandomAccessIterator>
    void sort (RandomAccessIterator first, RandomAccessIterator last);
custom (2) template <class RandomAccessIterator, class Compare>
    void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

### Sort elements in range

Sorts the elements in the range `[first, last)` into ascending order.

The elements are compared using operator`<` for the first version, and `comp` for the second.

Equivalent elements are not guaranteed to keep their original relative order (see [stable\\_sort](#)).

## Parameters

`first, last`

Random-access iterators to the initial and final positions of the sequence to be sorted. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
RandomAccessIterator shall point to a type for which `swap` is properly defined and which is both *move-constructible* and *move-assignable*.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

## Return value

none

## Example

```
1 // sort algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::sort
4 #include <vector>             // std::vector
5
6 bool myfunction (int i,int j) { return (i<j); }
7
8 struct myclass {
9     bool operator() (int i,int j) { return (i<j); }
10 } myobject;
11
12 int main () {
13     int myints[] = {32,71,12,45,26,80,53,33};
14     std::vector<int> myvector (myints, myints+8);           // 32 71 12 45 26 80 53 33
15
16     // using default comparison (operator <):
17     std::sort (myvector.begin(), myvector.begin()+4);        // (12 32 45 71)26 80 53 33
18
19     // using function as comp
20     std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)
21
22     // using object as comp
23     std::sort (myvector.begin(), myvector.end(), myobject);   // (12 26 32 33 45 53 71 80)
24
25     // print out content:
26     std::cout << "myvector contains:";
27     for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
28         std::cout << ' ' << *it;
29     std::cout << '\n';
30
31     return 0;
32 }
```

Output:

myvector contains: 12 26 32 33 45 53 71 80

## Complexity

On average, linearithmic in the *distance* between *first* and *last*: Performs approximately  $N \log_2(N)$  (where  $N$  is this distance) comparisons of elements, and up to that many element swaps (or moves).

## Data races

The objects in the range [*first*,*last*) are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">stable_sort</a>	Sort elements preserving order of equivalents (function template )
<a href="#">partial_sort</a>	Partially sort elements in range (function template )
<a href="#">search</a>	Search range for subsequence (function template )
<a href="#">reverse</a>	Reverse range (function template )

## /algorithm/sort\_heap

function template

### std::sort\_heap

<algorithm>

```
default (1) template <class RandomAccessIterator>
    void sort_heap (RandomAccessIterator first, RandomAccessIterator last);
    template <class RandomAccessIterator, class Compare>
custom (2)   void sort_heap (RandomAccessIterator first, RandomAccessIterator last,
                           Compare comp);
```

#### Sort elements of heap

Sorts the elements in the heap range [*first*,*last*) into ascending order.

The elements are compared using operator< for the first version, and *comp* for the second, which shall be the same as used to construct the heap.

The range loses its properties as a heap.

## Parameters

*first*, *last*

Random-access iterators to the initial and final positions of the heap range to be sorted. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*comp*

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.

Unless [*first*,*last*) is a one-element heap, this argument shall be the same as used to construct the heap.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

none

## Example

```
1 // range heap example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
4 #include <vector>              // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,5,15};
8     std::vector<int> v(myints,myints+5);
9
10    std::make_heap (v.begin(),v.end());
11    std::cout << "initial max heap : " << v.front() << '\n';
12
13    std::pop_heap (v.begin(),v.end()); v.pop_back();
14    std::cout << "max heap after pop : " << v.front() << '\n';
15
16    v.push_back(99); std::push_heap (v.begin(),v.end());
17    std::cout << "max heap after push: " << v.front() << '\n';
18
19    std::sort_heap (v.begin(),v.end());
20
21    std::cout << "final sorted range :";
22    for (unsigned i=0; i<v.size(); i++)
23        std::cout << ' ' << v[i];
24
25    std::cout << '\n';
26
27    return 0;
28 }
```

Output:

```

initial max heap    : 30
max heap after pop : 20
max heap after push: 99
final sorted range : 5 10 15 20 99

```

## Complexity

Up to linearithmic in the *distance* between *first* and *last*: Performs at most  $N \log(N)$  (where  $N$  is this distance) comparisons of elements, and up to that many element swaps (or moves).

## Data races

The objects in the range [*first*,*last*) are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">make_heap</a>	Make heap from range (function template )
<a href="#">push_heap</a>	Push element into heap range (function template )
<a href="#">pop_heap</a>	Pop element from heap range (function template )
<a href="#">reverse</a>	Reverse range (function template )

## /algorithm/stable\_partition

function template

### std::stable\_partition

<algorithm>

```

template <class BidirectionalIterator, class UnaryPredicate>
BidirectionalIterator stable_partition (BidirectionalIterator first,
                                      BidirectionalIterator last,
                                      UnaryPredicate pred);

```

#### Partition range in two - stable ordering

Rearranges the elements in the range [*first*,*last*), in such a way that all the elements for which *pred* returns *true* precede all those for which it returns *false*, and, unlike function [partition](#), the relative order of elements within each group is preserved.

This is generally implemented using an internal temporary buffer.

## Parameters

*first*, *last*

Bidirectional iterators to the initial and final positions of the sequence to partition. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
BidirectionalIterator shall point to a type for which [swap](#) is defined (and swaps the value of its arguments) and which is both *move-constructible* and *move-assignable*.

*pred*

Unary function that accepts an element in the range as argument, and returns a value convertible to *bool*. The value returned indicates whether the element is to be placed before (if *true*, it is placed before all the elements for which it returns *false*).  
The function shall not modify its argument.

This can either be a function pointer or a function object.

## Return value

An iterator that points to the first element of the second group of elements (those for which *pred* returns *false*), or *last* if this group is empty.

## Example

```

1 // stable_partition example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::stable_partition
4 #include <vector>             // std::vector
5
6 bool IsOdd (int i) { return (i%2)==1; }
7
8 int main () {
9     std::vector<int> myvector;
10
11    // set some values:
12    for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
13
14    std::vector<int>::iterator bound;
15    bound = std::stable_partition (myvector.begin(), myvector.end(), IsOdd);
16
17    // print out content:
18    std::cout << "odd elements:";
19    for (std::vector<int>::iterator it=myvector.begin(); it!=bound; ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    std::cout << "even elements:";
24    for (std::vector<int>::iterator it=bound; it!=myvector.end(); ++it)
25        std::cout << ' ' << *it;
26    std::cout << '\n';

```

```
27     return 0;
28 }
29 }
```

Output:

```
odd elements: 1 3 5 7 9
even elements: 2 4 6 8
```

## Complexity

If enough extra memory is available, linear in the *distance* between *first* and *last*: Applies *pred* exactly once to each element, and performs up to that many element moves.

Otherwise, up to linearithmic: Performs up to  $N \log(N)$  element swaps (where  $N$  is the distance above). It also applies *pred* exactly once to each element.

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">partition</a>	Partition range in two (function template )
<a href="#">sort</a>	Sort elements in range (function template )
<a href="#">reverse</a>	Reverse range (function template )
<a href="#">find_if</a>	Find element in range (function template )

## /algorithm/stable\_sort

function template

### **std::stable\_sort**

<algorithm>

```
template <class RandomAccessIterator>
void stable_sort ( RandomAccessIterator first, RandomAccessIterator last );

template <class RandomAccessIterator, class Compare>
void stable_sort ( RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp );
```

#### Sort elements preserving order of equivalents

Sorts the elements in the range `[first, last)` into ascending order, like [sort](#), but [stable\\_sort](#) preserves the relative order of the elements with equivalent values.

The elements are compared using `operator<` for the first version, and `comp` for the second.

## Parameters

*first*, *last*

Random-access iterators to the initial and final positions of the sequence to be sorted. The range used is `[first, last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
RandomAccessIterator shall point to a type for which [swap](#) is properly defined and which is both *move-constructible* and *move-assignable*.

*comp*

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as *first* argument is considered to go before the second in the specific *strict weak ordering* it defines.  
The function shall not modify any of its arguments.  
This can either be a function pointer or a function object.

## Return value

none

## Example

```
1 // stable_sort example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::stable_sort
4 #include <vector>             // std::vector
5
6 bool compare_as_ints (double i,double j)
7 {
8     return (int(i)<int(j));
9 }
10
11 int main () {
12     double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
13
14     std::vector<double> myvector;
15
16     myvector.assign(mydoubles,mydoubles+8);
17
18     std::cout << "using default comparison:";
19     std::stable_sort (myvector.begin(), myvector.end());
20 }
```

```

21 for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
22     std::cout << ' ' << *it;
23     std::cout << '\n';
24
25 myvector.assign(mydoubles,mydoubles+8);
26
27 std::cout << "using 'compare_as_ints' :";
28 std::stable_sort (myvector.begin(), myvector.end(), compare_as_ints);
29 for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
30     std::cout << ' ' << *it;
31     std::cout << '\n';
32
33 return 0;
}

```

`compare_as_ints` is a function that compares only the integral part of the elements, therefore, elements with the same integral part are considered equivalent. `stable_sort` preserves the relative order these had before the call.

Possible output:

```

using default comparison: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
using 'compare_as_ints' : 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67

```

## Complexity

If enough extra memory is available, linearithmic in the distance between `first` and `last`: Performs up to  $N \log_2(N)$  element comparisons (where  $N$  is this distance), and up to that many element moves.

Otherwise, polyloglinear in that distance: Performs up to  $N \log_2^2(N)$  element comparisons, and up to that many element swaps.

## Data races

The objects in the range `[first, last)` are modified.

## Exceptions

Throws if any of the element comparisons, the element swaps (or moves) or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

# /algorithm/swap

function template

## std::swap

C++98: <algorithm>, C++11: <utility>

```

template <class T> void swap (T& a, T& b);
    header // moved from <algorithm> to <utility> in C++11
non-array (1) template <class T> void swap (T& a, T& b)
    noexcept (is_nothrow_move_constructible<T>::value && is_nothrow_move_assignable<T>::value);
array (2) template <class T, size_t N> void swap(T (&a)[N], T (&b)[N])
    noexcept (noexcept(swap(*a,*b)));

```

### Exchange values of two objects

Exchanges the values of `a` and `b`.

The behavior of this function template is equivalent to:

```

1 template <class T> void swap ( T & a, T & b )
2 {
3     T c(a); a=b; b=c;
4 }

```

Notice how this function involves a copy construction and two assignment operations, which may not be the most efficient way of swapping the contents of classes that store large quantities of data, since each of these operations generally operate in linear time on their size.

Large data types can provide an overloaded version of this function optimizing its performance. Notably, all `standard containers` specialize it in such a way that only a few internal pointers are swapped instead of their entire contents, making them operate in constant time.

This function is no longer defined in header `<algorithm>`, but in `<utility>`.

The behavior of these function templates is equivalent to:

```

1 template <class T> void swap (T& a, T& b)
2 {
3     T c(std::move(a)); a=std::move(b); b=std::move(c);
4 }
5 template <class T, size_t N> void swap (T (&a)[N], T (&b)[N])
6 {
7     for (size_t i = 0; i<N; ++i) swap (a[i],b[i]);
8 }

```

Many components of the standard library (within `std`) call `swap` in an *unqualified* manner to allow custom overloads for non-fundamental types to be called instead of this generic version: Custom overloads of `swap` declared in the same namespace as the type for which they are provided get selected through *argument-dependent lookup* over this generic version.

## Parameters

a, b

Two objects, whose contents are swapped.

Type T shall be *copy-constructible* and *assignable*.

Type T shall be *move-constructible* and *move-assignable* (or have swap defined for it, for version (2)).

## Return value

none

## Example

```
1 // swap algorithm example (C++98)
2 #include <iostream>           // std::cout
3 #include <algorithm>         // std::swap
4 #include <vector>            // std::vector
5
6 int main () {
7
8     int x=10, y=20;           // x:10 y:20
9     std::swap(x,y);          // x:20 y:10
10
11    std::vector<int> foo (4,x), bar (6,y);      // foo:4x20 bar:6x10
12    std::swap(foo,bar);        // foo:6x10 bar:4x20
13
14    std::cout << "foo contains:";
15    for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
16        std::cout << ' ' << *it;
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
foo contains: 10 10 10 10 10 10
```

## Complexity

**Non-array:** Constant: Performs exactly one construction and two assignments (although notice that each of these operations works on its own complexity).

**Array:** Linear in N: performs a swap operation per element.

## Data races

Both a and b are modified.

## Exceptions

Throws if the construction or assignment of type T throws.

Never throws if T is *nothrow-move-constructible* and *nothrow-move-assignable*.

Note that if T does not fulfill the requirements specified above (in parameters), it causes *undefined behavior*.

## See also

<a href="#">copy</a>	Copy range of elements ( <a href="#">function template</a> )
<a href="#">fill</a>	Fill range with value ( <a href="#">function template</a> )
<a href="#">replace</a>	Replace value in range ( <a href="#">function template</a> )

# /algorithm/swap\_ranges

function template

## std::swap\_ranges

<algorithm>

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2);
```

### Exchange values of two ranges

Exchanges the values of each of the elements in the range [first1, last1) with those of their respective elements in the range beginning at first2.

The function calls `swap` (unqualified) to exchange the elements.

The behavior of this function template is equivalent to:

```
1 template<class ForwardIterator1, class ForwardIterator2>
2     ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardIterator1 last1,
3                                     ForwardIterator2 first2)
4 {
5     while (first1!=last1) {
6         swap (*first1, *first2);
7         ++first1; ++first2;
8     }
9     return first2;
10 }
```

## Parameters

`first1, last1`  
Forward iterators to the initial and final positions in one of the sequences to be swapped. The range used is  $[first1, last1)$ , which contains all the elements between `first1` and `last1`, including the element pointed by `first1` but not the element pointed by `last1`.

`first2`  
Forward iterator to the initial position in the other sequence to be swapped. The range used includes the same number of elements as the range  $[first1, last1)$ .  
The two ranges shall not overlap.

The ranges shall not overlap.  
`swap` shall be defined to exchange the types pointed by both iterator types symmetrically (in both orders).

## Return value

An iterator to the last element swapped in the second sequence.

## Example

```
1 // swap_ranges example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::swap_ranges
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> foo (5,10);      // foo: 10 10 10 10 10
8     std::vector<int> bar (5,33);      // bar: 33 33 33 33 33
9
10    std::swap_ranges(foo.begin() + 1, foo.end() - 1, bar.begin());
11
12    // print out results of swap:
13    std::cout << "foo contains:";
14    for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    std::cout << "bar contains:";
19    for (std::vector<int>::iterator it=bar.begin(); it!=bar.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
foo contains: 10 33 33 33 10
bar contains: 10 10 10 33 33
```

## Complexity

Linear in the `distance` between `first` and `last`: Performs a swap operation for each element in the range.

## Data races

The objects in both ranges are modified.

## Exceptions

Throws if either an element assignment or an operation on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<code>swap</code>	Exchange values of two objects ( <a href="#">function template</a> )
<code>iter_swap</code>	Exchange values of objects pointed to by two iterators ( <a href="#">function template</a> )
<code>replace</code>	Replace value in range ( <a href="#">function template</a> )

# /algorithm/transform

function template

## std::transform

<algorithm>

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
unary operation(1)    OutputIterator transform (InputIterator first1, InputIterator last1,
                                              OutputIterator result, UnaryOperation op);
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation>
binary operation(2)   OutputIterator transform (InputIterator1 first1, InputIterator1 last1,
                                              InputIterator2 first2, OutputIterator result,
                                              BinaryOperation binary_op);
```

### Transform range

Applies an operation sequentially to the elements of one (1) or two (2) ranges and stores the result in the range that begins at `result`.

#### (1) unary operation

Applies `op` to each of the elements in the range  $[first1, last1)$  and stores the value returned by each operation in the range that begins at `result`.

#### (2) binary operation

Calls `binary_op` using each of the elements in the range `[first1, last1]` as first argument, and the respective argument in the range that begins at `first2` as second argument. The value returned by each call is stored in the range that begins at `result`.

The behavior of this function template is equivalent to:

```
1 template <class InputIterator, class OutputIterator, class UnaryOperator>
2   OutputIterator transform (InputIterator first1, InputIterator last1,
3                           OutputIterator result, UnaryOperator op)
4 {
5   while (first1 != last1) {
6     *result = op(*first1); // or: *result=binary_op(*first1,*first2++);
7     ++result; ++first1;
8   }
9   return result;
10 }
```

The function allows for the destination range to be the same as one of the input ranges to make transformations *in place*.

## Parameters

`first1, last1`  
Input iterators to the initial and final positions of the first sequence. The range used is `[first1, last1)`, which contains all the elements between `first1` and `last1`, including the element pointed to by `first1` but not the element pointed to by `last1`.

`first2`  
Input iterator to the initial position of the second range. The range includes as many elements as `[first1, last1)`.

`result`  
Output iterator to the initial position of the range where the operation results are stored. The range includes as many elements as `[first1, last1)`.

`op`  
Unary function that accepts one element of the type pointed to by `InputIterator` as argument, and returns some result value convertible to the type pointed to by `OutputIterator`.  
This can either be a function pointer or a function object.

`binary_op`  
Binary function that accepts two elements as argument (one of each of the two sequences), and returns some result value convertible to the type pointed to by `OutputIterator`.  
This can either be a function pointer or a function object.

Neither `op` nor `binary_op` should directly modify the elements passed as its arguments: These are indirectly modified by the algorithm (using the return value) if the same range is specified for `result`.

## Return value

An iterator pointing to the element that follows the last element written in the `result` sequence.

## Example

```
1 // transform algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::transform
4 #include <vector>             // std::vector
5 #include <functional>         // std::plus
6
7 int op_increase (int i) { return ++i; }
8
9 int main () {
10   std::vector<int> foo;          // foo: 10 20 30 40 50
11   std::vector<int> bar;
12
13   // set some values:
14   for (int i=1; i<6; i++)
15     foo.push_back (i*10);
16
17   bar.resize(foo.size());        // allocate space
18
19   std::transform (foo.begin(), foo.end(), bar.begin(), op_increase);
20   // bar: 11 21 31 41 51
21
22   // std::plus adds together its two arguments:
23   std::transform (foo.begin(), foo.end(), bar.begin(), foo.begin(), std::plus<int>());
24   // foo: 21 41 61 81 101
25
26   std::cout << "foo contains:";
27   for (std::vector<int>::iterator it=foo.begin(); it!=foo.end(); ++it)
28     std::cout << ' ' << *it;
29   std::cout << '\n';
30
31   return 0;
32 }
```

Output:

```
foo contains: 21 41 61 81 101
```

## Complexity

Linear in the `distance` between `first1` and `last1`: Performs one assignment and one application of `op` (or `binary_op`) per element.

## Data races

The objects in the range `[first1, last1)` (and eventually those in the range beginning at `first2`) are accessed (each object is accessed exactly once). The objects in the range beginning at `result` are modified.

## Exceptions

Throws if any of the function calls, the assignments or the operations on iterators throws.  
Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">for_each</a>	Apply function to range (function template )
<a href="#">copy</a>	Copy range of elements (function template )

# /algorithm/unique

function template

## std::unique

<algorithm>

```
equality (1)  template <class ForwardIterator>
               ForwardIterator unique (ForwardIterator first, ForwardIterator last);
template <class ForwardIterator, class BinaryPredicate>
         ForwardIterator unique (ForwardIterator first, ForwardIterator last,
                                BinaryPredicate pred);
```

### Remove consecutive duplicates in range

Removes all but the first element from every consecutive group of equivalent elements in the range `[first, last)`.

The function cannot alter the properties of the object containing the range of elements (i.e., it cannot alter the size of an array or a container): The removal is done by replacing the duplicate elements by the next element that is not a duplicate, and signaling the new size of the shortened range by returning an iterator to the element that should be considered its new *past-the-end* element.

The relative order of the elements not removed is preserved, while the elements between the returned iterator and *last* are left in a valid but unspecified state.

The function uses operator`==` to compare the pairs of elements (or *pred*, in version (2)).

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator>
2   ForwardIterator unique (ForwardIterator first, ForwardIterator last)
3 {
4   if (first==last) return last;
5
6   ForwardIterator result = first;
7   while (++first != last)
8   {
9     if (!(*result == *first)) // or: if (!pred(*result,*first)) for version (2)
10    *(++result)=*first;
11  }
12  return ++result;
13 }
```

## Parameters

*first*, *last*

Forward iterators to the initial and final positions of the sequence of *move-assignable* elements. The range used is `[first, last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

*pred*

Binary function that accepts two elements in the range as argument, and returns a value convertible to `bool`. The value returned indicates whether both arguments are considered equivalent (if `true`, they are equivalent and one of them is removed).

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to the element that follows the last element not removed.

The range between *first* and this iterator includes all the elements in the sequence that were not considered duplicates.

## Example

```
1 // unique algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::unique, std::distance
4 #include <vector>             // std::vector
5
6 bool myfunction (int i, int j) {
7   return (i==j);
8 }
9
10 int main () {
11   int myints[] = {10,20,20,20,30,30,20,20,10};           // 10 20 20 20 30 30 20 20 10
12   std::vector<int> myvector (myints,myints+9);
13
14 // using default comparison:
15 std::vector<int>::iterator it;
16 it = std::unique (myvector.begin(), myvector.end());    // 10 20 30 20 10 ? ? ? ?
17 // ^
18
19 myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30 20 10
20
21 // using predicate comparison:
22 std::unique (myvector.begin(), myvector.end(), myfunction); // (no changes)
```

```

23 // print out content:
24 std::cout << "myvector contains:";
25 for (it=myvector.begin(); it!=myvector.end(); ++it)
26     std::cout << ' ' << *it;
27 std::cout << '\n';
28
29 return 0;
30 }

```

#### Output:

```
myvector contains: 10 20 30 20 10
```

### Complexity

For non-empty ranges, linear in one less than the [distance](#) between *first* and *last*: Compares each pair of consecutive elements, and possibly performs assignments on some of them.

### Data races

The objects in the range [*first*,*last*) are accessed and potentially modified.

### Exceptions

Throws if any of *pred*, the element comparisons, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

### See also

<a href="#">unique_copy</a>	Copy range removing duplicates ( <a href="#">function template</a> )
<a href="#">adjacent_find</a>	Find equal adjacent elements in range ( <a href="#">function template</a> )
<a href="#">remove</a>	Remove value from range ( <a href="#">function template</a> )
<a href="#">remove_if</a>	Remove elements from range ( <a href="#">function template</a> )

## /algorithm/unique\_copy

function template

### std::unique\_copy

<algorithm>

```

template <class InputIterator, class OutputIterator>
equality (1) OutputIterator unique_copy (InputIterator first, InputIterator last,
                                         OutputIterator result);
template <class InputIterator, class OutputIterator, class BinaryPredicate>
predicate (2) OutputIterator unique_copy (InputIterator first, InputIterator last,
                                         OutputIterator result, BinaryPredicate pred);

```

#### Copy range removing duplicates

Copies the elements in the range [*first*,*last*) to the range beginning at *result*, except consecutive duplicates (elements that compare equal to the element preceding).

Only the first element from every consecutive group of equivalent elements in the range [*first*,*last*) is copied.

The comparison between elements is performed by either applying operator==, or the template parameter *pred* (for the second version) between them.

The behavior of this function template is equivalent to:

```

1 template <class InputIterator, class OutputIterator>
2     OutputIterator unique_copy (InputIterator first, InputIterator last,
3                                 OutputIterator result)
4 {
5     if (first==last) return result;
6
7     *result = *first;
8     while (++first != last) {
9         typename iterator_traits<InputIterator>::value_type val = *first;
10        if (!(*result == val)) // or: if (!pred(*result, val)) for version (2)
11            *(++result)=val;
12    }
13    return ++result;
14 }

```

### Parameters

*first*, *last*

Forward iterators to the initial and final positions in a sequence. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

If *InputIterator* is a *single-pass iterator*, the type it points to shall be *copy-constructible* and *copy-assignable*.

*result*

Output iterator to the initial position of the range where the resulting range of values is stored.

The pointed type shall support being assigned the value of an element in the range [*first*,*last*).

*pred*

Binary function that accepts two elements in the range as argument, and returns a value convertible to *bool*. The value returned indicates whether both arguments are considered equivalent (if *true*, they are equivalent and one of them is removed).

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

The ranges shall not overlap.

## Return value

An iterator pointing to the end of the copied range, which contains no consecutive duplicates.

## Example

```
1 // unique_copy example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::unique_copy, std::sort, std::distance
4 #include <vector>             // std::vector
5
6 bool myfunction (int i, int j) {
7     return (i==j);
8 }
9
10 int main () {
11     int myints[] = {10,20,20,20,30,30,20,20,10};
12     std::vector<int> myvector (9);                      // 0 0 0 0 0 0 0 0 0
13
14     // using default comparison:
15     std::vector<int>::iterator it;
16     it=std::unique_copy (myints,myints+9,myvector.begin()); // 10 20 30 20 10 0 0 0 0
17     //
18
19     std::sort (myvector.begin(),it);                     // 10 10 20 20 30 0 0 0 0
20     //
21
22     // using predicate comparison:
23     it=std::unique_copy (myvector.begin(), it, myvector.begin(), myfunction);
24     // 10 20 30 20 30 0 0 0 0
25     //
26
27     myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30
28
29     // print out content:
30     std::cout << "myvector contains:";
31     for (it=myvector.begin(); it!=myvector.end(); ++it)
32         std::cout << ' ' << *it;
33     std::cout << '\n';
34
35     return 0;
36 }
```

Output:

```
myvector contains: 10 20 30
```

## Complexity

Up to linear in the *distance* between *first* and *last*: Compares each pair of elements, and performs an assignment operation for those elements not matching.

## Data races

The objects in the range [*first*,*last*] are accessed.

The objects in the range between *result* and the returned value are modified.

## Exceptions

Throws if any of *pred*, the element comparisons, the element assignments or the operations on iterators throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">unique</a>	Remove consecutive duplicates in range (function template )
<a href="#">adjacent_find</a>	Find equal adjacent elements in range (function template )
<a href="#">remove</a>	Remove value from range (function template )
<a href="#">remove_if</a>	Remove elements from range (function template )

## /algorithm/upper\_bound

function template

### std::upper\_bound

<algorithm>

```
template <class ForwardIterator, class T>
default (1)   ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last,
                                         const T& val);
template <class ForwardIterator, class T, class Compare>
custom (2)    ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last,
                                         const T& val, Compare comp);
```

#### Return iterator to upper bound

Returns an iterator pointing to the first element in the range [*first*,*last*] which compares greater than *val*.

The elements are compared using *operator<* for the first version, and *comp* for the second. The elements in the range shall already be *sorted* according to this same criterion (*operator<* or *comp*), or at least *partitioned* with respect to *val*.

The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is specially efficient for random-access iterators.

Unlike `lower_bound`, the value pointed by the iterator returned by this function cannot be equivalent to `val`, only greater.

The behavior of this function template is equivalent to:

```
1 template <class ForwardIterator, class T>
2     ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last, const T& val)
3 {
4     ForwardIterator it;
5     iterator_traits<ForwardIterator>::difference_type count, step;
6     count = std::distance(first, last);
7     while (count>0)
8     {
9         it = first; step=count/2; std::advance (it,step);
10        if (!(val<*it))           // or: if (!comp(val,*it)), for version (2)
11            { first=++it; count-=step+1; }
12        else count=step;
13    }
14    return first;
15 }
```

## Parameters

`first, last`

Forward iterators to the initial and final positions of a sorted (or properly partitioned) sequence. The range used is `[first, last)`, which contains all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

`val`

Value of the upper bound to search for in the range.

For (1), `T` shall be a type supporting being compared with elements of the range `[first, last)` as the left-hand side operand of operator`<`.

`comp`

Binary function that accepts two arguments (the first is always `val`, and the second of the type pointed by `ForwardIterator`), and returns a value convertible to `bool`. The value returned indicates whether the first argument is considered to go before the second.

The function shall not modify any of its arguments.

This can either be a function pointer or a function object.

## Return value

An iterator to the upper bound position for `val` in the range.

If no element in the range compares greater than `val`, the function returns `last`.

## Example

```
1 // lower_bound/upper_bound example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::lower_bound, std::upper_bound, std::sort
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,30,20,10,10,20};
8     std::vector<int> v(myints+8);           // 10 20 30 30 20 10 10 20
9
10    std::sort (v.begin(), v.end());          // 10 10 10 20 20 20 30 30
11
12    std::vector<int>::iterator low,up;
13    low=std::lower_bound (v.begin(), v.end(), 20); //
14    up= std::upper_bound (v.begin(), v.end(), 20); //
15
16    std::cout << "lower_bound at position " << (low- v.begin()) << '\n';
17    std::cout << "upper_bound at position " << (up - v.begin()) << '\n';
18
19    return 0;
20 }
```

Output:

```
lower_bound at position 3
upper_bound at position 6
```

## Complexity

On average, logarithmic in the `distance` between `first` and `last`: Performs approximately  $\log_2(N)+1$  element comparisons (where  $N$  is this distance).

On non-random-access iterators, the iterator advances produce themselves an additional linear complexity in  $N$  on average.

## Data races

The objects in the range `[first, last)` are accessed.

## Exceptions

Throws if either an element comparison or an operation on an iterator throws.

Note that invalid arguments cause *undefined behavior*.

## See also

<a href="#">lower_bound</a>	Return iterator to lower bound (function template )
<a href="#">equal_range</a>	Get subrange of equal elements (function template )
<a href="#">binary_search</a>	Test if value exists in sorted sequence (function template )

<a href="#">max_element</a>	Return largest element in range (function template )
-----------------------------	--

## /bitset

header

### <bitset>

#### Bitset header

Header that defines the `bitset` class:

#### Classes

<a href="#">bitset</a>	Bitset (class template )
------------------------	--------------------------

## /bitset/bitset

class template

### std::bitset

<bitset>

`template <size_t N> class bitset;`

#### Bitset

A `bitset` stores bits (elements with only two possible values: 0 or 1, true or false, ...).

The class emulates an array of `bool` elements, but optimized for space allocation: generally, each element occupies only one bit (which, on most systems, is eight times less than the smallest elemental type: `char`).

Each bit position can be accessed individually: for example, for a given `bitset` named `foo`, the expression `foo[3]` accesses its fourth bit, just like a regular array accesses its elements. But because no elemental type is a single bit in most C++ environments, the individual elements are accessed as special references type (see `bitset::reference`).

Bitsets have the feature of being able to be constructed from and converted to both integer values and binary strings (see its `constructor` and members `to_ulong` and `to_string`). They can also be directly inserted and extracted from streams in binary format (see `applicable operators`).

The `size` of a `bitset` is fixed at compile-time (determined by its template parameter). For a class that also optimizes for space allocation and allows for dynamic resizing, see the `bool` specialization of `vector` (`vector<bool>`).

#### Template parameters

N

Size of the `bitset`, in terms of number of bits.  
It is returned by member function `bitset::size`.  
`size_t` is an unsigned integral type.

#### Member types

<a href="#">reference</a>	Reference-like type (public member class )
---------------------------	--

#### Member functions

<a href="#">(constructor)</a>	Construct <code>bitset</code> (public member function )
<a href="#">applicable operators</a>	Bitset operators (function )

#### Bit access

<a href="#">operator[]</a>	Access bit (public member function )
<a href="#">count</a>	Count bits set (public member function )
<a href="#">size</a>	Return size (public member function )
<a href="#">test</a>	Return bit value (public member function )
<a href="#">any</a>	Test if any bit is set (public member function )
<a href="#">none</a>	Test if no bit is set (public member function )
<a href="#">all</a>	Test if all bits are set (public member function )

#### Bit operations

<a href="#">set</a>	Set bits (public member function )
<a href="#">reset</a>	Reset bits (public member function )
<a href="#">flip</a>	Flip bits (public member function )

#### Bitset operations

<a href="#">to_string</a>	Convert to string (public member function )
<a href="#">to_ulong</a>	Convert to unsigned long integer (public member function )
<a href="#">to_ullong</a>	Convert to unsigned long long (public member function )

#### Non-member function overloads

<a href="#">applicable operators</a>	Bitset operators (function )
--------------------------------------	------------------------------

## Non-member class specializations

hash<bitset>	Hash for bitset (class template specialization )
--------------	--

# /bitset/bitset/all

public member function

## std::bitset::all

<bitset>

bool all() const noexcept;

### Test if all bits are set

Returns whether all of the bits in the `bitset` are set (to `one`).

For this function to return true, all bits up to the `bitset size` shall be set.

### Parameters

none

### Return value

true if all of the bits in the `bitset` are set (to `one`), and false otherwise.

### Example

```
1 // bitset::all
2 #include <iostream>           // std::cin, std::cout, std::boolalpha
3 #include <bitset>            // std::bitset
4
5 int main ()
6 {
7     std::bitset<8> foo;
8
9     std::cout << "Please, enter an 8-bit binary number: ";
10    std::cin >> foo;
11
12    std::cout << std::boolalpha;
13    std::cout << "all: " << foo.all() << '\n';
14    std::cout << "any: " << foo.any() << '\n';
15    std::cout << "none: " << foo.none() << '\n';
16
17    return 0;
18 }
```

Possible output:

```
Please, enter an 8-bit binary number: 11111111
all: true
any: true
none: false
```

### Data races

The `bitset` is accessed.

### Exception safety

**No-throw guarantee:** never throws exceptions.

### See also

bitset::count	Count bits set (public member function )
bitset::any	Test if any bit is set (public member function )
bitset::none	Test if no bit is set (public member function )
bitset::test	Return bit value (public member function )

# /bitset/bitset/any

public member function

## std::bitset::any

<bitset>

bool any() const;

bool any() const noexcept;

### Test if any bit is set

Returns whether any of the bits is set (i.e., whether at least one bit in the `bitset` is set to `one`).

This is the opposite of `bitset::none`.

### Parameters

none

## Return value

true if any of the bits in the `bitset` is set (to one), and false otherwise.

## Example

```
1 // bitset::any
2 #include <iostream>           // std::cin, std::cout
3 #include <bitset>             // std::bitset
4
5 int main ()
6 {
7     std::bitset<16> foo;
8
9     std::cout << "Please, enter a binary number: ";
10    std::cin >> foo;
11
12    if (foo.any())
13        std::cout << foo << " has " << foo.count() << " bits set.\n";
14    else
15        std::cout << foo << " has no bits set.\n";
16
17    return 0;
18 }
```

Possible output:

```
Please, enter a binary number: 10110
000000000010110 has 3 bits set.
```

## Data races

The `bitset` is accessed.

## Exception safety

**No-throw guarantee:** never throws exceptions.

## See also

<code>bitset::count</code>	Count bits set (public member function )
<code>bitset::none</code>	Test if no bit is set (public member function )
<code>bitset::test</code>	Return bit value (public member function )

## /bitset/bitset/bitset

public member function

### std::bitset::bitset

<bitset>

```
default (1) bitset();
integer value (2) bitset (unsigned long val);
template<class charT, class traits, class Alloc>
explicit bitset (const basic_string<charT,traits,Alloc>& str,
string (3) typename basic_string<charT,traits,Alloc>::size_type pos = 0,
typename basic_string<charT,traits,Alloc>::size_type n =
basic_string<charT,traits,Alloc>::npos);

default (1) constexpr bitset() noexcept;
integer value (2) constexpr bitset (unsigned long long val) noexcept;
template <class charT, class traits, class Alloc>
explicit bitset (const basic_string<charT,traits,Alloc>& str,
string (3) typename basic_string<charT,traits,Alloc>::size_type pos = 0,
typename basic_string<charT,traits,Alloc>::size_type n =
basic_string<charT,traits,Alloc>::npos,
charT zero = charT('0'), chart one = charT('1'));
template <class charT>
explicit bitset (const charT* str,
typename basic_string<charT>::size_type n = basic_string<charT>::npos,
charT zero = charT('0'), chart one = charT('1'));
```

### Construct bitset

Constructs a `bitset` container object:

#### (1) default constructor

The object is initialized with zeros.

#### (2) initialization from integer value

Initializes the object with the bit values of `val`:

#### (3) initialization from string or (4) C-string

Uses the sequence of `zeros` and/or `ones` in `str` to initialize the first `n` bit positions of the constructed `bitset` object.

Note that `bitset` objects have a *fixed size* (determined by their class template argument) no matter the constructor used: Those bit positions not explicitly set by the constructor are initialized with a value of zero.

## Parameters

val	Integral value whose bits are copied to the bitset positions. - If the value representation of <i>val</i> is greater than the <i>bitset size</i> , only the least significant bits of <i>val</i> are taken into consideration. - If the value representation of <i>val</i> is less than the <i>bitset size</i> , the remaining bit positions are initialized to zero.
str	A <i>basic_string</i> whose contents are used to initialize the <i>bitset</i> : The constructor parses the string reading at most <i>n</i> characters beginning at <i>pos</i> , interpreting the character values '0' and '1' as zero and one, respectively. Note that the least significant bit is represented by the last character read (not the first); Thus, the first bit position is read from the right-most character, and the following bits use the characters preceding this, from right to left. If this sequence is shorter than the <i>bitset size</i> , the remaining bit positions are initialized to zero.
	A <i>basic_string</i> or <i>C-string</i> whose contents are used to initialize the <i>bitset</i> : The constructor parses the string reading at most <i>n</i> characters (beginning at <i>pos</i> for (3)), interpreting the character values specified as arguments <i>zero</i> and <i>one</i> as zeros and ones, respectively. Note that the least significant bit is represented by the last character read (not the first); Thus, the first bit position is read from the right-most character, and the following bits use the characters preceding this, from right to left. If this sequence is shorter than the <i>bitset size</i> , the remaining bit positions are initialized to zero.
pos	First character in the <i>basic_string</i> to be read and interpreted. If this is greater than the <i>length</i> of <i>str</i> , an <i>out_of_range</i> exception is thrown.
n	Number of characters to read. Any value greater than the <i>bitset size</i> (including <i>npos</i> ) is equivalent to specifying exactly the <i>bitset size</i> .
zero, one	Character values to represent <i>zero</i> and <i>one</i> .

## Example

```
1 // constructing bitsets
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <bitset>              // std::bitset
5
6 int main ()
7 {
8     std::bitset<16> foo;
9     std::bitset<16> bar (0xfa2);
10    std::bitset<16> baz (std::string("0101111001"));
11
12    std::cout << "foo: " << foo << '\n';
13    std::cout << "bar: " << bar << '\n';
14    std::cout << "baz: " << baz << '\n';
15
16    return 0;
17 }
```

Output:

```
foo: 0000000000000000
bar: 000011110100010
baz: 0000000101111001
```

## Data races

Constructors (3) and (4) access the characters in *str*.

## Exception safety

Neither the *default constructor* (1) nor the *constructor from integer value* (2) throw exceptions.

The other constructors cause no side effects in case an exception is thrown (strong guarantee).

Throws *out\_of\_range* if *pos* > *str.size()*.

## See also

<a href="#">bitset::set</a>	Set bits (public member function )
<a href="#">bitset::reset</a>	Reset bits (public member function )
<a href="#">bitset::operator[]</a>	Access bit (public member function )
<a href="#">bitset operators</a>	Bitset operators (function )

## /bitset/bitset/count

public member function

### std::bitset::count

<bitset>

```
size_t count() const;
size_t count() const noexcept;
```

#### Count bits set

Returns the number of bits in the *bitset* that are set (i.e., that have a value of *one*).

For the total number of bits in the *bitset* (including both *zeros* and *ones*), see *bitset::size*.

## Parameters

none

## Return value

The number of bits set.

`size_t` is an unsigned integral type.

## Example

```
1 // bitset::count           // std::cout
2 #include <iostream>        // std::string
3 #include <string>          // std::bitset
4 #include <bitset>          // std::bitset
5
6 int main ()
7 {
8     std::bitset<8> foo (std::string("10110011"));
9
10    std::cout << foo << " has ";
11    std::cout << foo.count() << " ones and ";
12    std::cout << (foo.size()-foo.count()) << " zeros.\n";
13
14    return 0;
15 }
```

Output:

```
10110011 has 5 ones and 3 zeros.
```

## Data races

The `bitset` is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `bitset`.  
If the `bitset size` is too big to be represented by the return type, `overflow_error` is thrown.

## See also

<code>bitset::size</code>	Return size (public member function )
<code>bitset::any</code>	Test if any bit is set (public member function )
<code>bitset::none</code>	Test if no bit is set (public member function )

# /bitset/bitset/flip

public member function

## std::bitset::flip

<bitset>

```
    all bits (1) bitset& flip();
    single bit (2) bitset& flip (size_t pos);
    all bits (1) bitset& flip() noexcept;
    single bit (2) bitset& flip (size_t pos);
```

### Flip bits

Flips bit values converting `zeros` into `ones` and `ones` into `zeros`:

#### (1) all bits

Flips all bits in the `bitset`.

#### (2) single bit

Flips the bit at position `pos`.

## Parameters

`pos`

Order position of the bit whose value is flipped.

Order positions are counted from the rightmost bit, which is order position 0.

If `pos` is equal or greater than the `bitset size`, an `out_of_range` exception is thrown.

`size_t` is an unsigned integral type.

## Return value

`*this`

## Example

```
1 // bitset::flip           // std::cout
2 #include <iostream>        // std::string
3 #include <string>          // std::bitset
4 #include <bitset>          // std::bitset
```

```

5 int main ()
6 {
7     std::bitset<4> foo (std::string("0001"));
8
9     std::cout << foo.flip(2) << '\n';      // 0101
10    std::cout << foo.flip() << '\n';       // 1010
11
12    return 0;
13 }

```

Output:

```

0101
1010

```

## Data races

Both the `bitset` and its bits are modified.

## Exception safety

For (1): it never throws exceptions (no-throw guarantee).

For (2): in case of exception, the object is in a valid state (basic guarantee).

If `pos` is equal or greater than the `bitset size`, the function throws an `out_of_range` exception.

## See also

<code>bitset::set</code>	Set bits (public member function )
<code>bitset::reset</code>	Reset bits (public member function )

## /bitset/bitset/hash

class template specialization

### std::hash<bitset>

<bitset>

```

template <class T> struct hash;           // unspecialized
template <size_t N> struct hash<bitset<N>>; // bitset specialization

```

#### Hash for bitset

Unary function object class that defines the `hash` specialization for `bitset`.

The functional call returns a hash value based on the `bitset` passed as argument: A *hash value* is a value that depends solely on its argument, returning always the same value for the same argument (for a given program execution). The value returned shall have a small likelihood of being the same as the one returned for a different argument (with chances of collision approaching `1/numeric_limits<size_t>::max`).

This allows the use of `bitset` objects as keys for *unordered containers* (such as `unordered_set` or `unordered_map`).

See `hash` for additional information.

## Member functions

### operator()

Returns a hash value for its argument, as a value of type `size_t`.  
`size_t` is an unsigned integral type.

## /bitset/bitset/none

public member function

### std::bitset::none

<bitset>

```

bool none() const;
bool none() const noexcept;

```

#### Test if no bit is set

Returns whether none of the bits is set (i.e., whether all bits in the `bitset` have a value of zero).

This is the opposite of `bitset::any`.

## Parameters

`none`

## Return value

`true` if none of the bits in the `bitset` is set (to `one`), and `false` otherwise.

## Example

```

1 // bitset::none
2 #include <iostream>           // std::cin, std::cout
3 #include <bitset>             // std::bitset
4
5 int main ()

```

```

6 {
7     std::bitset<16> foo;
8
9     std::cout << "Please, enter a binary number: ";
10    std::cin >> foo;
11
12    if (foo.none())
13        std::cout << foo << " has no bits set.\n";
14    else
15        std::cout << foo << " has " << foo.count() << " bits set.\n";
16
17    return 0;
18 }

```

Possible output:

```

Please, enter a binary number: 11010111
0000000011010111 has 6 bits set.

```

## Data races

The `bitset` is accessed.

## Exception safety

**No-throw guarantee:** never throws exceptions.

## See also

<code>bitset::count</code>	Count bits set (public member function )
<code>bitset::any</code>	Test if any bit is set (public member function )
<code>bitset::test</code>	Return bit value (public member function )

## /bitset/bitset/operator[]

public member function

### std::bitset::operator[]

<bitset>

```

    bool operator[] (size_t pos) const;
    reference operator[] (size_t pos);

```

#### Access bit

The function returns the value (or a reference) to the bit at position *pos*.

With this operator, no range check is performed. Use `bitset::test` to access the value with `bitset` bounds checked.

## Parameters

`pos` Order position of the bit whose value is accessed.  
Order positions are counted from the rightmost bit, which is order position 0.  
`size_t` is an unsigned integral type.

## Return value

The bit at position *pos*.

If the `bitset` object is const-qualified, the function returns a `bool` value. Otherwise, it returns a value of the special member type `reference`, which emulates a `bool` value with *reference-semantics* with respect to one bit in the `bitset` (see `bitset::reference`).

## Example

```

1 // bitset::operator[]
2 #include <iostream>           // std::cout
3 #include <bitset>            // std::bitset
4
5 int main ()
6 {
7     std::bitset<4> foo;
8
9     foo[1]=1;                  // 0010
10    foo[2]=foo[1];             // 0110
11
12    std::cout << "foo: " << foo << '\n';
13
14    return 0;
15 }

```

Output:

```

foo: 0110

```

## Data races

The `bitset` is accessed (neither the const nor the non-const versions modify the container).

The reference returned by the non-const version can be used to access or modify the bits in the `bitset`. Notice that modifying a single bit may have effects on an undetermined number of other bits in the `bitset`, thus rendering concurrent access/modification of different bits not thread-safe.

## Exception safety

If `pos` is not a valid bit position, it causes *undefined behavior*.

Otherwise, if an exception is thrown by this member function, the `bitset` is left in a valid state (basic guarantee).

## See also

<code>bitset::test</code>	Return bit value (public member function )
---------------------------	--

# /bitset/bitset/operators

function

## std::bitset operators

<bitset>

```
bitset& operator&= (const bitset& rhs);
bitset& operator|= (const bitset& rhs);
bitset& operator^= (const bitset& rhs);
bitset& operator<= (size_t pos);
bitset& operator>= (size_t pos);
bitset operator~() const;
bitset operator<<(size_t pos) const;
bitset operator>>(size_t pos) const;
bool operator== (const bitset& rhs) const;
bool operator!= (const bitset& rhs) const;

template<size_t N>
    bitset<N> operator& (const bitset<N>& lhs, const bitset<N>& rhs);
template<size_t N>
    bitset<N> operator| (const bitset<N>& lhs, const bitset<N>& rhs);
template<size_t N>
    bitset<N> operator^ (const bitset<N>& lhs, const bitset<N>& rhs);

template<class charT, class traits, size_t N>
    basic_istream<charT, traits>&
        operator>> (basic_istream<charT,traits>& is, bitset<N>& rhs);
template<class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
        operator<< (basic_ostream<charT,traits>& os, const bitset<N>& rhs);

member functions
```

```
bitset& operator&= (const bitset& rhs) noexcept;
bitset& operator|= (const bitset& rhs) noexcept;
bitset& operator^= (const bitset& rhs) noexcept;
bitset& operator<= (size_t pos) noexcept;
bitset& operator>= (size_t pos) noexcept;
bitset operator~() const noexcept;
bitset operator<<(size_t pos) const noexcept;
bitset operator>>(size_t pos) const noexcept;
bool operator== (const bitset& rhs) const noexcept;
bool operator!= (const bitset& rhs) const noexcept;

template<size_t N>
    bitset<N> operator& (const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
template<size_t N>
    bitset<N> operator| (const bitset<N>& lhs, const bitset<N>& rhs) noexcept;
template<size_t N>
    bitset<N> operator^ (const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

non-member functions
```

```
template<class charT, class traits, size_t N>
    basic_istream<charT, traits>&
        operator>> (basic_istream<charT,traits>& is, bitset<N>& rhs);
template<class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
        operator<< (basic_ostream<charT,traits>& os, const bitset<N>& rhs);

member functions
```

```
non-member functions
```

```
iostream inserters/extractors
```

```
iostream inserters/extractors
```

## Bitset operators

Performs the proper bitwise operation using the contents of the `bitset`.

## Parameters

`lhs`

Left-hand side `bitset` object (for non-member functions).

`rhs`

Right-hand side `bitset` object.

Both the *left-hand side* and *right-hand side* `bitset` objects must have the same amount of bits (i.e., have the same class template parameter, *N*).

`pos`

Number of bit locations to be shifted.

`is,os`

`basic_istream` or `basic_ostream` object from which a `bitset` object is respectively extracted or inserted. The format in which bitsets are inserted/extracted is a sequence of (suitably widened) '0' and '1' characters.

## Return value

If a reference: the left-hand side object (`*this, is or os`).

Otherwise: the result of the operation (either a `bitset` object, or true or false for the relational operators).

## Example

```

1 // bitset operators
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <bitset>              // std::bitset
5
6 int main ()
7 {
8     std::bitset<4> foo (std::string("1001"));          // 1010 (XOR,assign)
9     std::bitset<4> bar (std::string("0011"));          // 0010 (AND,assign)
10    std::cout << (foo^=bar) << '\n';                  // 0011 (OR,assign)
11    std::cout << (foo&=bar) << '\n';                  // 0010 (SHL,assign)
12    std::cout << (foo>>1) << '\n';                  // 0110 (SHR,assign)
13    std::cout << (~bar) << '\n';                     // 1100 (NOT)
14    std::cout << (bar<<1) << '\n';                  // 0110 (SHL)
15    std::cout << (bar>>1) << '\n';                  // 0001 (SHR)
16
17    std::cout << (foo==bar) << '\n';                // false (0110==0011)
18    std::cout << (foo!=bar) << '\n';                // true  (0110!=0011)
19
20    std::cout << (foo&bar) << '\n';                // 0010
21    std::cout << (foo|bar) << '\n';                // 0111
22    std::cout << (foo^bar) << '\n';                // 0101
23
24    return 0;
25 }

```

Output:

```

1010
0010
0011
1100
0110
1100
0110
0001
0
1
0010
0111
0101

```

## Data races

All bit position in the `bitset` objects involved in the operation are accessed, and -if in a compound assignment- also modified.

## Exception safety

The stream inserter extractors leave all objects in a valid state in case of exception (basic guarantee).  
The other operations never throw exceptions (no-throw guarantee).

## See also

<code>bitset::operator[]</code>	Access bit (public member function )
<code>bitset::set</code>	Set bits (public member function )
<code>bitset::reset</code>	Reset bits (public member function )
<code>bitset::flip</code>	Flip bits (public member function )

## /bitset/bitset/reference

public member class

### std::bitset::reference

<bitset>

`class reference;`

**Reference-like type**

This embedded class is the type returned by `bitset::operator[]` when applied to *non const-qualified* `bitset` objects. It accesses individual bits with an interface that emulates a reference to a `bool`.

Its prototype is:

```

1 class bitset::reference {
2     friend class bitset;
3     reference();                                // no public constructor
4 public:
5     ~reference();
6     operator bool() const;                    // convert to bool
7     reference& operator=(bool x);            // assign bool
8     reference& operator=(const reference& x); // assign bit
9     reference& flip();                      // flip bit value
10    bool operator~() const;                 // return inverse value
11 }

```

```

1 class bitset::reference {
2     friend class bitset;
3     reference() noexcept;                   // no public constructor

```

```

4 public:
5 ~reference();
6 operator bool() const noexcept; // convert to bool
7 reference& operator=(bool x) noexcept; // assign bool
8 reference& operator=(const reference& x) noexcept; // assign bit
9 reference& flip() noexcept; // flip bit value
10 bool operator~() const noexcept; // return inverse value
11 }

```

## /bitset/bitset/reset

public member function

### std::bitset::reset

<bitset>

<i>all bits (1)</i>	<code>bitset&amp; reset();</code>
<i>single bit (2)</i>	<code>bitset&amp; reset(size_t pos);</code>

<i>all bits (1)</i>	<code>bitset&amp; reset() noexcept;</code>
<i>single bit (2)</i>	<code>bitset&amp; reset(size_t pos);</code>

#### Reset bits

Resets bits to zero:

##### (1) all bits

Resets (to zero) all bits in the `bitset`.

##### (2) single bit

Resets (to zero) the bit at position `pos`.

---

#### Parameters

`pos`

Order position of the bit whose value is modified.

Order positions are counted from the rightmost bit, which is order position 0.

If `pos` is equal or greater than the `bitset size`, an `out_of_range` exception is thrown.

`size_t` is an unsigned integral type.

---

#### Return value

`*this`

---

#### Example

```

1 // bitset::reset
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <bitset>              // std::bitset
5
6 int main ()
7 {
8     std::bitset<4> foo (std::string("1011"));
9
10    std::cout << foo.reset(1) << '\n';      // 1001
11    std::cout << foo.reset() << '\n';      // 0000
12
13    return 0;
14 }

```

Output:

```

1001
0000

```

---

#### Data races

Both the `bitset` and its bits are modified.

---

#### Exception safety

For (1): it never throws exceptions (no-throw guarantee).

For (2): in case of exception, the object is in a valid state (basic guarantee).

If `pos` is equal or greater than the `bitset size`, the function throws an `out_of_range` exception.

---

#### See also

<code>bitset::set</code>	Set bits (public member function )
<code>bitset::flip</code>	Flip bits (public member function )

## /bitset/bitset/set

public member function

<bitset>

## std::bitset::set

```
all bits (1) bitset& set();
single bit (2) bitset& set (size_t pos, bool val = true);

all bits (1) bitset& set() noexcept;
single bit (2) bitset& set (size_t pos, bool val = true);
```

### Set bits

Sets bits:

#### (1) all bits

Sets (to one) all bits in the `bitset`.

#### (2) single bit

Sets `val` as the value for the bit at position `pos`.

### Parameters

`pos`

Order position of the bit whose value is modified.

Order positions are counted from the rightmost bit, which is order position 0.

If `pos` is equal or greater than the `bitset size`, an `out_of_range` exception is thrown.

`size_t` is an unsigned integral type.

`val`

Value to store in the bit (either `true` for one or `false` for zero).

### Return value

`*this`

### Example

```
1 // bitset::set
2 #include <iostream>           // std::cout
3 #include <bitset>            // std::bitset
4
5 int main ()
6 {
7     std::bitset<4> foo;
8
9     std::cout << foo.set() << '\n';      // 1111
10    std::cout << foo.set(2,0) << '\n';    // 1011
11    std::cout << foo.set(2) << '\n';      // 1111
12
13    return 0;
14 }
```

Output:

```
1111
1011
1111
```

### Data races

Both the `bitset` and its bits are modified.

### Exception safety

For (1): it never throws exceptions (no-throw guarantee).

For (2): in case of exception, the object is in a valid state (basic guarantee).

If `pos` is equal or greater than the `bitset size`, the function throws an `out_of_range` exception.

### See also

<code>bitset::reset</code>	Reset bits (public member function )
<code>bitset::flip</code>	Flip bits (public member function )

## /bitset/bitset/size

public member function

## std::bitset::size

<bitset>

```
size_t size() const;
constexpr size_t size() noexcept;
```

### Return size

Returns the number of bits in the `bitset`.

This is the template parameter with which the `bitset` class is instantiated (template parameter `N`).

### Parameters

none

## Return Value

The number of bits in the `bitset`.

`size_t` is an unsigned integral type.

## Example

```
1 // bitset::size
2 #include <iostream>           // std::cout
3 #include <bitset>             // std::bitset
4
5 int main ()
6 {
7     std::bitset<8> foo;
8     std::bitset<4> bar;
9
10    std::cout << "foo.size() is " << foo.size() << '\n';
11    std::cout << "bar.size() is " << bar.size() << '\n';
12
13    return 0;
14 }
```

Output:

```
foo.size() is 8
bar.size() is 4
```

## Data races

None (compile-time constant).

## Exception safety

**No-throw guarantee:** never throws exceptions.

## See also

<code>bitset::count</code>	Count bits set (public member function )
----------------------------	--

# /bitset/bitset/test

public member function

## std::bitset::test

<bitset>

`bool test (size_t pos) const;`

### Return bit value

Returns whether the bit at position `pos` is set (i.e., whether it is *one*).

Unlike the access operator (`operator[]`), this function performs a range check on `pos` before retrieving the bit value, throwing `out_of_range` if `pos` is equal or greater than the `bitset` size.

## Parameters

`pos`

Order position of the bit whose value is retrieved.

Order positions are counted from the rightmost bit, which is order position 0.

`size_t` is an unsigned integral type.

## Return value

true if the bit at position `pos` is set, and false if it is not set.

## Example

```
1 // bitset::test
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <cstdint>            // std::size_t
5 #include <bitset>              // std::bitset
6
7 int main ()
8 {
9     std::bitset<5> foo (std::string("01011"));
10
11    std::cout << "foo contains:\n";
12    std::cout << std::boolalpha;
13    for (std::size_t i=0; i<foo.size(); ++i)
14        std::cout << foo.test(i) << '\n';
15
16    return 0;
17 }
```

Output:

```
foo contains:  
true  
true  
false  
true  
false
```

## Data races

The `bitset` is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `bitset`.

If `pos` is not a valid bit position, `out_of_range` is thrown.

## See also

<code>bitset::operator[]</code>	Access bit (public member function )
<code>bitset::count</code>	Count bits set (public member function )
<code>bitset::any</code>	Test if any bit is set (public member function )
<code>bitset::none</code>	Test if no bit is set (public member function )

# /bitset/bitset/to\_string

public member function

## std::bitset::to\_string

<bitset>

```
template <class charT, class traits, class Alloc>
basic_string<charT,traits,Alloc> to_string() const;
template <class charT = char,
          class traits = char_traits<charT>,
          class Alloc = allocator<charT>>
basic_string<charT,traits,Alloc> to_string (charT zero = charT('0'),
                                              charT one = charT('1')) const;
```

### Convert to string

Constructs a `basic_string` object that represents the bits in the `bitset` as a succession of `zeros` and/or `ones`.

The string returned by this function has the same representation as the output produced by inserting the `bitset` directly into an output stream with `operator<<`.

Notice that this function template uses the template parameters to define the return type. Therefore, these are not implicitly deduced.

### Parameters

none

Constructs a `basic_string` object that represents the bits in the `bitset` as a succession of `zeros` and/or `ones`.

The string returned by this function has the same representation as the output produced by inserting the `bitset` directly into an output stream with `operator<<`.

Notice that this function template uses the template parameters to define the return type. The default types for these template parameters select `string` as the return type.

### Parameters

`zero, one`

Character values to represent `zero` and `one`.

### Return value

A string representing the bits in the `bitset`.

### Example

```
1 // bitset::to_string
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <bitset>              // std::bitset
5
6 int main ()
7 {
8     std::bitset<4> mybits;      // mybits: 0000
9     mybits.set();                // mybits: 1111
10
11    std::string mystring =
12        mybits.to_string<char,std::string::traits_type,std::string::allocator_type>();
13
14    std::cout << "mystring: " << mystring << '\n';
15
16    return 0;
17 }
```

Output:

```
mystring: 1111
```

## Data races

The `bitset` is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `bitset`.

## See also

<code>bitset::to_ulong</code>	Convert to unsigned long integer ( <a href="#">public member function</a> )
<code>bitset::bitset</code>	Construct bitset ( <a href="#">public member function</a> )

## /bitset/bitset/to\_ullong

public member function

### std::bitset::to\_ullong

<bitset>

```
unsigned long long to_ullong() const;
```

#### Convert to unsigned long long

Returns an unsigned long long with the integer value that has the same bits set as the bitset.

If the `bitset` size is too big to be represented in a value of type `unsigned long long`, the function throws an exception of type `overflow_error`.

## Parameters

none

## Return value

Integer value with the same bit representation as the `bitset` object.

## Example

```
1 // bitset::to_ullong
2 #include <iostream>           // std::cout
3 #include <bitset>            // std::bitset
4
5 int main ()
6 {
7     std::bitset<4> foo;      // foo: 0000
8     foo.set();                // foo: 1111
9
10    std::cout << foo << " as an integer is: " << foo.to_ullong() << '\n';
11
12    return 0;
13 }
```

Output:

```
1111 as an integer is: 15
```

## Data races

The `bitset` is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `bitset`.

If the `bitset` size is too big to be represented by the return type, `overflow_error` is thrown.

## See also

<code>bitset::to_ulong</code>	Convert to unsigned long integer ( <a href="#">public member function</a> )
<code>bitset::to_string</code>	Convert to string ( <a href="#">public member function</a> )
<code>bitset::bitset</code>	Construct bitset ( <a href="#">public member function</a> )

## /bitset/bitset/to\_ulong

public member function

### std::bitset::to\_ulong

<bitset>

```
unsigned long to_ulong() const;
```

#### Convert to unsigned long integer

Returns an unsigned long with the integer value that has the same bits set as the bitset.

If the `bitset` size is too big to be represented in a value of type `unsigned long`, the function throws an exception of type `overflow_error`.

## Parameters

none

## Return value

Integer value with the same bit representation as the `bitset` object.

## Example

```
1 // bitset::to_ulong
2 #include <iostream>           // std::cout
3 #include <bitset>            // std::bitset
4
5 int main ()
6 {
7     std::bitset<4> foo;      // foo: 0000
8     foo.set();                // foo: 1111
9
10    std::cout << foo << " as an integer is: " << foo.to_ulong() << '\n';
11
12    return 0;
13 }
```

Output:

```
1111 as an integer is: 15
```

## Data races

The `bitset` is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `bitset`.  
If the `bitset` size is too big to be represented by the return type, `overflow_error` is thrown.

## See also

<code>bitset::to_string</code>	Convert to string (public member function )
<code>bitset::bitset</code>	Construct bitset (public member function )

# /cctype

header

## <cctype> (ctype.h)

### Character handling functions

This header declares a set of functions to classify and transform individual characters.

## Functions

These functions take the `int` equivalent of one character as parameter and return an `int` that can either be another character or a value representing a boolean value: an `int` value of 0 means false, and an `int` value different from 0 represents true.

There are two sets of functions:

### Character classification functions

They check whether the character passed as parameter belongs to a certain category:

<code>isalnum</code>	Check if character is alphanumeric (function )
<code>isalpha</code>	Check if character is alphabetic (function )
<code>isblank</code>	Check if character is blank (function )
<code>iscntrl</code>	Check if character is a control character (function )
<code>isdigit</code>	Check if character is decimal digit (function )
<code>isgraph</code>	Check if character has graphical representation (function )
<code>islower</code>	Check if character is lowercase letter (function )
<code>isprint</code>	Check if character is printable (function )
<code>ispunct</code>	Check if character is a punctuation character (function )
<code>isspace</code>	Check if character is a white-space (function )
<code>isupper</code>	Check if character is uppercase letter (function )
<code>isxdigit</code>	Check if character is hexadecimal digit (function )

### Character conversion functions

Two functions that convert between letter cases:

<code>tolower</code>	Convert uppercase letter to lowercase (function )
<code>toupper</code>	Convert lowercase letter to uppercase (function )

For the first set, here is a map of how the original 127-character ASCII set is considered by each function (an x indicates that the function returns true on that character)

ASCII values	characters	iscntrl	isblank	isspace	isupper	islower	isalpha	isdigit	isxdigit	isalnum	ispunct	isgraph	isprint
0x00 .. 0x08	NUL, (other control codes)	x											
0x09	tab ('`t')	x	x	x									
0x0A .. 0x0D	(white-space control codes: '\f', '\v', '\n', '\r')	x		x									
0x0E .. 0x1F	(other control codes)	x											
0x20	space (' `')		x	x								x	
0x21 .. 0x2F	! "#%&`()*+-./									x	x	x	
0x30 .. 0x39	0123456789						x	x	x		x	x	
0x3a .. 0x40	:;<=>?@									x	x	x	
0x41 .. 0x46	ABCDEF				x	x		x	x		x	x	
0x47 .. 0x5A	GHIJKLMNOPQRSTUVWXYZ			x	x			x			x	x	
0x5B .. 0x60	[`\]^_									x	x	x	
0x61 .. 0x66	abcdef			x	x		x	x			x	x	
0x67 .. 0x7A	ghijklmnopqrstuvwxyz			x	x			x			x	x	
0x7B .. 0x7E	{ }~									x	x	x	
0x7F	(DEL)	x											

The characters in the extended character set (above 0x7F) may belong to diverse categories depending on the locale and the platform. As a general rule, `ispunct`, `isgraph` and `isprint` return true on these for the standard C locale on most platforms supporting extended character sets.

## /ctype/isalnum

function  
**isalnum** <cctype>

```
int isalnum ( int c );
```

### Check if character is alphanumeric

Checks whether `c` is either a decimal digit or an uppercase or lowercase letter.

The result is true if either `isalpha` or `isdigit` would also return true.

Notice that what is considered a letter may depend on the locale being used; In the default "c" locale, what constitutes a letter is what returns true by either `isupper` or `islower`.

For a detailed chart on what the different `ctype` functions return for each character of the standard ANSI character set, see the reference for the `<cctype>` header.

In C++, a locale-specific template version of this function (`isalnum`) exists in header `<locale>`.

### Parameters

`c`  
Character to be checked, casted as an `int`, or `EOF`.

### Return Value

A value different from zero (i.e., `true`) if indeed `c` is either a digit or a letter. Zero (i.e., `false`) otherwise.

### Example

```
1 /* isalnum example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     int i;
7     char str[]="c3po...";
8     i=0;
9     while (isalnum(str[i])) i++;
10    printf ("The first %d characters are alphanumeric.\n",i);
11    return 0;
12 }
```

Output:

```
The first 4 characters are alphanumeric.
```

### See also

<a href="#">isalpha</a>	Check if character is alphabetic ( <a href="#">function</a> )
<a href="#">isdigit</a>	Check if character is decimal digit ( <a href="#">function</a> )

## /ctype/isalpha

function

## isalpha

---

```
int isalpha ( int c );
```

### Check if character is alphabetic

Checks whether *c* is an alphabetic letter.

Notice that what is considered a letter depends on the locale being used; In the default "C" locale, what constitutes a letter is only what returns true by either *isupper* or *islower*.

Using other locales, an alphabetic character is a character for which *isupper* or *islower* would return true, or another character explicitly considered alphabetic by the locale (in this case, the character cannot be *iscntrl*, *isdigit*, *ispunct* or *isspace*).

For a detailed chart on what the different *ctype* functions return for each character of the standard ANSI character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function (*isalpha*) exists in header <locale>.

---

### Parameters

*c*

Character to be checked, casted to an *int*, or *EOF*.

---

### Return Value

A value different from zero (i.e., *true*) if indeed *c* is an alphabetic letter. Zero (i.e., *false*) otherwise.

---

### Example

```
1 /* isalpha example */
2 #include <stdio.h>
3 #include <cctype.h>
4 int main ()
5 {
6     int i=0;
7     char str[]="C++";
8     while (str[i])
9     {
10         if (isalpha(str[i])) printf ("character %c is alphabetic\n",str[i]);
11         else printf ("character %c is not alphabetic\n",str[i]);
12         i++;
13     }
14     return 0;
15 }
```

Output:

```
character C is alphabetic
character + is not alphabetic
character + is not alphabetic
```

---

### See also

<a href="#">isalnum</a>	Check if character is alphanumeric (function )
<a href="#">isdigit</a>	Check if character is decimal digit (function )

## /ctype/isblank

function

## isblank

&lt;cctype&gt;

---

```
int isblank ( int c );
```

### Check if character is blank

Checks whether *c* is a *blank character*.

A *blank character* is a *space character* used to separate words within a line of text.

The standard "C" locale considers blank characters the tab character ('\t') and the space character (' ').

Other locales may consider blank a different selection of characters, but they must all also be *space characters* by *isspace*.

For a detailed chart on what the different *ctype* functions return for each character of the standard ASCII character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function (*isblank*) exists in header <locale>.

**Compatibility note:** Standardized in C99 (C++11).

---

### Parameters

*c*

Character to be checked, casted to an *int*, or *EOF*.

---

### Return Value

A value different from zero (i.e., *true*) if indeed *c* is a blank character. Zero (i.e., *false*) otherwise.

## Example

```
1 /* isblank example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     char c;
7     int i=0;
8     char str[]="Example sentence to test isblank\n";
9     while (str[i])
10    {
11        c=str[i];
12        if (isblank(c)) c='\n';
13        putchar (c);
14        i++;
15    }
16    return 0;
17 }
```

This code prints out the C string character by character, replacing any blank character by a newline character. Output:

```
Example
sentence
to
test
isblank
```

## See also

<a href="#">isspace</a>	Check if character is a white-space ( <a href="#">function</a> )
<a href="#">isgraph</a>	Check if character has graphical representation ( <a href="#">function</a> )
<a href="#">ispunct</a>	Check if character is a punctuation character ( <a href="#">function</a> )
<a href="#">isalnum</a>	Check if character is alphanumeric ( <a href="#">function</a> )
<a href="#">isblank (locale)</a>	Check if character is blank using locale ( <a href="#">function template</a> )

## /ctype/iscntrl

function

### iscntrl

<cctype>

```
int iscntrl ( int c );
```

#### Check if character is a control character

Checks whether *c* is a *control character*.

A *control character* is a character that does not occupy a printing position on a display (this is the opposite of a *printable character*, checked with *isprint*).

For the standard ASCII character set (used by the "c" locale), control characters are those between ASCII codes 0x00 (NUL) and 0x1f (US), plus 0x7f (DEL).

For a detailed chart on what the different ctype functions return for each character of the standard ANSI character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function (*iscntrl*) exists in header <locale>.

## Parameters

*c*

Character to be checked, casted to an *int*, or EOF.

## Return Value

A value different from zero (i.e., true) if indeed *c* is a control character. Zero (i.e., false) otherwise.

## Example

```
1 /* iscntrl example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     int i=0;
7     char str[]="first line \n second line \n";
8     while (!iscntrl(str[i]))
9    {
10        putchar (str[i]);
11        i++;
12    }
13    return 0;
14 }
```

This code prints a string character by character until a control character that breaks the while-loop is encountered. In this case, only the first line would be printed, since the line ends with '\n', which is a control character (ASCII code 0x0a).

## See also

<a href="#">isgraph</a>	Check if character has graphical representation ( <a href="#">function</a> )
<a href="#">ispunct</a>	Check if character is a punctuation character ( <a href="#">function</a> )

## /ctype/isdigit

function

### isdigit

<cctype>

```
int isdigit ( int c );
```

#### Check if character is decimal digit

Checks whether *c* is a decimal digit character.

Decimal digits are any of: 0 1 2 3 4 5 6 7 8 9

For a detailed chart on what the different ctype functions return for each character of the standard ASCII character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function (*isdigit*) exists in header <locale>.

## Parameters

*c*

Character to be checked, casted to an *int*, or EOF.

## Return Value

A value different from zero (i.e., *true*) if indeed *c* is a decimal digit. Zero (i.e., *false*) otherwise.

## Example

```
1 /* isdigit example */
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <ctype.h>
5 int main ()
6 {
7     char str[]="1776ad";
8     int year;
9     if (isdigit(str[0]))
10    {
11        year = atoi (str);
12        printf ("The year that followed %d was %d.\n",year,year+1);
13    }
14    return 0;
15 }
```

Output

```
The year that followed 1776 was 1777
```

*isdigit* is used to check if the first character in *str* is a digit and therefore a valid candidate to be converted by *atoi* into an integer value.

## See also

<a href="#">isalnum</a>	Check if character is alphanumeric ( <a href="#">function</a> )
<a href="#">isalpha</a>	Check if character is alphabetic ( <a href="#">function</a> )

## /ctype/isgraph

function

### isgraph

<cctype>

```
int isgraph ( int c );
```

#### Check if character has graphical representation

Checks whether *c* is a character with graphical representation.

The characters with graphical representation are all those characters than can be printed (as determined by *isprint*) except the space character (' ').

For a detailed chart on what the different ctype functions return for each character of the standard ASCII character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function (*isgraph*) exists in header <locale>.

## Parameters

*c*

Character to be checked, casted to an *int*, or EOF.

## Return Value

A value different from zero (i.e., true) if indeed c has a graphical representation as character. Zero (i.e., false) otherwise.

## Example

```
1 /* isgraph example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     FILE * pFile;
7     int c;
8     pFile=fopen ("myfile.txt","r");
9     if (pFile)
10    {
11        do {
12            c = fgetc (pFile);
13            if (isgraph(c)) putchar (c);
14        } while (c != EOF);
15        fclose (pFile);
16    }
17 }
```

This example prints out the contents of "myfile.txt" without spaces and special characters, i.e. only prints out the characters that qualify as `isgraph`.

## See also

<a href="#">isprint</a>	Check if character is printable (function )
<a href="#">isspace</a>	Check if character is a white-space (function )
<a href="#">isalnum</a>	Check if character is alphanumeric (function )

## /cctype/islower

function

## islower

<cctype>

```
int islower ( int c );
```

### Check if character is lowercase letter

Checks whether c is a lowercase letter.

Notice that what is considered a letter may depend on the locale being used; In the default "c" locale, a lowercase letter is any of: a b c d e f g h i j k l m n o p q r s t u v w x y z.

Other locales may consider a different selection of characters as lowercase characters, but never characters that returns true for `iscntrl`, `isdigit`, `ispunct` or `isspace`.

For a detailed chart on what the different ctype functions return for each character of the standard ANSI character set, see the reference for the `<cctype>` header.

In C++, a locale-specific template version of this function (`islower`) exists in header `<locale>`.

## Parameters

c

Character to be checked, casted to an `int`, or `EOF`.

## Return Value

A value different from zero (i.e., true) if indeed c is a lowercase alphabetic letter. Zero (i.e., false) otherwise.

## Example

```
1 /* islower example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     int i=0;
7     char str[]="Test String.\n";
8     char c;
9     while (str[i])
10    {
11        c=str[i];
12        if (islower(c)) c=toupper(c);
13        putchar (c);
14        i++;
15    }
16    return 0;
17 }
```

Output:

```
TEST STRING.
```

## See also

<b>isupper</b>	Check if character is uppercase letter ( <a href="#">function</a> )
<b>isalpha</b>	Check if character is alphabetic ( <a href="#">function</a> )
<b>toupper</b>	Convert lowercase letter to uppercase ( <a href="#">function</a> )
<b>tolower</b>	Convert uppercase letter to lowercase ( <a href="#">function</a> )

## /cctype/isprint

function

### isprint

<cctype>

```
int isprint ( int c );
```

#### Check if character is printable

Checks whether *c* is a *printable character*.

A *printable character* is a character that occupies a printing position on a display (this is the opposite of a *control character*, checked with [iscntrl](#)).

For the standard ASCII character set (used by the "c" locale), printing characters are all with an ASCII code greater than 0x1f (US), except 0x7f (DEL).

[isgraph](#) returns true for the same cases as [isprint](#) except for the space character (' '), which returns true when checked with [isprint](#) but false when checked with [isgraph](#).

For a detailed chart on what the different ctype functions return for each character of the standard ANSI character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function ([isprint](#)) exists in header <locale>.

#### Parameters

*c*  
Character to be checked, casted to an `int`, or `EOF`.

#### Return Value

A value different from zero (i.e., `true`) if indeed *c* is a printable character. Zero (i.e., `false`) otherwise.

#### Example

```
1 /* isprint example */
2 #include <stdio.h>
3 #include <cctype.h>
4 int main ()
5 {
6     int i=0;
7     char str[]="first line \n second line \n";
8     while (isprint(str[i]))
9     {
10         putchar (str[i]);
11         i++;
12     }
13     return 0;
14 }
```

This code prints a string character by character until a character that is not printable is checked and breaks the while-loop. In this case, only the first line would be printed, since the line ends with a newline character ('\n'), which is not a printable character.

#### See also

<a href="#">iscntrl</a>	Check if character is a control character ( <a href="#">function</a> )
<a href="#">isspace</a>	Check if character is a white-space ( <a href="#">function</a> )
<a href="#">isalnum</a>	Check if character is alphanumeric ( <a href="#">function</a> )

## /cctype/ispunct

function

### ispunct

<cctype>

```
int ispunct ( int c );
```

#### Check if character is a punctuation character

Checks whether *c* is a punctuation character.

The standard "c" locale considers punctuation characters all graphic characters (as in [isgraph](#)) that are not alphanumeric (as in [isalnum](#)).

Other locales may consider a different selection of characters as punctuation characters, but in any case they are [isgraph](#) but not [isalnum](#).

For a detailed chart on what the different ctype functions return for each character of the standard ANSI character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function ([ispunct](#)) exists in header <locale>.

#### Parameters

c

Character to be checked, casted to an int, or EOF.

## Return Value

A value different from zero (i.e., true) if indeed c is a punctuation character. Zero (i.e., false) otherwise.

## Example

```
1 /* ispunct example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     int i=0;
7     int cx=0;
8     char str[]="Hello, welcome!";
9     while (str[i])
10    {
11        if (ispunct(str[i])) cx++;
12        i++;
13    }
14    printf ("Sentence contains %d punctuation characters.\n", cx);
15    return 0;
16 }
```

Output

```
Sentence contains 2 punctuation characters.
```

## See also

<a href="#">isgraph</a>	Check if character has graphical representation (function )
<a href="#">iscntrl</a>	Check if character is a control character (function )

## /ctype/isspace

function

### isspace

<cctype>

```
int isspace ( int c );
```

**Check if character is a white-space**

Checks whether c is a *white-space character*.

For the "c" locale, white-space characters are any of:

' '	(0x20)	space (SPC)
'\t'	(0x09)	horizontal tab (TAB)
'\n'	(0x0a)	newline (LF)
'\v'	(0x0b)	vertical tab (VT)
'\f'	(0x0c)	feed (FF)
'\r'	(0x0d)	carriage return (CR)

Other locales may consider a different selection of characters as white-spaces, but never a character that returns true for [isalnum](#).

For a detailed chart on what the different ctype functions return for each character of the standard ASCII character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function ([isspace](#)) exists in header <locale>.

## Parameters

c

Character to be checked, casted to an int, or EOF.

## Return Value

A value different from zero (i.e., true) if indeed c is a white-space character. Zero (i.e., false) otherwise.

## Example

```
1 /* isspace example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     char c;
7     int i=0;
8     char str[]="Example sentence to test isspace\n";
9     while (str[i])
10    {
11        c=str[i];
12        if (isspace(c)) c='\n';
13        putchar (c);
14        i++;
15    }
```

```
16 |     return 0;
17 }
```

This code prints out the C string character by character, replacing any white-space character by a newline character. Output:

```
Example
sentence
to
test
isspace
```

## See also

<a href="#">isgraph</a>	Check if character has graphical representation (function )
<a href="#">ispunct</a>	Check if character is a punctuation character (function )
<a href="#">isalnum</a>	Check if character is alphanumeric (function )
<a href="#">isspace (locale)</a>	Check if character is a white-space using locale (function template )

## /ctype/isupper

function  
**isupper** <cctype>

```
int isupper ( int c );
```

### Check if character is uppercase letter

Checks if parameter *c* is an uppercase alphabetic letter.

Notice that what is considered a letter may depend on the locale being used; In the default "c" locale, an uppercase letter is any of: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z.

Other locales may consider a different selection of characters as uppercase characters, but never characters that returns true for `iscntrl`, `isdigit`, `ispunct` or `isspace`.

For a detailed chart on what the different `ctype` functions return for each character of the standard ANSI character set, see the reference for the `<cctype>` header.

In C++, a locale-specific template version of this function (`isupper`) exists in header `<locale>`.

## Parameters

*c*  
Character to be checked, casted to an `int`, or `EOF`.

## Return Value

A value different from zero (i.e., `true`) if indeed *c* is an uppercase alphabetic letter. Zero (i.e., `false`) otherwise.

## Example

```
1 /* isupper example */
2 #include <stdio.h>
3 #include <ctype.h>
4 int main ()
5 {
6     int i=0;
7     char str[]="Test String.\n";
8     char c;
9     while (str[i])
10    {
11        c=str[i];
12        if (isupper(c)) c=tolower(c);
13        putchar (c);
14        i++;
15    }
16    return 0;
17 }
```

Output:

```
test string.
```

## See also

<a href="#">islower</a>	Check if character is lowercase letter (function )
<a href="#">isalpha</a>	Check if character is alphabetic (function )
<a href="#">toupper</a>	Convert lowercase letter to uppercase (function )
<a href="#">tolower</a>	Convert uppercase letter to lowercase (function )

## /ctype/isxdigit

function

## isxdigit

```
int isxdigit ( int c );
```

### Check if character is hexadecimal digit

Checks whether *c* is a hexdecimal digit character.

Hexadecimal digits are any of: 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

For a detailed chart on what the different ctype functions return for each character of the standard ANSI character set, see the reference for the <cctype> header.

In C++, a locale-specific template version of this function (*isxdigit*) exists in header <locale>.

### Parameters

<i>c</i>	Character to be checked, casted to an <code>int</code> , or <code>EOF</code> .
----------	--

### Return Value

A value different from zero (i.e., `true`) if indeed *c* is a hexadeciml digit. Zero (i.e., `false`) otherwise.

### Example

```
1 /* isxdigit example */
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <cctype.h>
5 int main ()
6 {
7     char str[]="fffff";
8     long int number;
9     if (isxdigit(str[0]))
10    {
11        number = strtol (str,NULL,16);
12        printf ("The hexadeciml number %lx is %ld.\n",number,number);
13    }
14    return 0;
15 }
```

*isxdigit* is used to check if the first character in *str* is a valid hexadeciml digit and therefore a valid candidate to be converted by *strtol* into an integral value.

Output:

The hexadeciml number ffff is 65535.

### See also

<a href="#">isdigit</a>	Check if character is decimal digit (function )
<a href="#">isalnum</a>	Check if character is alphanumeric (function )
<a href="#">isalpha</a>	Check if character is alphabetic (function )

## /cctype/tolower

function

## tolower

```
int tolower ( int c );
```

### Convert uppercase letter to lowercase

Converts *c* to its lowercase equivalent if *c* is an uppercase letter and has a lowercase equivalent. If no such conversion is possible, the value returned is *c* unchanged.

Notice that what is considered a letter may depend on the locale being used; In the default "c" locale, an uppercase letter is any of: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z, which translate respectively to: a b c d e f g h i j k l m n o p q r s t u v w x y z.

In other locales, if an uppercase character has more than one correspondent lowercase character, this function always returns the same character for the same value of *c*.

In C++, a locale-specific template version of this function (*tolower*) exists in header <locale>.

### Parameters

<i>c</i>	Character to be converted, casted to an <code>int</code> , or <code>EOF</code> .
----------	--

### Return Value

The lowercase equivalent to *c*, if such value exists, or *c* (unchanged) otherwise.

The value is returned as an `int` value that can be implicitly casted to `char`.

### Example

```
1 /* tolower example */
2 #include <stdio.h>
3 #include <cctype.h>
```

```

4 int main ()
5 {
6     int i=0;
7     char str[]="Test String.\n";
8     char c;
9     while (str[i])
10    {
11        c=str[i];
12        putchar (tolower(c));
13        i++;
14    }
15    return 0;
16 }
```

Output:

test string.

## See also

<a href="#">toupper</a>	Convert lowercase letter to uppercase (function )
<a href="#">isupper</a>	Check if character is uppercase letter (function )
<a href="#">islower</a>	Check if character is lowercase letter (function )
<a href="#">isalpha</a>	Check if character is alphabetic (function )

## /cctype/toupper

function

### toupper

<cctype>

`int toupper ( int c );`

#### Convert lowercase letter to uppercase

Converts *c* to its uppercase equivalent if *c* is a lowercase letter and has an uppercase equivalent. If no such conversion is possible, the value returned is *c* unchanged.

Notice that what is considered a letter may depend on the locale being used; In the default "C" locale, a lowercase letter is any of: a b c d e f g h i j k l m n o p q r s t u v w x y z, which translate respectively to: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z.

In other locales, if a lowercase character has more than one correspondent uppercase character, this function always returns the same character for the same value of *c*.

In C++, a locale-specific template version of this function (`toupper`) exists in header `<locale>`.

## Parameters

*c*

Character to be converted, casted to an `int`, or `EOF`.

## Return Value

The uppercase equivalent to *c*, if such value exists, or *c* (unchanged) otherwise. The value is returned as an `int` value that can be implicitly casted to `char`.

## Example

```

1 /* toupper example */
2 #include <stdio.h>
3 #include <cctype.h>
4 int main ()
5 {
6     int i=0;
7     char str[]="Test String.\n";
8     char c;
9     while (str[i])
10    {
11        c=str[i];
12        putchar ('toupper(c));
13        i++;
14    }
15    return 0;
16 }
```

Output:

TEST STRING.

## See also

<a href="#">tolower</a>	Convert uppercase letter to lowercase (function )
<a href="#">islower</a>	Check if character is lowercase letter (function )
<a href="#">isupper</a>	Check if character is uppercase letter (function )
<a href="#">isalpha</a>	Check if character is alphabetic (function )

# /cmath

header

## <cmath> (math.h)

### C numerics library

Header <cmath> declares a set of functions to compute common mathematical operations and transformations:

### Functions

#### Trigonometric functions

<a href="#">cos</a>	Compute cosine (function )
<a href="#">sin</a>	Compute sine (function )
<a href="#">tan</a>	Compute tangent (function )
<a href="#">acos</a>	Compute arc cosine (function )
<a href="#">asin</a>	Compute arc sine (function )
<a href="#">atan</a>	Compute arc tangent (function )
<a href="#">atan2</a>	Compute arc tangent with two parameters (function )

#### Hyperbolic functions

<a href="#">cosh</a>	Compute hyperbolic cosine (function )
<a href="#">sinh</a>	Compute hyperbolic sine (function )
<a href="#">tanh</a>	Compute hyperbolic tangent (function )
<a href="#">acosh</a>	Compute area hyperbolic cosine (function )
<a href="#">asinh</a>	Compute area hyperbolic sine (function )
<a href="#">atanh</a>	Compute area hyperbolic tangent (function )

#### Exponential and logarithmic functions

<a href="#">exp</a>	Compute exponential function (function )
<a href="#">frexp</a>	Get significand and exponent (function )
<a href="#">ldexp</a>	Generate value from significand and exponent (function )
<a href="#">log</a>	Compute natural logarithm (function )
<a href="#">log10</a>	Compute common logarithm (function )
<a href="#">modf</a>	Break into fractional and integral parts (function )
<a href="#">exp2</a>	Compute binary exponential function (function )
<a href="#">expm1</a>	Compute exponential minus one (function )
<a href="#">ilogb</a>	Integer binary logarithm (function )
<a href="#">log1p</a>	Compute logarithm plus one (function )
<a href="#">log2</a>	Compute binary logarithm (function )
<a href="#">logb</a>	Compute floating-point base logarithm (function )
<a href="#">scalbn</a>	Scale significand using floating-point base exponent (function )
<a href="#">scalbln</a>	Scale significand using floating-point base exponent (long) (function )

#### Power functions

<a href="#">pow</a>	Raise to power (function )
<a href="#">sqrt</a>	Compute square root (function )
<a href="#">cbrt</a>	Compute cubic root (function )
<a href="#">hypot</a>	Compute hypotenuse (function )

#### Error and gamma functions

<a href="#">erf</a>	Compute error function (function )
<a href="#">erfc</a>	Compute complementary error function (function )
<a href="#">tgamma</a>	Compute gamma function (function )
<a href="#">lgamma</a>	Compute log-gamma function (function )

#### Rounding and remainder functions

<a href="#">ceil</a>	Round up value (function )
<a href="#">floor</a>	Round down value (function )
<a href="#">fmod</a>	Compute remainder of division (function )
<a href="#">trunc</a>	Truncate value (function )
<a href="#">round</a>	Round to nearest (function )
<a href="#">lround</a>	Round to nearest and cast to long integer (function )
<a href="#">llround</a>	Round to nearest and cast to long long integer (function )
<a href="#">rint</a>	Round to integral value (function )
<a href="#">lrint</a>	Round and cast to long integer (function )
<a href="#">llrint</a>	Round and cast to long long integer (function )
<a href="#">nearbyint</a>	Round to nearby integral value (function )

<b>remainder</b>	Compute remainder (IEC 60559) ( <a href="#">function</a> )
<b>remquo</b>	Compute remainder and quotient ( <a href="#">function</a> )

#### Floating-point manipulation functions

<b>copysign</b>	Copy sign ( <a href="#">function</a> )
<b>nan</b>	Generate quiet NaN ( <a href="#">function</a> )
<b>nextafter</b>	Next representable value ( <a href="#">function</a> )
<b>nexttoward</b>	Next representable value toward precise value ( <a href="#">function</a> )

#### Minimum, maximum, difference functions

<b>fdim</b>	Positive difference ( <a href="#">function</a> )
<b>fmax</b>	Maximum value ( <a href="#">function</a> )
<b>fmin</b>	Minimum value ( <a href="#">function</a> )

#### Other functions

<b>fabs</b>	Compute absolute value ( <a href="#">function</a> )
<b>abs</b>	Compute absolute value ( <a href="#">function</a> )
<b>fma</b>	Multiply-add ( <a href="#">function</a> )

### Macros / Functions

These are implemented as macros in C and as functions in C++:

#### Classification macro / functions

<b>fpclassify</b>	Classify floating-point value ( <a href="#">macro/function</a> )
<b>isfinite</b>	Is finite value ( <a href="#">macro</a> )
<b>isinf</b>	Is infinity ( <a href="#">macro/function</a> )
<b>isnan</b>	Is Not-A-Number ( <a href="#">macro/function</a> )
<b>isnormal</b>	Is normal ( <a href="#">macro/function</a> )
<b>signbit</b>	Sign bit ( <a href="#">macro/function</a> )

#### Comparison macro / functions

<b>isgreater</b>	Is greater ( <a href="#">macro</a> )
<b>isgreaterequal</b>	Is greater or equal ( <a href="#">macro</a> )
<b>isless</b>	Is less ( <a href="#">macro</a> )
<b>islessequal</b>	Is less or equal ( <a href="#">macro</a> )
<b>islessgreater</b>	Is less or greater ( <a href="#">macro</a> )
<b>isunordered</b>	Is unordered ( <a href="#">macro</a> )

### Macro constants

<b>math_errhandling</b>	Error handling ( <a href="#">macro</a> )
<b>INFINITY</b>	Infinity ( <a href="#">constant</a> )
<b>NAN</b>	Not-A-Number ( <a href="#">constant</a> )
<b>HUGE_VAL</b>	Huge value ( <a href="#">constant</a> )
<b>HUGE_VALF</b>	Huge float value
<b>HUGE_VALL</b>	Huge long double value ( <a href="#">constant</a> )

This header also defines the following macro constants (since C99/C++11):

macro	type	description
<b>MATH_ERRNO</b> <b>MATH_ERREXCEPT</b>	int	Bitmask value with the possible values <code>math_errhandling</code> can take.
<b>FP_FAST_FMA</b> <b>FP_FAST_FMAF</b> <b>FP_FAST_FMAL</b>	int	Each, if defined, identifies for which type <code>fma</code> is at least as efficient as <code>x*y+z</code> .
<b>FP_INFINITE</b> <b>FP_NAN</b> <b>FP_NORMAL</b> <b>FP_SUBNORMAL</b> <b>FP_ZERO</b>	int	The possible values returned by <code>fpclassify</code> .
<b>FP_ILOGB0</b> <b>FP_ILOGBNAN</b>	int	Special values the <code>ilogb</code> function may return.

### Types

<b>double_t</b>	Floating-point type ( <a href="#">type</a> )
<b>float_t</b>	Floating-point type ( <a href="#">type</a> )

## /cmath/abs

function

**abs**

<cmath> <ctgmath>

```

double abs (double x);
float abs (float x);
long double abs (long double x);
double abs (double x);
float abs (float x);
long double abs (long double x);
double abs (T x);           // additional overloads for integral types

```

### Compute absolute value

Returns the *absolute value* of  $x$ :  $|x|$ .

These convenience `abs` overloads are exclusive of C++. In C, `abs` is only declared in `<stdlib.h>` (and operates on `int` values).

Since C++11, *additional overloads* are provided in this header (`<cmath>`) for the *integral types*: These overloads effectively cast  $x$  to a `double` before calculations (defined for  $T$  being any *integral type*).

### Parameters

$x$  Value whose absolute value is returned.

### Return Value

The absolute value of  $x$ .

### Example

```

1 // cmath's abs example
2 #include <iostream>          // std::cout
3 #include <cmath>              // std::abs
4
5 int main ()
6 {
7     std::cout << "abs (3.1416) = " << std::abs (3.1416) << '\n';
8     std::cout << "abs (-10.6) = " << std::abs (-10.6) << '\n';
9     return 0;
10 }

```

Output:

```

abs (3.1416) = 3.1416
abs (-10.6) = 10.6

```

### See also

<a href="#">abs (cstdlib)</a>	Absolute value (function )
<a href="#">fabs</a>	Compute absolute value (function )
<a href="#">labs</a>	Absolute value (function )

## /cmath/acos

function

### acos

`<cmath> <ctgmath>`

```

double acos (double x);
double acos (double x);
float acosf (float x);
long double acosl (long double x);
double acos (double x);
float acos (float x);
long double acos (long double x);
double acos (double x);
float acos (float x);
long double acos (long double x);
double acos (T x);           // additional overloads for integral types

```

### Compute arc cosine

Returns the principal value of the arc cosine of  $x$ , expressed in radians.

In trigonometrics, *arc cosine* is the inverse operation of *cosine*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<valarray>` (see `valarray acos`).

Additional overloads are provided in this header (`<cmath>`) for the *integral types*: These overloads effectively cast  $x$  to a `double` before calculations (defined for  $T$  being any *integral type*).

This function is also overloaded in `<complex>` and `<valarray>` (see `complex acos` and `valarray acos`).

### Parameters

$x$

Value whose arc cosine is computed, in the interval  $[-1, +1]$ . If the argument is out of this interval, a *domain error* occurs.

## Return Value

Principal arc cosine of  $x$ , in the interval  $[0,\pi]$  radians.  
One radian is equivalent to  $180/\pi$  degrees.

If a domain error occurs, the global variable `errno` is set to `EDOM`.

If a domain error occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

## Example

```
1 /* acos example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* acos */
4
5 #define PI 3.14159265
6
7 int main ()
8 {
9     double param, result;
10    param = 0.5;
11    result = acos (param) * 180.0 / PI;
12    printf ("The arc cosine of %f is %f degrees.\n", param, result);
13    return 0;
14 }
```

Output:

```
The arc cosine of 0.500000 is 60.000000 degrees.
```

## See also

<a href="#">cos</a>	Compute cosine (function )
<a href="#">asin</a>	Compute arc sine (function )

## /cmath/acosh

function

### acosh

<cmath> <ctgmath>

```
double acosh (double x);
float acoshf (float x);
long double acoshl (long double x);

double acosh (double x);
float acosh (float x);
long double acosh (long double x);
double acosh (T x);           // additional overloads for integral types
```

#### Compute area hyperbolic cosine

Returns the nonnegative area hyperbolic cosine of  $x$ .

The area hyperbolic cosine is the inverse operation of the [hyperbolic cosine](#).

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).

This function is also overloaded in `<complex>` (see [complex acosh](#)).

## Parameters

$x$

Value whose area hyperbolic cosine is computed.  
If the argument is less than 1, a [domain error](#) occurs.

## Return Value

Nonnegative area hyperbolic cosine of  $x$ , in the interval  $[0,+\infty]$ .

Note that the negative of this value is also a valid area hyperbolic cosine of  $x$

If a domain error occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

## Example

```
1 /* acosh example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* acosh, exp, sinh */
4
5 int main ()
6 {
7     double param, result;
8     param = exp(2) - sinh(2);
```

```

9 result = acosh(param) ;
10 printf ("The area hyperbolic cosine of %f is %f radians.\n", param, result);
11 return 0;
12 }
```

Output:

The area hyperbolic cosine of 3.762196 is 2.000000 radians.

### See also

<a href="#">cosh</a>	Compute hyperbolic cosine (function )
<a href="#">asinh</a>	Compute area hyperbolic sine (function )

## /cmath/asin

function

### asin

<cmath> <ctgmath>

```

double asin(double x);
    double asin (double x);
    float asinf (float x);
long double asinl (long double x);

    double asin (double x);
    float asin (float x);
long double asin (long double x);

    double asin (double x);
    float asin (float x);
long double asin (long double x);
    double asin (T x);           // additional overloads for integral types
```

#### Compute arc sine

Returns the principal value of the arc sine of  $x$ , expressed in radians.

In trigonometrics, *arc sine* is the inverse operation of *sine*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<valarray>` (see [valarray asin](#)).

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a `double` before calculations (defined for  $T$  being any [integral type](#)).

This function is also overloaded in `<complex>` and `<valarray>` (see [complex asin](#) and [valarray asin](#)).

### Parameters

$x$

Value whose arc sine is computed, in the interval  $[-1, +1]$ .  
If the argument is out of this interval, a *domain error* occurs.

### Return Value

Principal arc sine of  $x$ , in the interval  $[-\pi/2, +\pi/2]$  radians.  
One *radian* is equivalent to  $180/\pi$  *degrees*.

If a *domain error* occurs, the global variable `errno` is set to `EDOM`.

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

### Example

```

1 /* asin example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* asin */
4
5 #define PI 3.14159265
6
7 int main ()
8 {
9     double param, result;
10    param = 0.5;
11    result = asin (param) * 180.0 / PI;
12    printf ("The arc sine of %f is %f degrees\n", param, result);
13    return 0;
14 }
```

Output:

The arc sine of 0.500000 is 30.000000 degrees.

### See also

<a href="#">sin</a>	Compute sine (function )
---------------------	--------------------------

<a href="#">acos</a>	Compute arc cosine (function )
----------------------	--------------------------------

## /cmath/asinh

function  
**asinh** <cmath> <ctgmath>

```
double asinh (double x);
float asinhf (float x);
long double asinhl (long double x);

double asinh (double x);
float asinh (float x);
long double asinh (long double x);
double asinh (T x);           // additional overloads for integral types
```

### Compute area hyperbolic sine

Returns the area hyperbolic sine of  $x$ .

The *area hyperbolic sine* is the inverse operation of the *hyperbolic sine*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the **integral types**: These overloads effectively cast  $x$  to a **double** before calculations (defined for  $T$  being any *integral type*).

This function is also overloaded in `<complex>` (see **complex asinh**).

---

### Parameters

**x** Value whose area hyperbolic sine is computed.

---

### Return Value

Area hyperbolic sine of  $x$ .

---

### Example

```
1 /* asinh example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* asinh, exp, cosh */
4
5 int main ()
6 {
7     double param, result;
8     param = exp(2) - cosh(2);
9     result = asinh(param) ;
10    printf ("The area hyperbolic sine of %f is %f.\n", param, result);
11    return 0;
12 }
```

Output:

```
The area hyperbolic sine of 3.626860 is 2.000000.
```

---

### See also

<a href="#">sinh</a>	Compute hyperbolic sine (function )
<a href="#">acosh</a>	Compute area hyperbolic cosine (function )

## /cmath/atan

function  
**atan** <cmath> <ctgmath>

```
double atan(double x);
double atan (double x);
float atanf (float x);
long double atanl (long double x);

double atan (double x);
float atan (float x);
long double atan (long double x);

double atan (double x);
float atan (float x);
long double atan (long double x);
double atan (T x);           // additional overloads for integral types
```

### Compute arc tangent

Returns the principal value of the arc tangent of  $x$ , expressed in radians.

In trigonometrics, *arc tangent* is the inverse operation of *tangent*.

Notice that because of the sign ambiguity, the function cannot determine with certainty in which quadrant the angle falls only by its tangent value. See [atan2](#) for

an alternative that takes a fractional argument instead.

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<valarray>` (see `valarray atan`).

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast `x` to a double before calculations (defined for `T` being any `integral type`).

This function is also overloaded in `<complex>` and `<valarray>` (see `complex atan` and `valarray atan`).

## Parameters

x

Value whose arc tangent is computed.

## Return Value

Principal arc tangent of `x`, in the interval  $[-\pi/2, +\pi/2]$  radians.

One radian is equivalent to  $180/\pi$  degrees.

## Example

```
1 /* atan example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* atan */
4
5 #define PI 3.14159265
6
7 int main ()
8 {
9     double param, result;
10    param = 1.0;
11    result = atan (param) * 180 / PI;
12    printf ("The arc tangent of %f is %f degrees\n", param, result );
13    return 0;
14 }
```

Output:

```
The arc tangent of 1.000000 is 45.000000 degrees.
```

## See also

<a href="#">atan2</a>	Compute arc tangent with two parameters (function )
<a href="#">tan</a>	Compute tangent (function )
<a href="#">sin</a>	Compute sine (function )
<a href="#">cos</a>	Compute cosine (function )

## /cmath/atan2

function

### atan2

`<cmath> <ctgmath>`

```
double atan2(double y, double x);
double atan2 (double y      , double x);
float atan2f (float y      , float x);
long double atan2l (long double y, long double x);

double atan2 (double y      , double x);
float atan2 (float y      , float x);
long double atan2 (long double y, long double x);

double atan2 (double y      , double x);
float atan2 (float y      , float x);
long double atan2 (long double y, long double x);
double atan2 (Type1 y      , Type2 x); // additional overloads
```

#### Compute arc tangent with two parameters

Returns the principal value of the arc tangent of  $y/x$ , expressed in radians.

To compute the value, the function takes into account the sign of both arguments in order to determine the quadrant.

In C++, this function is overloaded in `<valarray>` (see `valarray atan2`).

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<valarray>` (see `valarray atan2`).

Additional overloads are provided in this header (`<cmath>`) for other combinations of `arithmetic types` (`Type1` and `Type2`): These overloads effectively cast its arguments to double before calculations, except if at least one of the arguments is of type `long double` (in which case both are casted to `long double` instead).

This function is also overloaded in `<valarray>` (see `valarray atan2`).

## Parameters

y

Value representing the proportion of the y-coordinate.

x Value representing the proportion of the x-coordinate.

If both arguments passed are zero, a *domain error* occurs.

### Return Value

Principal arc tangent of  $y/x$ , in the interval  $[-\pi, +\pi]$  radians.  
One radian is equivalent to  $180/\pi$  degrees.

If a *domain error* occurs, the global variable `errno` is set to `EDOM`.

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

### Example

```
1 /* atan2 example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* atan2 */
4
5 #define PI 3.14159265
6
7 int main ()
8 {
9     double x, y, result;
10    x = -10.0;
11    y = 10.0;
12    result = atan2 (y,x) * 180 / PI;
13    printf ("The arc tangent for (x=%f, y=%f) is %f degrees\n", x, y, result );
14    return 0;
15 }
```

Output:

```
The arc tangent for (x=-10.000000, y=10.000000) is 135.000000 degrees.
```

### See also

<a href="#">atan</a>	Compute arc tangent (function )
<a href="#">tan</a>	Compute tangent (function )
<a href="#">sin</a>	Compute sine (function )
<a href="#">cos</a>	Compute cosine (function )

## /cmath/atanh

function

### atanh

`<cmath> <ctgmath>`

```
double atanh (double x);
float atanhf (float x);
long double atanhl (long double x);

double atanh (double x);
float atanh (float x);
long double atanh (long double x);
double asinh (T x);           // additional overloads for integral types
```

#### Compute area hyperbolic tangent

Returns the area hyperbolic tangent of x.

The *area hyperbolic tangent* is the inverse operation of the *hyperbolic tangent*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast x to a `double` before calculations (defined for T being any *integral type*).

This function is also overloaded in `<complex>` (see `complex atanh`).

### Parameters

x

Value whose area hyperbolic tangent is computed, in the interval  $[-1, +1]$ .  
If the argument is out of this interval, a *domain error* occurs.  
For values of  $-1$  and  $+1$ , a *pole error* may occur.

### Return Value

Area hyperbolic tangent of x.

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If a *pole error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

## Example

```
1 /* atanh example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* tanh, atanh */
4
5 int main ()
6 {
7     double param, result;
8     param = tanh(1);
9     result = atanh(param) ;
10    printf ("The area hyperbolic tangent of %f is %f.\n", param, result);
11    return 0;
12 }
```

Output:

```
The area hyperbolic tangent of 0.761594 is 1.000000.
```

## See also

<a href="#">tanh</a>	Compute hyperbolic tangent ( <a href="#">function</a> )
<a href="#">asinh</a>	Compute area hyperbolic sine ( <a href="#">function</a> )
<a href="#">acosh</a>	Compute area hyperbolic cosine ( <a href="#">function</a> )

## /cmath/cbrt

function

### cbrt

`<cmath> <ctgmath>`

```
double cbrt (double x);
float cbrtf (float x);
long double cbctl (long double x);

double cbrt (double x);
float cbrt (float x);
long double cbtrt (long double x);
double cbrt (T x);           // additional overloads for integral types
```

#### Compute cubic root

Returns the *cubic root* of *x*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast *x* to a `double` before calculations (defined for *T* being any [integral type](#)).

## Parameters

*x*

Value whose cubit root is computed.

## Return Value

Cubic root of *x*.

## Example

```
1 /* cbrt example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* cbrt */
4
5 int main ()
6 {
7     double param, result;
8     param = 27.0;
9     result = cbrt (param);
10    printf ("cbrt (%f) = %f\n", param, result);
11    return 0;
12 }
```

Output:

```
cbrt (27.000000) = 3.000000
```

## See also

<a href="#">sqrt</a>	Compute square root ( <a href="#">function</a> )
<a href="#">pow</a>	Raise to power ( <a href="#">function</a> )

## /cmath/ceil

function

**ceil** <cmath> <ctgmath>

```
double ceil (double x);
    double ceil (double x);
    float ceilkf (float x);
long double ceill (long double x);

    double ceil (double x);
    float ceil (float x);
long double ceil (long double x);

    double ceil (double x);
    float ceil (float x);
long double ceil (long double x);
    double ceil (T x);           // additional overloads for integral types
```

### Round up value

Rounds  $x$  upward, returning the smallest integral value that is not less than  $x$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the **integral types**: These overloads effectively cast  $x$  to a **double** before calculations (defined for  $T$  being any *integral type*).

### Parameters

$x$

Value to round up.

### Return Value

The smallest integral value that is not less than  $x$  (as a floating-point value).

### Example

```
1 /* ceil example */
2 #include <stdio.h>      /* printf */
3 #include <cmath.h>        /* ceil */
4
5 int main ()
6 {
7     printf ( "ceil of 2.3 is %.1f\n", ceil(2.3) );
8     printf ( "ceil of 3.8 is %.1f\n", ceil(3.8) );
9     printf ( "ceil of -2.3 is %.1f\n", ceil(-2.3) );
10    printf ( "ceil of -3.8 is %.1f\n", ceil(-3.8) );
11    return 0;
12 }
```

Output:

```
ceil of 2.3 is 3.0
ceil of 3.8 is 4.0
ceil of -2.3 is -2.0
ceil of -3.8 is -3.0
```

### See also

<b>floor</b>	Round down value (function )
<b>fabs</b>	Compute absolute value (function )
<b>modf</b>	Break into fractional and integral parts (function )

## /cmath/copysign

function

**copysign** <cmath> <ctgmath>

```
double copysign (double x      , double y);
float copysignf (float x      , float y);
long double copysignl (long double x, long double y);

double copysign (double x      , double y);
float copysign (float x      , float y);
long double copysign (long double x, long double y);
double copysign (Type1 x      , Type2 y);           // additional overloads
```

### Copy sign

Returns a value with the magnitude of  $x$  and the sign of  $y$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for other combinations of **arithmetic types** ( $Type1$  and  $Type2$ ): These overloads effectively cast its arguments to **double** before calculations, except if at least one of the arguments is of type **long double** (in which case both are casted to **long double** instead).

## Parameters

x Value with the magnitude of the resulting value.

y Value with the sign of the resulting value.

## Return Value

The value with a magnitude of x and the sign of y.

## Example

```
1 /* copysign example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* copysign */
4
5 int main ()
6 {
7     printf ("copysign ( 10.0,-1.0) = %f\n", copysign( 10.0,-1.0));
8     printf ("copysign (-10.0,-1.0) = %f\n", copysign(-10.0,-1.0));
9     printf ("copysign (-10.0, 1.0) = %f\n", copysign(-10.0, 1.0));
10
11    return 0;
12 }
```

Output:

```
copysign ( 10.0,-1.0) = -10.0
copysign (-10.0,-1.0) = -10.0
copysign (-10.0, 1.0) = 10.0
```

## See also

[fabs](#) | Compute absolute value (function )

# /cmath/cos

function

## COS

<cmath> <ctgmath>

```
double cos (double x);
double cos (double x);
float cosf (float x);
long double cosl (long double x);

double cos (double x);
float cos (float x);
long double cos (long double x);

double cos (double x);
float cos (float x);
long double cos (long double x);
double cos (T x);           // additional overloads for integral types
```

### Compute cosine

Returns the cosine of an angle of x radians.

Header [`<tgmath.h>`](#) provides a type-generic macro version of this function.

This function is overloaded in [`<complex>`](#) and [`<valarray>`](#) (see [complex cos](#) and [valarray cos](#)).

Additional overloads are provided in this header ([`<cmath>`](#)) for the integral types: These overloads effectively cast x to a double before calculations (defined for T being any [integral type](#)).

This function is also overloaded in [`<complex>`](#) and [`<valarray>`](#) (see [complex cos](#) and [valarray cos](#)).

## Parameters

x Value representing an angle expressed in radians.  
One radian is equivalent to  $180/\pi$  degrees.

## Return Value

Cosine of x radians.

## Example

```
1 /* cos example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* cos */
4
5 #define PI 3.14159265
6
7 int main ()
```

```

8 {
9   double param, result;
10  param = 60.0;
11  result = cos ( param * PI / 180.0 );
12  printf ("The cosine of %f degrees is %f.\n", param, result );
13  return 0;
14 }

```

Output:

The cosine of 60.000000 degrees is 0.500000.

## See also

<a href="#">sin</a>	Compute sine (function )
<a href="#">tan</a>	Compute tangent (function )

## /cmath/cosh

function

### cosh

<cmath> <ctgmath>

```

double cosh (double x);
    double cosh (double x);
    float coshf (float x);
long double coshl (long double x);

double cosh (double x);
    float cosh (float x);
long double cosh (long double x);

double cosh (double x);
    float cosh (float x);
long double cosh (long double x);
    double cosh (T x);           // additional overloads for integral types

```

#### Compute hyperbolic cosine

Returns the hyperbolic cosine of *x*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see [complex cosh](#) and [valarray cosh](#)).

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast *x* to a double before calculations (defined for *T* being any [integral type](#)).

This function is also overloaded in `<complex>` and `<valarray>` (see [complex cosh](#) and [valarray cosh](#)).

## Parameters

*x*

Value representing a hyperbolic angle.

## Return Value

Hyperbolic cosine of *x*.

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* occurs:

If an overflow *range error* occurs, the global variable `errno` is set to `ERANGE`.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```

1 /* cosh example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* cosh, log */
4
5 int main ()
6 {
7   double param, result;
8   param = log(2.0);
9   result = cosh (param);
10  printf ("The hyperbolic cosine of %f is %f.\n", param, result );
11  return 0;
12 }

```

Output:

The hyperbolic cosine of 0.693147 is 1.250000.

## See also

<a href="#">sinh</a>	Compute hyperbolic sine (function )
----------------------	-------------------------------------

**tanh**

Compute hyperbolic tangent (function )

## /cmath/double\_t

type

**double\_t**

&lt;cmath&gt; &lt;ctgmath&gt;

**Floating-point type**

Alias of one of the fundamental floating-point types at least as wide as double.

It is the type used by the implementation to evaluate values of type double, as determined by [FLT\\_EVAL\\_METHOD](#):

<b>FLT_EVAL_METHOD</b>	<b>float_t</b>	<b>double_t</b>
0	float	double
1	double	double
2	long double	long double
other	implementation-defined	implementation-defined

## /cmath/erf

function

**erf**

&lt;cmath&gt; &lt;ctgmath&gt;

```
double erf (double x);
float erff (float x);
long double erfl (long double x);

double erf (double x);
float erf (float x);
long double erf (long double x);
double erf (T x);           // additional overloads for integral types
```

**Compute error function**

Returns the error function value for x.

Header [<tgmath.h>](#) provides a type-generic macro version of this function.Additional overloads are provided in this header ([<cmath>](#)) for the integral types: These overloads effectively cast x to a double before calculations (defined for T being any [integral type](#)).**Parameters**

x

Parameter for the error function.

**Return Value**

Error function value for x.

**Example**

```
1 /* erf example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* erf */
4
5 int main ()
6 {
7     double param, result;
8     param = 1.0;
9     result = erf (param);
10    printf ("erf (%f) = %f\n", param, result );
11    return 0;
12 }
```

Output:

erf (1.000000) = 0.842701

**See also**

<b>erfc</b>	Compute complementary error function (function )
<b>lgamma</b>	Compute log-gamma function (function )
<b>tgamma</b>	Compute gamma function (function )

## /cmath/erfc

function

**erfc**

&lt;cmath&gt; &lt;ctgmath&gt;

```

double erfc (double x);
float erfcf (float x);
long double erfc1 (long double x);

double erfc (double x);
float erfc (float x);
long double erfc (long double x);
double erfc (T x); // additional overloads for integral types

```

### Compute complementary error function

Returns the *complementary error function* value for  $x$ .

The *complementary error function* is equivalent to:

$$\text{erfc}(x) = 1 - \text{erf}(x)$$

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for the **integral types**: These overloads effectively cast  $x$  to a **double** before calculations (defined for  $T$  being any *integral type*).

### Parameters

$x$

Parameter for the *complementary error function*.

### Return Value

Complementary error function value for  $x$ .

If  $x$  is too large, an underflow *range error* occurs.

If an underflow *range error* occurs:  
- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.  
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_UNDERFLOW` is raised.



### Example

```

1 /* erfc example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* erfc */
4
5 int main ()
6 {
7     double param, result;
8     param = 1.0;
9     result = erfc (param);
10    printf ("erfc(%f) = %f\n", param, result );
11    return 0;
12 }

```

Output:

```
erfc (1.000000) = 0.157299
```

### See also

<a href="#">erf</a>	Compute error function (function )
<a href="#">lgamma</a>	Compute log-gamma function (function )
<a href="#">tgamma</a>	Compute gamma function (function )

## /cmath/exp

function

### exp

`<cmath> <ctgmath>`

```

double exp (double x);
double exp (double x);
float expf (float x);
long double expl (long double x);

double exp (double x);
float exp (float x);
long double exp (long double x);

double exp (double x);
float exp (float x);
long double exp (long double x);
double exp (T x); // additional overloads for integral types

```

### Compute exponential function

Returns the *base-e* exponential function of  $x$ , which is  $e$  raised to the power  $x$ :  $e^x$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see `complex exp` and `valarray exp`).

*Additional overloads* are provided in this header (`<cmath>`) for the **integral types**: These overloads effectively cast  $x$  to a **double** before calculations.

This function is also overloaded in `<complex>` and `<valarray>` (see `complex exp` and `valarray exp`).

### Parameters

x

Value of the exponent.

## Return Value

Exponential value of  $x$ .

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* occurs:

If an overflow *range error* occurs, the global variable `errno` is set to `ERANGE`.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* exp example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* exp */
4
5 int main ()
6 {
7     double param, result;
8     param = 5.0;
9     result = exp (param);
10    printf ("The exponential value of %f is %f.\n", param, result );
11    return 0;
12 }
```

Output:

```
The exponential value of 5.000000 is 148.413159.
```

## See also

<a href="#">log</a>	Compute natural logarithm (function)
<a href="#">pow</a>	Raise to power (function)

## /cmath/exp2

function

### exp2

`<cmath>` `<ctgmath>`

```
double exp2 (double x);
float exp2f (float x);
long double exp2l (long double x);

double exp2 (double x);
float exp2 (float x);
long double exp2 (long double x);
double exp2 (T x);           // additional overloads for integral types
```

#### Compute binary exponential function

Returns the base-2 exponential function of  $x$ , which is 2 raised to the power  $x$ :  $2^x$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any *integral type*).

## Parameters

x

Value of the exponent.

## Return Value

$2$  raised to the power of  $x$ .

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* occurs:

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* exp2 example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* exp2 */
4
5 int main ()
6 {
7     double param, result;
8     param = 8.0;
9     result = exp2 (param);
```

```

10 printf ("2 ^ %f = %f.\n", param, result );
11 return 0;
12 }
```

Output:

```
2 ^ 8.000000 is 256.000000.
```

### See also

<a href="#">log2</a>	Compute binary logarithm (function )
<a href="#">pow</a>	Raise to power (function )
<a href="#">exp</a>	Compute exponential function (function )

## /cmath/expm1

function

### expm1

<cmath> <ctgmath>

```

double expm1 (double x);
float expm1f (float x);
long double expm1l (long double x);

double expm1 (double x);
float expm1 (float x);
long double expm1 (long double x);
double expm1 (T x);           // additional overloads for integral types
```

#### Compute exponential minus one

Returns e raised to the power x minus one:  $e^x - 1$ .

For small magnitude values of x, `expm1` may be more accurate than `exp(x)-1`.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast x to a double before calculations (defined for T being any [integral type](#)).

### Parameters

x

Value of the exponent.

### Return Value

e raised to the power of x, minus one.

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* occurs:

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

### Example

```

1 /* expm1 example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* expm1 */
4
5 int main ()
6 {
7     double param, result;
8     param = 1.0;
9     result = expm1 (param);
10    printf ("expm1 (%f) = %f.\n", param, result );
11    return 0;
12 }
```

Output:

```
expm1 (1.000000) = 1.718282.
```

### See also

<a href="#">exp</a>	Compute exponential function (function )
<a href="#">log1p</a>	Compute logarithm plus one (function )
<a href="#">pow</a>	Raise to power (function )

## /cmath/fabs

function

## fabs

&lt;cmath&gt; &lt;ctgmath&gt;

```
double fabs (double x);
    double fabs (double x);
    float fabsf (float x);
long double fabsl (long double x);
    double fabs (double x);
    float fabs (float x);
long double fabs (long double x);
    double fabs (T x);           // additional overloads for integral types
```

### Compute absolute value

Returns the *absolute value* of  $x$ :  $|x|$ .

Header <[tgmath.h](#)> provides a type-generic macro version of this function.

Additional overloads are provided in this header (<[cmath](#)>) for the **integral types**: These overloads effectively cast  $x$  to a **double** (defined for  $T$  being any **integral type**).

### Parameters

x

Value whose absolute value is returned.

### Return Value

The absolute value of  $x$ .

### Example

```
1 /* fabs example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* fabs */
4
5 int main ()
6 {
7     printf ("The absolute value of 3.1416 is %f\n", fabs (3.1416) );
8     printf ("The absolute value of -10.6 is %f\n", fabs (-10.6) );
9     return 0;
10 }
```

Output:

```
The absolute value of 3.1416 is 3.141600
The absolute value of -10.6 is 10.600000
```

### See also

<a href="#">abs</a>	Absolute value (function)
<a href="#">labs</a>	Absolute value (function )
<a href="#">floor</a>	Round down value (function )
<a href="#">ceil</a>	Round up value (function )
<a href="#">modf</a>	Break into fractional and integral parts (function )

## /cmath/fdim

function

## fdim

&lt;cmath&gt; &lt;ctgmath&gt;

```
double fdim (double x      , double y);
float fdimf (float x       , float y);
long double fdiml (long double x, long double y);

double fdim (double x      , double y);
float fdim (float x       , float y);
long double fdim (long double x, long double y);
double fdim (Type1 x      , Type2 y);           // additional overloads
```

### Positive difference

Returns the *positive difference* between  $x$  and  $y$ .

The function returns  $x-y$  if  $x>y$ , and zero otherwise.

Header <[tgmath.h](#)> provides a type-generic macro version of this function.

Additional overloads are provided in this header (<[cmath](#)>) for other combinations of **arithmetic types** ( $Type1$  and  $Type2$ ): These overloads effectively cast its arguments to **double** before calculations, except if at least one of the arguments is of type **long double** (in which case both are casted to **long double** instead).

### Parameters

x, y

Values whose difference is calculated.

## Return Value

The positive difference between  $x$  and  $y$ .

## Example

```
1 /* fdim example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* fdim */
4
5 int main ()
6 {
7     printf ("fdim (2.0, 1.0) = %f\n", fdim(2.0,1.0));
8     printf ("fdim (1.0, 2.0) = %f\n", fdim(1.0,2.0));
9     printf ("fdim (-2.0, -1.0) = %f\n", fdim(-2.0,-1.0));
10    printf ("fdim (-1.0, -2.0) = %f\n", fdim(-1.0,-2.0));
11    return 0;
12 }
```

Output:

```
fdim (2.0, 1.0) = 1.000000
fdim (1.0, 2.0) = 0.000000
fdim (-2.0,-1.0) = 0.000000
fdim (-1.0,-2.0) = 1.000000
```

## See also

<a href="#">fmax</a>	Maximum value (function)
<a href="#">fmin</a>	Minimum value (function)

## /cmath/float\_t

type

## float\_t

<cmath> <ctgmath>

### Floating-point type

Alias of one of the fundamental floating-point types at least as wide as `float`.

It is the type used by the implementation to evaluate values of type `float`, as determined by `FLT_EVAL_METHOD`:

FLT_EVAL_METHOD	float_t	double_t
0	<code>float</code>	<code>double</code>
1	<code>double</code>	<code>double</code>
2	<code>long double</code>	<code>long double</code>
other	<i>implementation-defined</i>	<i>implementation-defined</i>

## /cmath/floor

function

## floor

<cmath> <ctgmath>

```
double floor (double x);
double floor (double x);
float floorf (float x);
long double floorl (long double x);

double floor (double x);
float floor (float x);
long double floor (long double x);

double floor (double x);
float floor (float x);
long double floor (long double x);
double floor (T x);           // additional overloads for integral types
```

### Round down value

Rounds  $x$  downward, returning the largest integral value that is not greater than  $x$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a `double` before calculations (defined for  $T$  being any *integral type*).

## Parameters

$x$

Value to round down.

## Return Value

The value of  $x$  rounded downward (as a floating-point value).

## Example

```
1 /* floor example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* floor */
4
5 int main ()
6 {
7     printf ( "floor of 2.3 is %.1lf\n", floor (2.3) );
8     printf ( "floor of 3.8 is %.1lf\n", floor (3.8) );
9     printf ( "floor of -2.3 is %.1lf\n", floor (-2.3) );
10    printf ( "floor of -3.8 is %.1lf\n", floor (-3.8) );
11    return 0;
12 }
```

Output:

```
floor of 2.3 is 2.0
floor of 3.8 is 3.0
floor of -2.3 is -3.0
floor of -3.8 is -4.0
```

## See also

<a href="#">ceil</a>	Round up value (function)
<a href="#">fabs</a>	Compute absolute value (function)
<a href="#">modf</a>	Break into fractional and integral parts (function)

## /cmath/fma

function  
**fma** <cmath> <ctgmath>

```
double fma (double x      , double y      , double z);
float fmaf (float x       , float y       , float z);
long double fmal (long double x, long double y, long double z);

double fma (double x      , double y      , double z);
float fma (float x       , float y       , float z);
long double fma (long double x, long double y, long double z);
double fma (Type1 x       , Type2 y       , Type3 z);           // additional overloads
```

### Multiply-add

Returns  $x*y+z$ .

The function computes the result without losing precision in any intermediate result.

The following macro constants may be defined in an implementation to signal that this function generally provides an efficiency improvement over performing the arithmetic operations in  $x*y+z$  (such as when a hardware multiply-add instruction is used):

macro	description
FP_FAST_FMA	For arguments of type <code>double</code> , it generally executes about as fast as, or faster than, $x*y+z$ .
FP_FAST_FMAF	For arguments of type <code>float</code> , it generally executes about as fast as, or faster than, $x*y+z$ .
FP_FAST_FMAL	For arguments of type <code>long double</code> , it generally executes about as fast as, or faster than, $x*y+z$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for other combinations of arithmetic types (`Type1`, `Type2` and `Type3`): These overloads effectively cast its arguments to `double` before calculations, except if at least one of the arguments is of type `long double` (in which case all of them are casted to `long double` instead).

## Parameters

`x, y` Values to be multiplied.

`z` Value to be added.

## Return Value

The result of  $x*y+z$

## Example

```
1 /* fma example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* fma, FP_FAST_FMA */
4
5 int main ()
6 {
7     double x,y,z,result;
8     x = 10.0, y = 20.0, z = 30.0;
9
10 #ifdef FP_FAST_FMA
```

```

11     result = fma(x,y,z);
12 #else
13     result = x*y+z;
14 #endif
15
16     printf ("10.0 * 20.0 + 30.0 = %f\n", result);
17     return 0;
18 }
```

Output:

```
10.0 * 20.0 + 30.0 = 230.000000
```

### See also

<a href="#">fmin</a>	Minimum value (function )
<a href="#">fdim</a>	Positive difference (function )

## /cmath/fmax

function

### fmax

<cmath> <ctgmath>

```

double fmax (double x      , double y);
float fmaxf (float x       , float y);
long double fmaxl (long double x, long double y);

double fmax (double x      , double y);
float fmax (float x       , float y);
long double fmax (long double x, long double y);
double fmax (Type1 x      , Type2 y);           // additional overloads
```

#### Maximum value

Returns the larger of its arguments: either *x* or *y*.

If one of the arguments in a *NaN*, the other is returned.

Header [`<tgmath.h>`](#) provides a type-generic macro version of this function.

Additional overloads are provided in this header ([`<cmath>`](#)) for other combinations of arithmetic types (*Type1* and *Type2*): These overloads effectively cast its arguments to double before calculations, except if at least one of the arguments is of type long double (in which case both are casted to long double instead).

### Parameters

*x, y*  
Values among which the function selects a maximum.

### Return Value

The maximum numeric value of its arguments.

### Example

```

1 /* fmax example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* fmax */
4
5 int main ()
6 {
7     printf ("fmax (100.0, 1.0) = %f\n", fmax(100.0,1.0));
8     printf ("fmax (-100.0, 1.0) = %f\n", fmax(-100.0,1.0));
9     printf ("fmax (-100.0, -1.0) = %f\n", fmax(-100.0,-1.0));
10    return 0;
11 }
```

Output:

```
fmax (100.0, 1.0) = 100.000000
fmax (-100.0, 1.0) = 1.000000
fmax (-100.0,-1.0) = -1.000000
```

### See also

<a href="#">fmin</a>	Minimum value (function )
<a href="#">fdim</a>	Positive difference (function )

## /cmath/fmin

function

### fmin

<cmath> <ctgmath>

```

double fmin (double x      , double y);
float fminf (float x       , float y);
long double fminl (long double x, long double y);

double fmin (double x      , double y);
float fmin (float x       , float y);
long double fmin (long double x, long double y);
double fmin (Type1 x      , Type2 y);           // additional overloads

```

### Minimum value

Returns the smaller of its arguments: either *x* or *y*.

If one of the arguments in a *NaN*, the other is returned.

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for other combinations of **arithmetic types** (*Type1* and *Type2*): These overloads effectively cast its arguments to *double* before calculations, except if at least one of the arguments is of type *long double* (in which case both are casted to *long double* instead).

### Parameters

*x, y*  
Values among which the function selects a minimum.

### Return Value

The minimum numeric value of its arguments.

### Example

```

1 /* fmin example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* fmin */
4
5 int main ()
6 {
7     printf ("fmin (100.0, 1.0) = %f\n", fmin(100.0,1.0));
8     printf ("fmin (-100.0, 1.0) = %f\n", fmin(-100.0,1.0));
9     printf ("fmin (-100.0, -1.0) = %f\n", fmin(-100.0,-1.0));
10    return 0;
11 }

```

Output:

```

fmin (100.0, 1.0) = 1.000000
fmin (-100.0, 1.0) = -100.000000
fmin (-100.0,-1.0) = -100.000000

```

### See also

<b>fmax</b>	Maximum value (function )
<b>fdim</b>	Positive difference (function )

## /cmath/fmod

function

### fmod

`<cmath> <ctgmath>`

```

double fmod (double numer, double denom);
double fmod (double numer      , double denom);
float fmodf (float numer      , float denom);
long double fmodl (long double numer, long double denom);

double fmod (double numer      , double denom);
float fmod (float numer      , float denom);
long double fmod (long double numer, long double denom);

double fmod (double numer      , double denom);
float fmod (float numer      , float denom);
long double fmod (long double numer, long double denom);
double fmod (Type1 numer      , Type2 denom);           // additional overloads

```

### Compute remainder of division

Returns the floating-point remainder of *numer/denom* (rounded towards zero):

*fmod* = *numer* - *tquot* \* *denom*

Where *tquot* is the truncated (i.e., rounded towards zero) result of: *numer/denom*.

A similar function, *remainder*, returns the same but with the quotient rounded to the nearest integer (instead of truncated).

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for other combinations of **arithmetic types** (*Type1* and *Type2*): These overloads effectively cast its arguments to *double* before calculations, except if at least one of the arguments is of type *long double* (in which case both are casted to *long double* instead).

### Parameters

**numer**  
Value of the quotient numerator.

**denom**  
Value of the quotient denominator.

### Return Value

The remainder of dividing the arguments.

If *denom* is zero, the function may either return zero or cause a *domain error* (depending on the library implementation).

If a *domain error* occurs, the global variable **errno** is set to **EDOM**.

If a *domain error* occurs:

- And **math\_errhandling** has **MATH\_ERRNO** set: the global variable **errno** is set to **EDOM**.
- And **math\_errhandling** has **MATH\_ERREXCEPT** set: **FE\_INVALID** is raised.

### Example

```
1 /* fmod example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* fmod */
4
5 int main ()
6 {
7     printf ( "fmod of 5.3 / 2 is %f\n", fmod (5.3,2) );
8     printf ( "fmod of 18.5 / 4.2 is %f\n", fmod (18.5,4.2) );
9     return 0;
10 }
```

Output:

```
fmod of 5.3 / 2 is 1.300000
fmod of 18.5 / 4.2 is 1.700000
```

### See also

<b>remainder</b>	Compute remainder (IEC 60559) ( <a href="#">function</a> )
<b>fabs</b>	Compute absolute value ( <a href="#">function</a> )
<b>modf</b>	Break into fractional and integral parts ( <a href="#">function</a> )

## /cmath/fpclassify

macro/function

### fpclassify

<cmath> <ctgmath>

macro **fpclassify(x)**

function **int fpclassify (float x);**  
**int fpclassify (double x);**  
**int fpclassify (long double x);**

#### Classify floating-point value

Returns a value of type **int** that matches one of the *classification macro constants*, depending on the value of *x*:

value	description
<b>FP_INFINITE</b>	Positive or negative infinity (overflow)
<b>FP_NAN</b>	Not-A-Number
<b>FP_ZERO</b>	Value of zero
<b>FP_SUBNORMAL</b>	Sub-normal value (underflow)
<b>FP_NORMAL</b>	Normal value (none of the above)

Note that each value pertains to a single category: zero is not a normal value.

These macro constants of type **int** are defined in header **<cmath>** (**<math.h>**).

In C, this is implemented as a macro, but the type of *x* shall be **float**, **double** or **long double**.

In C++, it is implemented with function overloads for each *floating-point type*.

### Parameters

**x**

The value to classify.

### Return value

One of the following **int** values: **FP\_INFINITE**, **FP\_NAN**, **FP\_ZERO**, **FP\_SUBNORMAL** or **FP\_NORMAL**.

### Example

```
1 /* fpclassify example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* fpclassify, signbit, FP_* */
4
5 int main()
```

```

6 {
7     double d = 1.0 / 0.0;
8     switch (fpclassify(d)) {
9         case FP_INFINITE: printf ("infinite"); break;
10        case FP_NAN: printf ("NaN"); break;
11        case FP_ZERO: printf ("zero"); break;
12        case FP_SUBNORMAL: printf ("subnormal"); break;
13        case FP_NORMAL: printf ("normal"); break;
14    }
15    if (signbit(d)) printf (" negative\n");
16    else printf (" positive or unsigned\n");
17    return 0;
18 }

```

Output:

```
infinite positive or unsigned
```

## See also

<a href="#">isfinite</a>	Is finite value (macro )
<a href="#">isnan</a>	Is Not-A-Number (macro/function )
<a href="#">isnormal</a>	Is normal (macro/function )
<a href="#">signbit</a>	Sign bit (macro/function )

## /cmath/frexp

function

### frexp

<cmath> <ctgmath>

```

double frexp (double x, int* exp);
    double frexp (double x      , int* exp);
    float frexpf (float x      , int* exp);
long double frexpl (long double x, int* exp);

    double frexp (double x      , int* exp);
    float frexp (float x      , int* exp);
long double frexp (long double x, int* exp);

    double frexp (double x      , int* exp);
    float frexp (float x      , int* exp);
long double frexp (long double x, int* exp);
    double frexp (T x      , int* exp); // additional overloads for integral types

```

#### Get significand and exponent

Breaks the floating point number *x* into its binary significand (a floating point with an absolute value between 0.5(included) and 1.0(excluded)) and an integral exponent for 2, such that:

*x* = significand \* 2<sup>exponent</sup>

The *exponent* is stored in the location pointed by *exp*, and the *significand* is the value returned by the function.

If *x* is zero, both parts (significand and exponent) are zero.

If *x* is negative, the *significand* returned by this function is negative.

Header <[tgmath.h](#)> provides a type-generic macro version of this function.

Additional overloads are provided in this header (<[cmath](#)>) for the integral types: These overloads effectively cast *x* to a double before calculations (defined for *T* being any [integral type](#)).

## Parameters

**x** Value to be decomposed.

**exp** Pointer to an *int* where the value of the exponent is stored.

## Return Value

The binary significand of *x*.

This value is the floating point value whose absolute value lays in the interval [0.5,1) which, once multiplied by 2 raised to the power of *exp*, yields *x*.

## Example

```

1 /* frexp example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* frexp */
4
5 int main ()
6 {
7     double param, result;
8     int n;
9
10    param = 8.0;
11    result = frexp (param , &n);
12    printf ("%f = %f * 2^%d\n", param, result, n);
13    return 0;
14 }

```

Output:

```
8.000000 = 0.500000 * 2^4
```

## See also

<a href="#">ldexp</a>	Generate value from significand and exponent ( <a href="#">function</a> )
<a href="#">log</a>	Compute natural logarithm ( <a href="#">function</a> )
<a href="#">pow</a>	Raise to power ( <a href="#">function</a> )

## /cmath/HUGE\_VAL

constant

### HUGE\_VAL

<cmath> <ctgmath>

#### Huge value

Macro constant that expands to a positive expression of type `double`.

A function returns this value when the result of a mathematical operation yields a value that is too large in magnitude to be representable with its return type. This is one of the possible *range errors*, and is signaled by setting `errno` to `ERANGE`.

Actually, functions may either return a positive or a negative `HUGE_VAL` (`HUGE_VAL` or `-HUGE_VAL`) to indicate the sign of the result.

## /cmath/HUGE\_VALF

### HUGE\_VALF

#### Huge float value

Macro constant that expands to a positive expression of type `float`.

A function returns this value when the result of a mathematical operation yields a value that is too large in magnitude to be representable with its return type. This is one of the possible *range errors*, and is signaled by setting `errno` to `ERANGE`.

Actually, functions may either return a positive or a negative `HUGE_VALF` (`HUGE_VALF` or `-HUGE_VALF`) to indicate the sign of the result.

## /cmath/HUGE\_VALL

constant

### HUGE\_VALL

<cmath> <ctgmath>

#### Huge long double value

Macro constant that expands to a positive expression of type `long double`.

A function returns this value when the result of a mathematical operation yields a value that is too large in magnitude to be representable with its return type. This is one of the possible *range errors*, and is signaled by setting `errno` to `ERANGE`.

Actually, functions may either return a positive or a negative `HUGE_VALL` (`HUGE_VALL` or `-HUGE_VALL`) to indicate the sign of the result.

## /cmath/hypot

function

### hypot

<cmath> <ctgmath>

```
double hypot (double x      , double y);
float hypotf (float x       , float y);
long double hypotl (long double x, long double y);

double hypot (double x      , double y);
float hypot (float x       , float y);
long double hypot (long double x, long double y);
double hypot (Type1 x      , Type2 y); // additional overloads
```

#### Compute hypotenuse

Returns the *hypotenuse* of a *right-angled triangle* whose legs are *x* and *y*.

The function returns what would be the square root of the sum of the squares of *x* and *y* (as per the Pythagorean theorem), but without incurring in undue overflow or underflow of intermediate values.

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for other combinations of *arithmetic types* (`Type1` and `Type2`): These overloads effectively cast its arguments to `double` before calculations, except if at least one of the arguments is of type `long double` (in which case both are casted to `long double` instead).

## Parameters

*x*, *y*

Floating point values corresponding to the legs of a *right-angled triangle* for which the hypotenuse is computed.

## Return Value

The *square root* of  $(x^2 + y^2)$ .

If the magnitude of the result is too large to be represented by a value of the return type, the function may return `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign (in which case, an overflow *range error* occurs):

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* hypot example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* hypot */
4
5 int main ()
6 {
7     double leg_x, leg_y, result;
8     leg_x = 3;
9     leg_y = 4;
10    result = hypot (leg_x, leg_y);
11    printf ("%f, %f and %f form a right-angled triangle.\n",leg_x,leg_y,result);
12    return 0;
13 }
```

Output:

```
3.000000, 4.000000 and 5.000000 form a right-angled triangle.
```

## See also

<a href="#">sqrt</a>	Compute square root ( <a href="#">function</a> )
<a href="#">pow</a>	Raise to power ( <a href="#">function</a> )

## /cmath/ilogb

function  
**ilogb**

`<cmath>` `<ctgmath>`

```
int ilogb (double x);
int ilogbf (float x);
int ilogbl (long double x);

int ilogb (double x);
int ilogb (float x);
int ilogb (long double x);
int ilogb (T x);           // additional overloads for integral types
```

### Integer binary logarithm

Returns the integral part of the logarithm of  $|x|$ , using `FLT_RADIX` as base for the logarithm.

This is the *exponent* used internally by the machine to express the floating-point value  $x$ , when it uses a *significand* between 1.0 and `FLT_RADIX`, so that, for a positive  $x$ :

$$x = \text{significand} * \text{FLT\_RADIX}^{\text{exponent}}$$

Generally, `FLT_RADIX` is 2, and the value returned by this function is one less than the *exponent* obtained with `frexp` (because of the different *significand* normalization as  $[1.0, 2.0)$  instead of  $[0.5, 1.0)$ ).

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a `double` before calculations (defined for  $T$  being any *integral type*).

Two specific macro constants may be returned by this function to indicate the following special cases:

macro	description
<code>FP_ILOGB0</code>	$x$ is zero
<code>FP_ILOGBNAN</code>	$x$ is <i>NaN</i>

These macro constants are defined in this same header (`<cmath>`).

## Parameters

$x$   
Value whose *ilogb* is returned.

## Return Value

If  $x$  is normal, the base-`FLT_RADIX` logarithm of  $x$ .

If  $x$  is subnormal, the value returned is the one corresponding to the normalized representation (negative exponent).

If  $x$  is zero, it returns `FP_ILOGB0` (a special value, only returned by this function, defined in `<cmath>`).

If  $x$  is infinite, it returns `INT_MAX`.

If  $x$  is *NaN*, it returns `FP_ILOGBNAN` (a special value, only returned by this function, defined in `<cmath>`).

If the magnitude of the result is too large to be represented by a value of the return type, the function returns an unspecified value, and an overflow *range error* occurs.

A zero, infinite or NaN value of *x* may also cause either a *domain error* or an overflow *range error*.

If an *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* ilogb example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* ilogb */
4
5 int main ()
6 {
7     double param;
8     int result;
9
10    param = 10.0;
11    result = ilogb (param);
12    printf ("ilogb(%f) = %d\n", param, result);
13    return 0;
14 }
```

Output:

```
ilogb(10.000000) = 3
```

## See also

<code>logb</code>	Compute floating-point base logarithm (function )
<code>log2</code>	Compute binary logarithm (function )
<code>pow</code>	Raise to power (function )

## /cmath/INFINITY

constant

## INFINITY

<cmath> <ctgmath>

### Infinity

Macro constant that expands to an expression of type `float`.

If the implementation supports *infinity* values, this is defined as the value that represents a positive or unsigned infinity. Otherwise, it is a positive constant that overflows at translation time.

This may be returned by a function that signals a *range error* by setting `errno` to `ERANGE`.

## /cmath/isfinite

macro

## isfinite

<cmath> <ctgmath>

```
macro isfinite(x)
```

```
function bool isfinite (float x);
function bool isfinite (double x);
function bool isfinite (long double x);
```

### Is finite value

Returns whether *x* is a *finite value*.

A *finite value* is any floating-point value that is neither *infinite* nor *NaN* (*Not-A-Number*).

In C, this is implemented as a macro that returns an `int` value. The type of *x* shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

## Parameters

*x*

A floating-point value.

## Return value

A non-zero value (`true`) if *x* is finite; and zero (`false`) otherwise.

## Example

```

1 /* isfinite example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* isfinite, sqrt */
4
5 int main()
6 {
7     printf ("isfinite(0.0)      : %d\n",isfinite(0.0));
8     printf ("isfinite(1.0/0.0)   : %d\n",isfinite(1.0/0.0));
9     printf ("isfinite(-1.0/0.0)  : %d\n",isfinite(-1.0/0.0));
10    printf ("isfinite(sqrt(-1.0)): %d\n",isfinite(sqrt(-1.0)));
11    return 0;
12 }

```

Output:

```

isfinite(0.0)      : 1
isfinite(1.0/0.0)   : 0
isfinite(-1.0/0.0)  : 0
isfinite(sqrt(-1.0)): 0

```

## See also

<a href="#">isinf</a>	Is infinity (macro/function )
<a href="#">isnan</a>	Is Not-A-Number (macro/function )
<a href="#">isnormal</a>	Is normal (macro/function )
<a href="#">fpclassify</a>	Classify floating-point value (macro/function )

## /cmath/isgreater

macro

## isgreater

<cmath> <ctgmath>

macro <b>isgreater</b> (x,y)
function    bool isgreater (float x      , float y); bool isgreater (double x     , double y); bool isgreater (long double x, long double y);

### Is greater

Returns whether *x* is greater than *y*.

If one or both arguments are *NaN*, the function returns *false*, but no **FE\_INVALID** exception is raised (note that the expression *x>y* may raise such an exception in this case).

In C, this is implemented as a macro that returns an *int* value. The type of both *x* and *y* shall be *float*, *double* or *long double*.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a *bool* value.

## Parameters

*x*, *y*  
Values to be compared.

## Return value

The same as *(x)>(y)*:  
*true* (1) if *x* is greater than *y*.  
*false* (0) otherwise.

## Example

```

1 /* isgreater example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* isgreater, log */
4
5 int main ()
6 {
7     double result;
8     result = log (10.0);
9
10    if (isgreater(result,0.0))
11        printf ("log(10.0) is positive");
12    else
13        printf ("log(10.0) is not positive");
14
15    return 0;
16 }

```

Output:

```
log(10.0) is positive
```

## See also

<a href="#">isgreaterequal</a>	Is greater or equal (macro )
--------------------------------	------------------------------

<b>isless</b>	Is less (macro )
<b>islessequal</b>	Is less or equal (macro )
<b>islessgreater</b>	Is less or greater (macro )
<b>isunordered</b>	Is unordered (macro )

## /cmath/isgreaterequal

macro

### isgreaterequal

<cmath> <ctgmath>

```
macro isgreaterequal(x,y)
    bool isgreaterequal (float x      , float y);
function bool isgreaterequal (double x     , double y);
    bool isgreaterequal (long double x, long double y);
```

#### Is greater or equal

Returns whether  $x$  is greater than or equal to  $y$ .

If one or both arguments are *NaN*, the function returns `false`, but no `FE_INVALID` exception is raised (note that the expression  $x \geq y$  may raise such an exception in this case).

In C, this is implemented as a macro that returns an `int` value. The type of both  $x$  and  $y$  shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

#### Parameters

$x, y$   
Values to be compared.

#### Return value

The same as  $(x) \geq (y)$ :  
`true` (1) if  $x$  is greater than or equal to  $y$ .  
`false` (0) otherwise.

#### Example

```
1 /* isgreaterequal example */
2 #include <stdio.h>      /* printf */
3 #include <cmath.h>        /* isgreaterequal, log */
4
5 int main ()
6 {
7     double result;
8     result = log (10.0);
9
10    if (isgreaterequal(result,0.0))
11        printf ("log(10.0) is not negative");
12    else
13        printf ("log(10.0) is negative");
14
15    return 0;
16 }
```

Output:

log(10.0) is not negative

#### See also

<b>isgreater</b>	Is greater (macro )
<b>isless</b>	Is less (macro )
<b>islessequal</b>	Is less or equal (macro )
<b>islessgreater</b>	Is less or greater (macro )
<b>isunordered</b>	Is unordered (macro )

## /cmath/isinf

macro/function

### isinf

<cmath> <ctgmath>

```
macro isinf(x)
    bool isinf (float x);
function bool isinf (double x);
    bool isinf (long double x);
```

#### Is infinity

Returns whether  $x$  is an *infinity value* (either *positive infinity* or *negative infinity*).

In C, this is implemented as a macro that returns an `int` value. The type of `x` shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

## Parameters

x

A floating-point value.

## Return value

A non-zero value (`true`) if `x` is an infinity; and zero (`false`) otherwise.

## Example

```
1 /* isinf example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* isinf, sqrt */
4
5 int main()
6 {
7     printf ("isinf(0.0)      : %d\n",isinf(0.0));
8     printf ("isinf(1.0/0.0)   : %d\n",isinf(1.0/0.0));
9     printf ("isinf(-1.0/0.0)  : %d\n",isinf(-1.0/0.0));
10    printf ("isinf(sqrt(-1.0)): %d\n",isinf(sqrt(-1.0)));
11    return 0;
12 }
```

Output:

```
isinf(0.0)      : 0
isinf(1.0/0.0)   : 1
isinf(-1.0/0.0)  : 1
isinf(sqrt(-1.0)): 0
```

## See also

<a href="#">isfinite</a>	Is finite value (macro )
<a href="#">isnan</a>	Is Not-A-Number (macro/function )
<a href="#">isnormal</a>	Is normal (macro/function )
<a href="#">fpclassify</a>	Classify floating-point value (macro/function )

## /cmath/isless

macro

## isless

<cmath> <ctgmath>

```
macro  isless(x,y)
function bool isless (float x      , float y);
       bool isless (double x     , double y);
       bool isless (long double x, long double y);
```

### Is less

Returns whether `x` is less than `y`.

If one or both arguments are *NaN*, the function returns `false`, but no `FE_INVALID` exception is raised (note that the expression `x<y` may raise such an exception in this case).

In C, this is implemented as a macro that returns an `int` value. The type of both `x` and `y` shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

## Parameters

x, y

Values to be compared.

## Return value

The same as `(x)<(y)`:

`true (1)` if `x` is less than `y`.

`false (0)` otherwise.

## Example

```
1 /* isless example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* isless, log */
4
5 int main ()
6 {
7     double result;
8     result = log (10.0);
9 }
```

```

10 if (isless(result,0.0))
11   printf ("log(10.0) is negative");
12 else
13   printf ("log(10.0) is not negative");
14
15 return 0;
16 }

```

Output:

```
log(10.0) is not negative
```

### See also

<a href="#">isgreater</a>	Is greater (macro )
<a href="#">isgreaterequal</a>	Is greater or equal (macro )
<a href="#">islessequal</a>	Is less or equal (macro )
<a href="#">islessgreater</a>	Is less or greater (macro )
<a href="#">isunordered</a>	Is unordered (macro )

## /cmath/islessequal

macro

### islessequal

<cmath> <ctgmath>

```
macro islessequal(x,y)
```

```
function bool islessequal (float x      , float y);
          bool islessequal (double x      , double y);
          bool islessequal (long double x, long double y);
```

#### Is less or equal

Returns whether *x* is less than or equal to *y*.

If one or both arguments are *NaN*, the function returns *false*, but no `FE_INVALID` exception is raised (note that the expression *x<=y* may raise such an exception in this case).

In C, this is implemented as a macro that returns an `int` value. The type of both *x* and *y* shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

### Parameters

*x*, *y*  
Values to be compared.

### Return value

The same as `(x)<=(y)`:  
*true* (1) if *x* is less than or equal to *y*.  
*false* (0) otherwise.

### Example

```

1 /* islessequal example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* islessequal, log */
4
5 int main ()
6 {
7   double result;
8   result = log (10.0);
9
10  if (islessequal(result,0.0))
11    printf ("log(10.0) is not positive");
12  else
13    printf ("log(10.0) is positive");
14
15 return 0;
16 }
```

Output:

```
log(10.0) is positive
```

### See also

<a href="#">isgreater</a>	Is greater (macro )
<a href="#">isgreaterequal</a>	Is greater or equal (macro )
<a href="#">isless</a>	Is less (macro )
<a href="#">islessgreater</a>	Is less or greater (macro )
<a href="#">isunordered</a>	Is unordered (macro )

## /cmath/islessgreater

macro

### islessgreater

<cmath> <ctgmath>

```
macro islessgreater(x,y)
function bool islessgreater (float x      , float y);
function bool islessgreater (double x     , double y);
function bool islessgreater (long double x, long double y);
```

#### Is less or greater

Returns whether  $x$  is less than or greater than  $y$ .

If one or both arguments are *NaN*, the function returns `false`, but no `FE_INVALID` exception is raised (note that the expression  $x < y \mid x > y$  may raise such an exception in this case).

In C, this is implemented as a macro that returns an `int` value. The type of both  $x$  and  $y$  shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

#### Parameters

$x, y$  Values to be compared.

#### Return value

The same as  $(x) < (y) \mid (x) > (y)$ :

`true` (1) if  $x$  is less than or greater than  $y$ .

`false` (0) otherwise.

#### Example

```
1 /* islessgreater example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>             /* islessgreater, log */
4
5 int main ()
6 {
7     double result;
8     result = log (10.0);
9
10    if (islessgreater(result,0.0))
11        printf ("log(10.0) is not zero");
12    else
13        printf ("log(10.0) is zero");
14
15    return 0;
16 }
```

Output:

```
log(10.0) is not zero
```

#### See also

<a href="#">isgreater</a>	Is greater (macro )
<a href="#">isgreaterequal</a>	Is greater or equal (macro )
<a href="#">isless</a>	Is less (macro )
<a href="#">islessequal</a>	Is less or equal (macro )
<a href="#">isunordered</a>	Is unordered (macro )

## /cmath/isnan

macro/function

### isnan

<cmath> <ctgmath>

```
macro isnan(x)
function bool isnan (float x);
function bool isnan (double x);
function bool isnan (long double x);
```

#### Is Not-A-Number

Returns whether  $x$  is a *NaN* (*Not-A-Number*) value.

The *NaN* values are used to identify undefined or non-representable values for floating-point elements, such as the square root of negative numbers or the result of  $0/0$ .

In C, this is implemented as a macro that returns an `int` value. The type of  $x$  shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

## Parameters

- x A floating-point value.

## Return value

A non-zero value (true) if x is a *NaN* value; and zero (false) otherwise.

## Example

```
1 /* isnan example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* isnan, sqrt */
4
5 int main()
6 {
7     printf ("isnan(0.0)      : %d\n",isnan(0.0));
8     printf ("isnan(1.0/0.0)   : %d\n",isnan(1.0/0.0));
9     printf ("isnan(-1.0/0.0) : %d\n",isnan(-1.0/0.0));
10    printf ("isnan(sqrt(-1.0)): %d\n",isnan(sqrt(-1.0)));
11    return 0;
12 }
```

Output:

```
isnan(0.0)      : 0
isnan(1.0/0.0)   : 0
isnan(-1.0/0.0) : 0
isnan(sqrt(-1.0)): 1
```

## See also

<a href="#">NAN</a>	Not-A-Number (constant )
<a href="#">isfinite</a>	Is finite value (macro )
<a href="#">isinf</a>	Is infinity (macro/function )
<a href="#">isnormal</a>	Is normal (macro/function )
<a href="#">fpclassify</a>	Classify floating-point value (macro/function )

## /cmath/isnormal

macro/function

## isnormal

<cmath> <ctgmath>

macro `isnormal(x)`

function `bool isnormal (float x);`  
`bool isnormal (double x);`  
`bool isnormal (long double x);`

### Is normal

Returns whether x is a *normal value*: i.e., whether it is neither *infinity*, *NaN*, zero or *subnormal*.

In C, this is implemented as a macro that returns an `int` value. The type of x shall be `float`, `double` or `long double`.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a `bool` value.

## Parameters

- x A floating-point value.

## Return value

A non-zero value (true) if x is *normal*; and zero (false) otherwise.

## Example

```
1 /* isnormal example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* isnormal */
4
5 int main()
6 {
7     printf ("isnormal(1.0)    : %d\n",isnormal(1.0));
8     printf ("isnormal(0.0)    : %d\n",isnormal(0.0));
9     printf ("isnormal(1.0/0.0): %d\n",isnormal(1.0/0.0));
10    return 0;
11 }
```

Output:

```
isnormal(1.0)    : 1
```

```
isnormal(0.0)      : 0
isnormal(1.0/0.0): 0
```

## See also

<a href="#">isfinite</a>	Is finite value (macro )
<a href="#">isinf</a>	Is infinity (macro/function )
<a href="#">isnan</a>	Is Not-A-Number (macro/function )
<a href="#">fpclassify</a>	Classify floating-point value (macro/function )

## /cmath/isunordered

macro

### isunordered

<cmath> <ctgmath>

```
macro isunordered(x,y)
```

```
function bool isunordered (float x      , float y);
          bool isunordered (double x     , double y);
          bool isunordered (long double x, long double y);
```

#### Is unordered

Returns whether *x* or *y* are *unordered values*:

If one or both arguments are *NaN*, the arguments are unordered and the function returns true. In no case the function raises a [FE\\_INVALID](#) exception.

In C, this is implemented as a macro that returns an *int* value. The type of both *x* and *y* shall be *float*, *double* or *long double*.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a *bool* value.

## Parameters

*x, y*  
Values to check whether they are unordered.

## Return value

true (1) if either *x* or *y* is *NaN*.  
false (0) otherwise.

## Example

```
1 /* isunordered example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>       /* isunordered, sqrt */
4
5 int main ()
6 {
7     double result;
8     result = sqrt (-1.0);
9
10    if (isunordered(result,0.0))
11        printf ("sqrt(-1.0) and 0.0 cannot be ordered");
12    else
13        printf ("sqrt(-1.0) and 0.0 can be ordered");
14
15    return 0;
16 }
```

Output:

```
sqrt(-1.0) and 0.0 cannot be ordered
```

## See also

<a href="#">isgreater</a>	Is greater (macro )
<a href="#">isgreaterequal</a>	Is greater or equal (macro )
<a href="#">isless</a>	Is less (macro )
<a href="#">islessgreater</a>	Is less or greater (macro )
<a href="#">isunordered</a>	Is unordered (macro )

## /cmath/ldexp

function

### ldexp

<cmath> <ctgmath>

```
double ldexp (double x, int exp);

double ldexp (double x      , int exp);
float ldexpf (float x      , int exp);
long double ldexpl (long double x, int exp);
```

```

double ldexp (double x      , int exp);
float ldexp (float x       , int exp);
long double ldexp (long double x, int exp);

double ldexp (double x      , int exp);
float ldexp (float x       , int exp);
long double ldexp (long double x, int exp);
double ldexp (T x          , int exp); // additional overloads for integral types

```

### Generate value from significand and exponent

Returns the result of multiplying *x* (the significand) by 2 raised to the power of *exp* (the exponent).

`ldexp(x,exp) = x * 2exp`

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast *x* to a double before calculations (defined for *T* being any *integral type*).

### Parameters

*x* Floating point value representing the *significand*.

*exp* Value of the *exponent*.

### Return Value

The function returns:

$x * 2^{\text{exp}}$

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* occurs:

If an overflow *range error* occurs, the global variable `errno` is set to `ERANGE`.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

### Example

```

1 /* ldexp example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* ldexp */
4
5 int main ()
6 {
7     double param, result;
8     int n;
9
10    param = 0.95;
11    n = 4;
12    result = ldexp (param , n);
13    printf ("%f * 2^%d = %f\n", param, n, result);
14    return 0;
15 }

```

Output:

0.950000 \* 2^4 = 15.200000

### See also

<code>frexp</code>	Get significand and exponent (function)
<code>log</code>	Compute natural logarithm (function)
<code>pow</code>	Raise to power (function)

## /cmath/Igamma

function

### Igamma

`<cmath> <ctgmath>`

```

double lgamma (double x);
float lgammaf (float x);
long double lgammal (long double x);

double lgamma (double x);
float lgamma (float x);
long double lgamma (long double x);
double lgamma (T x); // additional overloads for integral types

```

### Compute log-gamma function

Returns the natural logarithm of the absolute value of the *gamma function* of *x*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast `x` to a double before calculations (defined for `T` being any *integral type*).



## Parameters

`x`

Parameter for the *log-gamma* function.

## Return Value

Log-gamma function of `x`.

If `x` is too large, an overflow *range error* occurs.

If `x` is zero or a negative integer for which the function is asymptotic, it may cause a *pole error* (depending on implementation).

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

If a *pole error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

## Example

```
1 /* lgamma example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* lgamma */
4
5 int main ()
6 {
7     double param, result;
8     param = 0.5;
9     result = lgamma (param);
10    printf ("lgamma(%f) = %f\n", param, result );
11    return 0;
12 }
```

Output:

```
lgamma (0.500000) = 0.572365
```

## See also

<code>tgamma</code>	Compute gamma function (function )
<code>erf</code>	Compute error function (function )
<code>erfc</code>	Compute complementary error function (function )

## /cmath/llrint

function

### llrint

`<cmath> <ctgmath>`

```
long long int llrint (double x);
long long int llrintf (float x);
long long int llrintl (long double x);

long long int llrint (double x);
long long int llrint (float x);
long long int llrint (long double x);
long long int llrint (T x);           // additional overloads for integral types
```

#### Round and cast to long long integer

Rounds `x` to an integral value, using the rounding direction specified by `fegetround`, and returns it as a value of type `long long int`.

See `lrint` for an equivalent function that returns a `long int`.

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast `x` to a double before calculations (defined for `T` being any *integral type*).

## Parameters

`x`

Value to round.

## Return Value

The value of `x` rounded to a nearby integral, casted to a value of type `long long int`.

If the rounded value is outside the range of the return type, the value returned is unspecified, and a *domain error* or an overflow *range error* may occur (or none, depending on implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.

- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.

- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* llrint example */
2 #include <stdio.h>      /* printf */
3 #include <fenv.h>        /* fegetround, FE_* */
4 #include <math.h>         /* llrint */
5
6 int main ()
7 {
8     printf ("rounding using ");
9     switch (fegetround()) {
10     case FE_DOWNWARD: printf ("downward"); break;
11     case FE_TONEAREST: printf ("to-nearest"); break;
12     case FE_TOWARDZERO: printf ("toward-zero"); break;
13     case FE_UPWARD: printf ("upward"); break;
14     default: printf ("unknown");
15 }
16 printf (" rounding:\n");
17
18 printf ( "llrint (2.3) = %lld\n", llrint(2.3) );
19 printf ( "llrint (3.8) = %lld\n", llrint(3.8) );
20 printf ( "llrint (-2.3) = %lld\n", llrint(-2.3) );
21 printf ( "llrint (-3.8) = %lld\n", llrint(-3.8) );
22 return 0;
23 }
```

Possible output:

```
Rounding using to-nearest rounding:
llrint (2.3) = 2
llrint (3.8) = 4
llrint (-2.3) = -2
llrint (-3.8) = -4
```

## See also

<a href="#">nearbyint</a>	Round to nearby integral value ( <a href="#">function</a> )
<a href="#">rint</a>	Round to integral value ( <a href="#">function</a> )
<a href="#">lrint</a>	Round and cast to long integer ( <a href="#">function</a> )
<a href="#">round</a>	Round to nearest ( <a href="#">function</a> )
<a href="#">floor</a>	Round down value ( <a href="#">function</a> )
<a href="#">ceil</a>	Round up value ( <a href="#">function</a> )
<a href="#">trunc</a>	Truncate value ( <a href="#">function</a> )

## /cmath/llround

function

### llround

`<cmath>` `<ctgmath>`

```
long long int llround (double x);
long long int llroundf (float x);
long long int llroundl (long double x);

long long int llround (double x);
long long int llround (float x);
long long int llround (long double x);
long long int llround (T x);           // additional overloads for integral types
```

#### Round to nearest and cast to long long integer

Returns the integer value that is nearest in value to *x*, with halfway cases rounded away from zero.

The rounded value is returned as a value of type `long long int`. See `lround` for an equivalent function that returns a `long int` instead.

Header `<ctgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast *x* to a `double` before calculations (defined for *T* being any [integral type](#)).

#### Parameters

*x*

Value to round.

#### Return Value

The value of *x* rounded to the nearest integral, casted to a value of type `long long int`.

If the rounded value is outside the range of the return type, the value returned is unspecified, and a *domain error* or an overflow *range error* may occur (or none, depending on implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.

- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.

- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* llround example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* llround */
4
5 int main ()
6 {
7     printf ( "llround (2.3) = %lld\n", llround(2.3) );
8     printf ( "llround (3.8) = %lld\n", llround(3.8) );
9     printf ( "llround (-2.3) = %lld\n", llround(-2.3) );
10    printf ( "llround (-3.8) = %lld\n", llround(-3.8) );
11    return 0;
12 }
```

Possible output:

```
Rounding using to-nearest rounding:
llround (2.3) = 2
llround (3.8) = 4
llround (-2.3) = -2
llround (-3.8) = -4
```

## See also

<code>llrint</code>	Round and cast to long long integer ( <a href="#">function</a> )
<code>round</code>	Round to nearest ( <a href="#">function</a> )
<code>lround</code>	Round to nearest and cast to long integer ( <a href="#">function</a> )
<code>nearbyint</code>	Round to nearby integral value ( <a href="#">function</a> )
<code>floor</code>	Round down value ( <a href="#">function</a> )
<code>ceil</code>	Round up value ( <a href="#">function</a> )
<code>trunc</code>	Truncate value ( <a href="#">function</a> )

## /cmath/log

function

### log

`<cmath>` `<ctgmath>`

```
double log (double x);
double log (    double x);
float logf (    float x);
long double logl (long double x);

double log (    double x);
float log (    float x);
long double log (long double x);
double log (T x);           // additional overloads for integral types
```

#### Compute natural logarithm

Returns the *natural logarithm* of *x*.

The *natural logarithm* is the base-e logarithm: the inverse of the natural exponential function ([exp](#)). For common (base-10) logarithms, see [log10](#).

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see [complex log](#) and [valarray log](#)).

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast *x* to a double before calculations.

This function is also overloaded in `<complex>` and `<valarray>` (see [complex log](#) and [valarray log](#)).

## Parameters

*x*

Value whose logarithm is calculated.  
If the argument is negative, a *domain error* occurs.

## Return Value

Natural logarithm of *x*.

If *x* is negative, it causes a *domain error*.

If *x* is zero, it may cause a *pole error* (depending on the library implementation).

If a *domain error* occurs, the global variable `errno` is set to `EDOM`.

If a *pole error* occurs, the global variable `errno` is set `ERANGE`.

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.

- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.
- If a *pole error* occurs:
- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
  - And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

### Example

```
1 /* log example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* log */
4
5 int main ()
6 {
7     double param, result;
8     param = 5.5;
9     result = log (param);
10    printf ("log(%f) = %f\n", param, result );
11    return 0;
12 }
```

Output:

```
log(5.500000) = 1.704748
```

### See also

<a href="#">log10</a>	Compute common logarithm ( <a href="#">function</a> )
<a href="#">exp</a>	Compute exponential function ( <a href="#">function</a> )
<a href="#">pow</a>	Raise to power ( <a href="#">function</a> )

## /cmath/log10

function

**log10** <cmath> <ctgmath>

```
double log10 (double x);
       double log10 (double x);
       float log10f (float x);
long double log10l (long double x);

       double log10 (double x);
       float log10 (float x);
long double log10 (long double x);

       double log10 (double x);
       float log10 (float x);
long double log10 (long double x); // additional overloads for integral types
```

### Compute common logarithm

Returns the *common* (base-10) *logarithm* of *x*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see `complex log10` and `valarray log10`).

*Additional overloads* are provided in this header (`<cmath>`) for the *integral types*: These overloads effectively cast *x* to a `double` before calculations.

This function is also overloaded in `<complex>` and `<valarray>` (see `complex log10` and `valarray log10`).

### Parameters

*x*  
Value whose logarithm is calculated.  
If the argument is negative, a *domain error* occurs.

### Return Value

Common logarithm of *x*.  
If *x* is negative, it causes a *domain error*.  
If *x* is zero, it may cause a *pole error* (depending on the library implementation).

If a *domain error* occurs, the global variable `errno` is set to `EDOM`.  
If a *pole error* occurs, the global variable `errno` is set `ERANGE`.

If a *domain error* occurs:  

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If a *pole error* occurs:  

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

### Example

```
1 /* log10 example */
2 #include <stdio.h>      /* printf */
```

```

3 #include <math.h>      /* log10 */
4
5 int main ()
6 {
7     double param, result;
8     param = 1000.0;
9     result = log10 (param);
10    printf ("log10(%f) = %f\n", param, result );
11    return 0;
12 }

```

Output:

```
log10(1000.000000) = 3.000000
```

### See also

<a href="#">log</a>	Compute natural logarithm (function )
<a href="#">exp</a>	Compute exponential function (function )
<a href="#">pow</a>	Raise to power (function )

## /cmath/log1p

function  
**log1p** <cmath> <ctgmath>

```

double log1p (double x);
float log1pf (float x);
long double log1pl (long double x);

double log1p (double x);
float log1p (float x);
long double log1p (long double x);
double log1p (T x);           // additional overloads for integral types

```

### Compute logarithm plus one

Returns the *natural logarithm* of one plus *x*.

For small magnitude values of *x*, `logp1` may be more accurate than `log(1+x)`.

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for the *integral types*: These overloads effectively cast *x* to a *double* before calculations (defined for *T* being any *integral type*).

### Parameters

*x*  
Value whose logarithm is calculated.  
If the argument is less than -1, a *domain error* occurs.

### Return Value

The *natural logarithm* of  $(1+x)$ .  
If *x* is less than -1, it causes a *domain error*.  
If *x* is -1, it may cause a *pole error* (depending on the library implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If a *pole error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

### Example

```

1 /* log1p example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* log1p */
4
5 int main ()
6 {
7     double param, result;
8     param = 1.0;
9     result = log1p (param);
10    printf ("log1p (%f) = %f.\n", param, result );
11    return 0;
12 }

```

Output:

```
log1p (1.000000) = 0.693147
```

### See also

<b>exp</b>	Compute exponential function (function )
<b>log</b>	Compute natural logarithm (function )
<b>pow</b>	Raise to power (function )

## /cmath/log2

function  
**log2** <cmath> <ctgmath>

```
double log2 (double x);
float log2f (float x);
long double log2l (long double x);

double log2 (double x);
float log2 (float x);
long double log2 (long double x);
double log2 (T x); // additional overloads for integral types
```

### Compute binary logarithm

Returns the *binary* (base-2) *logarithm* of *x*.

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for the *integral types*: These overloads effectively cast *x* to a *double* before calculations (defined for *T* being any *integral type*).

---

### Parameters

*x*

Value whose logarithm is calculated.  
If the argument is negative, a *domain error* occurs.

---

### Return Value

The *binary logarithm* of *x*:  $\log_2 x$ .

If *x* is negative, it causes a *domain error*.

If *x* is zero, it may cause a *pole error* (depending on the library implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If a *pole error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

---

### Example

```
1 /* log2 example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* log2 */
4
5 int main ()
6 {
7     double param, result;
8     param = 1024.0;
9     result = log2 (param);
10    printf ("log2 (%f) = %f.\n", param, result );
11    return 0;
12 }
```

Output:

```
log2 (1024.000000) = 10.000000
```

---

### See also

<b>exp2</b>	Compute binary exponential function (function )
<b>log</b>	Compute natural logarithm (function )
<b>pow</b>	Raise to power (function )

## /cmath/logb

function  
**logb** <cmath> <ctgmath>

```
double logb (double x);
float logbf (float x);
long double logbl (long double x);

double logb (double x);
float logb (float x);
long double logb (long double x);
double logb (T x); // additional overloads for integral types
```

## Compute floating-point base logarithm

Returns the logarithm of  $|x|$ , using `FLT_RADIX` as base for the logarithm.

On most platforms, `FLT_RADIX` is 2, and thus this function is equivalent to `log2` for positive values.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).

### Parameters

- $x$  Value whose logarithm is calculated.

### Return Value

The base-`FLT_RADIX` logarithm of  $x$ .

If  $x$  is zero it may cause a *domain error* or a *pole error* (or no error, depending on the library implementation).

If an *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If a *pole error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

### Example

```
1 /* logb example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* logb */
4
5 int main ()
6 {
7     double param, result;
8     param = 1024.0;
9     result = logb (param);
10    printf ("logb (%f) = %f.\n", param, result );
11    return 0;
12 }
```

Output:

```
logb (1024.000000) = 10.000000
```

### See also

<a href="#">ilogb</a>	Integer binary logarithm (function)
<a href="#">log2</a>	Compute binary logarithm (function)
<a href="#">pow</a>	Raise to power (function)

## /cmath/lrint

function

### lrint

`<cmath> <ctgmath>`

```
long int lrint (double x);
long int lrintf (float x);
long int lrintl (long double x);

long int lrint (double x);
long int lrint (float x);
long int lrint (long double x);
long int lrint (T x);           // additional overloads for integral types
```

#### Round and cast to long integer

Rounds  $x$  to an integral value, using the rounding direction specified by `fegetround`, and returns it as a value of type `long int`.

See `llrint` for an equivalent function that returns a `long long int`.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).

### Parameters

- $x$  Value to round.

### Return Value

The value of  $x$  rounded to a nearby integral, casted to a value of type `long int`. If the rounded value is outside the range of the return type, the value returned is unspecified, and a *domain error* or an *overflow range error* may occur (or none, depending on implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* lrint example */
2 #include <stdio.h>           /* printf */
3 #include <fenv.h>             /* fegetround, FE_* */
4 #include <math.h>              /* lrint */
5
6 int main ()
7 {
8     printf ("rounding using ");
9     switch (fegetround()) {
10         case FE_DOWNWARD: printf ("downward"); break;
11         case FE_TONEAREST: printf ("to-nearest"); break;
12         case FE_TOWARDZERO: printf ("toward-zero"); break;
13         case FE_UPWARD: printf ("upward"); break;
14         default: printf ("unknown");
15     }
16     printf (" lrint:\n");
17
18     printf ("lrint (2.3) = %ld\n", lrint(2.3) );
19     printf ("lrint (3.8) = %ld\n", lrint(3.8) );
20     printf ("lrint (-2.3) = %ld\n", lrint(-2.3) );
21     printf ("lrint (-3.8) = %ld\n", lrint(-3.8) );
22     return 0;
23 }
```

Possible output:

```
Rounding using to-nearest rounding:
lrint (2.3) = 2
lrint (3.8) = 4
lrint (-2.3) = -2
lrint (-3.8) = -4
```

## See also

<a href="#">nearbyint</a>	Round to nearby integral value ( <a href="#">function</a> )
<a href="#">rint</a>	Round to integral value ( <a href="#">function</a> )
<a href="#">llrint</a>	Round and cast to long long integer ( <a href="#">function</a> )
<a href="#">round</a>	Round to nearest ( <a href="#">function</a> )
<a href="#">floor</a>	Round down value ( <a href="#">function</a> )
<a href="#">ceil</a>	Round up value ( <a href="#">function</a> )
<a href="#">trunc</a>	Truncate value ( <a href="#">function</a> )

## /cmath/lround

function

### **lround**

`<cmath>` `<ctgmath>`

```
long int lround (double x);
long int lroundf (float x);
long int lroundl (long double x);

long int lround (double x);
long int lround (float x);
long int lround (long double x);
long int lround (T x);           // additional overloads for integral types
```

#### Round to nearest and cast to long integer

Returns the integer value that is nearest in value to  $x$ , with halfway cases rounded away from zero.

The rounded value is returned as a value of type `long int`. See `llround` for an equivalent function that returns a `long long int` instead.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any *integral type*).

## Parameters

`x`

Value to round.

## Return Value

The value of  $x$  rounded to the nearest integral, casted to a value of type `long int`.  
If the rounded value is outside the range of the return type, the value returned is unspecified, and a *domain error* or an *overflow range error* may occur (or none, depending on implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* lround example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* lround */
4
5 int main ()
6 {
7     printf ( "lround (2.3) = %ld\n", lround(2.3) );
8     printf ( "lround (3.8) = %ld\n", lround(3.8) );
9     printf ( "lround (-2.3) = %ld\n", lround(-2.3) );
10    printf ( "lround (-3.8) = %ld\n", lround(-3.8) );
11    return 0;
12 }
```

Possible output:

```
Rounding using to-nearest rounding:
lround (2.3) = 2
lround (3.8) = 4
lround (-2.3) = -2
lround (-3.8) = -4
```

## See also

<a href="#">lrint</a>	Round and cast to long integer ( <a href="#">function</a> )
<a href="#">round</a>	Round to nearest ( <a href="#">function</a> )
<a href="#">llround</a>	Round to nearest and cast to long long integer ( <a href="#">function</a> )
<a href="#">nearbyint</a>	Round to nearby integral value ( <a href="#">function</a> )
<a href="#">floor</a>	Round down value ( <a href="#">function</a> )
<a href="#">ceil</a>	Round up value ( <a href="#">function</a> )
<a href="#">trunc</a>	Truncate value ( <a href="#">function</a> )

## /cmath/math\_errhandling

macro

### math\_errhandling

<cmath> <ctgmath>

`int`

#### Error handling

Expands to an expression that identifies the error handling mechanism employed by the functions in the `<cmath>` header:

constant	value	description
<code>MATH_ERRNO</code>	1	<code>errno</code> is used to signal errors: - On <i>domain error</i> : <code>errno</code> is set to <code>EDOM</code> . - On <i>range error</i> (including <i>pole error</i> , <i>overflow</i> , and possibly <i>underflow</i> ): <code>errno</code> is set to <code>ERANGE</code> .
<code>MATH_ERREXCEPT</code>	2	The proper C exception is raised: - On <i>domain error</i> : <code>FE_INVALID</code> is raised. - On <i>pole error</i> : <code>FE_DIVBYZERO</code> is raised. - On <i>overflow</i> : <code>FE_OVERFLOW</code> is raised. - On <i>underflow</i> : <code>FE_UNDERFLOW</code> may be raised.
<code>MATH_ERRNO MATH_ERREXCEPT</code>	3	Both of the above

Both `MATH_ERRNO` and `MATH_ERREXCEPT` are macro constant expressions defined in `<cmath>` as 1 and 2 respectively.

## Example

```
1 /* math_errhandling example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* math_errhandling */
4 #include <errno.h>            /* errno, EDOM */
5 #include <fenv.h>              /* feclearexcept, fetestexcept, FE_ALL_EXCEPT, FE_INVALID */
6 #pragma STDC FENV_ACCESS on
7
8 int main () {
9     errno = 0;
10    if (math_errhandling & MATH_ERREXCEPT) feclearexcept(FE_ALL_EXCEPT);
11
12    printf ("Error handling: %d",math_errhandling);
13
14    sqrt (-1);
15    if (math_errhandling & MATH_ERRNO) {
16        if (errno==EDOM) printf("errno set to EDOM\n");
17    }
}
```

```

18 if (math_errhandling &MATH_ERREXCEPT) {
19     if (fetestexcept(FE_INVALID)) printf("FE_INVALID raised\n");
20 }
21
22 return 0;
23 }

```

Possible output:

```
Error handling: 3
errno set to EDOM
FE_INVALID raised
```

## Data races

Libraries that support multi-threading shall implement `errno` and/or floating-point exception state in a per-thread basis: With each thread having its own local `errno` and floating-point state.

This is a requirement for libraries compliant with C11 and C++11 standards.

<code>errno</code>	Last error number (macro)
--------------------	---------------------------

## /cmath/modf

function  
**modf** <cmath> <ctgmath>

```

double modf (double x, double* intpart);
    double modf (double x      , double* intpart);
    float modff (float x      , float* intpart);
long double modfl (long double x, long double* intpart);

    double modf (double x      , double* intpart);
    float modf (float x      , float* intpart);
long double modf (long double x, long double* intpart);

    double modf (double x      , double* intpart);
    float modf (float x      , float* intpart);
long double modf (long double x, long double* intpart);
    double modf (T x      , double* intpart);           // additional overloads

```

### Break into fractional and integral parts

Breaks `x` into an integral and a fractional part.

The integer part is stored in the object pointed by `intpart`, and the fractional part is returned by the function.

Both parts have the same sign as `x`.

*Additional overloads are provided in this header (<cmath>) for the integral types: These overloads effectively cast `x` to a double before calculations (defined for `T` being any [integral type](#)).*

### Parameters

`x`  
Floating point value to break into parts.

`intpart`  
Pointer to an object (of the same type as `x`) where the integral part is stored with the same sign as `x`.

### Return Value

The fractional part of `x`, with the same sign.

### Example

```

1 /* modf example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* modf */
4
5 int main ()
6 {
7     double param, fractpart, intpart;
8
9     param = 3.14159265;
10    fractpart = modf (param , &intpart);
11    printf ("%f = %f + %f \n", param, intpart, fractpart);
12    return 0;
13 }

```

Output:

```
3.141593 = 3.000000 + 0.141593
```

### See also

<a href="#">ldexp</a>	Generate value from significand and exponent (function )
<a href="#">frexp</a>	Get significand and exponent (function )

## /cmath/NAN

constant  
**NAN**

<cmath> <ctgmath>

**float**

### Not-A-Number

Macro constant that expands to an expression of type **float** that represents a NaN if the implementation supports quiet NaNs (otherwise, it is not defined).

#### See also

<b>nan</b>	Generate quiet NaN ( <a href="#">function</a> )
<b>nanf</b>	Generate quiet NaN (float) ( <a href="#">function</a> )
<b>isnan</b>	Is Not-A-Number ( <a href="#">macro/function</a> )

## /cmath/nanf

function  
**nanf**

<cmath> <ctgmath>

**float nanf (const char\* tagp);**

### Generate quiet NaN (float)

Returns a quiet *NaN* (Not-A-Number) value of type **float**.

The *NaN* values are used to identify undefined or non-representable values for floating-point elements, such as the square root of negative numbers or the result of 0/0.

The argument can be used by library implementations to distinguish different *NaN* values in a implementation-specific manner.

Similarly, [nan](#) and [nanl](#) return *NaN* values of type **double** and **long double**, respectively.

#### Parameters

**tagp**  
An implementation-specific C-string.  
If this is an empty string (""), the function returns a generic *NaN* value (the same as returned by passing "NAN" to [strtod](#)).

#### Return Value

A quiet *NaN* value.

#### See also

<b>NAN</b>	Not-A-Number ( <a href="#">constant</a> )
<b>nan</b>	Generate quiet NaN ( <a href="#">function</a> )
<b>isnan</b>	Is Not-A-Number ( <a href="#">macro/function</a> )
<b>nextafter</b>	Next representable value ( <a href="#">function</a> )
<b>nexttoward</b>	Next representable value toward precise value ( <a href="#">function</a> )

## /cmath/nan-function

function  
**nan**

<cmath> <ctgmath>

**double nan (const char\* tagp);**

### Generate quiet NaN

Returns a quiet *NaN* (Not-A-Number) value of type **double**.

The *NaN* values are used to identify undefined or non-representable values for floating-point elements, such as the square root of negative numbers or the result of 0/0.

The argument can be used by library implementations to distinguish different *NaN* values in a implementation-specific manner.

Similarly, [nanf](#) and [nanl](#) return *NaN* values of type **float** and **long double**, respectively.

#### Parameters

**tagp**  
An implementation-specific C-string.  
If this is an empty string (""), the function returns a generic *NaN* value (the same as returned by passing "NAN" to [strtod](#)).

#### Return Value

A quiet *NaN* value.

## See also

<a href="#">isnan</a>	Is Not-A-Number (macro/function )
<a href="#">nextafter</a>	Next representable value (function )
<a href="#">nexttoward</a>	Next representable value toward precise value (function )
<a href="#">NAN</a>	Not-A-Number (constant )

## /cmath/nanl

function  
**nanl** <cmath> <ctgmath>

```
float nanl (const char* tagp);
```

### Generate quiet NaN (long double)

Returns a quiet *Nan* (Not-A-Number) value of type *long double*.

The *Nan* values are used to identify undefined or non-representable values for floating-point elements, such as the square root of negative numbers or the result of 0/0.

The argument can be used by library implementations to distinguish different *Nan* values in a implementation-specific manner.

Similarly, [nan](#) and [nanf](#) return *Nan* values of type *double* and *float*, respectively.

## Parameters

tagp An implementation-specific C-string.  
If this is an empty string (""), the function returns a generic *Nan* value (the same as returned by passing "NAN" to [strtold](#)).

## Return Value

A quiet *Nan* value.

## See also

<a href="#">NAN</a>	Not-A-Number (constant )
<a href="#">nan</a>	Generate quiet NaN (function )
<a href="#">isnan</a>	Is Not-A-Number (macro/function )
<a href="#">nextafter</a>	Next representable value (function )
<a href="#">nexttoward</a>	Next representable value toward precise value (function )

## /cmath/nearbyint

function  
**nearbyint** <cmath> <ctgmath>

```
double nearbyint (double x);
float nearbyintf (float x);
long double nearbyintl (long double x);

double nearbyint (double x);
float nearbyint (float x);
long double nearbyint (long double x);
double nearbyint (T x); // additional overloads for integral types
```

### Round to nearby integral value

Rounds *x* to an integral value, using the rounding direction specified by [fegetround](#).

This function does not raise [FE\\_INEXACT](#) exceptions. See [rint](#) for an equivalent function that may do.

Header [<tgmath.h>](#) provides a type-generic macro version of this function.

Additional overloads are provided in this header ([<cmath>](#)) for the integral types: These overloads effectively cast *x* to a *double* before calculations (defined for *T* being any [integral type](#)).

## Parameters

*x* Value to round.

## Return Value

The value of *x* rounded to a nearby integral (as a floating-point value).

## Example

```
1 /* nearbyint example */
2 #include <stdio.h>           /* printf */
3 #include <fenv.h>             /* fegetround, FE_* */
4 #include <math.h>              /* nearbyint */
5
```

```

6 int main ()
7 {
8     printf ("rounding using ");
9     switch (fegetround ()) {
10     case FE_DOWNWARD: printf ("downward"); break;
11     case FE_TONEAREST: printf ("to-nearest"); break;
12     case FE_TOWARDZERO: printf ("toward-zero"); break;
13     case FE_UPWARD: printf ("upward"); break;
14     default: printf ("unknown");
15 }
16 printf (" rounding:\n");
17
18 printf ("nearbyint (2.3) = %.1f\n", nearbyint(2.3));
19 printf ("nearbyint (3.8) = %.1f\n", nearbyint(3.8));
20 printf ("nearbyint (-2.3) = %.1f\n", nearbyint(-2.3));
21 printf ("nearbyint (-3.8) = %.1f\n", nearbyint(-3.8));
22 return 0;
23 }
```

Possible output:

```
Rounding using to-nearest rounding:
nearbyint (2.3) = 2.0
nearbyint (3.8) = 4.0
nearbyint (-2.3) = -2.0
nearbyint (-3.8) = -4.0
```

## See also

<a href="#">rint</a>	Round to integral value ( <a href="#">function</a> )
<a href="#">lrint</a>	Round and cast to long integer ( <a href="#">function</a> )
<a href="#">round</a>	Round to nearest ( <a href="#">function</a> )
<a href="#">floor</a>	Round down value ( <a href="#">function</a> )
<a href="#">ceil</a>	Round up value ( <a href="#">function</a> )
<a href="#">trunc</a>	Truncate value ( <a href="#">function</a> )

## /cmath/nextafter

function **nextafter** <cmath> <ctgmath>

```

double nextafter (double x      , double y);
float nextafterf (float x      , float y);
long double nextafterl (long double x, long double y);

double nextafter (double x      , double y );
float nextafter (float x      , float y );
long double nextafter (long double x, long double y );
double nextafter (Type1 x      , Type2 y);           // additional overloads
```

### Next representable value

Returns the next representable value after *x* in the direction of *y*.

The similar function, [nexttoward](#) has the same behavior, but it takes a *long double* as second argument.

Header [<tgmath.h>](#) provides a type-generic macro version of this function.

Additional overloads are provided in this header ([<cmath>](#)) for other combinations of arithmetic types (*Type1* and *Type2*): These overloads effectively cast its arguments to *double* before calculations, except if at least one of the arguments is of type *long double* (in which case both are casted to *long double* instead).

### Parameters

*x* Base value.

*y* Value toward which the return value is approximated.

If both parameters compare equal, the function returns *y*.

### Return Value

The next representable value after *x* in the direction of *y*.

If *x* is the largest finite value representable in the type, and the result is infinite or not representable, an overflow *range error* occurs.

If an overflow *range error* occurs:

- And [math\\_errhandling](#) has *MATH\_ERRNO* set: the global variable *errno* is set to *ERANGE*.
- And [math\\_errhandling](#) has *MATH\_ERREXCEPT* set: *FE\_OVERFLOW* is raised.

### Example

```

1 /* nextafter example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* nextafter */
4
5 int main ()
6 {
```

```

7 printf ("first representable value greater than zero: %e\n", nextafter(0.0,1.0));
8 printf ("first representable value less than zero: %e\n", nextafter(0.0,-1.0));
9 return 0;
10 }

```

Possible output:

```
first representable value greater than zero: 4.940656e-324
first representable value less than zero: -4.940656e-324
```

## See also

[nexttoward](#) Next representable value toward precise value (function )

## /cmath/nexttoward

function [nexttoward](#) <cmath> <ctgmath>

```

double nexttoward (double x      , long double y);
float nexttowardf (float x      , long double y);
long double nexttowardl (long double x, long double y);

double nexttoward (double x      , long double y);
float nexttoward (float x      , long double y);
long double nexttoward (long double x, long double y);
double nexttoward (T x      , long double y); // additional overloads

```

### Next representable value toward precise value

Returns the next representable value after *x* in the direction of *y*.

This function behaves as [nextafter](#), but with a potentially more precise *y*.

Header [<tgmath.h>](#) provides a type-generic macro version of this function.

Additional overloads are provided in this header ([<cmath>](#)) for the integral types: These overloads effectively cast *x* to a double before calculations (defined for *T* being any [integral type](#)).

## Parameters

*x* Base value.

*y* Value toward which the return value is approximated.

If both parameters compare equal, the function returns *y* (converted to the return type).

## Return Value

The next representable value after *x* in the direction of *y*.

If *x* is the largest finite value representable in the type, and the result is infinite or not representable, an overflow *range error* occurs.

If an overflow *range error* occurs:

- And [math\\_errhandling](#) has [MATH\\_ERRNO](#) set: the global variable [errno](#) is set to [ERANGE](#).
- And [math\\_errhandling](#) has [MATH\\_ERREXCEPT](#) set: [FE\\_OVERFLOW](#) is raised.

## Example

```

1 /* nexttoward example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* nexttoward */
4
5 int main ()
6 {
7     printf ("first representable value greater than zero: %e\n", nexttoward(0.0,1.0L));
8     printf ("first representable value less than zero: %e\n", nexttoward(0.0,-1.0L));
9     return 0;
10 }

```

Possible output:

```
first representable value greater than zero: 4.940656e-324
first representable value less than zero: -4.940656e-324
```

## See also

[nextafter](#) Next representable value (function )

## /cmath/pow

function [pow](#) <cmath> <ctgmath>

```

double pow (double base, double exponent);
    double pow (double base      , double exponent);
    float  powf (float base     , float exponent);
long double powl (long double base, long double exponent);
    double pow (double base      , double exponent);
    float  pow (float base     , float exponent);
long double pow (long double base, long double exponent);
    double pow (Type1 base     , Type2 exponent);           // additional overloads

```

## Raise to power

Returns *base* raised to the power *exponent*:

*base**exponent*

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see [complex pow](#) and [valarray pow](#)).

*Additional overloads* are provided in this header (`<cmath>`) for other combinations of [arithmetic types](#) (*Type1* and *Type2*): These overloads effectively cast its arguments to `double` before calculations, except if at least one of the arguments is of type `long double` (in which case both are casted to `long double` instead).

This function is also overloaded in `<complex>` and `<valarray>` (see [complex pow](#) and [valarray pow](#)).

## Parameters

*base*

Base value.

*exponent*

Exponent value.

## Return Value

The result of raising *base* to the power *exponent*.

If the *base* is [finite](#) negative and the *exponent* is [finite](#) but not an integer value, it causes a [domain error](#).

If both *base* and *exponent* are zero, it may also cause a [domain error](#) on certain implementations.

If *base* is zero and *exponent* is negative, it may cause a [domain error](#) or a [pole error](#) (or none, depending on the library implementation).

The function may also cause a [range error](#) if the result is too great or too small to be represented by a value of the return type.

If a [domain error](#) occurs, the global variable `errno` is set to `EDOM`.

If a [pole](#) or [range error](#) occurs, the global variable `errno` is set `ERANGE`.

If a [domain error](#) occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If a [pole error](#) occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_DIVBYZERO` is raised.

If a [range error](#) occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: either `FE_OVERFLOW` or `FE_UNDERFLOW` is raised.

## Example

```

1 /* pow example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* pow */
4
5 int main ()
6 {
7     printf ("7 ^ 3 = %f\n", pow (7.0, 3.0) );
8     printf ("4.73 ^ 12 = %f\n", pow (4.73, 12.0) );
9     printf ("32.01 ^ 1.54 = %f\n", pow (32.01, 1.54) );
10    return 0;
11 }

```

Output:

```

7 ^ 3 = 343.000000
4.73 ^ 12 = 125410439.217423
32.01 ^ 1.54 = 208.036691

```

## See also

<a href="#">log</a>	Compute natural logarithm (function)
<a href="#">exp</a>	Compute exponential function (function )
<a href="#">sqrt</a>	Compute square root (function )

## /cmath/remainder

function **remainder** <cmath> <ctgmath>

```
double remainder (double numer      , double denom);
float remainderf (float numer      , float denom);
long double remainderl (long double numer, long double denom);

double remainder (double numer      , double denom);
float remainder (float numer      , float denom);
long double remainder (long double numer, long double denom);
double remainder (Type1 numer      , Type2 denom);           // additional overloads
```

### Compute remainder (IEC 60559)

Returns the floating-point remainder of *numer/denom* (rounded to nearest):

```
remainder = numer - rquot * denom
```

Where **rquot** is the result of: *numer/denom*, rounded toward the nearest integral value (with halfway cases rounded toward the even number).

A similar function, **fmod**, returns the same but with the quotient truncated (rounded towards zero) instead.

The function **remquo** has a behavior identical to this function, but it additionally provides access to the intermediate quotient value used.

Header **<tgmath.h>** provides a type-generic macro version of this function.

Additional overloads are provided in this header (**<cmath>**) for other combinations of arithmetic types (**Type1** and **Type2**): These overloads effectively cast its arguments to **double** before calculations, except if at least one of the arguments is of type **long double** (in which case both are casted to **long double** instead).

### Parameters

**numer**  
Value of the quotient numerator.

**denom**  
Value of the quotient denominator.

### Return Value

The remainder of dividing the arguments.

If this remainder is zero, its sign shall be that of *numer*.

If *denom* is zero, the function may either return zero or cause a *domain error* (depending on the library implementation).

If a *domain error* occurs:

- And **math\_errhandling** has **MATH\_ERRNO** set: the global variable **errno** is set to **EDOM**.
- And **math\_errhandling** has **MATH\_ERREXCEPT** set: **FE\_INVALID** is raised.

### Example

```
1 /* remainder example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* remainder */
4
5 int main ()
6 {
7     printf ( "remainder of 5.3 / 2 is %f\n", remainder (5.3,2) );
8     printf ( "remainder of 18.5 / 4.2 is %f\n", remainder (18.5,4.2) );
9     return 0;
10 }
```

Output:

```
remainder of 5.3 / 2 is -0.700000
remainder of 18.5 / 4.2 is 1.700000
```

### See also

<b>fmod</b>	Compute remainder of division (function )
<b>fabs</b>	Compute absolute value (function )
<b>round</b>	Round to nearest (function )

## /cmath/remquo

function **remquo** <cmath> <ctgmath>

```
double remquo (double numer      , double denom      , int* quot);
float remquof (float numer      , float denom      , int* quot);
long double remquol (long double numer, long double denom, int* quot);

double remquo (double numer      , double denom      , int* quot);
float remquo (float numer      , float denom      , int* quot);
long double remquo (long double numer, long double denom, int* quot);
double remquo (Type1 numer      , Type2 denom      , int* quot); // additional overloads
```

## Compute remainder and quotient

Returns the same as [remainder](#), but it additionally stores the quotient internally used to determine its result in the object pointed by *quot*.

The value pointed by *quot* contains the congruent modulo with at least 3 bits of the integral quotient *numer/denom*.

Header [`<tgmath.h>`](#) provides a type-generic macro version of this function.

Additional overloads are provided in this header ([`<cmath>`](#)) for other combinations of arithmetic types (Type1 and Type2): These overloads effectively cast its arguments to double before calculations, except if at least one of the arguments is of type long double (in which case both are casted to long double instead).

### Parameters

<code>numer</code>	Floating point value with the quotient numerator.
<code>denom</code>	Floating point value with the quotient denominator.
<code>quot</code>	Pointer to an object where the quotient internally used to determine the remainder is stored as a value of type <code>int</code> .

### Return Value

The remainder of dividing the arguments.

If this remainder is zero, its sign shall be that of *x*; In this case, the value stored in *quot* is unspecified.

If *denominator* is zero, the function may either return zero or cause a *domain error* (depending on the library implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

### Example

```
1 /* remquo example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* remquo */
4
5 int main ()
6 {
7     double numer = 10.3;
8     double denom = 4.5;
9     int quot;
10    double result = remquo (numer,denom,&quot);
11    printf ("numerator: %f\n", numer);
12    printf ("denominator: %f\n", denom);
13    printf ("remainder: %f\n", result);
14    printf ("quotient: %d\n", quot);
15    return 0;
16 }
```

Output:

```
numerator: 10.300000
denominator: 4.500000
remainder: 1.300000
quotient: 2
```

### See also

[remainder](#) Compute remainder (IEC 60559) (function )

[fmod](#) Compute remainder of division (function )

[fabs](#) Compute absolute value (function )

[round](#) Round to nearest (function )

## /cmath/rint

function

### rint

[`<cmath>`](#) [`<ctgmath>`](#)

```
double rint (double x);
float rintf (float x);
long double rintl (long double x);

double rint (double x);
float rint (float x);
long double rint (long double x);
double rint (T x);           // additional overloads for integral types
```

### Round to integral value

Rounds *x* to an integral value, using the rounding direction specified by [fegetround](#).

This function may raise an `FE_INEXACT` exception if the value returned differs in value from *x*. See [nearbyint](#) for an equivalent function that cannot raise such exception.

Header [`<tgmath.h>`](#) provides a type-generic macro version of this function.

Additional overloads are provided in this header ([`<cmath>`](#)) for the integral types: These overloads effectively cast *x* to a double before calculations (defined for *T* being any [integral type](#)).

## Parameters

x Value to round.

## Return Value

The value of x rounded to a nearby integral (as a floating-point value). If this value differs from x, a `FE_INEXACT` exception may be raised (depending on the implementation).

## Example

```
1 /* rint example */
2 #include <stdio.h>           /* printf */
3 #include <fenv.h>             /* fegetround, FE_* */
4 #include <math.h>              /* rint */
5
6 int main ()
7 {
8     printf ("rounding using ");
9     switch (fegetround()) {
10         case FE_DOWNTARD: printf ("downward"); break;
11         case FE_TONEAREST: printf ("to-nearest"); break;
12         case FE_TOWARDZERO: printf ("toward-zero"); break;
13         case FE_UPWARD: printf ("upward"); break;
14         default: printf ("unknown");
15     }
16     printf (" rint(%f)\n", rint(2.3));
17     printf (" rint(%f)\n", rint(3.8));
18     printf (" rint(%f)\n", rint(-2.3));
19     printf (" rint(%f)\n", rint(-3.8));
20
21     return 0;
22 }
```

Possible output:

```
Rounding using to-nearest rounding:
rint (2.3) = 2.0
rint (3.8) = 4.0
rint (-2.3) = -2.0
rint (-3.8) = -4.0
```

## See also

<a href="#">nearbyint</a>	Round to nearby integral value ( <a href="#">function</a> )
<a href="#">lrint</a>	Round and cast to long integer ( <a href="#">function</a> )
<a href="#">round</a>	Round to nearest ( <a href="#">function</a> )
<a href="#">floor</a>	Round down value ( <a href="#">function</a> )
<a href="#">ceil</a>	Round up value ( <a href="#">function</a> )
<a href="#">trunc</a>	Truncate value ( <a href="#">function</a> )

## /cmath/round

function

### round

`<cmath>` `<ctgmath>`

```
double round (double x);
float roundf (float x);
long double roundl (long double x);

double round (double x);
float round (float x);
long double round (long double x);
double round (T x);           // additional overloads for integral types
```

#### Round to nearest

Returns the integral value that is nearest to x, with halfway cases rounded away from zero.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast x to a double before calculations (defined for T being any [integral type](#)).

## Parameters

x Value to round.

## Return Value

The value of x rounded to the nearest integral (as a floating-point value).

## Example

```
1 /* round vs floor vs ceil vs trunc */
2 #include <stdio.h>           /* printf */
3 #include <math.h>             /* round, floor, ceil, trunc */
4
5 int main ()
6 {
7     const char * format = "% .1f \t% .1f \t% .1f \t% .1f \t% .1f\n";
8     printf ("value\tround\tfloor\tceil\ttrunc\n");
9     printf ("----\t----\t----\t----\t----\n");
10    printf (format, 2.3,round( 2.3),floor( 2.3),ceil( 2.3),trunc( 2.3));
11    printf (format, 3.8,round( 3.8),floor( 3.8),ceil( 3.8),trunc( 3.8));
12    printf (format, 5.5,round( 5.5),floor( 5.5),ceil( 5.5),trunc( 5.5));
13    printf (format,-2.3,round(-2.3),floor(-2.3),ceil(-2.3),trunc(-2.3));
14    printf (format,-3.8,round(-3.8),floor(-3.8),ceil(-3.8),trunc(-3.8));
15    printf (format,-5.5,round(-5.5),floor(-5.5),ceil(-5.5),trunc(-5.5));
16    return 0;
17 }
```

Output:

value	round	floor	ceil	trunc
2.3	2.0	2.0	3.0	2.0
3.8	4.0	3.0	4.0	3.0
5.5	6.0	5.0	6.0	5.0
-2.3	-2.0	-3.0	-2.0	-2.0
-3.8	-4.0	-4.0	-3.0	-3.0
-5.5	-6.0	-6.0	-5.0	-5.0

## See also

<a href="#">floor</a>	Round down value (function )
<a href="#">ceil</a>	Round up value (function )
<a href="#">trunc</a>	Truncate value (function )
<a href="#">nearbyint</a>	Round to nearby integral value (function )
<a href="#">rint</a>	Round to integral value (function )

## /cmath/scalbln

function **scalbln** <cmath> <ctgmath>

```
double scalbln (double x      , long int n);
float scalblnf (float x       , long int n);
long double scalblnl (long double x, long int n);

double scalbln (double x      , long int n);
float scalbln (float x       , long int n);
long double scalbln (long double x, long int n);
double scalbln (T x          , long int n); // additional overloads for integral types
```

### Scale significand using floating-point base exponent (long)

Scales *x* by `FLOAT_RADIX` raised to the power of *n*, returning the result of computing:

$$\text{scalbn}(x, n) = x * \text{FLOAT_RADIX}^n$$

Presumably, *x* and *n* are the components of a floating-point number in the system; In such a case, this function may be optimized to be more efficient than the theoretical operations to compute the value explicitly.

There also exists another version of this function: `scalbn`, which is identical, except that it takes an `int` as second argument.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast *x* to a `double` before calculations (defined for *T* being any `integral type`).

## Parameters

*Value* representing the *significand*.

*exp*

Value of the *exponent*.

## Return Value

Returns  $x * \text{FLOAT_RADIX}^n$ .

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* may occur (if too small, the function returns zero, and an underflow *range error* may occur).

If a *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: either `FE_OVERFLOW` or `FE_UNDERFLOW` is raised.

## Example

```

1 /* scalbn example */
2 #include <stdio.h>      /* printf */
3 #include <float.h>        /* FLT_RADIX */
4 #include <math.h>         /* scalbn */
5
6 int main ()
7 {
8     double param, result;
9     long n;
10
11    param = 1.50;
12    n = 4L;
13    result = scalbn (param , n);
14    printf ("%f * %d^%d = %f\n", param, FLT_RADIX, n, result);
15    return 0;
16 }

```

Output:

```
1.500000 * 2^4 = 24.000000
```

## See also

<a href="#">scalbn</a>	Scale significand using floating-point base exponent (function )
<a href="#">ldexp</a>	Generate value from significand and exponent (function )
<a href="#">logb</a>	Compute floating-point base logarithm (function )

## /cmath/scalbn

function

### scalbn

<cmath> <ctgmath>

```

double scalbn (double x      , int n);
float scalbnf (float x       , int n);
long double scalbnl (long double x, int n);

double scalbn (double x      , int n);
float scalbn (float x       , int n);
long double scalbn (long double x, int n);
double scalbn (T x           , int n); // additional overloads for integral types

```

#### Scale significand using floating-point base exponent

Scales *x* by `FLT_RADIX` raised to the power of *n*, returning the same as:

`scalbn(x,n) = x * FLT_RADIXn`

Presumably, *x* and *n* are the components of a floating-point number in the system; In such a case, this function may be optimized to be more efficient than the theoretical operations to compute the value explicitly.

On most platforms, `FLT_RADIX` is 2, making this function equivalent to `ldexp`.

Header `<tgmath.h>` provides a type-generic macro version of this function.

Additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast *x* to a double before calculations (defined for *T* being any *integral type*).

There also exists another version of this function: `scalbln`, which is identical, except that it takes a `long int` as second argument.

## Parameters

*x* Value representing the *significand*.

*exp* Value of the *exponent*.

## Return Value

Returns  $x * \text{FLT\_RADIX}^n$ .

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* may occur (if too small, the function returns zero, and an underflow *range error* may occur).

If a *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: either `FE_OVERFLOW` or `FE_UNDERFLOW` is raised.

## Example

```

1 /* scalbn example */
2 #include <stdio.h>      /* printf */
3 #include <float.h>        /* FLT_RADIX */
4 #include <math.h>         /* scalbn */
5
6 int main ()
7 {
8     double param, result;
9     int n;
10
11    param = 1.50;
12    n = 4L;
13    result = scalbn (param , n);
14    printf ("%f * %d^%d = %f\n", param, FLT_RADIX, n, result);
15    return 0;
16 }

```

```

11 param = 1.50;
12 n = 4;
13 result = scalbn (param , n);
14 printf ("%f * %d^%d = %f\n", param, FLT_RADIX, n, result);
15 return 0;
16 }

```

Output:

```
1.500000 * 2^4 = 24.000000
```

## See also

<a href="#">ldexp</a>	Generate value from significand and exponent (function )
<a href="#">logb</a>	Compute floating-point base logarithm (function )

## /cmath/signbit

macro/function

### signbit

<cmath> <ctgmath>

macro signbit(x)
bool signbit (float x);
function bool signbit (double x);
bool signbit (long double x);

#### Sign bit

Returns whether the sign of *x* is negative.

This can be also applied to *infinities*, *NaNs* and *zeroes* (if zero is unsigned, it is considered positive).

In C, this is implemented as a macro that returns an *int* value. The type of *x* shall be *float*, *double* or *long double*.

In C++, it is implemented with function overloads for each *floating-point type*, each returning a *bool* value.

## Parameters

*x*

A floating-point value.

## Return value

A non-zero value (*true*) if the sign of *x* is negative; and zero (*false*) otherwise.

## Example

```

1 /* signbit example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>        /* signbit, sqrt */
4
5 int main()
6 {
7     printf ("signbit(0.0)      : %d\n",signbit(0.0));
8     printf ("signbit(1.0/0.0)   : %d\n",signbit(1.0/0.0));
9     printf ("signbit(-1.0/0.0)  : %d\n",signbit(-1.0/0.0));
10    printf ("signbit(sqrt(-1.0)): %d\n",signbit(sqrt(-1.0)));
11    return 0;
12 }

```

Output:

```
signbit(0.0)      : 0
signbit(1.0/0.0)   : 0
signbit(-1.0/0.0)  : 1
signbit(sqrt(-1.0): 1
```

## See also

<a href="#">isinf</a>	Is infinity (macro/function )
<a href="#">isnormal</a>	Is normal (macro/function )
<a href="#">isnan</a>	Is Not-A-Number (macro/function )
<a href="#">fpclassify</a>	Classify floating-point value (macro/function )

## /cmath/sin

function

### sin

<cmath> <ctgmath>

```
double sin(double x);
```

```

double sin (double x);
float sinf (float x);
long double sinl (long double x);
double sin (double x);
float sin (float x);
long double sin (long double x);
double sin (T x);           // additional overloads for integral types

```

### Compute sine

Returns the sine of an angle of  $x$  radians.

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see [complex sin](#) and [valarray sin](#)).

*Additional overloads* are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).

This function is also overloaded in `<complex>` and `<valarray>` (see [complex sin](#) and [valarray sin](#)).

### Parameters

$x$

Value representing an angle expressed in radians.  
One radian is equivalent to  $180/\pi$  degrees.

### Return Value

Sine of  $x$  radians.

### Example

```

1 /* sin example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>          /* sin */
4
5 #define PI 3.14159265
6
7 int main ()
8 {
9     double param, result;
10    param = 30.0;
11    result = sin (param*PI/180);
12    printf ("The sine of %f degrees is %f.\n", param, result );
13    return 0;
14 }

```

Output:

```
The sine of 30.000000 degrees is 0.500000.
```

### See also

<a href="#">cos</a>	Compute cosine (function)
<a href="#">tan</a>	Compute tangent (function)

## /cmath/sinh

function

### sinh

`<cmath>` `<ctgmath>`

```

double sinh (double x);
double sinh (double x);
float sinhf (float x);
long double sinhl (long double x);
double sinh (double x);
float sinh (float x);
long double sinh (long double x);
double sinh (T x);           // additional overloads for integral types

```

### Compute hyperbolic sine

Returns the hyperbolic sine of  $x$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see [complex sinh](#) and [valarray sinh](#)).

*Additional overloads* are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).

This function is also overloaded in `<complex>` and `<valarray>` (see [complex sinh](#) and [valarray sinh](#)).

## Parameters

- x Value representing a hyperbolic angle.

## Return Value

Hyperbolic sine of x.

If the magnitude of the result is too large to be represented by a value of the return type, the function returns `HUGE_VAL` (or `HUGE_VALF` or `HUGE_VALL`) with the proper sign, and an overflow *range error* occurs:

If an overflow *range error* occurs, the global variable `errno` is set to `ERANGE`.

If an overflow *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_OVERFLOW` is raised.

## Example

```
1 /* sinh example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* sinh, log */
4
5 int main ()
6 {
7     double param, result;
8     param = log(2.0);
9     result = sinh (param);
10    printf ("The hyperbolic sine of %f is %f.\n", param, result );
11    return 0;
12 }
```

Output:

```
The hyperbolic sine of 0.693147 is 0.750000.
```

## See also

<a href="#">cosh</a>	Compute hyperbolic cosine (function )
<a href="#">tanh</a>	Compute hyperbolic tangent (function )

# /cmath/sqrt

function

## sqrt

`<cmath> <ctgmath>`

```
double sqrt (double x);
double sqrt (double x);
float sqrtf (float x);
long double sqrtl (long double x);

double sqrt (double x);
float sqrt (float x);
long double sqrt (long double x);

double sqrt (double x);
float sqrt (float x);
long double sqrt (long double x);
double sqrt (T x);           // additional overloads for integral types
```

### Compute square root

Returns the *square root* of x.

Header `<tgmath.h>` provides a type-generic macro version of this function.

This function is overloaded in `<complex>` and `<valarray>` (see `complex sqrt` and `valarray sqrt`).

Additional overloads are provided in this header (`<cmath>`) for the *integral types*: These overloads effectively cast x to a `double` before calculations (defined for T being any *integral type*).

This function is also overloaded in `<complex>` and `<valarray>` (see `complex sqrt` and `valarray sqrt`).

## Parameters

- x Value whose square root is computed.  
If the argument is negative, a *domain error* occurs.

## Return Value

Square root of x.

If x is negative, a *domain error* occurs:

If a *domain error* occurs, the global variable `errno` is set to `EDOM`.

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

## Example

```
1 /* sqrt example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* sqrt */
4
5 int main ()
6 {
7     double param, result;
8     param = 1024.0;
9     result = sqrt (param);
10    printf ("sqrt(%f) = %f\n", param, result );
11    return 0;
12 }
```

Output:

```
sqrt(1024.000000) = 32.000000
```

## See also

[pow](#) | Raise to power (function )

[log](#) | Compute natural logarithm (function )

## /cmath/tan

function

### **tan**

<cmath> <ctgmath>

```
double tan (double x);
double tan (double x);
float tanf (float x);
long double tanl (long double x);

double tan (double x);
float tan (float x);
long double tan (long double x);

double tan (double x);
float tan (float x);
long double tan (long double x);
double tan (T x);           // additional overloads for integral types
```

#### Compute tangent

Returns the tangent of an angle of  $x$  radians.

Header [`<tgmath.h>`](#) provides a type-generic macro version of this function.

This function is overloaded in [`<complex>`](#) and [`<valarray>`](#) (see [complex tan](#) and [valarray tan](#)).

Additional overloads are provided in this header ([`<cmath>`](#)) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).

This function is also overloaded in [`<complex>`](#) and [`<valarray>`](#) (see [complex tan](#) and [valarray tan](#)).

## Parameters

**x**

Value representing an angle, expressed in radians.  
One *radian* is equivalent to  $180/\pi$  degrees.

## Return Value

Tangent of  $x$  radians.

## Example

```
1 /* tan example */
2 #include <stdio.h>      /* printf */
3 #include <math.h>         /* tan */
4
5 #define PI 3.14159265
6
7 int main ()
8 {
9     double param, result;
10    param = 45.0;
11    result = tan ( param * PI / 180.0 );
12    printf ("The tangent of %f degrees is %f.\n", param, result );
13    return 0;
14 }
```

Output:

```
The tangent of 45.000000 degrees is 1.000000.
```

## See also

<a href="#">sin</a>	Compute sine (function )
<a href="#">cos</a>	Compute cosine (function )
<a href="#">atan</a>	Compute arc tangent (function )

## /cmath/tanh

function	
<b>tanh</b>	<cmath> <ctgmath>
<pre>double tanh (double x); double tanh (double x); float tanhf (float x); long double tanhl (long double x);  double tanh (double x); float tanh (float x); long double tanh (long double x); double tanh (T x);           // additional overloads for integral types</pre>	

### Compute hyperbolic tangent

Returns the hyperbolic tangent of  $x$ .

Header <a href="#"><code>&lt;tgmath.h&gt;</code></a> provides a type-generic macro version of this function.
This function is overloaded in <a href="#"><code>&lt;complex&gt;</code></a> and <a href="#"><code>&lt;valarray&gt;</code></a> (see <a href="#">complex tanh</a> and <a href="#">valarray tanh</a> ).
<i>Additional overloads</i> are provided in this header ( <a href="#"><code>&lt;cmath&gt;</code></a> ) for the integral types: These overloads effectively cast $x$ to a double before calculations (defined for $T$ being any <a href="#">integral type</a> ).
This function is also overloaded in <a href="#"><code>&lt;complex&gt;</code></a> and <a href="#"><code>&lt;valarray&gt;</code></a> (see <a href="#">complex tanh</a> and <a href="#">valarray tanh</a> ).

### Parameters

$x$	Value representing a hyperbolic angle.
-----	--

### Return Value

Hyperbolic tangent of  $x$ .

### Example

```
1 /* tanh example */
2 #include <stdio.h>          /* printf */
3 #include <math.h>             /* tanh, log */
4
5 int main ()
6 {
7     double param, result;
8     param = log(2.0);
9     result = tanh (param);
10    printf ("The hyperbolic tangent of %f is %f.\n", param, result);
11    return 0;
12 }
```

Output:

```
The hyperbolic tangent of 0.693147 is 0.600000.
```

## See also

<a href="#">sinh</a>	Compute hyperbolic sine (function )
<a href="#">cosh</a>	Compute hyperbolic cosine (function )

## /cmath/tgamma

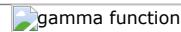
function	
<b>tgamma</b>	<cmath> <ctgmath>
<pre>double tgamma (      double x); float tgammaf (      float x); long double tgammal (long double x);  double tgamma (      double x); float tgamma (      float x); long double tgamma (long double x); double tgamma (T x);           // additional overloads for integral types</pre>	

### Compute gamma function

Returns the *gamma function* of  $x$ .

Header <a href="#"><code>&lt;tgmath.h&gt;</code></a> provides a type-generic macro version of this function.
--

*Additional overloads* are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).



## Parameters

$x$

Parameter for the *gamma function*.

## Return Value

Gamma function of  $x$ .

If the magnitude of  $x$  is too large, an overflow *range error* occurs. If too small, an underflow *range error* may occur.

If  $x$  is zero or a negative integer for which the function is asymptotic, it may cause a *domain error* or a *pole error* (or none, depending on implementation).

If a *domain error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `EDOM`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: `FE_INVALID` is raised.

If a *range error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: either `FE_OVERFLOW` or `FE_UNDERFLOW` is raised.

If a *pole error* occurs:

- And `math_errhandling` has `MATH_ERRNO` set: the global variable `errno` is set to `ERANGE`.
- And `math_errhandling` has `MATH_ERREXCEPT` set: either `FE_DIVBYZERO` is raised.

## Example

```
1 /* tgamma example */
2 #include <stdio.h>           /* printf */
3 #include <math.h>              /* tgamma */
4
5 int main ()
6 {
7     double param, result;
8     param = 0.5;
9     result = tgamma (param);
10    printf ("tgamma(%f) = %f\n", param, result );
11    return 0;
12 }
```

Output:

```
tgamma (0.500000) = 1.772454
```

## See also

<a href="#">lgamma</a>	Compute log-gamma function ( <a href="#">function</a> )
<a href="#">erf</a>	Compute error function ( <a href="#">function</a> )
<a href="#">erfc</a>	Compute complementary error function ( <a href="#">function</a> )

# /cmath/trunc

function

## trunc

`<cmath> <ctgmath>`

```
double trunc (      double x);
float truncf (     float x);
long double truncl (long double x);

double trunc (      double x);
float trunc (     float x);
long double trunc (long double x);
double trunc (T x);           // additional overloads for integral types
```

### Truncate value

Rounds  $x$  toward zero, returning the nearest integral value that is not larger in magnitude than  $x$ .

Header `<tgmath.h>` provides a type-generic macro version of this function.

*Additional overloads* are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast  $x$  to a double before calculations (defined for  $T$  being any [integral type](#)).

## Parameters

$x$

Value to truncate.

## Return Value

The nearest integral value that is not larger in magnitude than  $x$  (as a floating-point value).

## Example

```
1 /* round vs floor vs ceil vs trunc */
2 #include <stdio.h>           /* printf */
3 #include <math.h>             /* round, floor, ceil, trunc */
4
5 int main ()
6 {
7     const char * format = "%.1f \t%.1f \t%.1f \t%.1f \t%.1f\n";
8     printf ("value\tround\tfloor\tceil\ttrunc\n");
9     printf ("-----\t-----\t-----\t-----\n");
10    printf (format, 2.3,round( 2.3),floor( 2.3),ceil( 2.3),trunc( 2.3));
11    printf (format, 3.8,round( 3.8),floor( 3.8),ceil( 3.8),trunc( 3.8));
12    printf (format, 5.5,round( 5.5),floor( 5.5),ceil( 5.5),trunc( 5.5));
13    printf (format,-2.3,round(-2.3),floor(-2.3),ceil(-2.3),trunc(-2.3));
14    printf (format,-3.8,round(-3.8),floor(-3.8),ceil(-3.8),trunc(-3.8));
15    printf (format,-5.5,round(-5.5),floor(-5.5),ceil(-5.5),trunc(-5.5));
16    return 0;
17 }
```

Output:

value	round	floor	ceil	trunc
2.3	2.0	2.0	3.0	2.0
3.8	4.0	3.0	4.0	3.0
5.5	6.0	5.0	6.0	5.0
-2.3	-2.0	-3.0	-2.0	-2.0
-3.8	-4.0	-4.0	-3.0	-3.0
-5.5	-6.0	-6.0	-5.0	-5.0

## See also

<a href="#">floor</a>	Round down value (function )
<a href="#">ceil</a>	Round up value (function )
<a href="#">round</a>	Round to nearest (function )
<a href="#">rint</a>	Round to integral value (function )

## /cstdio

header

### <cstdio> (stdio.h)

#### C library to perform Input/Output operations

Input and Output operations can also be performed in C++ using the **C Standard Input and Output Library** (**cstdio**, known as **stdio.h** in the C language). This library uses what are called *streams* to operate with physical devices such as keyboards, printers, terminals or with any other type of files supported by the system. Streams are an abstraction to interact with these in an uniform way; All streams have similar properties independently of the individual characteristics of the physical media they are associated with.

Streams are handled in the **cstdio** library as pointers to **FILE** objects. A pointer to a **FILE** object uniquely identifies a stream, and is used as a parameter in the operations involving that stream.

There also exist three standard streams: **stdin**, **stdout** and **stderr**, which are automatically created and opened for all programs using the library.

#### Stream properties

Streams have some properties that define which functions can be used on them and how these will treat the data input or output through them. Most of these properties are defined at the moment the stream is associated with a file (opened) using the **fopen** function:

##### Read/Write Access

Specifies whether the stream has read or write access (or both) to the physical media they are associated with.

##### Text / Binary

Text streams are thought to represent a set of text lines, each one ending with a new-line character. Depending on the environment where the application is run, some character translation may occur with text streams to adapt some special characters to the text file specifications of the environment. A binary stream, on the other hand, is a sequence of characters written or read from the physical media with no translation, having a one-to-one correspondence with the characters read or written to the stream.

##### Buffer

A buffer is a block of memory where data is accumulated before being physically read or written to the associated file or device. Streams can be either *fully buffered*, *line buffered* or *unbuffered*. On fully buffered streams, data is read/written when the buffer is filled, on line buffered streams this happens when a new-line character is encountered, and on unbuffered streams characters are intended to be read/written as soon as possible.

##### Orientation

On opening, streams have no orientation. As soon as an input/output operation is performed on them, they become either *byte-oriented* or *wide-oriented*, depending on the operation performed (generally, functions defined in **<cstdio>** are *byte-oriented*, while functions in **<cwchar>** are *wide-oriented*). See **cwchar** for more info.

#### Indicators

Streams have certain internal indicators that specify their current state and which affect the behavior of some input and output operations performed on them:

##### Error indicator

This indicator is set when an error has occurred in an operation related to the stream. This indicator can be checked with the **ferror** function, and can be reset by calling either to **clearerr**, **freopen** or **rewind**.

##### End-Of-File indicator

When set, indicates that the last reading or writing operation performed with the stream reached the *End of File*. It can be checked with the **feof** function, and can be reset by calling either to **clearerr** or **freopen** or by calling to any repositioning function (**rewind**, **fseek** and **fsetpos**).

##### Position indicator

It is an internal pointer of each stream which points to the next character to be read or written in the next I/O operation. Its value can be obtained by the **f<sub>tell</sub>** and **fgetpos** functions, and can be changed using the repositioning functions **rewind**, **fseek** and **fsetpos**.

## Functions

### Operations on files:

<b>remove</b>	Remove file (function )
<b>rename</b>	Rename file (function )
<b>tmpfile</b>	Open a temporary file (function )
<b>tmpnam</b>	Generate temporary filename (function )

### File access:

<b>fclose</b>	Close file (function )
<b>fflush</b>	Flush stream (function )
<b>fopen</b>	Open file (function )
<b>freopen</b>	Reopen stream with different file or mode (function )
<b>setbuf</b>	Set stream buffer (function )
<b>setvbuf</b>	Change stream buffering (function )

### Formatted input/output:

<b>fprintf</b>	Write formatted data to stream (function )
<b>fscanf</b>	Read formatted data from stream (function )
<b>printf</b>	Print formatted data to stdout (function )
<b>scanf</b>	Read formatted data from stdin (function )
<b>sprintf</b>	Write formatted output to sized buffer (function )
<b>fprintf</b>	Write formatted data to string (function )
<b>sscanf</b>	Read formatted data from string (function )
<b>vfprintf</b>	Write formatted data from variable argument list to stream (function )
<b>vfscanf</b>	Read formatted data from stream into variable argument list (function )
<b>vprintf</b>	Print formatted data from variable argument list to stdout (function )
<b>vscanf</b>	Read formatted data into variable argument list (function )
<b>vsnprintf</b>	Write formatted data from variable argument list to sized buffer (function )
<b>vsprintf</b>	Write formatted data from variable argument list to string (function )
<b>vscanf</b>	Read formatted data from string into variable argument list (function )

### Character input/output:

<b>fgetc</b>	Get character from stream (function )
<b>fgets</b>	Get string from stream (function )
<b>fputc</b>	Write character to stream (function )
<b>fputs</b>	Write string to stream (function )
<b>getc</b>	Get character from stream (function )
<b>getchar</b>	Get character from stdin (function )
<b>gets</b>	Get string from stdin (function )
<b>putc</b>	Write character to stream (function )
<b>putchar</b>	Write character to stdout (function )
<b>puts</b>	Write string to stdout (function )
<b>ungetc</b>	Unget character from stream (function )

### Direct input/output:

<b>fread</b>	Read block of data from stream (function )
<b>fwrite</b>	Write block of data to stream (function )

### File positioning:

<b>fgetpos</b>	Get current position in stream (function )
<b>fseek</b>	Reposition stream position indicator (function )
<b>fsetpos</b>	Set position indicator of stream (function )
<b>f<sub>tell</sub></b>	Get current position in stream (function )
<b>rewind</b>	Set position of stream to the beginning (function )

### Error-handling:

<b>clearerr</b>	Clear error indicators (function )
<b>feof</b>	Check end-of-file indicator (function )
<b>ferror</b>	Check error indicator (function )
<b>perror</b>	Print error message (function )

## Macros

<b>BUFSIZ</b>	Buffer size (constant )
<b>EOF</b>	End-of-File (constant )

<b>FILENAME_MAX</b>	Maximum length of file names ( <a href="#">constant</a> )
<b>FOPEN_MAX</b>	Potential limit of simultaneous open streams ( <a href="#">constant</a> )
<b>L_tmpnam</b>	Minimum length for temporary file name ( <a href="#">constant</a> )
<b>NULL</b>	Null pointer ( <a href="#">macro</a> )
<b>TMP_MAX</b>	Number of temporary files ( <a href="#">constant</a> )

Additionally: `_IOFBF`, `_IOLBF`, `_IONBF` (used with `setvbuf`)  
and `SEEK_CUR`, `SEEK_END` and `SEEK_SET` (used with `fseek`).

## Types

<b>FILE</b>	Object containing information to control a stream ( <a href="#">type</a> )
<b>fpos_t</b>	Object containing information to specify a position within a file ( <a href="#">type</a> )
<b>size_t</b>	Unsigned integral type ( <a href="#">type</a> )

## /cstdio/BUFSIZ

constant

## BUFSIZ

<cstdio>

### Buffer size

This macro constant expands to an integral expression with the size of the buffer used by the `setbuf` function.

## See also

<a href="#">setbuf</a>	Set stream buffer ( <a href="#">function</a> )
------------------------	--

## /cstdio/clearerr

function

## clearerr

<cstdio>

`void clearerr ( FILE * stream );`

### Clear error indicators

Resets both the *error* and the *eof* indicators of the *stream*.

When a i/o function fails either because of an error or because the end of the file has been reached, one of these internal indicators may be set for the *stream*. The state of these indicators is cleared by a call to this function, or by a call to any of: `rewind`, `fseek`, `fsetpos` and `freopen`.

## Parameters

**stream**

Pointer to a `FILE` object that identifies the stream.

## Return Value

None

## Example

```

1 /* writing errors */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile;
6     pFile = fopen("myfile.txt","r");
7     if (pFile==NULL) perror ("Error opening file");
8     else {
9         fputc ('x',pFile);
10        if (ferror (pFile)) {
11            printf ("Error Writing to myfile.txt\n");
12            clearerr (pFile);
13        }
14        fgetc (pFile);
15        if (!ferror (pFile))
16            printf ("No errors reading myfile.txt\n");
17        fclose (pFile);
18    }
19    return 0;
20 }
```

This program opens an existing file called `myfile.txt` for reading and causes an I/O error trying to write on it. That error is cleared using `clearerr`, so a second error checking returns false.

Output:

```
Error Writing to myfile.txt
No errors reading myfile.txt
```

## See also

<b>feof</b>	Check end-of-file indicator (function )
<b>ferror</b>	Check error indicator (function )
<b>rewind</b>	Set position of stream to the beginning (function )

## /cstdio/EOF

constant

### EOF

<cstdio>

#### End-of-File

It is a macro definition of type `int` that expands into a negative integral constant expression (generally, `-1`).

It is used as the value returned by several functions in header `<cstdio>` to indicate that the End-of-File has been reached or to signal some other failure conditions.

It is also used as the value to represent an invalid character.

In C++, this macro corresponds to the value of `char_traits<char>::eof()`.

#### See also

<b>feof</b>	Check end-of-file indicator (function )
<b>ferror</b>	Check error indicator (function )

## /cstdio/fclose

function

### fclose

<cstdio>

```
int fclose ( FILE * stream );
```

#### Close file

Closes the file associated with the `stream` and disassociates it.

All internal buffers associated with the stream are disassociated from it and flushed: the content of any unwritten output buffer is written and the content of any unread input buffer is discarded.

Even if the call fails, the stream passed as parameter will no longer be associated with the file nor its buffers.

#### Parameters

`stream`

Pointer to a `FILE` object that specifies the stream to be closed.

#### Return Value

If the stream is successfully closed, a zero value is returned.

On failure, `EOF` is returned.

#### Example

```
1 /* fclose example */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile;
6     pFile = fopen ("myfile.txt","wt");
7     fprintf (pFile, "fclose example");
8     fclose (pFile);
9     return 0;
10 }
```

This simple code creates a new text file, then writes a sentence to it, and then closes it.

#### See also

<b>fopen</b>	Open file (function )
<b>fflush</b>	Flush stream (function )

## /cstdio/feof

function

### feof

<cstdio>

```
int feof ( FILE * stream );
```

#### Check end-of-file indicator

Checks whether the *end-of-File indicator* associated with `stream` is set, returning a value different from zero if it is.

This indicator is generally set by a previous operation on the *stream* that attempted to read at or past the end-of-file.

Notice that *stream*'s internal position indicator may point to the *end-of-file* for the next operation, but still, the *end-of-file* indicator may not be set until an operation attempts to read at that point.

This indicator is cleared by a call to `clearerr`, `rewind`, `fseek`, `fsetpos` or `freopen`. Although if the *position indicator* is not repositioned by such a call, the next i/o operation is likely to set the indicator again.

## Parameters

**stream**  
Pointer to a `FILE` object that identifies the stream.

## Return Value

A non-zero value is returned in the case that the *end-of-file indicator* associated with the stream is set.  
Otherwise, zero is returned.

## Example

```
1 /* feof example: byte counter */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     int n = 0;
8     pFile = fopen ("myfile.txt","rb");
9     if (pFile==NULL) perror ("Error opening file");
10    else
11    {
12        while (fgetc(pFile) != EOF) {
13            ++n;
14        }
15        if (feof(pFile)) {
16            puts ("End-of-File reached.");
17            printf ("Total number of bytes read: %d\n", n);
18        }
19        else puts ("End-of-File was not reached.");
20        fclose (pFile);
21    }
22    return 0;
23 }
```

This code opens the file called `myfile.txt`, and counts the number of characters that it contains by reading all of them one by one. The program checks whether the end-of-file was reached, and if so, prints the total number of bytes read.

## See also

<code>clearerr</code>	Clear error indicators ( <a href="#">function</a> )
<code>ferror</code>	Check error indicator ( <a href="#">function</a> )

## /cstdio/ferror

function

## ferror

`<cstdio>`

`int ferror ( FILE * stream );`

### Check error indicator

Checks if the *error indicator* associated with *stream* is set, returning a value different from zero if it is.

This indicator is generally set by a previous operation on the *stream* that failed, and is cleared by a call to `clearerr`, `rewind` or `freopen`.

## Parameters

**stream**  
Pointer to a `FILE` object that identifies the stream.

## Return Value

A non-zero value is returned in the case that the *error indicator* associated with the stream is set.  
Otherwise, zero is returned.

## Example

```
1 /* ferror example: writing error */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile;
6     pFile=fopen("myfile.txt","r");
7     if (pFile==NULL) perror ("Error opening file");
8     else {
9         fputc ('x',pFile);
10        if (ferror (pFile))
```

```

11     printf ("Error Writing to myfile.txt\n");
12     fclose (pFile);
13 }
14 return 0;
15 }
```

This program opens an existing file called `myfile.txt` in read-only mode but tries to write a character to it, generating an error that is detected by `ferror`.

Output:

```
Error Writing to myfile.txt
```

## See also

<a href="#">feof</a>	Check end-of-file indicator (function )
<a href="#">clearerr</a>	Clear error indicators (function )
<a href="#">perror</a>	Print error message (function )

## /cstdio/fflush

function

### fflush

<cstdio>

```
int fflush ( FILE * stream );
```

#### Flush stream

If the given `stream` was open for writing (or if it was open for updating and the last i/o operation was an output operation) any unwritten data in its output buffer is written to the file.

If `stream` is a null pointer, all such streams are flushed.

In all other cases, the behavior depends on the specific library implementation. In some implementations, flushing a stream open for reading causes its input buffer to be cleared (but this is not portable expected behavior).

The stream remains open after this call.

When a file is closed, either because of a call to `fclose` or because the program terminates, all the buffers associated with it are automatically flushed.

## Parameters

`stream`

Pointer to a `FILE` object that specifies a buffered stream.

## Return Value

A zero value indicates success.

If an error occurs, `EOF` is returned and the error indicator is set (see `ferror`).

## Example

In files open for update (i.e., open for both reading and writing), the stream shall be flushed after an output operation before performing an input operation. This can be done either by repositioning (`fseek`, `fsetpos`, `rewind`) or by calling explicitly `fflush`, like in this example:

```

1 /* fflush example */
2 #include <stdio.h>
3 char mybuffer[80];
4 int main()
5 {
6     FILE * pFile;
7     pFile = fopen ("example.txt", "r+");
8     if (pFile == NULL) perror ("Error opening file");
9     else {
10         fputs ("test",pFile);
11         fflush (pFile);    // flushing or repositioning required
12         fgets (mybuffer,80,pFile);
13         puts (mybuffer);
14         fclose (pFile);
15         return 0;
16     }
17 }
```

## See also

<a href="#">fclose</a>	Close file (function )
<a href="#">fopen</a>	Open file (function )
<a href="#">setbuf</a>	Set stream buffer (function )
<a href="#">setvbuf</a>	Change stream buffering (function )

## /cstdio/fgetc

function

<cstdio>

## fgetc

```
int fgetc ( FILE * stream );
```

### Get character from stream

Returns the character currently pointed by the internal file position indicator of the specified *stream*. The internal file position indicator is then advanced to the next character.

If the stream is at the end-of-file when called, the function returns **EOF** and sets the *end-of-file indicator* for the stream (**feof**).

If a read error occurs, the function returns **EOF** and sets the *error indicator* for the stream (**ferror**).

**fgetc** and **getc** are equivalent, except that **getc** may be implemented as a macro in some libraries.

### Parameters

**stream**

Pointer to a **FILE** object that identifies an input stream.

### Return Value

On success, the character read is returned (promoted to an **int** value).

The return type is **int** to accommodate for the special value **EOF**, which indicates failure:

If the position indicator was at the *end-of-file*, the function returns **EOF** and sets the *eof indicator* (**feof**) of *stream*.

If some other reading error happens, the function also returns **EOF**, but sets its *error indicator* (**ferror**) instead.

### Example

```
1 /* fgetc example: money counter */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile;
6     int c;
7     int n = 0;
8     pFile=fopen ("myfile.txt","r");
9     if (pFile==NULL) perror ("Error opening file");
10    else
11    {
12        do {
13            c = fgetc (pFile);
14            if (c == '$') n++;
15        } while (c != EOF);
16        fclose (pFile);
17        printf ("The file contains %d dollar sign characters ($).\n",n);
18    }
19    return 0;
20 }
```

This program reads an existing file called **myfile.txt** character by character and uses the **n** variable to count how many dollar characters (\$) the file contains.

### See also

<b>getc</b>	Get character from stream (function )
<b>fputc</b>	Write character to stream (function )
<b>fread</b>	Read block of data from stream (function )
<b>fscanf</b>	Read formatted data from stream (function )

## /cstdio/fgetpos

function

## fgetpos

<cstdio>

```
int fgetpos ( FILE * stream, fpos_t * pos );
```

### Get current position in stream

Retrieves the current position in the *stream*.

The function fills the **fpos\_t** object pointed by *pos* with the information needed from the *stream's position indicator* to restore the *stream* to its current position (and multibyte state, if *wide-oriented*) with a call to **fsetpos**.

The **f tell** function can be used to retrieve the current position in the *stream* as an integer value.

### Parameters

**stream**

Pointer to a **FILE** object that identifies the stream.

**pos**

Pointer to a **fpos\_t** object.

This should point to an object already allocated.

### Return Value

On success, the function returns zero.

In case of error, **errno** is set to a platform-specific positive value and the function returns a non-zero value.

## Example

```
1 /* fgetpos example */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile;
6     int c;
7     int n;
8     fpos_t pos;
9
10    pFile = fopen ("myfile.txt", "r");
11    if (pFile==NULL) perror ("Error opening file");
12    else
13    {
14        c = fgetc (pFile);
15        printf ("1st character is %c\n",c);
16        fgetpos (pFile,&pos);
17        for (n=0;n<3;n++)
18        {
19            fsetpos (pFile,&pos);
20            c = fgetc (pFile);
21            printf ("2nd character is %c\n",c);
22        }
23        fclose (pFile);
24    }
25    return 0;
26 }
```

Possible output (with myfile.txt containing ABC):

```
1st character is A
2nd character is B
2nd character is B
2nd character is B
```

The example opens `myfile.txt`, then reads the first character once, and then reads 3 times the same second character.

## See also

<a href="#">fsetpos</a>	Set position indicator of stream ( <a href="#">function</a> )
<a href="#">ftell</a>	Get current position in stream ( <a href="#">function</a> )
<a href="#">fseek</a>	Reposition stream position indicator ( <a href="#">function</a> )

## /cstdio/fgets

function  
**fgets** <cstdio>

```
char * fgets ( char * str, int num, FILE * stream );
```

### Get string from stream

Reads characters from *stream* and stores them as a C string into *str* until (*num*-1) characters have been read or either a newline or the *end-of-file* is reached, whichever happens first.

A newline character makes `fgets` stop reading, but it is considered a valid character by the function and included in the string copied to *str*.

A terminating null character is automatically appended after the characters copied to *str*.

Notice that `fgets` is quite different from `gets`: not only `fgets` accepts a *stream* argument, but also allows to specify the maximum size of *str* and includes in the string any ending newline character.

### Parameters

**str**      Pointer to an array of chars where the string read is copied.  
**num**      Maximum number of characters to be copied into *str* (including the terminating null-character).  
**stream**      Pointer to a `FILE` object that identifies an input stream.  
              `stdin` can be used as argument to read from the *standard input*.

### Return Value

On success, the function returns *str*.  
If the *end-of-file* is encountered while attempting to read a character, the *eof indicator* is set (`feof`). If this happens before any characters could be read, the pointer returned is a null pointer (and the contents of *str* remain unchanged).  
If a read error occurs, the *error indicator* (`errno`) is set and a null pointer is also returned (but the contents pointed by *str* may have changed).

## Example

```
1 /* fgets example */
2 #include <stdio.h>
3
4 int main()
5 {
6     FILE * pFile;
```

```

8     char mystring [100];
9
10    pFile = fopen ("myfile.txt" , "r");
11    if (pFile == NULL) perror ("Error opening file");
12    else {
13        if ( fgets (mystring , 100 , pFile) != NULL )
14            puts (mystring);
15        fclose (pFile);
16    }
17    return 0;
}

```

This example reads the first line of `myfile.txt` or the first 99 characters, whichever comes first, and prints them on the screen.

### See also

<a href="#">fputs</a>	Write string to stream (function )
<a href="#">fgetc</a>	Get character from stream (function )
<a href="#">gets</a>	Get string from stdin (function )

## /cstdio/FILE

type

### FILE

<cstdio>

#### Object containing information to control a stream

Object type that identifies a stream and contains the information needed to control it, including a pointer to its buffer, its position indicator and all its state indicators.

FILE objects are usually created by a call to either `fopen` or `tmpfile`, which both return a pointer to one of these objects.

The content of a FILE object is not meant to be accessed from outside the functions of the `<cstdio>` and `<cwchar>` headers; In fact, portable programs shall only use them in the form of pointers to identify streams, since for some implementations, even the value of the pointer itself could be significant to identify the stream (i.e., the pointer to a copy of a FILE object could be interpreted differently than a pointer to the original).

Its memory allocation is automatically managed: it is allocated by either `fopen` or `tmpfile`, and it is the responsibility of the library to free the resources once either the stream has been closed using `fclose` or the program terminates normally.

On inclusion of the `<cstdio>` header file, three objects of this type are automatically created, and pointers to them are declared: `stdin`, `stdout` and `stderr`, associated with the standard input stream, standard output stream and standard error stream, respectively.

### Example

```

1 /* FEOF example */
2 #include <stdio.h>
3
4 int main()
5 {
6     FILE * pFile;
7     char buffer [100];
8
9     pFile = fopen ("myfile.txt" , "r");
10    if (pFile == NULL) perror ("Error opening file");
11    else
12    {
13        while ( ! feof (pFile) )
14        {
15            if ( fgets (buffer , 100 , pFile) == NULL ) break;
16            fputs (buffer , stdout);
17        }
18        fclose (pFile);
19    }
20    return 0;
21 }

```

This example reads the content of a text file called `myfile.txt` and sends it to the standard output stream.

### See also

<a href="#">fopen</a>	Open file (function )
<a href="#">fclose</a>	Close file (function )

## /cstdio/FILENAME\_MAX

constant

### FILENAME\_MAX

<cstdio>

#### Maximum length of file names

This macro constant expands to an integral expression corresponding to the size needed for an array of `char` elements to hold the longest file name string allowed by the library. Or, if the library imposes no such restriction, it is set to the recommended size for character arrays intended to hold a file name.

### See also

<b>L_tmpnam</b>	Minimum length for temporary file name ( <a href="#">constant</a> )
<b>FOPEN_MAX</b>	Potential limit of simultaneous open streams ( <a href="#">constant</a> )
<b>TMP_MAX</b>	Number of temporary files ( <a href="#">constant</a> )

## /cstdio/fopen

function

### fopen

<cstdio>

```
FILE * fopen ( const char * filename, const char * mode );
```

#### Open file

Opens the file whose name is specified in the parameter *filename* and associates it with a stream that can be identified in future operations by the **FILE** pointer returned.

The operations that are allowed on the stream and how these are performed are defined by the *mode* parameter.

The returned stream is *fully buffered* by default if it is known to not refer to an interactive device (see [setbuf](#)).

The returned pointer can be disassociated from the file by calling [fclose](#) or [freopen](#). All opened files are automatically closed on normal program termination.

The running environment supports at least **FOPEN\_MAX** files open simultaneously.

#### Parameters

##### filename

C string containing the name of the file to be opened.

Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

##### mode

C string containing a file access mode. It can be:

"r"	<b>read:</b> Open file for input operations. The file must exist.
"w"	<b>write:</b> Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.
"a"	<b>append:</b> Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations ( <a href="#">fseek</a> , <a href="#">fsetpos</a> , <a href="#">rewind</a> ) are ignored. The file is created if it does not exist.
"r+"	<b>read/update:</b> Open a file for update (both for input and output). The file must exist.
"w+"	<b>write/update:</b> Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
"a+"	<b>append/update:</b> Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations ( <a href="#">fseek</a> , <a href="#">fsetpos</a> , <a href="#">rewind</a> ) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

With the *mode* specifiers above the file is open as a *text file*. In order to open a file as a *binary file*, a "b" character has to be included in the *mode* string. This additional "b" character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the "+" sign for the mixed modes ("rb+", "wb+", "ab+").

The new C standard (C2011, which is not part of C++) adds a new standard subspecifier ("x"), that can be appended to any "w" specifier (to form "wx", "wbx", "w+x" or "w+bx"/"wb+x"). This subspecifier forces the function to fail if the file exists, instead of overwriting it.

If additional characters follow the sequence, the behavior depends on the library implementation: some implementations may ignore additional characters so that for example an additional "t" (sometimes used to explicitly state a *text file*) is accepted.

On some library implementations, opening or creating a text file with update mode may treat the stream instead as a binary file.

*Text files* are files containing sequences of lines of text. Depending on the environment where the application runs, some special character conversion may occur in input/output operations in *text mode* to adapt them to a system-specific text file format. Although on some environments no conversions occur and both *text files* and *binary files* are treated the same way, using the appropriate mode improves portability.

For files open for update (those which include a "+" sign), on which both input and output operations are allowed, the stream shall be flushed ([fflush](#)) or repositioned ([fseek](#), [fsetpos](#), [rewind](#)) before a reading operation that follows a writing operation. The stream shall be repositioned ([fseek](#), [fsetpos](#), [rewind](#)) before a writing operation that follows a reading operation (whenever that operation did not reach the end-of-file).

#### Return Value

If the file is successfully opened, the function returns a pointer to a **FILE** object that can be used to identify the stream on future operations. Otherwise, a null pointer is returned.

On most library implementations, the **errno** variable is also set to a system-specific error code on failure.

#### Example

```
1 /* fopen example */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile;
6     pFile = fopen ( "myfile.txt", "w" );
7     if (pFile!=NULL)
8     {
9         fputs ( "fopen example", pFile );
10        fclose (pFile);
11    }
12    return 0;
13 }
```

#### See also

<a href="#">fclose</a>	Close file (function )
<a href="#">freopen</a>	Reopen stream with different file or mode (function )
<a href="#">setbuf</a>	Set stream buffer (function )
<a href="#">setvbuf</a>	Change stream buffering (function )
<a href="#">tmpfile</a>	Open a temporary file (function )
<a href="#">tmpnam</a>	Generate temporary filename (function )

## /cstdio/FOPEN\_MAX

constant

### FOPEN\_MAX

<cstdio>

#### Potential limit of simultaneous open streams

This macro constant expands to an integral expression that represents the maximum number of files that can be opened simultaneously.

Particular library implementations may count files opened by [tmpfile](#) towards this limit. Implementations may also allow more files to be opened beyond this limit.

FOPEN\_MAX shall be greater than 7 on all systems.

#### See also

<a href="#">TMP_MAX</a>	Number of temporary files (constant )
-------------------------	---------------------------------------

## /cstdio/fpos\_t

type

### fpos\_t

<cstdio>

#### Object containing information to specify a position within a file

This type of object is used to specify a position within a file. An object of this type is capable of specifying uniquely any position within a file.

The information in fpos\_t objects is usually filled by a call to [fgetpos](#), which takes a pointer to an object of this type as argument.

The content of an fpos\_t object is not meant to be read directly, but only to be used as an argument in a call to [fsetpos](#).

## /cstdio/fprintf

function

### fprintf

<cstdio>

int fprintf ( FILE \* stream, const char \* format, ... );

#### Write formatted data to stream

Writes the C string pointed by *format* to the *stream*. If *format* includes *format specifiers* ( subsequences beginning with %), the additional arguments following *format* are formatted and inserted in the resulting string replacing their respective specifiers.

After the *format* parameter, the function expects at least as many additional arguments as specified by *format*.

#### Parameters

*stream*

Pointer to a [FILE](#) object that identifies an output stream.

*format*

C string that contains the text to be written to the stream.

It can optionally contain embedded *format specifiers* that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A *format specifier* follows this prototype:

%[flags][width][.precision][length]specifier

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2

c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	description
-	Left-justify within the given field width; Right justification is the default (see <i>width</i> sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see <i>width</i> sub-specifier).

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The <i>width</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	description
	For integer specifiers (d, i, o, u, x): <i>precision</i> specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A <i>precision</i> of 0 means that no character is written for the value 0.
.number	For a, A, e, E, f and F specifiers: this is the number of digits to be printed <b>after</b> the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for <i>precision</i> , 0 is assumed.
.*	The <i>precision</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

specifiers							
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void* int*	
hh	signed char	unsigned char				signed char*	
h	short int	unsigned short int				short int*	
l	long int	unsigned long int		wint_t	wchar_t*	long int*	
ll	long long int	unsigned long long int				long long int*	
j	intmax_t	uintmax_t				intmax_t*	
z	size_t	size_t				size_t*	
t	ptrdiff_t	ptrdiff_t				ptrdiff_t*	
L			long double				

Note that the c specifier takes an int (or wint\_t) as argument, but performs the proper conversion to a char value (or a wchar\_t) before formatting it for output.

**Note:** Yellow rows indicate specifiers and sub-specifiers introduced by C99. See [<cinttypes>](#) for the specifiers for extended types.

### ... (additional arguments)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for n).

There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

## Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (*errno*) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, *errno* is set to *EILSEQ* and a negative number is returned.

## Example

```
1 /* fprintf example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     int n;
8     char name [100];
9
10    pFile = fopen ("myfile.txt", "w");
11    for (n=0 ; n<3 ; n++)
12    {
13        puts ("please, enter a name: ");
14        gets (name);
15        fprintf (pFile, "Name %d [%-10.10s]\n", n+1, name);
16    }
17    fclose (pFile);
```

```
18
19     return 0;
20 }
```

This example prompts 3 times the user for a name and then writes them to `myfile.txt` each one in a line with a fixed length (a total of 19 characters + newline).

Two format tags are used:

`%d` : Signed decimal integer  
`%-10.10s` : left-justified (-), minimum of ten characters (10), maximum of ten characters (.10), string (s).

Assuming that we have entered John, Jean-Francois and Yoko as the 3 names, `myfile.txt` would contain:

```
Name 1 [John      ]
Name 2 [Jean-Franc]
Name 3 [Yoko      ]
```

For more examples on formatting see [printf](#).

## Compatibility

Particular library implementations may support additional *specifiers* and *sub-specifiers*.

Those listed here are supported by the latest C and C++ standards (both published in 2011), but those in yellow were introduced in C99 (only required for C++ implementations since C++11), and may not be supported by libraries that comply with older standards.

## See also

<a href="#">printf</a>	Print formatted data to stdout ( <a href="#">function</a> )
<a href="#">fscanf</a>	Read formatted data from stream ( <a href="#">function</a> )
<a href="#">fwrite</a>	Write block of data to stream ( <a href="#">function</a> )
<a href="#">fputs</a>	Write string to stream ( <a href="#">function</a> )

## /cstdio/fputc

function  
**fputc** <cstdio>

```
int fputc ( int character, FILE * stream );
```

### Write character to stream

Writes a *character* to the *stream* and advances the position indicator.

The character is written at the position indicated by the *internal position indicator* of the *stream*, which is then automatically advanced by one.

## Parameters

**character**  
The *int* promotion of the character to be written.  
The value is internally converted to an *unsigned char* when written.

**stream**  
Pointer to a [FILE](#) object that identifies an output stream.

## Return Value

On success, the *character* written is returned.  
If a writing error occurs, `EOF` is returned and the *error indicator* (`ferror`) is set.

## Example

```
1 /* fputc example: alphabet writer */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     char c;
8
9     pFile = fopen ("alphabet.txt", "w");
10    if (pFile!=NULL) {
11
12        for (c = 'A' ; c <= 'Z' ; c++)
13            fputc ( c , pFile );
14
15        fclose (pFile);
16    }
17    return 0;
18 }
```

This program creates a file called `alphabet.txt` and writes ABCDEFGHIJKLMNOPQRSTUVWXYZ to it.

## See also

<a href="#">putc</a>	Write character to stream ( <a href="#">function</a> )
<a href="#">fgetc</a>	Get character from stream ( <a href="#">function</a> )
<a href="#">fwrite</a>	Write block of data to stream ( <a href="#">function</a> )

## /cstdio/fputs

function  
**fputs** <cstdio>

---

```
int fputs ( const char * str, FILE * stream );
```

**Write string to stream**

Writes the C string pointed by *str* to the *stream*.

The function begins copying from the address specified (*str*) until it reaches the terminating null character ('\0'). This terminating null-character is not copied to the stream.

Notice that **fputs** not only differs from **puts** in that the destination *stream* can be specified, but also **fputs** does not write additional characters, while **puts** appends a newline character at the end automatically.

### Parameters

**str** C string with the content to be written to *stream*.

**stream** Pointer to a **FILE** object that identifies an output stream.

### Return Value

On success, a non-negative value is returned.

On error, the function returns **EOF** and sets the *error indicator* (**ferror**).

### Example

```
1 /* fputs example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     char sentence [256];
8
9     printf ("Enter sentence to append: ");
10    fgets (sentence,256,stdin);
11    pFile = fopen ("mylog.txt","a");
12    fputs (sentence,pFile);
13    fclose (pFile);
14    return 0;
15 }
```

This program allows to append a line to a file called mylog.txt each time it is run.

### See also

<b>puts</b>	Write string to stdout (function )
<b>fgets</b>	Get string from stream (function )
<b>fputc</b>	Write character to stream (function )
<b>fprintf</b>	Write formatted data to stream (function )
<b>fwrite</b>	Write block of data to stream (function )

## /cstdio/fread

function  
**fread** <cstdio>

---

```
size_t fread ( void * ptr, size_t size, size_t count, FILE * stream );
```

**Read block of data from stream**

Reads an array of *count* elements, each one with a size of *size* bytes, from the *stream* and stores them in the block of memory specified by *ptr*.

The position indicator of the stream is advanced by the total amount of bytes read.

The total amount of bytes read if successful is (*size*\**count*).

### Parameters

**ptr** Pointer to a block of memory with a size of at least (*size*\**count*) bytes, converted to a **void\***.

**size** Size, in bytes, of each element to be read.  
*size\_t* is an unsigned integral type.

**count**

Number of elements, each one with a size of *size* bytes.  
*size\_t* is an unsigned integral type.

**stream**  
Pointer to a `FILE` object that specifies an input stream.

### Return Value

The total number of elements successfully read is returned.  
If this number differs from the *count* parameter, either a reading error occurred or the *end-of-file* was reached while reading. In both cases, the proper indicator is set, which can be checked with `ferror` and `feof`, respectively.  
If either *size* or *count* is zero, the function returns zero and both the stream state and the content pointed by *ptr* remain unchanged.  
*size\_t* is an unsigned integral type.

### Example

```
1 /* fread example: read an entire file */
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main () {
6     FILE * pFile;
7     long lSize;
8     char * buffer;
9     size_t result;
10
11    pFile = fopen ( "myfile.bin" , "rb" );
12    if (pFile==NULL) {fputs ("File error",stderr); exit (1);}
13
14    // obtain file size:
15    fseek (pFile , 0 , SEEK_END);
16    lSize = ftell (pFile);
17    rewind (pFile);
18
19    // allocate memory to contain the whole file:
20    buffer = (char*) malloc (sizeof(char)*lSize);
21    if (buffer == NULL) {fputs ("Memory error",stderr); exit (2);}
22
23    // copy the file into the buffer:
24    result = fread (buffer,1,lSize,pFile);
25    if (result != lSize) {fputs ("Reading error",stderr); exit (3);}
26
27    /* the whole file is now loaded in the memory buffer. */
28
29    // terminate
30    fclose (pFile);
31    free (buffer);
32    return 0;
33 }
```

This code loads `myfile.bin` into a dynamically allocated memory buffer, which can be used to manipulate the content of a file as an array.

### See also

<b>fwrite</b>	Write block of data to stream <a href="#">(function )</a>
<b>fgetc</b>	Get character from stream <a href="#">(function )</a>
<b>fscanf</b>	Read formatted data from stream <a href="#">(function )</a>

## /cstdio/freopen

function  
**freopen** <cstdio>

```
FILE * freopen ( const char * filename, const char * mode, FILE * stream );
```

### Reopen stream with different file or mode

Reuses *stream* to either open the file specified by *filename* or to change its access *mode*.

If a new *filename* is specified, the function first attempts to close any file already associated with *stream* (third parameter) and disassociates it. Then, independently of whether that stream was successfully closed or not, `freopen` opens the file specified by *filename* and associates it with the *stream* just as `fopen` would do using the specified *mode*.

If *filename* is a null pointer, the function attempts to change the *mode* of the stream. Although a particular library implementation is allowed to restrict the changes permitted, and under which circumstances.

The *error indicator* and *eof indicator* are automatically cleared (as if `clearerr` was called).

This function is especially useful for redirecting predefined streams like `stdin`, `stdout` and `stderr` to specific files (see the example below).

### Parameters

**filename**

C string containing the name of the file to be opened.

Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

If this parameter is a null pointer, the function attempts to change the mode of the *stream*, as if the file name currently associated with that stream had been used.

**mode**

C string containing a file access mode. It can be:

"r"	<b>read:</b> Open file for input operations. The file must exist.
"w"	<b>write:</b> Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.
"a"	<b>append:</b> Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations ( <code>fseek</code> , <code>fsetpos</code> , <code>rewind</code> ) are ignored. The file is created if it does not exist.
"r+"	<b>read/update:</b> Open a file for update (both for input and output). The file must exist.
"w+"	<b>write/update:</b> Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file.
"a+"	<b>append/update:</b> Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations ( <code>fseek</code> , <code>fsetpos</code> , <code>rewind</code> ) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist.

With the *mode* specifiers above the file is open as a *text file*. In order to open a file as a *binary file*, a "b" character has to be included in the *mode* string. This additional "b" character can either be appended at the end of the string (thus making the following compound modes: "rb", "wb", "ab", "r+b", "w+b", "a+b") or be inserted between the letter and the "+" sign for the mixed modes ("rb+", "wb+", "ab+").

The new C standard (C2011, which is not part of C++) adds a new standard subspecifier ("x"), that can be appended to any "w" specifier (to form "wx", "wbx", "w+x" or "w+bx" / "wb+x"). This subspecifier forces the function to fail if the file exists, instead of overwriting it.

If additional characters follow the sequence, the behavior depends on the library implementation: some implementations may ignore additional characters so that for example an additional "t" (sometimes used to explicitly state a *text file*) is accepted.

On some library implementations, opening or creating a text file with update mode may treat the stream instead as a binary file.

**stream**  
pointer to a [FILE](#) object that identifies the stream to be reopened.

## Return Value

If the file is successfully reopened, the function returns the pointer passed as parameter *stream*, which can be used to identify the reopened stream. Otherwise, a null pointer is returned.

On most library implementations, the `errno` variable is also set to a system-specific error code on failure.

## Example

```
1 /* freopen example: redirecting stdout */
2 #include <stdio.h>
3
4 int main ()
5 {
6     freopen ("myfile.txt", "w", stdout);
7     printf ("This sentence is redirected to a file.");
8     fclose (stdout);
9     return 0;
10 }
```

This sample code redirects the output that would normally go to the standard output to a file called `myfile.txt`, that after this program is executed contains:  
 This sentence is redirected to a file.

## See also

<a href="#">fopen</a>	Open file (function )
<a href="#">fclose</a>	Close file (function )

## /cstdio/fscanf

function

### fscanf

<cstdio>

`int fscanf ( FILE * stream, const char * format, ... );`

#### Read formatted data from stream

Reads data from the *stream* and stores them according to the parameter *format* into the locations pointed by the additional arguments.

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

## Parameters

**stream**

Pointer to a [FILE](#) object that identifies the input stream to read data from.

**format**

C string that contains a sequence of characters that control how characters extracted from the stream are treated:

- Whitespace character:** the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters -- see `isspace`). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- Non-whitespace character, except format specifier (%):** Any character that is not either a whitespace character (blank, newline or tab) or part of a *format specifier* (which begin with a % character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
- Format specifiers:** A sequence formed by an initial percentage sign (%) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the *stream* and stored into the locations pointed by the additional arguments.

A *format specifier* for `fscanf` follows this prototype:

```
%[*][width][length]specifier
```

Where the *specifier* character at the end is the most significant component, since it defines which characters are extracted, their interpretation and the type of its corresponding argument:

<b>specifier</b>	<b>Description</b>	<b>Characters extracted</b>
i, u	Integer	Any number of digits, optionally preceded by a sign (+ or -). Decimal digits assumed by default (0-9), but a 0 prefix introduces octal digits (0-7), and 0x hexadecimal digits (0-f).
d	Decimal integer	Any number of decimal digits (0-9), optionally preceded by a sign (+ or -).
o	Octal integer	Any number of octal digits (0-7), optionally preceded by a sign (+ or -).
x	Hexadecimal integer	Any number of hexadecimal digits (0-9, a-f, A-F), optionally preceded by 0x or 0X, and all optionally preceded by a sign (+ or -).
f, e, g	Floating point number	A series of decimal digits, optionally containing a decimal point, optionally preceded by a sign (+ or -) and optionally followed by the e or E character and a decimal integer (or some of the other sequences supported by <code>strtod</code> ).
a		Implementations complying with C99 also support hexadecimal floating-point format when preceded by 0x or 0X.
c	Character	The next character. If a <i>width</i> other than 1 is specified, the function reads exactly <i>width</i> characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
s	String of characters	Any number of non-whitespace characters, stopping at the first whitespace character found. A terminating null character is automatically added at the end of the stored sequence.
p	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format %p in <code>fprintf</code> .
[characters]	Scanset	Any number of the characters specified between the brackets. A dash (-) that is not the first character may produce non-portable behavior in some library implementations.
[^characters]	Negated scanset	Any number of characters none of them specified as characters between the brackets.
n	Count	No input is consumed. The number of characters read so far from <i>stream</i> is stored in the pointed location.
%	%	A % followed by another % matches a single %.

Except for n, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends there.

The *format specifier* can also contain sub-specifiers: asterisk (\*), width and length (in that order), which are optional and follow these specifications:

<b>sub-specifier</b>	<b>description</b>
*	An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument).
width	Specifies the maximum number of characters to be read in the current reading operation (optional).
length	One of hh, h, l, ll, j, z, t, L (optional). This alters the expected type of the storage pointed by the corresponding argument (see below).

This is a chart showing the types expected for the corresponding arguments where input is stored (both with and without a length sub-specifier):

<b>specifiers</b>						
<b>length</b>	<b>d i</b>	<b>u o x</b>	<b>f e g a</b>	<b>c s [] [^]</b>	<b>p</b>	<b>n</b>
(none)	int*	unsigned int*	float*	char*	void**	int*
hh	signed char*	unsigned char*				signed char*
h	short int*	unsigned short int*				short int*
l	long int*	unsigned long int*	double*	wchar_t*		long int*
ll	long long int*	unsigned long long int*				long long int*
j	intmax_t*	uintmax_t*				intmax_t*
z	size_t*	size_t*				size_t*
t	ptrdiff_t*	ptrdiff_t*				ptrdiff_t*
L			long double*			

**Note:** Yellow rows indicate specifiers and sub-specifiers introduced by C99.

... (additional arguments)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type.

There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.

These arguments are expected to be pointers: to store the result of a `fscanf` operation on a regular variable, its name should be preceded by the reference operator (&) (see [example](#)).

## Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (`feof` or `ferror`). And, if either happens before any data could be successfully read, `EOF` is returned.

If an encoding error happens interpreting wide characters, the function sets `errno` to `EILSEQ`.

## Example

```
1 /* fscanf example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char str [80];
7     float f;
8     FILE * pFile;
9
10    pFile = fopen ("myfile.txt","w+");
11    fprintf (pFile, "%f %s", 3.1416, "PI");
12    rewind (pFile);
13    fscanf (pFile, "%f", &f);
14    fscanf (pFile, "%s", str);
15    fclose (pFile);
```

```

16 printf ("I have read: %f and %s \n",f,str);
17 return 0;
18 }

```

This sample code creates a file called `myfile.txt` and writes a float number and a string to it. Then, the stream is rewinded and both values are read with `fscanf`. It finally produces an output similar to:

```
I have read: 3.141600 and PI
```

## Compatibility

Particular library implementations may support additional specifiers and sub-specifiers.

Those listed here are supported by the latest C and C++ standards (both published in 2011), but those in yellow were introduced by C99 (only required for C++ implementations since C++11), and may not be supported by libraries that comply with older standards.

## See also

<b>scanf</b>	Read formatted data from <code>stdin</code> ( <a href="#">function</a> )
<b>fprintf</b>	Write formatted data to stream ( <a href="#">function</a> )
<b>fread</b>	Read block of data from stream ( <a href="#">function</a> )
<b>fgets</b>	Get string from stream ( <a href="#">function</a> )

## /cstdio/fseek

function  
**fseek** <cstdio>

```
int fseek ( FILE * stream, long int offset, int origin );
```

### Reposition stream position indicator

Sets the position indicator associated with the *stream* to a new position.

For streams open in binary mode, the new position is defined by adding *offset* to a reference position specified by *origin*.

For streams open in text mode, *offset* shall either be zero or a value returned by a previous call to `fseek`, and *origin* shall necessarily be `SEEK_SET`.

If the function is called with other values for these arguments, support depends on the particular system and library implementation (non-portable).

The *end-of-file internal indicator* of the *stream* is cleared after a successful call to this function, and all effects from previous calls to `ungetc` on this *stream* are dropped.

On streams open for update (read+write), a call to `fseek` allows to switch between reading and writing.

## Parameters

<b>stream</b>	Pointer to a <code>FILE</code> object that identifies the stream.
<b>offset</b>	Binary files: Number of bytes to offset from <i>origin</i> . Text files: Either zero, or a value returned by <code>fseek</code> .
<b>origin</b>	Position used as reference for the <i>offset</i> . It is specified by one of the following constants defined in <code>&lt;cstdio&gt;</code> exclusively to be used as arguments for this function:
<b>Constant</b>	<b>Reference position</b>

`SEEK_SET` Beginning of file  
`SEEK_CUR` Current position of the file pointer  
`SEEK_END` End of file \*

\* Library implementations are allowed to not meaningfully support `SEEK_END` (therefore, code using it has no real standard portability).

## Return Value

If successful, the function returns zero.

Otherwise, it returns non-zero value.

If a read or write error occurs, the *error indicator* (`ferror`) is set.

## Example

```

1 /* fseek example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     pFile = fopen ( "example.txt" , "wb" );
8     fputs ( "This is an apple." , pFile );
9     fseek ( pFile , 9 , SEEK_SET );
10    fputs ( " sam" , pFile );
11    fclose ( pFile );
12    return 0;
13 }

```

After this code is successfully executed, the file `example.txt` contains:

```
This is a sample.
```

## See also

<a href="#">ftell</a>	Get current position in stream (function )
<a href="#">fsetpos</a>	Set position indicator of stream (function )
<a href="#">rewind</a>	Set position of stream to the beginning (function )

## /cstdio/fsetpos

function

### fsetpos

<cstdio>

```
int fsetpos ( FILE * stream, const fpos_t * pos );
```

#### Set position indicator of stream

Restores the current position in the *stream* to *pos*.

The *internal file position indicator* associated with *stream* is set to the position represented by *pos*, which is a pointer to an *fpos\_t* object whose value shall have been previously obtained by a call to [fgetpos](#).

The *end-of-file internal indicator* of the *stream* is cleared after a successful call to this function, and all effects from previous calls to [ungetc](#) on this *stream* are dropped.

On streams open for update (read+write), a call to [fsetpos](#) allows to switch between reading and writing.

A similar function, [fseek](#), can be used to set arbitrary positions on streams open in binary mode.

## Parameters

stream

Pointer to a [FILE](#) object that identifies the stream.

position

Pointer to a *fpos\_t* object containing a position previously obtained with [fgetpos](#).

## Return Value

If successful, the function returns zero.

On failure, a non-zero value is returned and [errno](#) is set to a system-specific positive value.

## Example

```
1 /* fsetpos example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     fpos_t position;
8
9     pFile = fopen ("myfile.txt","w");
10    fgetpos (pFile, &position);
11    fputs ("That is a sample",pFile);
12    fsetpos (pFile, &position);
13    fputs ("This",pFile);
14    fclose (pFile);
15    return 0;
16 }
```

After this code is successfully executed, a file called *myfile.txt* will contain:

This is a sample

## See also

<a href="#">fgetpos</a>	Get current position in stream (function )
<a href="#">fseek</a>	Reposition stream position indicator (function )
<a href="#">rewind</a>	Set position of stream to the beginning (function )

## /cstdio/ftell

function

### ftell

<cstdio>

```
long int ftell ( FILE * stream );
```

#### Get current position in stream

Returns the current value of the position indicator of the *stream*.

For binary streams, this is the number of bytes from the beginning of the file.

For text streams, the numerical value may not be meaningful but can still be used to restore the position to the same position later using [fseek](#) (if there are characters put back using [ungetc](#) still pending of being read, the behavior is undefined).

## Parameters

### stream

Pointer to a `FILE` object that identifies the stream.

## Return Value

On success, the current value of the position indicator is returned.

On failure, `-1L` is returned, and `errno` is set to a system-specific positive value.

## Example

```
1 /* ftell example : getting size of a file */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     long size;
8
9     pFile = fopen ("myfile.txt","rb");
10    if (pFile==NULL) perror ("Error opening file");
11    else
12    {
13        fseek (pFile, 0, SEEK_END); // non-portable
14        size=ftell (pFile);
15        fclose (pFile);
16        printf ("Size of myfile.txt: %ld bytes.\n",size);
17    }
18    return 0;
19 }
```

This program prints out the size of `myfile.txt` in bytes (where supported).

## See also

<code>fseek</code>	Reposition stream position indicator (function )
<code>fgetpos</code>	Get current position in stream (function )
<code>rewind</code>	Set position of stream to the beginning (function )

## /cstdio/fwrite

### function

## fwrite

<cstdio>

```
size_t fwrite ( const void * ptr, size_t size, size_t count, FILE * stream );
```

### Write block of data to stream

Writes an array of `count` elements, each one with a size of `size` bytes, from the block of memory pointed by `ptr` to the current position in the `stream`.

The *position indicator* of the stream is advanced by the total number of bytes written.

Internally, the function interprets the block pointed by `ptr` as if it was an array of `(size*count)` elements of type `unsigned char`, and writes them sequentially to `stream` as if `fputc` was called for each byte.

## Parameters

`ptr` Pointer to the array of elements to be written, converted to a `const void*`.

`size` Size in bytes of each element to be written.  
`size_t` is an unsigned integral type.

`count` Number of elements, each one with a size of `size` bytes.  
`size_t` is an unsigned integral type.

`stream` Pointer to a `FILE` object that specifies an output stream.

## Return Value

The total number of elements successfully written is returned.

If this number differs from the `count` parameter, a writing error prevented the function from completing. In this case, the *error indicator* (`errno`) will be set for the `stream`.

If either `size` or `count` is zero, the function returns zero and the *error indicator* remains unchanged.  
`size_t` is an unsigned integral type.

## Example

```
1 /* fwrite example : write buffer */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
```

```

8 |     char buffer[] = { 'x' , 'y' , 'z' };
9 |     pFile = fopen ("myfile.bin", "wb");
10|     fwrite (buffer , sizeof(char), sizeof(buffer), pFile);
11|     fclose (pFile);
12|     return 0;
}

```

A file called `myfile.bin` is created and the content of the buffer is stored into it. For simplicity, the buffer contains `char` elements but it can contain any other type.  
`sizeof(buffer)` is the length of the array in bytes (in this case it is three, because the array has three elements of one byte each).

## See also

---

## /cstdio/getc

function  
**getc** <cstdio>

```
int getc ( FILE * stream );
```

### Get character from stream

Returns the character currently pointed by the internal file position indicator of the specified *stream*. The internal file position indicator is then advanced to the next character.

If the stream is at the end-of-file when called, the function returns `EOF` and sets the *end-of-file indicator* for the stream (`feof`).

If a read error occurs, the function returns `EOF` and sets the *error indicator* for the stream (`ferror`).

`getc` and `fgetc` are equivalent, except that `getc` may be implemented as a macro in some libraries. See `getchar` for a similar function that reads directly from `stdin`.

## Parameters

---

### stream

Pointer to a `FILE` object that identifies an input stream.

Because some libraries may implement this function as a macro, and this may evaluate the *stream* expression more than once, this should be an expression without side effects.

## Return Value

---

On success, the character read is returned (promoted to an `int` value).

The return type is `int` to accommodate for the special value `EOF`, which indicates failure:

If the position indicator was at the *end-of-file*, the function returns `EOF` and sets the *eof indicator* (`feof`) of *stream*.

If some other reading error happens, the function also returns `EOF`, but sets its *error indicator* (`ferror`) instead.

## Example

---

```

1 /* getc example: money counter */
2 #include <stdio.h>
3 int main ()
4 {
5     FILE * pFile;
6     int c;
7     int n = 0;
8     pFile=fopen ("myfile.txt","r");
9     if (pFile==NULL) perror ("Error opening file");
10    else
11    {
12        do {
13            c = getc (pFile);
14            if (c == '$') n++;
15        } while (c != EOF);
16        fclose (pFile);
17        printf ("File contains %d$\n",n);
18    }
19    return 0;
20 }

```

This program reads an existing file called `myfile.txt` character by character and uses the `n` variable to count how many dollar characters (\$) does the file contain.

## See also

---

<b>fgetc</b>	Get character from stream (function )
<b>fputc</b>	Write character to stream (function )
<b>fread</b>	Read block of data from stream (function )
<b>fwrite</b>	Write block of data to stream (function )

## /cstdio/getchar

function  
**getchar** <cstdio>

---

```
int getchar ( void );
Get character from stdin
```

Returns the next character from the standard input ([stdin](#)).

It is equivalent to calling [getc](#) with [stdin](#) as argument.

### Parameters

(none)

### Return Value

On success, the character read is returned (promoted to an [int](#) value).

The return type is [int](#) to accommodate for the special value [EOF](#), which indicates failure:

If the standard input was at the *end-of-file*, the function returns [EOF](#) and sets the *eof indicator* ([feof](#)) of [stdin](#).

If some other reading error happens, the function also returns [EOF](#), but sets its *error indicator* ([ferror](#)) instead.

### Example

```
1 /* getchar example : typewriter */
2 #include <stdio.h>
3
4 int main ()
5 {
6     int c;
7     puts ("Enter text. Include a dot ('.') in a sentence to exit:");
8     do {
9         c=getchar();
10        putchar (c);
11     } while (c != '.');
12     return 0;
13 }
```

A simple typewriter. Every sentence is echoed once ENTER has been pressed until a dot (.) is included in the text.

### See also

<a href="#">get</a>	Get character from stream (function )
<a href="#">putchar</a>	Write character to stdout (function )
<a href="#">scanf</a>	Read formatted data from stdin (function )

## /cstdio/gets

function

### gets

<cstdio>

```
char * gets ( char * str );
```

#### Get string from stdin

Reads characters from the *standard input* ([stdin](#)) and stores them as a C string into *str* until a newline character or the *end-of-file* is reached.

The newline character, if found, is not copied into *str*.

A terminating null character is automatically appended after the characters copied to *str*.

Notice that [gets](#) is quite different from [fgets](#): not only [gets](#) uses [stdin](#) as source, but it does not include the ending newline character in the resulting string and does not allow to specify a maximum size for *str* (which can lead to buffer overflows).

### Parameters

*str*

Pointer to a block of memory (array of [char](#)) where the string read is copied as a C *string*.

### Return Value

On success, the function returns *str*.

If the *end-of-file* is encountered while attempting to read a character, the *eof indicator* is set ([feof](#)). If this happens before any characters could be read, the pointer returned is a null pointer (and the contents of *str* remain unchanged).

If a read error occurs, the *error indicator* ([ferror](#)) is set and a null pointer is also returned (but the contents pointed by *str* may have changed).

### Compatibility

The most recent revision of the C standard (2011) has definitively removed this function from its specification.  
The function is deprecated in C++ (as of 2011 standard, which follows C99+TC3).

### Example

```
1 /* gets example */
2 #include <stdio.h>
3
4 int main()
5 {
6     char string [256];
7     printf ("Insert your full address: ");
8     gets (string);      // warning: unsafe (see fgets instead)
```

```

9 |     printf ("Your address is: %s\n",string);
10|     return 0;
11|

```

## See also

<a href="#">fgets</a>	Get string from stream (function )
<a href="#">getchar</a>	Get character from stdin (function )
<a href="#">scanf</a>	Read formatted data from stdin (function )

## /cstdio/L\_tmpnam

constant

### L\_tmpnam

<cstdio>

#### Minimum length for temporary file name

This macro constant expands to an integral expression corresponding to the size needed for an array of `char` elements to hold the longest file name string possibly generated by [tmpnam](#).

## See also

<a href="#">FILENAME_MAX</a>	Maximum length of file names (constant )
<a href="#">FOPEN_MAX</a>	Potential limit of simultaneous open streams (constant )
<a href="#">TMP_MAX</a>	Number of temporary files (constant )

## /cstdio/NULL

macro

### NULL

<cstdint> <cstdlib> <cstring> <cwchar> <ctime> <clocale> <cstdio>

#### Null pointer

This macro expands to a *null pointer constant*.

A *null-pointer constant* is an integral constant expression that evaluates to zero (like 0 or 0L), or the cast of such value to type `void*` (like `(void*)0`).

A *null-pointer constant* is an integral constant expression that evaluates to zero (such as 0 or 0L).

A *null-pointer constant* is either an integral constant expression that evaluates to zero (such as 0 or 0L), or a value of type [nullptr\\_t](#) (such as `nullptr`).

A null pointer constant can be converted to any *pointer type* (or *pointer-to-member type*), which acquires a *null pointer value*. This is a special value that indicates that the pointer is not pointing to any object.

## /cstdio/perror

function

### perror

<cstdio>

`void perror ( const char * str );`

#### Print error message

Interprets the value of [errno](#) as an error message, and prints it to [stderr](#) (the standard error output stream, usually the console), optionally preceding it with the custom message specified in *str*.

`errno` is an integral variable whose value describes the error condition or diagnostic information produced by a call to a library function (any function of the C standard library may set a value for `errno`, even if not explicitly specified in this reference, and even if no error happened), see [errno](#) for more info.

The error message produced by `perror` is platform-depend.

If the parameter *str* is not a null pointer, *str* is printed followed by a colon (:) and a space. Then, whether *str* was a null pointer or not, the generated error description is printed followed by a newline character ('\n').

`perror` should be called right after the error was produced, otherwise it can be overwritten by calls to other functions.

## Parameters.

*str*

C string containing a custom message to be printed before the error message itself.  
If it is a null pointer, no preceding custom message is printed, but the error message is still printed.  
By convention, the name of the application itself is generally used as parameter.

## Return Value

none

## Example

```

1 /* perror example */
2 #include <stdio.h>

```

```

3
4 int main ()
5 {
6     FILE * pFile;
7     pFile=fopen ("unexist.ent","rb");
8     if (pfile==NULL)
9         perror ("The following error occurred");
10    else
11        fclose (pFile);
12    return 0;
13 }

```

If the file `unexist.ent` does not exist, something similar to this may be expected as program output:

`The following error occurred: No such file or directory`

## See also

<a href="#">clearerr</a>	Clear error indicators (function)
<a href="#">ferror</a>	Check error indicator (function)

## /cstdio/printf

function

### printf

<cstdio>

`int printf ( const char * format, ... );`

#### Print formatted data to stdout

Writes the C string pointed by `format` to the standard output (`stdout`). If `format` includes `format specifiers` (subsequences beginning with %), the additional arguments following `format` are formatted and inserted in the resulting string replacing their respective specifiers.

#### Parameters

`format`

C string that contains the text to be written to `stdout`.

It can optionally contain embedded `format specifiers` that are replaced by the values specified in subsequent additional arguments and formatted as requested.

A `format specifier` follows this prototype: [see compatibility note below]

`%[flags][width].[precision]specifier`

Where the `specifier character` at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

The `format specifier` can also contain sub-specifiers: `flags`, `width`, `.precision` and `modifiers` (in that order), which are optional and follow these specifications:

flags	description
-	Left-justify within the given field width; Right justification is the default (see <code>width</code> sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see <code>width</code> sub-specifier).

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The <code>width</code> is not specified in the <code>format</code> string, but as an additional integer value argument preceding the argument that has to be formatted.

precision	description

.number	For integer specifiers (d, i, o, u, x, x): <i>precision</i> specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A <i>precision</i> of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed <b>after</b> the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for <i>precision</i> , 0 is assumed.
.*	The <i>precision</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

The *length* sub-specifier modifies the length of the data type. This is a chart showing the types used to interpret the corresponding arguments with and without *length* specifier (if a different type is used, the proper type promotion or conversion is performed, if allowed):

specifiers							
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void* int*	
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Note regarding the c specifier: it takes an int (or wint\_t) as argument, but performs the proper conversion to a char value (or a wchar\_t) before formatting it for output.

**Note:** Yellow rows indicate specifiers and sub-specifiers introduced by C99. See [<cinttypes>](#) for the specifiers for extended types.

... (additional arguments)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for n).

There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

## Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (errno) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, errno is set to EILSEQ and a negative number is returned.

## Example

```
1 /* printf example */
2 #include <stdio.h>
3
4 int main()
5 {
6     printf ("Characters: %c %c \n", 'a', 65);
7     printf ("Decimals: %d %ld\n", 1977, 650000L);
8     printf ("Preceding with blanks: %10d \n", 1977);
9     printf ("Preceding with zeros: %010d \n", 1977);
10    printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
11    printf ("floats: %4.2f %+0e %E \n", 3.1416, 3.1416, 3.1416);
12    printf ("Width trick: %*d \n", 5, 10);
13    printf ("%s \n", "A string");
14    return 0;
15 }
```

Output:

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks: 1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick: 10
A string
```

## Compatibility

Particular library implementations may support additional *specifiers* and *sub-specifiers*.

Those listed here are supported by the latest C and C++ standards (both published in 2011), but those in yellow were introduced in C99 (only required for C++ implementations since C++11), and may not be supported by libraries that comply with older standards.

## See also

<a href="#">puts</a>	Write string to stdout ( <a href="#">function</a> )
<a href="#">scanf</a>	Read formatted data from stdin ( <a href="#">function</a> )
<a href="#">fprintf</a>	Write formatted data to stream ( <a href="#">function</a> )
<a href="#">fwrite</a>	Write block of data to stream ( <a href="#">function</a> )

## /cstdio/putc

function  
**putc** <cstdio>

```
int putc ( int character, FILE * stream );
```

### Write character to stream

Writes a *character* to the *stream* and advances the position indicator.

The character is written at the position indicated by the *internal position indicator* of the *stream*, which is then automatically advanced by one.

**putc** and **fputc** are equivalent, except that **putc** may be implemented as a macro in some libraries. See **putchar** for a similar function that writes directly to **stdout**.

### Parameters

#### character

The *int* promotion of the character to be written.

The value is internally converted to an *unsigned char* when written.

Because some libraries may implement this function as a macro, and this may evaluate the *stream* expression more than once, this should be an expression without side effects.

#### stream

Pointer to a **FILE** object that identifies an output stream.

### Return Value

On success, the *character* written is returned.

If a writing error occurs, **EOF** is returned and the *error indicator* (**ferror**) is set.

### Example

```
1 /* putc example: alphabet writer */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     char c;
8
9     pFile=fopen("alphabet.txt","wt");
10    for (c = 'A' ; c <= 'Z' ; c++) {
11        putc (c , pFile);
12    }
13    fclose (pFile);
14    return 0;
15 }
```

This example program creates a file called **alphabet.txt** and writes ABCDEFGHIJKLMNOPQRSTUVWXYZ to it.

### See also

<b>putchar</b>	Write character to <b>stdout</b> (function )
<b>fputc</b>	Write character to stream (function )
<b>getc</b>	Get character from stream (function )
<b>fwrite</b>	Write block of data to stream (function )
<b>fprintf</b>	Write formatted data to stream (function )

## /cstdio/putchar

function  
**putchar** <cstdio>

```
int putchar ( int character );
```

### Write character to **stdout**

Writes a *character* to the *standard output* (**stdout**).

It is equivalent to calling **putc** with **stdout** as second argument.

### Parameters

#### character

The *int* promotion of the character to be written.

The value is internally converted to an *unsigned char* when written.

### Return Value

On success, the *character* written is returned.

If a writing error occurs, **EOF** is returned and the *error indicator* (**ferror**) is set.

## Example

```
1 /* putchar example: printing the alphabet */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char c;
7     for (c = 'A' ; c <= 'Z' ; c++) putchar (c);
8
9     return 0;
10 }
```

This program writes ABCDEFGHIJKLMNOPQRSTUVWXYZ to the standard output.

## See also

<a href="#">putc</a>	Write character to stream (function )
<a href="#">fputc</a>	Write character to stream (function )
<a href="#">getchar</a>	Get character from stdin (function )

## /cstdio/puts

function

### puts

<cstdio>

```
int puts ( const char * str );
```

#### Write string to stdout

Writes the C string pointed by *str* to the *standard output* (*stdout*) and appends a newline character ('\n').

The function begins copying from the address specified (*str*) until it reaches the terminating null character ('\0'). This terminating null-character is not copied to the stream.

Notice that *puts* not only differs from *fputs* in that it uses *stdout* as destination, but it also appends a newline character at the end automatically (which *fputs* does not).

## Parameters

*str* C string to be printed.

## Return Value

On success, a non-negative value is returned.

On error, the function returns EOF and sets the *error indicator* (*errno*).

## Example

```
1 /* puts example : hello world! */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char string [] = "Hello world!";
7     puts (string);
8 }
```

## See also

<a href="#">fputs</a>	Write string to stream (function )
<a href="#">printf</a>	Print formatted data to stdout (function )
<a href="#">putchar</a>	Write character to stdout (function )
<a href="#">gets</a>	Get string from stdin (function )

## /cstdio/remove

function

### remove

<cstdio>

```
int remove ( const char * filename );
```

#### Remove file

Deletes the file whose name is specified in *filename*.

This is an operation performed directly on a file identified by its *filename*; No streams are involved in the operation.

Proper file access shall be available.

## Parameters

### filename

C string containing the name of the file to be deleted.

Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

### Return value

If the file is successfully deleted, a zero value is returned.

On failure, a nonzero value is returned.

On most library implementations, the `errno` variable is also set to a system-specific error code on failure.

### Example

```
1 /* remove example: remove myfile.txt */
2 #include <stdio.h>
3
4 int main ()
5 {
6     if( remove( "myfile.txt" ) != 0 )
7         perror( "Error deleting file" );
8     else
9         puts( "File successfully deleted" );
10    return 0;
11 }
```

If the file `myfile.txt` exists before the execution and the program has write access to it, the file would be deleted and this message would be written to `stdout`:

```
File successfully deleted
```

Otherwise, a message similar to this would be written to `stderr`:

```
Error deleting file: No such file or directory
```

### See also

[rename](#)

Rename file (function)

## /cstdio/ rename

function

### rename

<cstdio>

```
int rename ( const char * oldname, const char * newname );
```

#### Rename file

Changes the name of the file or directory specified by `oldname` to `newname`.

This is an operation performed directly on a file; No streams are involved in the operation.

If `oldname` and `newname` specify different paths and this is supported by the system, the file is moved to the new location.

If `newname` names an existing file, the function may either fail or override the existing file, depending on the specific system and library implementation.

Proper file access shall be available.

### Parameters

#### oldname

C string containing the name of an existing file to be renamed and/or moved.

Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

#### newname

C string containing the new name for the file.

Its value shall follow the file name specifications of the running environment and can include a path (if supported by the system).

### Return value

If the file is successfully renamed, a zero value is returned.

On failure, a nonzero value is returned.

On most library implementations, the `errno` variable is also set to a system-specific error code on failure.

### Example

```
1 /* rename example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     int result;
7     char oldname[] = "oldname.txt";
8     char newname[] = "newname.txt";
9     result= rename( oldname , newname );
10    if ( result == 0 )
11        puts ( "File successfully renamed" );
12    else
13        perror( "Error renaming file" );
14    return 0;
15 }
```

If the file `oldname.txt` could be successfully renamed to `newname.txt` the following message would be written to `stdout`:

`File successfully renamed`

Otherwise, a message similar to this will be written to `stderr`:

`Error renaming file: Permission denied`

## See also

[remove](#) Remove file (function )

# /cstdio/rewind

function

## rewind

<cstdio>

`void rewind ( FILE * stream );`

### Set position of stream to the beginning

Sets the position indicator associated with `stream` to the beginning of the file.

The `end-of-file` and `error` internal indicators associated to the `stream` are cleared after a successful call to this function, and all effects from previous calls to `ungetc` on this `stream` are dropped.

On streams open for update (read+write), a call to `rewind` allows to switch between reading and writing.

## Parameters

`stream`

Pointer to a `FILE` object that identifies the stream.

## Return Value

none

## Example

```
1 /* rewind example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     int n;
7     FILE * pFile;
8     char buffer [27];
9
10    pFile = fopen ("myfile.txt","w+");
11    for ( n='A' ; n<='Z' ; n++)
12        fputc ( n, pFile);
13    rewind (pFile);
14    fread (buffer,1,26,pFile);
15    fclose (pFile);
16    buffer[26]='\0';
17    puts (buffer);
18    return 0;
19 }
```

A file called `myfile.txt` is created for reading and writing and filled with the alphabet. The file is then rewinded, read and its content is stored in a buffer, that then is written to the standard output:

ABCDEFIGHJKLMNOPQRSTUVWXYZ

## See also

[fseek](#) Reposition stream position indicator (function )

[fsetpos](#) Set position indicator of stream (function )

[fflush](#) Flush stream (function )

# /cstdio/scanf

function

## scanf

<cstdio>

`int scanf ( const char * format, ... );`

### Read formatted data from stdin

Reads data from `stdin` and stores them according to the parameter `format` into the locations pointed by the additional arguments.

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the `format` string.

## Parameters

## format

C string that contains a sequence of characters that control how characters extracted from the stream are treated:

- **Whitespace character:** the function will read and ignore any whitespace characters encountered before the next non-whitespace character (whitespace characters include spaces, newline and tab characters -- see [isspace](#)). A single whitespace in the *format* string validates any quantity of whitespace characters extracted from the *stream* (including none).
- **Non-whitespace character, except format specifier (%):** Any character that is not either a whitespace character (blank, newline or tab) or part of a *format specifier* (which begin with a % character) causes the function to read the next character from the stream, compare it to this non-whitespace character and if it matches, it is discarded and the function continues with the next character of *format*. If the character does not match, the function fails, returning and leaving subsequent characters of the stream unread.
- **Format specifiers:** A sequence formed by an initial percentage sign (%) indicates a format specifier, which is used to specify the type and format of the data to be retrieved from the *stream* and stored into the locations pointed by the additional arguments.

A *format specifier* for `scanf` follows this prototype:

```
%[*][width]specifier
```

Where the *specifier* character at the end is the most significant component, since it defines which characters are extracted, their interpretation and the type of its corresponding argument:

<b>specifier</b>	<b>Description</b>	<b>Characters extracted</b>
i	Integer	Any number of digits, optionally preceded by a sign (+ or -). <a href="#">Decimal digits</a> assumed by default (0–9), but a 0 prefix introduces octal digits (0–7), and 0x <a href="#">hexadecimal digits</a> (0–f). <i>Signed</i> argument.
d or u	Decimal integer	Any number of <a href="#">decimal digits</a> (0–9), optionally preceded by a sign (+ or -). d is for a <i>signed</i> argument, and u for an <i>unsigned</i> .
o	Octal integer	Any number of octal digits (0–7), optionally preceded by a sign (+ or -). <i>Unsigned</i> argument.
x	Hexadecimal integer	Any number of <a href="#">hexadecimal digits</a> (0–9, a–f, A–F), optionally preceded by 0x or 0X, and all optionally preceded by a sign (+ or -). <i>Unsigned</i> argument.
f, e, g	Floating point number	A series of <a href="#">decimal digits</a> , optionally containing a decimal point, optionally preceded by a sign (+ or -) and optionally followed by the e or E character and a decimal integer (or some of the other sequences supported by <a href="#">strtod</a> ). Implementations complying with C99 also support hexadecimal floating-point format when preceded by 0x or 0X.
a		
c	Character	The next character. If a <i>width</i> other than 1 is specified, the function reads exactly <i>width</i> characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.
s	String of characters	Any number of non-whitespace characters, stopping at the first <a href="#">whitespace</a> character found. A terminating null character is automatically added at the end of the stored sequence.
p	Pointer address	A sequence of characters representing a pointer. The particular format used depends on the system and library implementation, but it is the same as the one used to format %p in <a href="#">fprintf</a> .
[characters]	Scanset	Any number of the characters specified between the brackets. A dash (-) that is not the first character may produce non-portable behavior in some library implementations.
[^characters]	Negated scanset	Any number of characters none of them specified as <i>characters</i> between the brackets.
n	Count	No input is consumed. The number of characters read so far from <a href="#">stdin</a> is stored in the pointed location.
%	%	A % followed by another % matches a single %.

Except for n, at least one character shall be consumed by any specifier. Otherwise the match fails, and the scan ends there.

The *format specifier* can also contain sub-specifiers: asterisk (\*), width and length (in that order), which are optional and follow these specifications:

<b>sub-specifier</b>	<b>description</b>
*	An optional starting asterisk indicates that the data is to be read from the stream but ignored (i.e. it is not stored in the location pointed by an argument).
width	Specifies the maximum number of characters to be read in the current reading operation (optional).
length	One of hh, h, l, ll, j, z, t, L (optional). This alters the expected type of the storage pointed by the corresponding argument (see below).

This is a chart showing the types expected for the corresponding arguments where input is stored (both with and without a *length* sub-specifier):

	specifiers						
<b>length</b>	d i	u o x	f e g a	c s [] [^]	p	n	
(none)	int*	unsigned int*	float*	char*	void**	int*	
hh	signed char*	unsigned char*				signed char*	
h	short int*	unsigned short int*				short int*	
l	long int*	unsigned long int*	double*	wchar_t*		long int*	
ll	long long int*	unsigned long long int*				long long int*	
j	intmax_t*	uintmax_t*				intmax_t*	
z	size_t*	size_t*				size_t*	
t	ptrdiff_t*	ptrdiff_t*				ptrdiff_t*	
L			long double*				

**Note:** Yellow rows indicate specifiers and sub-specifiers introduced by C99.

... (additional arguments)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type.

There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.

These arguments are expected to be pointers: to store the result of a `scanf` operation on a regular variable, its name should be preceded by the *reference operator* (&) (see [example](#)).

## Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set ([feof](#) or [ferror](#)). And, if either happens before any data could be successfully read, [EOF](#) is returned.

If an encoding error happens interpreting wide characters, the function sets [errno](#) to [EILSEQ](#).

## Example

```
1 /* scanf example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char str [80];
7     int i;
8
9     printf ("Enter your family name: ");
10    scanf ("%79s",str);
11    printf ("Enter your age: ");
12    scanf ("%d",&i);
13    printf ("Mr. %s , %d years old.\n",str,i);
14    printf ("Enter a hexadecimal number: ");
15    scanf ("%x",&i);
16    printf ("You have entered %#x (%d).\n",i,i);
17
18    return 0;
19 }
```

This example demonstrates some of the types that can be read with `scanf`:

```
Enter your family name: Soulie
Enter your age: 29
Mr. Soulie , 29 years old.
Enter a hexadecimal number: ff
You have entered 0xff (255).
```

## Compatibility

Particular library implementations may support additional specifiers and sub-specifiers.

Those listed here are supported by the latest C and C++ standards (both published in 2011), but those in yellow were introduced by C99 (only required for C++ implementations since C++11), and may not be supported by libraries that comply with older standards.

## See also

<a href="#">fscanf</a>	Read formatted data from stream ( <a href="#">function</a> )
<a href="#">printf</a>	Print formatted data to stdout ( <a href="#">function</a> )
<a href="#">gets</a>	Get string from stdin ( <a href="#">function</a> )
<a href="#">fopen</a>	Open file ( <a href="#">function</a> )

## /cstdio/setbuf

function  
**setbuf** <cstdio>

```
void setbuf ( FILE * stream, char * buffer );
```

### Set stream buffer

Specifies the *buffer* to be used by the *stream* for I/O operations, which becomes a *fully buffered* stream. Or, alternatively, if *buffer* is a null pointer, buffering is disabled for the *stream*, which becomes an *unbuffered* stream.

This function should be called once the *stream* has been associated with an open file, but before any input or output operation is performed with it.

The buffer is assumed to be at least `BUFSIZ` bytes in size (see `setvbuf` to specify a size of the buffer).

A *stream buffer* is a block of data that acts as intermediary between the i/o operations and the physical file associated to the stream: For output buffers, data is output to the buffer until its maximum capacity is reached, then it is *flushed* (i.e.: all data is sent to the physical file at once and the buffer cleared). Likewise, input buffers are filled from the physical file, from which data is sent to the operations until exhausted, at which point new data is acquired from the file to fill the buffer again.

Stream buffers can be explicitly flushed by calling `fflush`. They are also automatically flushed by `fclose` and `freopen`, or when the program terminates normally.

A *full buffered stream* uses the entire size of the buffer as buffer whenever enough data is available (see `setvbuf` for other buffer modes).

All files are opened with a default allocated buffer (*fully buffered*) if they are known to not refer to an interactive device. This function can be used to either set a specific memory block to be used as buffer or to disable buffering for the stream.

The default streams `stdin` and `stdout` are *fully buffered* by default if they are known to not refer to an interactive device. Otherwise, they may either be *line buffered* or *unbuffered* by default, depending on the system and library implementation. The same is true for `stderr`, which is always either *line buffered* or *unbuffered* by default.

A call to this function is equivalent to calling `setvbuf` with `_IOFBF` as *mode* and `BUFSIZ` as *size* (when *buffer* is not a null pointer), or equivalent to calling it with `_IONBF` as *mode* (when it is a null pointer).

## Parameters

### stream

Pointer to a `FILE` object that identifies an open stream.

### buffer

User allocated buffer. Shall be at least `BUFSIZ` bytes long.  
Alternatively, a null pointer can be specified to disable buffering.

## Return Value

none.

## Example

```
1 /* setbuf example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char buffer[BUFSIZ];
7     FILE *pFile1, *pFile2;
8
9     pFile1=fopen ("myfile1.txt","w");
10    pFile2=fopen ("myfile2.txt","a");
11
12    setbuf ( pFile1 , buffer );
13    fputs ("This is sent to a buffered stream",pFile1);
14    fflush (pFile1);
15
16    setbuf ( pFile2 , NULL );
17    fputs ("This is sent to an unbuffered stream",pFile2);
18
19    fclose (pFile1);
20    fclose (pFile2);
21
22    return 0;
23 }
```

In this example, two files are opened for writing. The stream associated with the file `myfile1.txt` is set to a user allocated buffer; a writing operation to it is performed; the data is logically part of the stream, but it has not been written to the device until the `fflush` function is called. The second buffer in the example, associated with the file `myfile2.txt`, is set to unbuffered, so the subsequent output operation is written to the device as soon as possible. The final state, however, is the same for both buffered and unbuffered streams once the files have been closed (closing a file flushes its buffer).

## See also

<a href="#">setvbuf</a>	Change stream buffering (function )
<a href="#">fopen</a>	Open file (function )
<a href="#">fflush</a>	Flush stream (function )

## /cstdio/setvbuf

function

### setvbuf

<cstdio>

```
int setvbuf ( FILE * stream, char * buffer, int mode, size_t size );
```

#### Change stream buffering

Specifies a *buffer* for *stream*. The function allows to specify the *mode* and *size* of the buffer (in bytes).

If *buffer* is a null pointer, the function automatically allocates a buffer (using *size* as a hint on the size to use). Otherwise, the array pointed by *buffer* may be used as a buffer of *size* bytes.

This function should be called once the *stream* has been associated with an open file, but before any input or output operation is performed with it.

A *stream buffer* is a block of data that acts as intermediary between the i/o operations and the physical file associated to the stream: For output buffers, data is output to the buffer until its maximum capacity is reached, then it is **flushed** (i.e.: all data is sent to the physical file at once and the buffer cleared). Likewise, input buffers are filled from the physical file, from which data is sent to the operations until exhausted, at which point new data is acquired from the file to fill the buffer again.

Stream buffers can be explicitly flushed by calling `fflush`. They are also automatically flushed by `fclose` and `freopen`, or when the program terminates normally.

All files are opened with a default allocated buffer (*fully buffered*) if they are known to not refer to an interactive device. This function can be used to either redefine the buffer *size* or *mode*, to define a user-allocated buffer or to disable buffering for the stream.

The default streams `stdin` and `stdout` are *fully buffered* by default if they are known to not refer to an interactive device. Otherwise, they may either be *line buffered* or *unbuffered* by default, depending on the system and library implementation. The same is true for `stderr`, which is always either *line buffered* or *unbuffered* by default.

## Parameters

*stream*

Pointer to a `FILE` object that identifies an open stream.

*buffer*

User allocated buffer. Shall be at least *size* bytes long.

If set to a null pointer, the function automatically allocates a buffer.

*mode*

Specifies a mode for file buffering. Three special macro constants (`_IOFBF`, `_IOLBF` and `_IONBF`) are defined in `<cstdio>` to be used as the value for this parameter:

<code>_IOFBF</code>	<b>Full buffering:</b> On output, data is written once the buffer is full (or <code>flushed</code> ). On Input, the buffer is filled when an input operation is requested and the buffer is empty.
<code>_IOLBF</code>	<b>Line buffering:</b> On output, data is written when a newline character is inserted into the stream or when the buffer is full (or <code>flushed</code> ), whatever happens first. On Input, the buffer is filled up to the next newline character when an input operation is requested and the buffer is empty.
<code>_IONBF</code>	<b>No buffering:</b> No buffer is used. Each I/O operation is written as soon as possible. In this case, the <i>buffer</i> and <i>size</i> parameters are ignored.

**size**  
Buffer size, in bytes.  
If the *buffer* argument is a null pointer, this value may determine the size automatically allocated by the function for the buffer.

## Return Value

If the buffer is correctly assigned to the file, a zero value is returned.  
Otherwise, a non-zero value is returned; This may be due to an invalid *mode* parameter or to some other error allocating or assigning the buffer.

## Example

```
1 /* setvbuf example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE *pFile;
7
8     pFile=fopen ("myfile.txt","w");
9
10    setvbuf ( pFile , NULL , _IOFBF , 1024 );
11
12    // File operations here
13
14    fclose (pFile);
15
16    return 0;
17 }
```

In this example, a file called `myfile.txt` is created and a full buffer of 1024 bytes is requested for the associated stream, so the data output to this stream should only be written to the file each time the 1024-byte buffer is filled.

## See also

<b>setbuf</b>	Set stream buffer ( <a href="#">function</a> )
<b>fopen</b>	Open file ( <a href="#">function</a> )
<b>fflush</b>	Flush stream ( <a href="#">function</a> )

## /cstdio/size\_t

type

## size\_t

`<cstdint> <cstdio> <cstdlib> <cstring> <ctime> <cwchar>`

### Unsigned integral type

Alias of one of the fundamental unsigned integer types.

It is a type able to represent the size of any object in bytes: `size_t` is the type returned by the `sizeof` operator and is widely used in the standard library to represent sizes and counts.

In `<cstdio>`, it is used as the type of some parameters in the functions `fread`, `fwrite` and `setvbuf`, and in the case of `fread` and `fwrite` also as its returning type.

## /cstdio/snprintf

function

## snprintf

`<cstdio>`

```
int snprintf ( char * s, size_t n, const char * format, ... );
```

### Write formatted output to sized buffer

Composes a string with the same text that would be printed if `printf` was used on `printf`, but instead of being printed, the content is stored as a C *string* in the buffer pointed by *s* (taking *n* as the maximum buffer capacity to fill).

If the resulting string would be longer than *n*-1 characters, the remaining characters are discarded and not stored, but counted for the value returned by the function.

A terminating null character is automatically appended after the content written.

After the *format* parameter, the function expects at least as many additional arguments as needed for *format*.

## Parameters

*s*

Pointer to a buffer where the resulting C-string is stored.  
The buffer should have a size of at least *n* characters.

*n*

Maximum number of bytes to be used in the buffer.  
The generated string has a length of at most *n*-1, leaving space for the additional terminating null character.  
`size_t` is an unsigned integral type.

*format*

C string that contains a format string that follows the same specifications as *format* in `printf` (see `printf` for details).

*... (additional arguments)*

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for *n*).  
There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

## Return Value

The number of characters that would have been written if *n* had been sufficiently large, not counting the terminating *null character*.  
If an encoding error occurs, a negative number is returned.  
Notice that only when this returned value is non-negative and less than *n*, the string has been completely written.

## Example

```
1 /* snprintf example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char buffer [100];
7     int cx;
8
9     cx = sprintf ( buffer, 100, "The half of %d is %d", 60, 60/2 );
10
11    if (cx>=0 && cx<100)      // check returned value
12
13        sprintf ( buffer+cx, 100-cx, ", and the half of that is %d.", 60/2/2 );
14
15    puts (buffer);
16
17    return 0;
18 }
```

Output:

```
The half of 60 is 30, and the half of that is 15.
```

For more examples on formatting see [printf](#).

## See also

<a href="#">printf</a>	Print formatted data to stdout ( <a href="#">function</a> )
<a href="#">sprintf</a>	Write formatted data to string ( <a href="#">function</a> )
<a href="#">strcat</a>	Concatenate strings ( <a href="#">function</a> )
<a href="#">sscanf</a>	Read formatted data from string ( <a href="#">function</a> )

# /cstdio/sprintf

function

## sprintf

<cstdio>

```
int sprintf ( char * str, const char * format, ... );
```

### Write formatted data to string

Composes a string with the same text that would be printed if *format* was used on [printf](#), but instead of being printed, the content is stored as a C *string* in the buffer pointed by *str*.

The size of the buffer should be large enough to contain the entire resulting string (see [snprintf](#) for a safer version).

A terminating null character is automatically appended after the content.

After the *format* parameter, the function expects at least as many additional arguments as needed for *format*.

## Parameters

*str*

Pointer to a buffer where the resulting C-string is stored.  
The buffer should be large enough to contain the resulting string.

*format*

C string that contains a format string that follows the same specifications as *format* in [printf](#) (see [printf](#) for details).

... (*additional arguments*)

Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a *format specifier* in the *format* string (or a pointer to a storage location, for *n*).  
There should be at least as many of these arguments as the number of values specified in the *format specifiers*. Additional arguments are ignored by the function.

## Return Value

On success, the total number of characters written is returned. This count does not include the additional null-character automatically appended at the end of the string.

On failure, a negative number is returned.

## Example

```
1 /* sprintf example */
2 #include <stdio.h>
```

```

3
4 int main ()
5 {
6     char buffer [50];
7     int n, a=5, b=3;
8     n=sprintf (buffer, "%d plus %d is %d", a, b, a+b);
9     printf ("[%s] is a string %d chars long\n",buffer,n);
10    return 0;
11 }

```

Output:

```
[5 plus 3 is 8] is a string 13 chars long
```

## See also

---

## /cstdio/sscanf

function

### sscanf

<cstdio>

```
int sscanf ( const char * s, const char * format, ...);
```

#### Read formatted data from string

Reads data from *s* and stores them according to parameter *format* into the locations given by the additional arguments, as if [scanf](#) was used, but reading from *s* instead of the standard input ([stdin](#)).

The additional arguments should point to already allocated objects of the type specified by their corresponding format specifier within the *format* string.

#### Parameters

---

*s* C string that the function processes as its source to retrieve the data.  
*format* C string that contains a format string that follows the same specifications as *format* in [scanf](#) (see [scanf](#) for details).  
*...* (*additional arguments*) Depending on the *format* string, the function may expect a sequence of additional arguments, each containing a pointer to allocated storage where the interpretation of the extracted characters is stored with the appropriate type.  
There should be at least as many of these arguments as the number of values stored by the *format specifiers*. Additional arguments are ignored by the function.

#### Return Value

---

On success, the function returns the number of items in the argument list successfully filled. This count can match the expected number of items or be less (even zero) in the case of a matching failure.

In the case of an input failure before any data could be successfully interpreted, [EOF](#) is returned.

#### Example

---

```

1 /* sscanf example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char sentence []="Rudolph is 12 years old";
7     char str [20];
8     int i;
9
10    sscanf (sentence,"%s %*s %d",str,&i);
11    printf ("%s -> %d\n",str,i);
12
13    return 0;
14 }

```

Output:

```
Rudolph -> 12
```

## See also

---

<a href="#">scanf</a>	Read formatted data from <a href="#">stdin</a> ( <a href="#">function</a> )
<a href="#">sprintf</a>	Write formatted data to <a href="#">string</a> ( <a href="#">function</a> )

## /cstdio/stderr

object

### stderr

<cstdio>

[FILE](#) \* [stderr](#);

#### Standard error stream

The standard error stream is the default destination for error messages and other diagnostic warnings. Like [stdout](#), it is usually also directed by default to the text console (generally, on the screen).

`stderr` can be used as an argument for any function that takes an argument of type `FILE*` expecting an output stream, like `fputs` or `fprintf`.

Although in many cases both `stdout` and `stderr` are associated with the same output device (like the console), applications may differentiate between what is sent to `stdout` and what to `stderr` for the case that one of them is redirected. For example, it is frequent to redirect the regular output of a console program (`stdout`) to a file while expecting the error messages to keep appearing in the console.

It is also possible to redirect `stderr` to some other destination from within a program using the `freopen` function.

`stderr` is never *fully buffered* on startup. It is library-dependent whether the stream is *line buffered* or *not buffered* by default (see `setvbuf`).

## See also

<code>stdin</code>	Standard input stream (object )
<code>stdout</code>	Standard output stream (object )

## /cstdio/stdin

object

### stdin

<cstdio>

`FILE * stdin;`

#### Standard input stream

The standard input stream is the default source of data for applications. In most systems, it is usually directed by default to the keyboard.

`stdin` can be used as an argument for any function that expects an input stream (`FILE*`) as one of its parameters, like `fgets` or `fscanf`.

Although it is commonly assumed that the source of data for `stdin` is going to be a keyboard, this may not be the case even in regular console systems, since `stdin` can generally be redirected on most operating systems at the time of invoking the application. For example, many systems, among them DOS/Windows and most UNIX shells, support the following command syntax:

```
myapplication < example.txt
```

to use the content of the file `example.txt` as the primary source of data for `myapplication` instead of the console keyboard.

It is also possible to redirect `stdin` to some other source of data from within a program by using the `freopen` function.

If `stdin` is known to not refer to an interactive device, the stream is *fully buffered*. Otherwise, it is library-dependent whether the stream is *line buffered* or *not buffered* by default (see `setvbuf`).

## See also

<code>stdout</code>	Standard output stream (object )
<code>stderr</code>	Standard error stream (object )

## /cstdio/stdout

object

### stdout

<cstdio>

`FILE * stdout;`

#### Standard output stream

The standard output stream is the default destination of output for applications. In most systems, it is usually directed by default to the text console (generally, on the screen).

`stdout` can be used as an argument for any function that takes an argument of type `FILE*` expecting an output stream, like `fputs` or `fprintf`.

Although it is commonly assumed that the default destination for `stdout` is going to be the screen, this may not be the case even in regular console systems, since `stdout` can generally be redirected on most operating systems at the time of invoking the application. For example, many systems, among them DOS/Windows and most UNIX shells, support the following command syntax:

```
myapplication > example.txt  
to redirect the output of myapplication to the file example.txt instead of the console.
```

It is also possible to redirect `stdout` to some other source of data from within a program using the `freopen` function.

If `stdout` is known to not refer to an interactive device, the stream is *fully buffered*. Otherwise, it is library-dependent whether the stream is *line buffered* or *not buffered* by default (see `setvbuf`).

## See also

<code>stdin</code>	Standard input stream (object )
<code>stderr</code>	Standard error stream (object )

## /cstdio/tmpfile

function

### tmpfile

<cstdio>

`FILE * tmpfile ( void );`

## Open a temporary file

Creates a temporary binary file, open for update ("wb+" mode, see [fopen](#) for details) with a filename guaranteed to be different from any other existing file.

The temporary file created is automatically deleted when the stream is closed ([fclose](#)) or when the program terminates normally. If the program terminates abnormally, whether the file is deleted depends on the specific system and library implementation.

### Parameters

none

### Return Value

If successful, the function returns a stream pointer to the temporary file created.

On failure, [NULL](#) is returned.

### Example

```
1 /* tmpfile example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char buffer [256];
8     FILE * pFile;
9     pFile = tmpfile ();
10
11    do {
12        if (!fgets(buffer,256,stdin)) break;
13        fputs (buffer,pFile);
14    } while (strlen(buffer)>1);
15
16    rewind(pFile);
17
18    while (!feof(pFile)) {
19        if (fgets (buffer,256,pFile) == NULL) break;
20        fputs (buffer,stdout);
21    }
22
23    fclose (pFile);
24    return 0;
25 }
```

This program creates a temporary file to store the lines entered by the user. When the user enters an empty line, the program rewinds the temporary file and prints its contents to [stdout](#).

### See also

<a href="#">fopen</a>	Open file (function )
<a href="#">tmpnam</a>	Generate temporary filename (function )

## /cstdio/TMP\_MAX

constant

### TMP\_MAX

<cstdio>

#### Number of temporary files

This macro expands to the minimum number of unique temporary file names that are guaranteed to be possible to generate using [tmpnam](#).

This value cannot be lower than 25.

Particular library implementations may count file names used by files created with [tmpfile](#) towards this limit.

### See also

<a href="#">FOPEN_MAX</a>	Potential limit of simultaneous open streams (constant )
---------------------------	--

## /cstdio/tmpnam

function

### tmpnam

<cstdio>

```
char * tmpnam ( char * str );
```

#### Generate temporary filename

Returns a string containing a *file name* different from the name of any existing file, and thus suitable to safely create a temporary file without risking to overwrite an existing file.

If *str* is a null pointer, the resulting string is stored in an internal static array that can be accessed by the return value. The content of this string is preserved at least until a subsequent call to this same function, which may overwrite it.

If *str* is not a null pointer, it shall point to an array of at least [L\\_tmpnam](#) characters that will be filled with the proposed temporary file name.

The file name returned by this function can be used to create a regular file using [fopen](#) to be used as a temporary file. The file created this way, unlike those created with [tmpfile](#) is not automatically deleted when closed; A program shall call [remove](#) to delete this file once closed.

## Parameters

str

Pointer to an array of characters where the proposed temporary name will be stored as a C string. The suggested size of this array is at least [L\\_tmpnam](#) characters.  
Alternatively, a null pointer can be specified to use an internal static array to store the proposed temporary name, whose pointer is returned by the function.

## Return Value

On success, a pointer to the C string containing the proposed name for a temporary file:

- If str was a null pointer, this points to an internal buffer (whose content is preserved at least until the next call to this function).
- If str was not a null pointer, str is returned.

If the function fails to create a suitable filename, it returns a null pointer.

## Example

```
1 /* tmpnam example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     char buffer [L_tmpnam];
7     char * pointer;
8
9     tmpnam (buffer);
10    printf ("Tempname #1: %s\n",buffer);
11
12    pointer = tmpnam (NULL);
13    printf ("Tempname #2: %s\n",pointer);
14
15    return 0;
16 }
```

This program will generate two different names for temporary files. Each one has been created by one of the two methods in which `tmpnam` can be used.

Possible output:

```
Tempname #1: /s4s4.
Tempname #2: /s4s4.1
```

## See also

<a href="#">fopen</a>	Open file (function )
<a href="#">tmpfile</a>	Open a temporary file (function )

# /cstdio/ungetc

function

## ungetc

<cstdio>

```
int ungetc ( int character, FILE * stream );
```

### Unget character from stream

A *character* is virtually put back into an input *stream*, decreasing its *internal file position* as if a previous `getc` operation was undone.

This *character* may or may not be the one read from the *stream* in the preceding input operation. In any case, the next character retrieved from *stream* is the *character* passed to this function, independently of the original one.

Notice though, that this only affects further input operations on that *stream*, and not the content of the physical file associated with it, which is not modified by any calls to this function.

Some library implementations may support this function to be called multiple times, making the characters available in the reverse order in which they were *put back*. Although this behavior has no standard portability guarantees, and further calls may simply fail after any number of calls beyond the first.

If successful, the function clears the *end-of-file indicator* of *stream* (if it was currently set), and decrements its internal *file position indicator* if it operates in binary mode; In text mode, the *position indicator* has unspecified value until all characters put back with `ungetc` have been read or discarded.

A call to `fseek`, `fsetpos` or `rewind` on *stream* will discard any characters previously put back into it with this function.

If the argument passed for the *character* parameter is `EOF`, the operation fails and the input *stream* remains unchanged.

## Parameters

character

The `int` promotion of the character to be put back.  
The value is internally converted to an `unsigned char` when put back.

stream

Pointer to a `FILE` object that identifies an input stream.

## Return Value

On success, the *character* put back is returned.  
If the operation fails, `EOF` is returned.

## Example

```
1 /* ungetc example */
2 #include <stdio.h>
3
4 int main ()
5 {
6     FILE * pFile;
7     int c;
8     char buffer [256];
9
10    pFile = fopen ("myfile.txt","rt");
11    if (pFile==NULL) perror ("Error opening file");
12    else while (!feof (pFile)) {
13        c=getc (pFile);
14        if (c == EOF) break;
15        if (c == '#') ungetc ('\0',pFile);
16        else ungetc (c,pFile);
17        if (fgets (buffer,255,pFile) != NULL)
18            fputs (buffer,stdout);
19        else break;
20    }
21    return 0;
22 }
```

This example opens an existing file called `myfile.txt` for reading and prints its lines, but first gets the first character of every line and puts it back into the stream replacing any starting `#` by an `\0`.

## See also

<a href="#">getc</a>	Get character from stream (function )
<a href="#">fgetc</a>	Get character from stream (function )
<a href="#">putc</a>	Write character to stream (function )

## /cstdio/vfprintf

function  
**vfprintf** <cstdio>

```
int vfprintf ( FILE * stream, const char * format, va_list arg );
```

### Write formatted data from variable argument list to stream

Writes the C string pointed by *format* to the *stream*, replacing any *format specifier* in the same way as `printf` does, but using the elements in the variable argument list identified by *arg* instead of additional function arguments.

Internally, the function retrieves arguments from the list identified by *arg* as if `va_arg` was used on it, and thus the state of *arg* is likely altered by the call.

In any case, *arg* should have been initialized by `va_start` at some point before the call, and it is expected to be released by `va_end` at some point after the call.

## Parameters

### stream

Pointer to a `FILE` object that identifies an output stream.

### format

C string that contains a format string that follows the same specifications as *format* in `printf` (see `printf` for details).

### arg

A value identifying a variable arguments list initialized with `va_start`.

`va_list` is a special type defined in `<cstdarg.h>`.

## Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (`ferror`) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, `errno` is set to `EILSEQ` and a negative number is returned.

## Example

```
1 /* vfprintf example */
2 #include <stdio.h>
3 #include <stdarg.h>
4
5 void WriteFormatted (FILE * stream, const char * format, ...)
6 {
7     va_list args;
8     va_start (args, format);
9     vfprintf (stream, format, args);
10    va_end (args);
11 }
12
13 int main ()
14 {
15     FILE * pFile;
```

```

16 pFile = fopen ("myfile.txt","w");
17
18 WriteFormatted (pFile,"Call with %d variable argument.\n",1);
19 WriteFormatted (pFile,"Call with %d variable %s.\n",2,"arguments");
20
21 fclose (pFile);
22
23 return 0;
24 }

```

The example demonstrates how the `WriteFormatted` can be called with a different number of arguments, which are on their turn passed to the `vfprintf` function.

`myfile.txt` would contain:

myfile.txt
Call with 1 variable argument.
Call with 2 variable arguments.

## See also

<a href="#">vprintf</a>	Print formatted data from variable argument list to stdout (function )
<a href="#">vsprintf</a>	Write formatted data from variable argument list to string (function )
<a href="#">fprintf</a>	Write formatted data to stream (function )
<a href="#">printf</a>	Print formatted data to stdout (function )

## /cstdio/vfscanf

function  
**vfscanf** <cstdio>

```
int vfscanf ( FILE * stream, const char * format, va_list arg );
```

### Read formatted data from stream into variable argument list

Reads data from the *stream* and stores them according to parameter *format* into the locations pointed by the elements in the variable argument list identified by *arg*.

Internally, the function retrieves arguments from the list identified by *arg* as if `va_arg` was used on it, and thus the state of *arg* is likely to be altered by the call.

In any case, *arg* should have been initialized by `va_start` at some point before the call, and it is expected to be released by `va_end` at some point after the call.

## Parameters

<code>stream</code>	Pointer to a <code>FILE</code> object that identifies an input stream.
<code>format</code>	C string that contains a format string that follows the same specifications as <i>format</i> in <code>scanf</code> (see <code>scanf</code> for details).
<code>arg</code>	A value identifying a variable arguments list initialized with <code>va_start</code> . <code>va_list</code> is a special type defined in <code>&lt;cstdarg.h&gt;</code> .

## Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (`feof` or `ferror`). And, if either happens before any data could be successfully read, `EOF` is returned.

If an encoding error happens interpreting wide characters, the function sets `errno` to `EILSEQ`.

## Example

```

1 /* vfscanf example */
2 #include <stdio.h>
3 #include <stdarg.h>
4
5 void ReadStuff (FILE * stream, const char * format, ...)
6 {
7     va_list args;
8     va_start (args, format);
9     vfscanf (stream, format, args);
10    va_end (args);
11 }
12
13 int main ()
14 {
15     FILE * pFile;
16     int val;
17     char str[100];
18
19     pFile = fopen ("myfile.txt","r");
20
21     if (pFile!=NULL) {
22         ReadStuff ( pFile, " %s %d ", str, &val );
23         printf ("I have read %s and %d", str, val);
24         fclose (pFile);

```

```
25 }
26 return 0;
27 }
```

## See also

<a href="#">vscanf</a>	Read formatted data into variable argument list (function )
<a href="#">vscanf</a>	Read formatted data from string into variable argument list (function )
<a href="#">fscanf</a>	Read formatted data from stream (function )
<a href="#">scanf</a>	Read formatted data from stdin (function )

## /cstdio/vprintf

function

### vprintf

<cstdio>

```
int vprintf ( const char * format, va_list arg );
```

#### Print formatted data from variable argument list to stdout

Writes the C string pointed by *format* to the standard output (`stdout`), replacing any *format specifier* in the same way as `printf` does, but using the elements in the variable argument list identified by *arg* instead of additional function arguments.

Internally, the function retrieves arguments from the list identified by *arg* as if `va_arg` was used on it, and thus the state of *arg* is likely altered by the call.

In any case, *arg* should have been initialized by `va_start` at some point before the call, and it is expected to be released by `va_end` at some point after the call.

## Parameters

format

C string that contains a format string that follows the same specifications as *format* in `printf` (see `printf` for details).

arg

A value identifying a variable arguments list initialized with `va_start`.

`va_list` is a special type defined in `<cstdarg.h>`.

## Return Value

On success, the total number of characters written is returned.

If a writing error occurs, the *error indicator* (`ferror`) is set and a negative number is returned.

If a multibyte character encoding error occurs while writing wide characters, `errno` is set to `EILSEQ` and a negative number is returned.

## Example

```
1 /* vprintf example */
2 #include <stdio.h>
3 #include <stdarg.h>
4
5 void WriteFormatted ( const char * format, ... )
6 {
7     va_list args;
8     va_start (args, format);
9     vprintf (format, args);
10    va_end (args);
11 }
12
13 int main ()
14 {
15     WriteFormatted ("Call with %d variable argument.\n",1);
16     WriteFormatted ("Call with %d variable %s.\n",2,"arguments");
17
18     return 0;
19 }
```

The example illustrates how the `WriteFormatted` can be called with a different number of arguments, which are on their turn passed to the `vprintf` function, showing the following output:

```
Call with 1 variable argument.
Call with 2 variable arguments.
```

## See also

<a href="#">vfprintf</a>	Write formatted data from variable argument list to stream (function )
<a href="#">vsprintf</a>	Write formatted data from variable argument list to string (function )
<a href="#">printf</a>	Print formatted data to stdout (function )

## /cstdio/vscanf

function

### vscanf

<cstdio>

```
int vscanf ( const char * format, va_list arg );
```

#### Read formatted data into variable argument list

Reads data from the standard input (`stdin`) and stores them according to parameter `format` into the locations pointed by the elements in the variable argument list identified by `arg`.

Internally, the function retrieves arguments from the list identified by `arg` as if `va_arg` was used on it, and thus the state of `arg` is likely to be altered by the call.

In any case, `arg` should have been initialized by `va_start` at some point before the call, and it is expected to be released by `va_end` at some point after the call.

#### Parameters

##### format

C string that contains a format string that follows the same specifications as `format` in `scanf` (see `scanf` for details).

##### arg

A value identifying a variable arguments list initialized with `va_start`.

`va_list` is a special type defined in `<cstdarg>`.

#### Return Value

On success, the function returns the number of items of the argument list successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*.

If a reading error happens or the *end-of-file* is reached while reading, the proper indicator is set (`feof` or `ferror`). And, if either happens before any data could be successfully read, `EOF` is returned.

If an encoding error happens interpreting wide characters, the function sets `errno` to `EILSEQ`.

#### Example

```
1 /* vscanf example */
2 #include <stdio.h>
3 #include <stdarg.h>
4
5 void GetMatches ( const char * format, ... )
6 {
7     va_list args;
8     va_start (args, format);
9     vscanf (format, args);
10    va_end (args);
11 }
12
13 int main ()
14 {
15     int val;
16     char str[100];
17
18     printf ("Please enter a number and a word: ");
19     fflush (stdout);
20     GetMatches (" %d %99s ", &val, str);
21     printf ("Number read: %d\nWord read: %s\n", val, str);
22
23     return 0;
24 }
```

Possible output:

```
Please enter a number and a word: 911 airport
Number read: 911
Word read: airport
```

#### See also

<a href="#">vfscanf</a>	Read formatted data from stream into variable argument list (function )
<a href="#">vsscanf</a>	Read formatted data from string into variable argument list (function )
<a href="#">fscanf</a>	Read formatted data from stream (function )
<a href="#">scanf</a>	Read formatted data from <code>stdin</code> (function )

## /cstdio/vsnprintf

function

#### vsnprintf

`<cstdio>`

```
int vsnprintf (char * s, size_t n, const char * format, va_list arg );
```

#### Write formatted data from variable argument list to sized buffer

Composes a string with the same text that would be printed if `printf` was used on `printf`, but using the elements in the variable argument list identified by `arg` instead of additional function arguments and storing the resulting content as a C string in the buffer pointed by `s` (taking `n` as the maximum buffer capacity to fill).

If the resulting string would be longer than `n-1` characters, the remaining characters are discarded and not stored, but counted for the value returned by the function.

Internally, the function retrieves arguments from the list identified by `arg` as if `va_arg` was used on it, and thus the state of `arg` is likely to be altered by the call.

In any case, `arg` should have been initialized by `va_start` at some point before the call, and it is expected to be released by `va_end` at some point after the call.

## Parameters

---

s	Pointer to a buffer where the resulting C-string is stored. The buffer should have a size of at least <i>n</i> characters.
n	Maximum number of bytes to be used in the buffer. The generated string has a length of at most <i>n</i> -1, leaving space for the additional terminating null character. <i>size_t</i> is an unsigned integral type.
format	C string that contains a format string that follows the same specifications as <i>format</i> in <code>printf</code> (see <code>printf</code> for details).
arg	A value identifying a variable arguments list initialized with <code>va_start</code> . <code>va_list</code> is a special type defined in <code>&lt;cstdarg&gt;</code> .

## Return Value

---

The number of characters that would have been written if *n* had been sufficiently large, not counting the terminating *null character*. If an encoding error occurs, a negative number is returned. Notice that only when this returned value is non-negative and less than *n*, the string has been completely written.

## Example

---

```
1 /* vsnprintf example */
2 #include <stdio.h>
3 #include <stdarg.h>
4
5 void PrintFError ( const char * format, ... )
6 {
7     char buffer[256];
8     va_list args;
9     va_start (args, format);
10    vsnprintf (buffer,256,format, args);
11    perror (buffer);
12    va_end (args);
13 }
14
15 int main ()
16 {
17     FILE * pFile;
18     char szFileName[]="myfile.txt";
19
20     pFile = fopen (szFileName,"r");
21     if (pFile == NULL)
22         PrintFError ("Error opening '%s'",szFileName);
23     else
24     {
25         // file successfully open
26         fclose (pFile);
27     }
28     return 0;
29 }
```

In this example, if the file `myfile.txt` does not exist, `perror` is called to show an error message similar to:

```
Error opening file 'myfile.txt': No such file or directory
```

## See also

---

<a href="#">vfprintf</a>	Write formatted data from variable argument list to stream (function )
<a href="#">vprintf</a>	Print formatted data from variable argument list to stdout (function )
<a href="#">sprintf</a>	Write formatted data to string (function )
<a href="#">printf</a>	Print formatted data to stdout (function )

## /cstdio/vsprintf

function  
**vprintf** `<cstdio>`  
`int vsprintf (char * s, const char * format, va_list arg );`

### Write formatted data from variable argument list to string

Composes a string with the same text that would be printed if `format` was used on `printf`, but using the elements in the variable argument list identified by `arg` instead of additional function arguments and storing the resulting content as a *C string* in the buffer pointed to by `s`.

Internally, the function retrieves arguments from the list identified by `arg` as if `va_arg` was used on it, and thus the state of `arg` is likely to be altered by the call.

In any case, `arg` should have been initialized by `va_start` at some point before the call, and it is expected to be released by `va_end` at some point after the call.

## Parameters

---

s	Pointer to a buffer where the resulting C-string is stored. The buffer should be large enough to contain the resulting string.
---	---

**format**  
 C string that contains a format string that follows the same specifications as *format* in `printf` (see `printf` for details).

**arg**  
 A value identifying a variable arguments list initialized with `va_start`.  
`va_list` is a special type defined in `<cstdarg.h>`.

## Return Value

On success, the total number of characters written is returned.  
 On failure, a negative number is returned.

## Example

```

1 /* vsprintf example */
2 #include <stdio.h>
3 #include <stdarg.h>
4
5 void PrintFError ( const char * format, ... )
6 {
7     char buffer[256];
8     va_list args;
9     va_start (args, format);
10    vsprintf (buffer,format, args);
11    perror (buffer);
12    va_end (args);
13 }
14
15 int main ()
16 {
17     FILE * pFile;
18     char szFileName[]="myfile.txt";
19
20     pFile = fopen (szFileName,"r");
21     if (pFile == NULL)
22         PrintFError ("Error opening '%s'",szFileName);
23     else
24     {
25         // file successfully open
26         fclose (pFile);
27     }
28     return 0;
29 }
```

In this example, if the file `myfile.txt` does not exist, `perror` is called to show an error message similar to:

```
Error opening file 'myfile.txt': No such file or directory
```

## See also

<a href="#">vfprintf</a>	Write formatted data from variable argument list to stream (function )
<a href="#">vprintf</a>	Print formatted data from variable argument list to stdout (function )
<a href="#">sprintf</a>	Write formatted data to string (function )
<a href="#">printf</a>	Print formatted data to stdout (function )

## /cstdio/vscanf

function

**vscanf** <cstdio>

---

```
int vscanf ( const char * s, const char * format, va_list arg );
```

**Read formatted data from string into variable argument list**

Reads data from *s* and stores them according to parameter *format* into the locations pointed by the elements in the variable argument list identified by *arg*.

Internally, the function retrieves arguments from the list identified by *arg* as if `va_arg` was used on it, and thus the state of *arg* is likely to be altered by the call.

In any case, *arg* should have been initialized by `va_start` at some point before the call, and it is expected to be released by `va_end` at some point after the call.

## Parameters

**s**  
 C string that the function processes as its source to retrieve the data.

**format**  
 C string that contains a format string that follows the same specifications as *format* in `scanf` (see `scanf` for details).

**arg**  
 A value identifying a variable arguments list initialized with `va_start`.  
`va_list` is a special type defined in `<cstdarg.h>`.

## Return Value

On success, the function returns the number of items in the argument list successfully filled. This count can match the expected number of items or be less - even zero- in the case of a matching failure.

In the case of an input failure before any data could be successfully interpreted, `EOF` is returned.

## Example

```
1 /* vsscanf example */
2 #include <stdio.h>
3 #include <stdarg.h>
4
5 void GetMatches ( const char * str, const char * format, ... )
6 {
7     va_list args;
8     va_start (args, format);
9     vsscanf (str, format, args);
10    va_end (args);
11 }
12
13 int main ()
14 {
15     int val;
16     char buf[100];
17
18     GetMatches ( "99 bottles of beer on the wall", " %d %s ", &val, buf );
19
20     printf ("Product: %s\nQuantity: %d\n", buf, val );
21
22     return 0;
23 }
```

Possible output:

```
Product: bottles
Quantity: 99
```

## See also

<b>vscanf</b>	Read formatted data into variable argument list ( <a href="#">function</a> )
<b>vfscanf</b>	Read formatted data from stream into variable argument list ( <a href="#">function</a> )
<b>sscanf</b>	Read formatted data from string ( <a href="#">function</a> )
<b>scanf</b>	Read formatted data from stdin ( <a href="#">function</a> )
<b>vsprintf</b>	Write formatted data from variable argument list to string ( <a href="#">function</a> )

## /cstdlib

header

### <cstdlib> (stdlib.h)

#### C Standard General Utilities Library

This header defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetics, searching, sorting and converting.

#### Functions

##### String conversion

<b>atof</b>	Convert string to double ( <a href="#">function</a> )
<b>atoi</b>	Convert string to integer ( <a href="#">function</a> )
<b>atol</b>	Convert string to long integer ( <a href="#">function</a> )
<b>atoll</b>	Convert string to long long integer ( <a href="#">function</a> )
<b>strtod</b>	Convert string to double ( <a href="#">function</a> )
<b>strtof</b>	Convert string to float ( <a href="#">function</a> )
<b>strtol</b>	Convert string to long integer ( <a href="#">function</a> )
<b>strtold</b>	Convert string to long double ( <a href="#">function</a> )
<b>strtoll</b>	Convert string to long long integer ( <a href="#">function</a> )
<b>strtoul</b>	Convert string to unsigned long integer ( <a href="#">function</a> )
<b>strtoull</b>	Convert string to unsigned long long integer ( <a href="#">function</a> )

##### Pseudo-random sequence generation

<b>rand</b>	Generate random number ( <a href="#">function</a> )
<b>srand</b>	Initialize random number generator ( <a href="#">function</a> )

##### Dynamic memory management

<b>calloc</b>	Allocate and zero-initialize array ( <a href="#">function</a> )
<b>free</b>	Deallocate memory block ( <a href="#">function</a> )
<b>malloc</b>	Allocate memory block ( <a href="#">function</a> )
<b>realloc</b>	Reallocate memory block ( <a href="#">function</a> )

##### Environment

<b>abort</b>	Abort current process ( <a href="#">function</a> )
<b>atexit</b>	Set function to be executed on exit ( <a href="#">function</a> )
<b>at_quick_exit</b>	Set function to be executed on quick exit ( <a href="#">function</a> )

<a href="#">exit</a>	Terminate calling process (function )
<a href="#">getenv</a>	Get environment string (function )
<a href="#">quick_exit</a>	Terminate calling process quick (function )
<a href="#">system</a>	Execute system command (function )
<a href="#">_Exit</a>	Terminate calling process (function )

#### Searching and sorting

<a href="#">bsearch</a>	Binary search in array (function )
<a href="#">qsort</a>	Sort elements of array (function )

#### Integer arithmetics

<a href="#">abs</a>	Absolute value (function )
<a href="#">div</a>	Integral division (function )
<a href="#">labs</a>	Absolute value (function )
<a href="#">ldiv</a>	Integral division (function )
<a href="#">llabs</a>	Absolute value (function )
<a href="#">lldiv</a>	Integral division (function )

#### Multibyte characters

<a href="#">mblen</a>	Get length of multibyte character (function )
<a href="#">mbtowc</a>	Convert multibyte sequence to wide character (function )
<a href="#">wctomb</a>	Convert wide character to multibyte sequence (function )

#### Multibyte strings

<a href="#">mbstowcs</a>	Convert multibyte string to wide-character string (function )
<a href="#">wcstombs</a>	Convert wide-character string to multibyte string (function )

#### Macro constants

<a href="#">EXIT_FAILURE</a>	Failure termination code (macro )
<a href="#">EXIT_SUCCESS</a>	Success termination code (macro )
<a href="#">MB_CUR_MAX</a>	Maximum size of multibyte characters (macro )
<a href="#">NULL</a>	Null pointer (macro )
<a href="#">RAND_MAX</a>	Maximum value returned by rand (macro )

#### Types

<a href="#">div_t</a>	Structure returned by div (type )
<a href="#">ldiv_t</a>	Structure returned by ldiv (type )
<a href="#">lldiv_t</a>	Structure returned by lldiv (type )
<a href="#">size_t</a>	Unsigned integral type (type )

## /cstdlib/abort

function  
**abort** <cstdlib>  

```
void abort (void);
[[noreturn]] void abort() noexcept;
```

#### Abort current process

Aborts the current process, producing an abnormal program termination.

The function raises the SIGABRT signal (as if `raise(SIGABRT)` was called). This, if uncaught, causes the program to terminate returning a platform-dependent unsuccessful termination error code to the host environment.

The program is terminated without destroying any object and without calling any of the functions passed to `atexit` or `at_quick_exit`.

#### Parameters

none

#### Return Value

none (the function never returns).

#### Example

```
1 /* abort example */
2 #include <stdio.h>      /* fopen, fputs, fclose, stderr */
3 #include <stdlib.h>      /* abort, NULL */
4
5 int main ()
6 {
7     FILE * pFile;
8     pFile= fopen ("myfile.txt","r");
```

```

9  if (pFile == NULL)
10 {
11   fputs ("error opening file\n",stderr);
12   abort();
13 }
14
15 /* regular process here */
16
17 fclose (pFile);
18 return 0;
19 }

```

If `myfile.txt` does not exist, a message is printed and `abort` is called.

## Data races

Concurrently calling this function is safe, causing no data races.  
Note though that its *handling* process may affect all threads.

## Exceptions (C++)

If no function handlers have been defined with `signal` to handle `SIGABRT`, the function never throws exceptions (no-throw guarantee). Otherwise, the behavior depends on the particular library implementation.

## See also

<code>exit</code>	Terminate calling process (function )
<code>atexit</code>	Set function to be executed on exit (function )

## /cstdlib/abs

function  
**abs** <cstdlib>

```

int abs (int n);
int abs (    int n);
long int abs (long int n);
long int abs (    long int n);
long long int abs (long long int n);

```

### Absolute value

Returns the absolute value of parameter *n* (  $|n|$  ).

In C++, this function is also overloaded in header `<cmath>` for floating-point types (see `cmath abs`), in header `<complex>` for complex numbers (see `complex abs`), and in header `<valarray>` for valarrays (see `valarray abs`).

## Parameters

*n*  
 Integral value.

## Return Value

The absolute value of *n*.

## Portability

In C, only the `int` version exists.

For the `long int` equivalent see `labs`.

For the `long long int` equivalent see `llabs`.

## Example

```

1 /* abs example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* abs */
4
5 int main ()
6 {
7   int n,m;
8   n=abs(23);
9   m=abs(-11);
10  printf ("n=%d\n",n);
11  printf ("m=%d\n",m);
12  return 0;
13 }

```

Output:

```

n=23
m=11

```

## Data races

Concurrently calling this function is safe, causing no data races.

## Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If the result cannot be represented by the returned type (such as `abs(INT_MIN)` in an implementation with two's complement signed values), it causes *undefined behavior*.

## See also

<code>labs</code>	Absolute value (function)
<code>fabs</code>	Compute absolute value (function)
<code>div</code>	Integral division (function)

# /cstdlib/atexit

function **atexit** <cstdlib>

```
int atexit (void (*func)(void));
extern "C" int atexit (void (*func)(void));
extern "C++" int atexit (void (*func)(void));
extern "C" int atexit (void (*func)(void)) noexcept;
extern "C++" int atexit (void (*func)(void)) noexcept;
```

**Set function to be executed on exit**

The function pointed by `func` is automatically called without arguments when the program terminates normally.

If more than one `atexit` function has been specified by different calls to this function, they are all executed in reverse order as a stack (i.e. the last function specified is the first to be executed at exit).

A single function can be registered to be executed at exit more than once.

If `atexit` is called after `exit`, the call may or may not succeed depending on the particular system and library implementation (*unspecified behavior*).

If a function registered with `atexit` throws an exception for which it does not provide a handler when called on termination, `terminate` is automatically called (C++).

Particular library implementations may impose a limit on the number of functions call that can be registered with `atexit`, but this cannot be less than 32 function calls.

## Parameters

function  
Function to be called. The function shall return no value and take no arguments.

## Return Value

A zero value is returned if the function was successfully registered.

If it failed, a non-zero value is returned.

## Example

```
1 /* atexit example */
2 #include <stdio.h>          /* puts */
3 #include <stdlib.h>          /* atexit */
4
5 void fnExit1 (void)
6 {
7     puts ("Exit function 1.");
8 }
9
10 void fnExit2 (void)
11 {
12     puts ("Exit function 2.");
13 }
14
15 int main ()
16 {
17     atexit (fnExit1);
18     atexit (fnExit2);
19     puts ("Main function.");
20     return 0;
21 }
```

Output:

```
Main function.
Exit function 2.
Exit function 1.
```

## Data races

Concurrently calling this function introduces no data races (C++): Calls are properly synchronized at the process level.

## Exceptions (C++)

No-throw guarantee: this function never throws exceptions.

### See also

<a href="#">exit</a>	Terminate calling process (function )
<a href="#">abort</a>	Abort current process (function )

## /cstdlib/atof

function  
**atof** <cstdlib>

```
double atof (const char* str);
```

### Convert string to double

Parses the C string *str*, interpreting its content as a floating point number and returns its value as a *double*.

The function first discards as many whitespace characters (as in [isspace](#)) as necessary until the first non-whitespace character is found. Then, starting from this character, takes as many characters as possible that are valid following a syntax resembling that of floating point literals (see below), and interprets them as a numerical value. The rest of the string after the last valid character is ignored and has no effect on the behavior of this function.

A valid floating point number for *atof* using the "c" locale is formed by an optional sign character (+ or -), followed by a sequence of digits, optionally containing a decimal-point character (.), optionally followed by an exponent part (an e or E character followed by an optional sign and a sequence of digits).

A valid floating point number for *atof* using the "c" locale is formed by an optional sign character (+ or -), followed by one of:

- A sequence of digits, optionally containing a decimal-point character (.), optionally followed by an exponent part (an e or E character followed by an optional sign and a sequence of digits).
- A 0x or 0X prefix, then a sequence of hexadecimal digits (as in [isxdigit](#)) optionally containing a period which separates the whole and fractional number parts. Optionally followed by a power of 2 exponent (a p or P character followed by an optional sign and a sequence of hexadecimal digits).
- INF or INFINITY (ignoring case).
- NAN or NANsequence (ignoring case), where *sequence* is a sequence of characters, where each character is either an alphanumeric character (as in [isalnum](#)) or the underscore character (\_).

If the first sequence of non-whitespace characters in *str* does not form a valid floating-point number as just defined, or if no such sequence exists because either *str* is empty or contains only whitespace characters, no conversion is performed and the function returns 0.0.

### Parameters

*str* C-string beginning with the representation of a floating-point number.

### Return Value

On success, the function returns the converted floating point number as a *double* value.

If no valid conversion could be performed, the function returns zero (0.0).

If the converted value would be out of the range of representable values by a *double*, it causes *undefined behavior*. See [strtod](#) for a more robust cross-platform alternative when this is a possibility.

### Example

```
1 /* atof example: sine calculator */
2 #include <stdio.h>           /* printf, fgets */
3 #include <stdlib.h>           /* atof */
4 #include <math.h>              /* sin */
5
6 int main ()
7 {
8     double n,m;
9     double pi=3.1415926535;
10    char buffer[256];
11    printf ("Enter degrees: ");
12    fgets (buffer,256,stdin);
13    n = atof (buffer);
14    m = sin (n*pi/180);
15    printf ("The sine of %f degrees is %f\n" , n, m);
16    return 0;
17 }
```

Output:

```
Enter degrees: 45
The sine of 45.000000 degrees is 0.707101
```

### Data races

The array pointed by *str* is accessed.

## Exceptions (C++)

No-throw guarantee: this function never throws exceptions.

If *str* does not point to a valid C-string, or if the converted value would be out of the range of values representable by a *double*, it causes *undefined behavior*.

### See also

<b>strtod</b>	Convert string to double (function )
<b>atoi</b>	Convert string to integer (function )
<b>atol</b>	Convert string to long integer (function )

## /cstdlib/atoi

function

### atoi

<cstdlib>

```
int atoi (const char * str);
```

#### Convert string to integer

Parses the C-string *str* interpreting its content as an integral number, which is returned as a value of type *int*.

The function first discards as many whitespace characters (as in *isspace*) as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial *plus* or *minus* sign followed by as many base-10 digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in *str* is not a valid integral number, or if no such sequence exists because either *str* is empty or it contains only whitespace characters, no conversion is performed and zero is returned.

#### Parameters

*str*

C-string beginning with the representation of an integral number.

#### Return Value

On success, the function returns the converted integral number as an *int* value.

If the converted value would be out of the range of representable values by an *int*, it causes *undefined behavior*. See *strtol* for a more robust cross-platform alternative when this is a possibility.

#### Example

```
1 /* atoi example */
2 #include <stdio.h>      /* printf, fgets */
3 #include <stdlib.h>      /* atoi */
4
5 int main ()
6 {
7     int i;
8     char buffer[256];
9     printf ("Enter a number: ");
10    fgets (buffer, 256, stdin);
11    i = atoi (buffer);
12    printf ("The value entered is %d. Its double is %.2f.\n",i,i*2);
13    return 0;
14 }
```

Output:

```
Enter a number: 73
The value entered is 73. Its double is 146.
```

#### Data races

The array pointed by *str* is accessed.

#### Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if the converted value would be out of the range of values representable by an *int*, it causes *undefined behavior*.

#### See also

<b>atol</b>	Convert string to long integer (function )
<b>atof</b>	Convert string to double (function )
<b>strtol</b>	Convert string to long integer (function )

## /cstdlib/atol

function

### atol

<cstdlib>

```
long int atol ( const char * str );
```

#### Convert string to long integer

Parses the C-string *str* interpreting its content as an integral number, which is returned as a value of type *long int*.

The function first discards as many whitespace characters (as in *isspace*) as necessary until the first non-whitespace character is found. Then, starting from this

character, takes an optional initial *plus* or *minus* sign followed by as many base-10 digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in *str* is not a valid integral number, or if no such sequence exists because either *str* is empty or it contains only whitespace characters, no conversion is performed and zero is returned.

## Parameters

**str** C-string containing the representation of an integral number.

## Return Value

On success, the function returns the converted integral number as a `long int` value.

If no valid conversion could be performed, a zero value is returned.

If the converted value would be out of the range of representable values by a `long int`, it causes *undefined behavior*. See [strtol](#) for a more robust cross-platform alternative when this is a possibility.

## Example

```
1 /* atol example */
2 #include <stdio.h>      /* printf, fgets */
3 #include <stdlib.h>      /* atol */
4
5 int main ()
6 {
7     long int li;
8     char buffer[256];
9     printf ("Enter a long number: ");
10    fgets (buffer, 256, stdin);
11    li = atol(buffer);
12    printf ("The value entered is %ld. Its double is %ld.\n",li,li*2);
13    return 0;
14 }
```

Output:

```
Enter a number: 567283
The value entered is 567283. Its double is 1134566.
```

## Data races

The array pointed by *str* is accessed.

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if the converted value would be out of the range of values representable by an `long int`, it causes *undefined behavior*.

## See also

<a href="#">atoi</a>	Convert string to integer (function)
<a href="#">atof</a>	Convert string to double (function)
<a href="#">strtol</a>	Convert string to long integer (function)

## /cstdlib/atoll

function

### atoll

<cstdlib>

```
long long int atoll ( const char * str );
```

#### Convert string to long long integer

Parses the C-string *str* interpreting its content as an integral number, which is returned as a value of type `long long int`.

This function operates like [atol](#) to interpret the string, but produces numbers of type `long long int` (see [atol](#) for details on the interpretation process).

## Parameters

**str** C-string containing the representation of an integral number.

## Return Value

On success, the function returns the converted integral number as a `long long int` value.

If no valid conversion could be performed, a zero value is returned.

If the converted value would be out of the range of representable values by a `long long int`, it causes *undefined behavior*. See [strtoll](#) for a more robust cross-platform alternative when this is a possibility.

## Example

```

1 /* atoll example */
2 #include <stdio.h>      /* printf, fgets */
3 #include <stdlib.h>      /* atoll */
4
5 int main ()
6 {
7     long long int lli;
8     char buffer[256];
9     printf ("Enter a long number: ");
10    fgets (buffer, 256, stdin);
11    lli = atoll(buffer);
12    printf ("The value entered is %lld. Its double is %lld.\n", lli, lli*2);
13    return 0;
14 }

```

Output:

```

Enter a number: 9275806
The value entered is 9275806. Its double is 18551612.

```

## Data races

The array pointed by *str* is accessed.

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if the converted value would be out of the range of values representable by a `long long int`, it causes *undefined behavior*.

## See also

<a href="#">atoi</a>	Convert string to integer (function )
<a href="#">atol</a>	Convert string to long integer (function )
<a href="#">strtoll</a>	Convert string to long long integer (function )

## /cstdlib/at\_quick\_exit

function **at\_quick\_exit** <cstdlib>

```

int at_quick_exit (void (*func)(void));
extern "C" int at_quick_exit (void (*func)(void)) noexcept;
extern "C++" int at_quick_exit (void (*func)(void)) noexcept;

```

### Set function to be executed on quick exit

The function pointed by *func* is automatically called (without arguments) when `quick_exit` is called.

If more than one `at_quick_exit` function has been specified by different calls to this function, they are all executed in reverse order.

If a function registered with `at_quick_exit` throws an exception for which it does not provide a handler while called by `quick_exit`, `terminate` is automatically called (C++).

Notice that the `at_quick_exit` stack of functions is separate from the `atexit` stack (and each is triggered by different circumstances), but the same function may be passed to both functions so that it is called in both cases.

Particular library implementations may impose a limit on the number of functions that can be registered with `at_quick_exit`, but this cannot be less than 32 functions.

## Parameters

function  
Function to be called. The function shall return no value and take no arguments.

## Return Value

A zero value is returned if the function was successfully registered.  
If it failed, a non-zero value is returned.

## Example

```

1 /* at_quick_exit example */
2 #include <stdio.h>      /* puts */
3 #include <stdlib.h>      /* at_quick_exit, quick_exit, EXIT_SUCCESS */
4
5 void fnQExit (void)
6 {
7     puts ("Quick exit function.");
8 }
9
10 int main ()
11 {
12     at_quick_exit (fnQExit);
13     puts ("Main function: Beginning");
14     quick_exit (EXIT_SUCCESS);

```

```

15 puts ("Main function: End"); // never executed
16 return 0;
17 }

```

Output:

```
Main function: Beginning
Quick exit function.
```

## Data races

Concurrently calling this function introduces no data races: Calls are properly synchronized at the process level, although notice that the relative order of calls from different threads is indeterminate.

Calls to `at_quick_exit` that do not complete before a call to `quick_exit` may not succeed (depends on particular library implementation).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

## See also

<code>atexit</code>	Set function to be executed on exit (function )
<code>quick_exit</code>	Terminate calling process quick (function )
<code>abort</code>	Abort current process (function )

## /cstdlib/bsearch

function  
**bsearch** <cstdlib>

```
void* bsearch (const void* key, const void* base,
               size_t num, size_t size,
               int (*compar)(const void*,const void*));
```

### Binary search in array

Searches the given `key` in the array pointed to by `base` (which is formed by `num` elements, each of `size` bytes), and returns a `void*` pointer to a matching element, if found.

To perform the search, the function performs a series of calls to `compar` with `key` as first argument and elements of the array pointed to by `base` as second argument.

Because this function may be optimized to use a non-linear search algorithm (presumably a binary search), the elements that compare less than `key` using `compar` should precede those that compare equal, and these should precede those that compare greater. This requirement is fulfilled by any array ordered with the same criteria used by `compar` (as if sorted with `qsort`).

## Parameters

`key` Pointer to the object that serves as key for the search, type-casted to a `void*`.

`base` Pointer to the first object of the array where the search is performed, type-casted to a `void*`.

`num` Number of elements in the array pointed to by `base`.  
`size_t` is an unsigned integral type.

`size` Size in bytes of each element in the array.  
`size_t` is an unsigned integral type.

`compar` Pointer to a function that compares two elements.  
This function is called repeatedly by `bsearch` to compare `key` against individual elements in `base`. It shall follow the following prototype:

```
int compar (const void* pkey, const void* pelem);
```

Taking two pointers as arguments: the first is always `key`, and the second points to an element of the array (both type-casted to `const void*`). The function shall return (in a stable and transitive manner):

return value	meaning
<0	The element pointed to by <code>pkey</code> goes before the element pointed to by <code>pelem</code>
0	The element pointed to by <code>pkey</code> is equivalent to the element pointed to by <code>pelem</code>
>0	The element pointed to by <code>pkey</code> goes after the element pointed to by <code>pelem</code>

For types that can be compared using regular relational operators, a general `compar` function may look like:

```

1 int compareMyType (const void * a, const void * b)
2 {
3     if ( *(MyType*)a < *(MyType*)b ) return -1;
4     if ( *(MyType*)a == *(MyType*)b ) return 0;
5     if ( *(MyType*)a > *(MyType*)b ) return 1;
6 }
```

## Return Value

A pointer to an entry in the array that matches the search *key*. If there are more than one matching elements (i.e., elements for which *compar* would return 0), this may point to any of them (not necessarily the first one).  
If *key* is not found, a null pointer is returned.

## Example

```
1 /* bsearch example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* qsort, bsearch, NULL */
4
5 int compareints (const void * a, const void * b)
6 {
7     return ( *(int*)a - *(int*)b );
8 }
9
10 int values[] = { 50, 20, 60, 40, 10, 30 };
11
12 int main ()
13 {
14     int * pItem;
15     int key = 40;
16     qsort (values, 6, sizeof (int), compareints);
17     pItem = (int*) bsearch (&key, values, 6, sizeof (int), compareints);
18     if (pItem!=NULL)
19         printf ("%d is in the array.\n",*pItem);
20     else
21         printf ("%d is not in the array.\n",key);
22     return 0;
23 }
```

In the example, *compareints* compares the values pointed to by the two parameters as *int* values and returns the result of subtracting their pointed values, which gives 0 as result if they are equal, a positive result if the value pointed to by *a* is greater than the one pointed to by *b* or a negative result if the value pointed to by *b* is greater.

In the main function the target array is sorted with *qsort* before calling *bsearch* to search for a value.

Output:

```
40 is in the array.
```

For C strings, *strcmp* can directly be used as the *compar* argument for *bsearch*:

```
1 /* bsearch example with strings */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* qsort, bsearch, NULL */
4 #include <string.h>      /* strcmp */
5
6 char strvalues[][20] = {"some", "example", "strings", "here"};
7
8 int main ()
9 {
10    char * pItem;
11    char key[20] = "example";
12
13    /* sort elements in array: */
14    qsort (strvalues, 4, 20, (int*)(const void*,const void*)) strcmp);
15
16    /* search for the key: */
17    pItem = (char*) bsearch (key, strvalues, 4, 20, (int*)(const void*,const void*)) strcmp);
18
19    if (pItem!=NULL)
20        printf ("%s is in the array.\n",pItem);
21    else
22        printf ("%s is not in the array.\n",key);
23    return 0;
24 }
```

Output:

```
example is in the array.
```

## Complexity

Unspecified, but binary searches are generally logarithmic in *num*, on average, calling *compar* approximately  $\log_2(\text{num})+2$  times.

## Data races

The function accesses the object pointed to by *key* and any number of the *num* elements in the array pointed to by *base*, but does not modify any of them.

## Exceptions (C++)

If *comp* does not throw exceptions, this function throws no exceptions (no-throw guarantee).

If *key* does not point to an object *size* bytes long, or if *base* does not point to an array of at least *num* properly arranged elements of *size* bytes each, or if *comp* does not behave as described above, it causes *undefined behavior*.

## See also

<a href="#">qsort</a>	Sort elements of array (function )
-----------------------	------------------------------------

function  
**calloc**

<cstdlib>

```
void* calloc (size_t num, size_t size);
```

**Allocate and zero-initialize array**

Allocates a block of memory for an array of *num* elements, each of them *size* bytes long, and initializes all its bits to zero.

The effective result is the allocation of a zero-initialized memory block of (*num*\**size*) bytes.

If *size* is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.

**Parameters**

**num** Number of elements to allocate.

**size** Size of each element.

**size\_t** is an unsigned integral type.

**Return Value**

On success, a pointer to the memory block allocated by the function.

The type of this pointer is always **void\***, which can be cast to the desired type of data pointer in order to be dereferenceable.

If the function failed to allocate the requested block of memory, a *null pointer* is returned.

**Example**

```
1 /* calloc example */
2 #include <stdio.h>      /* printf, scanf, NULL */
3 #include <stdlib.h>      /* calloc, exit, free */
4
5 int main ()
6 {
7     int i,n;
8     int * pData;
9     printf ("Amount of numbers to be entered: ");
10    scanf ("%d",&i);
11    pData = (int*) calloc (i,sizeof(int));
12    if (pData==NULL) exit (1);
13    for (n=0;n<i;n++)
14    {
15        printf ("Enter number #d: ",n+1);
16        scanf ("%d",&pData[n]);
17    }
18    printf ("You have entered: ");
19    for (n=0;n<i;n++) printf ("%d ",pData[n]);
20    free (pData);
21    return 0;
22 }
```

This program simply stores numbers and then prints them out. But the number of items it stores can be adapted each time the program is executed because it allocates the needed memory during runtime.

**Data races**

Only the storage referenced by the returned pointer is modified. No other storage locations are accessed by the call.

If the function reuses the same unit of storage released by a *deallocation function* (such as **free** or **realloc**), the functions are synchronized in such a way that the deallocation happens entirely before the next allocation.

**Exceptions (C++)**

**No-throw guarantee:** this function never throws exceptions.

**See also**

<b>free</b>	Deallocate memory block (function )
<b>malloc</b>	Allocate memory block (function )
<b>realloc</b>	Reallocate memory block (function )

## /cstdlib/div

function  
**div**

<cstdlib>

```
div_t div (int numer, int denom);
div_t div (    int numer,    int denom);
ldiv_t div (long int numer, long int denom);

div_t div (    int numer,    int denom);
ldiv_t div (    long int numer,    long int denom);
lldiv_t div (long long int numer, long long int denom);
```

**Integral division**

Returns the integral quotient and remainder of the division of *numer* by *denom* ( *numer/denom* ) as a structure of type `div_t`, `ldiv_t` or `lldiv_t`, which has two members: *quot* and *rem*.

## Parameters

`numer`  
    Numerator.  
`denom`  
    Denominator.

## Return Value

The result is returned by value in a structure defined in `<cstdlib>`, which has two members. For `div_t`, these are, in either order:

```
1 int quot; // quotient
2 int rem; // remainder
```

## Portability

In C, only the `int` version exists.

For the `long` `int` equivalent, see `ldiv`.

For the `long long` `int` equivalent, see `lldiv`.

## Example

```
1 /* div example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* div, div_t */
4
5 int main ()
6 {
7     div_t divresult;
8     divresult = div (38,5);
9     printf ("38 div 5 => %d, remainder %d.\n", divresult.quot, divresult.rem);
10    return 0;
11 }
```

Output:

```
38 div 5 => 7, remainder 3.
```

## Data races

Concurrently calling this function is safe, causing no data races.

## Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If either part of the result cannot be represented, it causes *undefined behavior*.

## See also

<a href="#">ldiv</a>	<a href="#">Integral division (function)</a>
----------------------	--

# /cstdlib/div\_t

type

## div\_t

`<cstdlib>`

### Structure returned by div

Structure to represent the result value of an integral division performed by function `div`.

It has two `int` members: *quot* and *rem*, defined in either order. A possible definition could be:

```
1 typedef struct {
2     int quot;
3     int rem;
4 } div_t;
```

## Members

`quot`

Represents the *quotient* of the integral division operation performed by `div`, which is the integer of lesser magnitude that is nearest to the algebraic quotient.

`rem`

Represents the *remainder* of the integral division operation performed by `div`, which is the integer resulting from subtracting *quot* to the numerator of the operation.

## See also

<b>ldiv_t</b>	Structure returned by ldiv (type )
<b>div</b>	Integral division (function )

## /cstdlib/exit

function  
**exit** <cstdlib>

```
void exit (int status);
[[noreturn]] void exit (int status);
```

### Terminate calling process

Terminates the process normally, performing the regular cleanup for terminating programs.

Normal program termination performs the following (in the same order):

- Objects associated with the current thread with thread storage duration are destroyed (C++11 only).
- Objects with static storage duration are destroyed (C++) and functions registered with [atexit](#) are called.
- All C streams (open with functions in [<cstdio>](#)) are closed (and flushed, if buffered), and all files created with [tmpfile](#) are removed.
- Control is returned to the host environment.

Note that objects with automatic storage are not destroyed by calling `exit` (C++).

If `status` is zero or [EXIT\\_SUCCESS](#), a *successful termination* status is returned to the host environment.

If `status` is [EXIT\\_FAILURE](#), an *unsuccessful termination* status is returned to the host environment.

Otherwise, the status returned depends on the system and library implementation.

For a similar function that does not perform the cleanup described above, see [quick\\_exit](#).

---

### Parameters

#### status

Status code.

If this is 0 or [EXIT\\_SUCCESS](#), it indicates success.

If it is [EXIT\\_FAILURE](#), it indicates failure.

---

### Return Value

none (the function never returns).

---

### Example

```
1 /* exit example */
2 #include <stdio.h>      /* printf, fopen */
3 #include <stdlib.h>      /* exit, EXIT_FAILURE */
4
5 int main ()
6 {
7     FILE * pFile;
8     pFile = fopen ("myfile.txt","r");
9     if (pFile==NULL)
10    {
11        printf ("Error opening file");
12        exit (EXIT_FAILURE);
13    }
14    else
15    {
16        /* file operations here */
17    }
18    return 0;
19 }
```

---

### Data races

Calling this function destroys all objects with static duration: A program with multiple threads running shall not call `exit` (see [quick\\_exit](#) for a similar function that does not affect static objects).

---

### Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If the program termination process described above throws an exception, [terminate](#) is automatically called.

---

### See also

<b>abort</b>	Abort current process (function )
<b>atexit</b>	Set function to be executed on exit (function )

## /cstdlib/\_Exit

function  
**\_Exit** <cstdlib>

```
void _Exit (int status);
[[noreturn]] void _Exit (int status) noexcept;
```

#### Terminate calling process

Terminates the process normally by returning control to the host environment, but without performing any of the regular cleanup tasks for terminating processes (as function `exit` does).

No object destructors, nor functions registered by `atexit` or `at_quick_exit` are called.

Whether C streams are closed and/or flushed, and files open with `tmpfile` are removed depends on the particular system or library implementation.

If `status` is zero or `EXIT_SUCCESS`, a *successful termination* status is returned to the host environment.

If `status` is `EXIT_FAILURE`, an *unsuccessful termination* status is returned to the host environment.

Otherwise, the status returned depends on the system and library implementation.

#### Parameters

##### status

Status code.  
If this is 0 or `EXIT_SUCCESS`, it indicates success.  
If it is `EXIT_FAILURE`, it indicates failure.

#### Return Value

none (the function never returns).

#### Example

```
1 /* _Exit example */
2 #include <stdio.h>      /* printf, fopen */
3 #include <stdlib.h>      /* _Exit, EXIT_FAILURE */
4
5 int main ()
6 {
7     FILE * pFile;
8     pFile = fopen ("myfile.txt", "r");
9     if (pFile==NULL)
10    {
11        printf ("Error opening file");
12        _Exit (EXIT_FAILURE);
13    }
14    else
15    {
16        /* file operations here */
17    }
18    return 0;
19 }
```

#### Data races

Concurrently calling this function multiple times has no effect.

#### Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

#### See also

<a href="#">exit</a>	Terminate calling process ( <a href="#">function</a> )
<a href="#">quick_exit</a>	Terminate calling process quick ( <a href="#">function</a> )
<a href="#">abort</a>	Abort current process ( <a href="#">function</a> )

## /cstdlib/EXIT\_FAILURE

macro

### EXIT\_FAILURE

<cstdlib>

#### Failure termination code

This macro expands to a system-dependent integral expression that, when used as the argument for function `exit`, signifies that the application failed.

The opposite meaning can be specified with `EXIT_SUCCESS`.

## /cstdlib/EXIT\_SUCCESS

macro

### EXIT\_SUCCESS

<cstdlib>

#### Success termination code

This macro expands to a system-dependent integral expression that, when used as the argument for function `exit`, signifies that the application was successful.

The opposite meaning can be specified with `EXIT_FAILURE`.

# /cstdlib/free

function  
**free** <cstdlib>

```
void free (void* ptr);
```

## Deallocate memory block

A block of memory previously allocated by a call to `malloc`, `calloc` or `realloc` is deallocated, making it available again for further allocations.

If `ptr` does not point to a block of memory allocated with the above functions, it causes *undefined behavior*.

If `ptr` is a *null pointer*, the function does nothing.

Notice that this function does not change the value of `ptr` itself, hence it still points to the same (now invalid) location.

## Parameters

`ptr` Pointer to a memory block previously allocated with `malloc`, `calloc` or `realloc`.

## Return Value

none

## Example

```
1 /* free example */
2 #include <stdlib.h>      /* malloc, calloc, realloc, free */
3
4 int main ()
5 {
6     int * buffer1, * buffer2, * buffer3;
7     buffer1 = (int*) malloc (100*sizeof(int));
8     buffer2 = (int*) calloc (100,sizeof(int));
9     buffer3 = (int*) realloc (buffer2,500*sizeof(int));
10    free (buffer1);
11    free (buffer3);
12    return 0;
13 }
```

This program has no output. It just demonstrates some ways to allocate and free dynamic memory using the C stdlib functions.

## Data races

Only the storage referenced by `ptr` is modified. No other storage locations are accessed by the call.

If the function releases a unit of storage that is reused by a call to *allocation functions* (such as `calloc` or `malloc`), the functions are synchronized in such a way that the deallocation happens entirely before the next allocation.

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If `ptr` does not point to a memory block previously allocated with `malloc`, `calloc` or `realloc`, and is not a *null pointer*, it causes *undefined behavior*.

## See also

<a href="#">malloc</a>	Allocate memory block (function )
<a href="#">calloc</a>	Allocate and zero-initialize array (function )
<a href="#">realloc</a>	Reallocate memory block (function )

# /cstdlib/getenv

function  
**getenv** <cstdlib>

```
char* getenv (const char* name);
```

## Get environment string

Retrieves a C-string containing the value of the environment variable whose `name` is specified as argument. If the requested variable is not part of the environment list, the function returns a null pointer.

The pointer returned points to an internal memory block, whose content or validity may be altered by further calls to `getenv` (but not by other library functions).

The string pointed by the pointer returned by this function shall not be modified by the program. Some systems and library implementations may allow to change environmental variables with specific functions (`putenv`, `setenv`...), but such functionality is non-portable.

## Parameters

`name`

C-string containing the name of the requested variable.  
Depending on the platform, this may either be case sensitive or not.

## Return Value

A C-string with the value of the requested environment variable, or a *null pointer* if such environment variable does not exist.

## Example

```
1 /* getenv example: getting path */
2 #include <stdio.h>           /* printf */
3 #include <stdlib.h>          /* getenv */
4
5 int main ()
6 {
7     char* pPath;
8     pPath = getenv ("PATH");
9     if (pPath!=NULL)
10     printf ("The current path is: %s",pPath);
11     return 0;
12 }
```

The example above prints the PATH environment variable, if such a variable exists in the hosting environment.

## Data races

Concurrently calling this function is safe, provided that the environment remains unchanged.

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

## See also

<a href="#">system</a>	Execute system command (function )
------------------------	------------------------------------

# /cstdlib/itoa

function

## itoa

<stdlib.h>

`char * itoa ( int value, char * str, int base );`

**Convert integer to string (non-standard function)**

Converts an integer *value* to a null-terminated string using the specified *base* and stores the result in the array given by *str* parameter.

If *base* is 10 and *value* is negative, the resulting string is preceded with a minus sign (-). With any other *base*, *value* is always considered unsigned.

*str* should be an array long enough to contain any possible value: (`(sizeof(int)*8+1)` for radix=2, i.e. 17 bytes in 16-bits platforms and 33 in 32-bits platforms.

## Parameters

*value*

Value to be converted to a string.

*str*

Array in memory where to store the resulting null-terminated string.

*base*

Numerical base used to represent the *value* as a string, between 2 and 36, where 10 means decimal base, 16 hexadecimal, 8 octal, and 2 binary.

## Return Value

A pointer to the resulting null-terminated string, same as parameter *str*.

## Portability

This function is **not** defined in ANSI-C and is **not** part of C++, but is supported by some compilers.

A standard-compliant alternative for some cases may be [sprintf](#):

- `sprintf(str,"%d",value)` converts to decimal base.
- `sprintf(str,"%x",value)` converts to hexadecimal base.
- `sprintf(str,"%o",value)` converts to octal base.

## Example

```
1 /* itoa example */
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main ()
6 {
7     int i;
8     char buffer [33];
9     printf ("Enter a number: ");
10    scanf ("%d",&i);
11    itoa (i,buffer,10);
```

```

12 printf ("decimal: %s\n",buffer);
13 itoa (i,buffer,16);
14 printf ("hexadecimal: %s\n",buffer);
15 itoa (i,buffer,2);
16 printf ("binary: %s\n",buffer);
17 return 0;
18 }

```

Output:

```

Enter a number: 1750
decimal: 1750
hexadecimal: 6d6
binary: 11011010110

```

## See also

<a href="#">sprintf</a>	Write formatted data to string (function )
<a href="#">atoi</a>	Convert string to integer (function )
<a href="#">atol</a>	Convert string to long integer (function )

## /cstdlib/labs

function

### labs

<cstdlib>

```
long int labs (long int n);
```

#### Absolute value

Returns the absolute value of parameter *n* ( */n/* ).

This is the *long int* version of [abs](#).

#### Parameters

*n*

Integral value.

#### Return Value

The absolute value of *n*.

#### Example

```

1 /* labs example */
2 #include <stdio.h>          /* printf */
3 #include <stdlib.h>           /* labs */
4
5 int main ()
6 {
7     long int n,m;
8     n=labs(65537L);
9     m=labs(-100000L);
10    printf ("n=%ld\n",n);
11    printf ("m=%ld\n",m);
12    return 0;
13 }

```

Output:

```

n=65537
m=100000

```

#### Data races

Concurrently calling this function is safe, causing no data races.

#### Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If the result cannot be represented as a *long int* (such as *labs(LONG\_MIN)* in an implementation with two's complement signed values), it causes *undefined behavior*.

## See also

<a href="#">abs</a>	Absolute value (function )
<a href="#">fabs</a>	Compute absolute value (function )
<a href="#">ldiv</a>	Integral division (function )

## /cstdlib/ldiv

## function **ldiv**

<cstdlib>

```
ldiv_t ldiv (long int numer, long int denom);
```

### Integral division

Returns the integral quotient and remainder of the division of *numer* by *denom* (*numer/denom*) as a structure of type `ldiv_t`, which has two members: *quot* and *rem*.

### Parameters

`numer`  
Numerator.

`denom`  
Denominator.

### Return Value

The result is returned by value in a `ldiv_t` structure, which has two members (in either order):

```
1 long int quot; // quotient
2 long int rem; // remainder
```

### Example

```
1 /* ldiv example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* ldiv, ldiv_t */
4
5 int main ()
6 {
7     ldiv_t ldivresult;
8     ldivresult = ldiv (1000000L,132L);
9     printf ("1000000 div 132 => %ld, remainder %ld.\n", ldivresult.quot, ldivresult.rem);
10    return 0;
11 }
```

Output:

```
1000000 div 132 => 7575, remainder 100.
```

### Data races

Concurrently calling this function is safe, causing no data races.

### Exceptions (C++)

No-throw guarantee: this function throws no exceptions.

If either part of the result cannot be represented, it causes *undefined behavior*.

### See also

<code>div</code>	Integral division (function )
<code>lldiv</code>	Integral division (function )
<code>ldiv_t</code>	Structure returned by ldiv (type )

## /cstdlib/ldiv\_t

type

## **ldiv\_t**

<cstdlib>

### Structure returned by ldiv

Structure to represent the result value of an integral division performed by function `ldiv` (and, in C++, possibly also by `div`).

It has two data members of type `long int`: `quot` and `rem`, defined in either order. A possible definition could be:

```
1 typedef struct {
2     long int quot;
3     long int rem;
4 } ldiv_t;
```

### Members

`quot`

Represents the *quotient* of the integral division operation performed by `ldiv`, which is the integer of lesser magnitude that is nearest to the algebraic quotient.

`rem`

Represents the *remainder* of the integral division operation performed by `ldiv`, which is the integer resulting from subtracting `quot` to the numerator of the operation.

## See also

<a href="#">div_t</a>	Structure returned by div (type )
<a href="#">div</a>	Integral division (function )
<a href="#">ldiv</a>	Integral division (function )

## /cstdlib/llabs

function  
**llabs** <cstdlib>

```
long long int llabs (long long int n);
```

### Absolute value

Returns the absolute value of parameter *n* ( */n/* ).

This is the long long int version of [abs](#).

## Parameters

*n*  
Integral value.

## Return Value

The absolute value of *n*.

## Example

```
1 /* llabs example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* llabs */
4
5 int main ()
6 {
7     long long int n,m;
8     n=llabs(31558149LL);
9     m=llabs(-10000000LL);
10    printf ("n=%lld\n",n);
11    printf ("m=%lld\n",m);
12    return 0;
13 }
```

Output:

```
n=31558149
m=100000000
```

## Data races

Concurrently calling this function is safe, causing no data races.

## Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If the result cannot be represented as a long long int (such as llabs(LLONG\_MIN) in an implementation with two's complement signed values), it causes *undefined behavior*.

## See also

<a href="#">abs</a>	Absolute value (function )
<a href="#">labs</a>	Absolute value (function )
<a href="#">fabs</a>	Compute absolute value (function )
<a href="#">ldiv</a>	Integral division (function )

## /cstdlib/lldiv

function  
**lldiv** <cstdlib>

```
lldiv_t lldiv (long long int numer, long long int denom);
```

### Integral division

Returns the integral quotient and remainder of the division of *numer* by *denom* ( *numer/denom* ) as a structure of type [lldiv\\_t](#), which has two members: *quot* and *rem*.

## Parameters

*numer*  
Numerator.

**denom**  
Denominator.

## Return Value

The result is returned by value in a `lldiv_t` structure, which has two members (in either order):

```
1 long long int quot; // quotient
2 long long int rem; // remainder
```

## Example

```
1 /* lldiv example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* lldiv, lldiv_t */
4
5 int main ()
6 {
7     lldiv_t res;
8     res = lldiv (31558149LL,3600LL);
9     printf ("Earth orbit: %lld hours and %lld seconds.\n", res.quot, res.rem);
10    return 0;
11 }
```

Output:

```
Earth orbit: 8766 hours and 549 seconds.
```

## Data races

Concurrently calling this function is safe, causing no data races.

## Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If either part of the result cannot be represented, it causes *undefined behavior*.

## See also

<code>div</code>	Integral division ( <a href="#">function</a> )
<code>ldiv</code>	Integral division ( <a href="#">function</a> )
<code>lldiv_t</code>	Structure returned by <code>lldiv</code> ( <a href="#">type</a> )

## /cstdlib/lldiv\_t

type

## lldiv\_t

<cstdlib>

### Structure returned by lldiv

Structure to represent the result value of an integral division performed by function `lldiv` (and, in C++, possibly also by `div`).

It has two data members of type `long long int`: `quot` and `rem`, defined in either order. A possible definition could be:

```
1 typedef struct {
2     long long quot;
3     long long rem;
4 } lldiv_t;
```

## Members

`quot`

Represents the *quotient* of the integral division operation performed by `lldiv`, which is the integer of lesser magnitude that is nearest to the algebraic quotient.

`rem`

Represents the *remainder* of the integral division operation performed by `lldiv`, which is the integer resulting from subtracting `quot` to the numerator of the operation.

## See also

<code>div_t</code>	Structure returned by <code>div</code> ( <a href="#">type</a> )
<code>div</code>	Integral division ( <a href="#">function</a> )
<code>lldiv</code>	Integral division ( <a href="#">function</a> )

## /cstdlib/malloc

function

<cstdlib>

## malloc

```
void* malloc (size_t size);
```

### Allocate memory block

Allocates a block of `size` bytes of memory, returning a pointer to the beginning of the block.

The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.

If `size` is zero, the return value depends on the particular library implementation (it may or may not be a *null pointer*), but the returned pointer shall not be dereferenced.

### Parameters

`size`

Size of the memory block, in bytes.  
`size_t` is an unsigned integral type.

### Return Value

On success, a pointer to the memory block allocated by the function.

The type of this pointer is always `void*`, which can be cast to the desired type of data pointer in order to be dereferenceable.

If the function failed to allocate the requested block of memory, a *null pointer* is returned.

### Example

```
1 /* malloc example: random string generator*/
2 #include <stdio.h>          /* printf, scanf, NULL */
3 #include <stdlib.h>          /* malloc, free, rand */
4
5 int main ()
6 {
7     int i,n;
8     char * buffer;
9
10    printf ("How long do you want the string? ");
11    scanf ("%d", &i);
12
13    buffer = (char*) malloc (i+1);
14    if (buffer==NULL) exit (1);
15
16    for (n=0; n<i; n++)
17        buffer[n]=rand()%26+'a';
18    buffer[i]='\0';
19
20    printf ("Random string: %s\n",buffer);
21    free (buffer);
22
23    return 0;
24 }
```

This program generates a string of the length specified by the user and fills it with alphabetic characters. The possible length of this string is only limited by the amount of memory available to `malloc`.

### Data races

Only the storage referenced by the returned pointer is modified. No other storage locations are accessed by the call.

If the function reuses the same unit of storage released by a *deallocation function* (such as `free` or `realloc`), the functions are synchronized in such a way that the deallocation happens entirely before the next allocation.

### Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

### See also

<code>free</code>	Deallocate memory block (function )
<code>calloc</code>	Allocate and zero-initialize array (function )
<code>realloc</code>	Reallocate memory block (function )

## /cstdlib/MB\_CUR\_MAX

macro

### MB\_CUR\_MAX

`<cstdlib>`

#### Maximum size of multibyte characters

This macro expands to a positive integer expression of type `size_t`, the value of which is the maximum number of bytes in a multibyte character with the current locale (category `LC_CTYPE`).

Its value is never greater than `MB_LEN_MAX` (see `<climits>`).

## /cstdlib/mblen

```
int mblen (const char* pmb, size_t max);
```

#### Get length of multibyte character

Returns the size of the multibyte character pointed by *pmb*, examining at most *max* bytes.

*mblen* has its own internal *shift state*, which is altered as necessary only by calls to this function. A call to the function with a null pointer as *pmb* resets the state (and returns whether multibyte characters are state-dependent).

The behavior of this function depends on the `LC_CTYPE` category of the selected C locale.

#### Parameters

*pmb*

Pointer to the first byte of a multibyte character.

Alternatively, the function may be called with a *null pointer*, in which case the function resets its internal shift state to the initial value and returns whether multibyte characters use a state-dependent encoding.

*max*

Maximum number of bytes of *pmb* to consider for the multibyte character.

No more than `MB_CUR_MAX` characters are examined in any case.

`size_t` is an unsigned integral type.

#### Return Value

If the argument passed as *pmb* is not a *null pointer*, the size in bytes of the character pointed by *pmb* is returned when it forms a valid multibyte character and is not the *terminating null character*. If it is the *terminating null character*, the function returns zero, and in the case they do not form a valid multibyte character, -1 is returned.

If the argument passed as *pmb* is a *null pointer*, the function returns a nonzero value if multibyte character encodings are state-dependent, and zero otherwise.

#### Example

```
1 /* mblen example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* mblen, mbtowc, wchar_t(C) */
4
5 void printbuffer (const char* pt, size_t max)
6 {
7     int length;
8     wchar_t dest;
9
10    mblen (NULL, 0);      /* reset mblen */
11    mbtowc (NULL, NULL, 0); /* reset mbtowc */
12
13    while (max>0) {
14        length = mblen (pt, max);
15        if (length<1) break;
16        mbtowc (&dest, pt, length);
17        printf ("[%lc]", dest);
18        pt+=length; max-=length;
19    }
20}
21
22 int main()
23 {
24     const char str [] = "test string";
25
26     printbuffer (str,sizeof(str));
27
28     return 0;
29 }
```

`printbuffer` prints a multibyte string character by character.

The example uses a trivial string using the "c" locale, but locales that interpret multibyte string are supported by the function.

Output:

```
[t][e][s][t][ ][s][t][r][i][n][g]
```

#### Data races

The function accesses the array pointed by *pmb*.

The function also accesses and modifies an internal state object, which may cause data races on concurrent calls to this function (see `mbrlen` for an alternative that may use an external state object).

Concurrently changing locale settings may also introduce data races.

#### Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If *pmb* is neither a *null pointer* nor a pointer to an array long enough (as described above), it causes *undefined behavior*.

#### See also

<a href="#">mbtowc</a>	Convert multibyte sequence to wide character ( <a href="#">function</a> )
<a href="#">wctomb</a>	Convert wide character to multibyte sequence ( <a href="#">function</a> )
<a href="#">mbstowcs</a>	Convert multibyte string to wide-character string ( <a href="#">function</a> )

## /cstdlib/mbstowcs

function

### mbstowcs

&lt;cstdlib&gt;

```
size_t mbstowcs (wchar_t* dest, const char* src, size_t max);
```

#### Convert multibyte string to wide-character string

Translates the multibyte sequence pointed by *src* to the equivalent sequence of wide-characters (which is stored in the array pointed by *dest*), up until either *max* wide characters have been translated or until a null character is encountered in the multibyte sequence *src* (which is also translated and stored, but not counted in the length returned by the function).

If *max* characters are successfully translated, the resulting string stored in *dest* is not null-terminated.

The behavior of this function depends on the `LC_CTYPE` category of the selected C locale.

#### Parameters

<code>dest</code>	Pointer to an array of <code>wchar_t</code> elements long enough to contain the resulting sequence (at most, <i>max</i> wide characters).
<code>src</code>	C-string with the multibyte characters to be interpreted. The multibyte sequence shall begin in the initial shift state.
<code>max</code>	Maximum number of <code>wchar_t</code> characters to write to <i>dest</i> . <code>size_t</code> is an unsigned integral type.

#### Return Value

The number of wide characters written to *dest*, not including the eventual *terminating null character*.

If an invalid multibyte character is encountered, a value of (`size_t`) -1 is returned.

Notice that `size_t` is an unsigned integral type, and thus none of the values possibly returned is less than zero.

#### Data races

The function accesses the array pointed by *src*, and modifies the array pointed by *dest*.

The function may also access and modify an internal state object, which may cause data races on concurrent calls to this function if the implementation uses a static object (see `mbsrtowcs` for an alternative that can use an external state object). Concurrently changing locale settings may also introduce data races.

#### Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If *dest* does not point to an array long enough to contain the translated sequence, or if *src* is either not null-terminated or does not contain enough bytes to generate *max* wide characters (or if it does not begin in the initial shift state), it causes *undefined behavior*.

#### See also

<code>mblen</code>	Get length of multibyte character (function )
<code>mbtowc</code>	Convert multibyte sequence to wide character (function )
<code>wcstombs</code>	Convert wide-character string to multibyte string (function )

## /cstdlib/mbtowc

function

### mbtowc

&lt;cstdlib&gt;

```
int mbtowc (wchar_t* pwc, const char* pmb, size_t max);
```

#### Convert multibyte sequence to wide character

The multibyte character pointed by *pmb* is converted to a value of type `wchar_t` and stored at the location pointed by *pwc*. The function returns the length in bytes of the multibyte character.

`mbtowc` has its own internal *shift state*, which is altered as necessary only by calls to this function. A call to the function with a *null pointer* as *pmb* resets the state (and returns whether multibyte characters are state-dependent).

The behavior of this function depends on the `LC_CTYPE` category of the selected C locale.

#### Parameters

<code>pwc</code>	Pointer to an object of type <code>wchar_t</code> . Alternatively, this argument can be a <i>null pointer</i> , in which case the function does not store the <code>wchar_t</code> translation, but still returns the length in bytes of the multibyte character.
<code>pmb</code>	Pointer to the first byte of a multibyte character. Alternatively, this argument can be a <i>null pointer</i> , in which case the function resets its internal shift state to the initial value and returns whether multibyte characters have a state-dependent encoding.

max

Maximum number of bytes of *pmb* to consider for the multibyte character.  
No more than `MB_CUR_MAX` characters are examined in any case.  
`size_t` is an unsigned integral type.

## Return Value

If the argument passed as *pmb* is not a *null pointer*, the size in bytes of the multibyte character pointed by *pmb* is returned when it forms a valid multibyte character and is not the *terminating null character*. If it is the *terminating null character*, the function returns zero, and in the case they do not form a valid multibyte character, -1 is returned.

If the argument passed as *pmb* is a *null pointer*, the function returns a nonzero value if multibyte character encodings are state-dependent, and zero otherwise.

## Example

```
1 /* mbtowc example */
2 #include <stdio.h>           /* printf */
3 #include <stdlib.h>          /* mbtowc, wchar_t(C) */
4
5 void printbuffer (const char* pt, size_t max)
6 {
7     int length;
8     wchar_t dest;
9
10    mbtowc (NULL, NULL, 0); /* reset mbtowc */
11
12    while (max>0) {
13        length = mbtowc(&dest,pt,max);
14        if (length<1) break;
15        printf ("[%lc]",dest);
16        pt+=length; max-=length;
17    }
18}
19
20 int main()
21 {
22     const char str [] = "mbtowc example";
23
24     printbuffer (str,sizeof(str));
25
26     return 0;
27 }
```

`printbuffer` prints a multibyte string character by character.

The example uses a trivial string using the "c" locale, but locales that interpret multibyte strings are supported by the function.

Output:

```
[m][b][t][o][w][c][ ][e][x][a][m][p][l][e]
```

## Data races

The function accesses the array pointed by *pmb*, and modifies the object pointed by *pwc* (if not null).  
The function also accesses and modifies an internal state object, which may cause data races on concurrent calls to this function (see `mbrtowc` for an alternative that may use an external state object).  
Concurrently changing locale settings may also introduce data races.

## Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If *pwc* is neither a *null pointer* nor points to a valid object, or if *pmb* is neither a *null pointer* nor a pointer to an array long enough (as described above), it causes *undefined behavior*.

## See also

<a href="#">mblen</a>	Get length of multibyte character ( <a href="#">function</a> )
<a href="#">wctomb</a>	Convert wide character to multibyte sequence ( <a href="#">function</a> )
<a href="#">mbstowcs</a>	Convert multibyte string to wide-character string ( <a href="#">function</a> )
<a href="#">wcstombs</a>	Convert wide-character string to multibyte string ( <a href="#">function</a> )

## /cstdlib/NULL

macro

**NULL**

`<cstddef> <cstdlib> <cstring> <cwchar> <ctime> <clocale> <cstdio>`

### Null pointer

This macro expands to a *null pointer constant*.

A *null-pointer constant* is an integral constant expression that evaluates to zero (like 0 or 0L), or the cast of such value to type `void*` (like `(void*)0`).

A *null-pointer constant* is an integral constant expression that evaluates to zero (such as 0 or 0L).

A *null-pointer constant* is either an integral constant expression that evaluates to zero (such as 0 or 0L), or a value of type `nullptr_t` (such as `nullptr`).

A null pointer constant can be converted to any *pointer type* (or *pointer-to-member type*), which acquires a *null pointer value*. This is a special value that

indicates that the pointer is not pointing to any object.

## /cstdlib/qsort

function  
**qsort** <cstdlib>

```
void qsort (void* base, size_t num, size_t size,
           int (*compar)(const void*,const void*));
```

### Sort elements of array

Sorts the *num* elements of the array pointed to by *base*, each element *size* bytes long, using the *compar* function to determine the order.

The sorting algorithm used by this function compares pairs of elements by calling the specified *compar* function with pointers to them as argument.

The function does not return any value, but modifies the content of the array pointed to by *base* reordering its elements as defined by *compar*.

The order of equivalent elements is undefined.

### Parameters

**base** Pointer to the first object of the array to be sorted, converted to a *void\**.

**num** Number of elements in the array pointed to by *base*.  
*size\_t* is an unsigned integral type.

**size** Size in bytes of each element in the array.  
*size\_t* is an unsigned integral type.

**compar** Pointer to a function that compares two elements.  
This function is called repeatedly by *qsort* to compare two elements. It shall follow the following prototype:

```
int compar (const void* p1, const void* p2);
```

Taking two pointers as arguments (both converted to *const void\**). The function defines the order of the elements by returning (in a stable and transitive manner):

return value	meaning
<0	The element pointed to by <i>p1</i> goes before the element pointed to by <i>p2</i>
0	The element pointed to by <i>p1</i> is equivalent to the element pointed to by <i>p2</i>
>0	The element pointed to by <i>p1</i> goes after the element pointed to by <i>p2</i>

For types that can be compared using regular relational operators, a general *compar* function may look like:

```
1 int compareMyType (const void * a, const void * b)
2 {
3     if ( *(MyType*)a < *(MyType*)b ) return -1;
4     if ( *(MyType*)a == *(MyType*)b ) return 0;
5     if ( *(MyType*)a > *(MyType*)b ) return 1;
6 }
```

### Return Value

none

### Example

```
1 /* qsort example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>       /* qsort */
4
5 int values[] = { 40, 10, 100, 90, 20, 25 };
6
7 int compare (const void * a, const void * b)
8 {
9     return ( *(int*)a - *(int*)b );
10 }
11
12 int main ()
13 {
14     int n;
15     qsort (values, 6, sizeof(int), compare);
16     for (n=0; n<6; n++)
17         printf ("%d ",values[n]);
18     return 0;
19 }
```

Output:

```
10 20 25 40 90 100
```

### Complexity

Unspecified, but quicksorts are generally linearithmic in *num*, on average, calling *compar* approximately  $\text{num} \cdot \log_2(\text{num})$  times.

## Data races

The function accesses and/or modifies the *num* elements in the array pointed to by *base*.

## Exceptions (C++)

If *comp* does not throw exceptions, this function throws no exceptions (no-throw guarantee).

If *base* does not point to an array of at least *num\*size* bytes, or if *comp* does not behave as described above, it causes *undefined behavior*.

## See also

<a href="#">bsearch</a>	Binary search in array (function )
-------------------------	------------------------------------

# /cstdlib/quick\_exit

function  
**quick\_exit** <cstdlib>

```
_Noreturn void quick_exit (int status);
[[noreturn]] void quick_exit (int status) noexcept;
```

### Terminate calling process quick

Terminates the process normally by returning control to the host environment after calling all functions registered using [at\\_quick\\_exit](#).

No additional cleanup tasks are performed: No object destructors are called. Although whether C streams are closed and/or flushed, and files open with [tmpfile](#) are removed depends on the particular system or library implementation.

If *status* is zero or [EXIT\\_SUCCESS](#), a *successful termination* status is returned to the host environment.

If *status* is [EXIT\\_FAILURE](#), an *unsuccessful termination* status is returned to the host environment.

Otherwise, the status returned depends on the system and library implementation.

If a program calls both [exit](#) and [quick\\_exit](#), or [quick\\_exit](#) more than once, it causes *undefined behavior*.

## Parameters

**status**  
Status code.  
If this is 0 or [EXIT\\_SUCCESS](#), it indicates success.  
If it is [EXIT\\_FAILURE](#), it indicates failure.

## Return Value

none (the function never returns).

## Example

```
1 /* quick_exit example */
2 #include <stdio.h>           /* puts */
3 #include <stdlib.h>          /* at_quick_exit, quick_exit, EXIT_SUCCESS */
4
5 void fnQExit (void)
6 {
7     puts ("Quick exit function.");
8 }
9
10 int main ()
11 {
12     at_quick_exit (fnQExit);
13     puts ("Main function: Beginning");
14     quick_exit (EXIT_SUCCESS);
15     puts ("Main function: End"); // never executed
16     return 0;
17 }
```

Output:

```
Main function: Beginning
Quick exit function.
```

## Data races

Concurrently calling this function multiple times has no effect.

Calls to [at\\_quick\\_exit](#) that do not complete before the call to this function may not succeed (depends on particular library implementation).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If any of the functions registered with [at\\_quick\\_exit](#) throws an exception, [terminate](#) is automatically called.

## See also

<a href="#">at_quick_exit</a>	Set function to be executed on quick exit (function )
<a href="#">exit</a>	Terminate calling process (function )
<a href="#">abort</a>	Abort current process (function )

# /cstdlib/rand

function  
**rand** <cstdlib>

```
int rand (void);
```

## Generate random number

Returns a pseudo-random integral number in the range between 0 and [RAND\\_MAX](#).

This number is generated by an algorithm that returns a sequence of apparently non-related numbers each time it is called. This algorithm uses a seed to generate the series, which should be initialized to some distinctive value using function [srand](#).

[RAND\\_MAX](#) is a constant defined in [<cstdlib>](#).

A typical way to generate trivial pseudo-random numbers in a determined range using [rand](#) is to use the modulo of the returned value by the range span and add the initial value of the range:

```
1 v1 = rand() % 100;           // v1 in the range 0 to 99
2 v2 = rand() % 100 + 1;        // v2 in the range 1 to 100
3 v3 = rand() % 30 + 1985;     // v3 in the range 1985-2014
```

Notice though that this modulo operation does not generate uniformly distributed random numbers in the span (since in most cases this operation makes lower numbers slightly more likely).

C++ supports a wide range of powerful tools to generate random and pseudo-random numbers (see [<random>](#) for more info).

## Parameters

(none)

## Return Value

An integer value between 0 and [RAND\\_MAX](#).

## Example

```
1 /* rand example: guess the number */
2 #include <stdio.h>          /* printf, scanf, puts, NULL */
3 #include <stdlib.h>          /* srand, rand */
4 #include <time.h>            /* time */
5
6 int main ()
7 {
8     int iSecret, iGuess;
9
10    /* initialize random seed: */
11    srand (time(NULL));
12
13    /* generate secret number between 1 and 10: */
14    iSecret = rand() % 10 + 1;
15
16    do {
17        printf ("Guess the number (1 to 10): ");
18        scanf ("%d",&iGuess);
19        if (iSecret<iGuess) puts ("The secret number is lower");
20        else if (iSecret>iGuess) puts ("The secret number is higher");
21    } while (iSecret!=iGuess);
22
23    puts ("Congratulations!");
24    return 0;
25 }
```

In this example, the random seed is initialized to a value representing the current time (calling [time](#)) to generate a different value every time the program is run.

Possible output:

```
Guess the number (1 to 10): 5
The secret number is higher
Guess the number (1 to 10): 8
The secret number is lower
Guess the number (1 to 10): 7
Congratulations!
```

## Compatibility

In C, the generation algorithm used by [rand](#) is guaranteed to only be advanced by calls to this function. In C++, this constraint is relaxed, and a library implementation is allowed to advance the generator on other circumstances (such as calls to elements of [<random>](#)).

## Data races

The function accesses and modifies internal state objects, which may cause data races with concurrent calls to [rand](#) or [srand](#).

Some libraries provide an alternative function that explicitly avoids this kind of data race: [rand\\_r](#) (non-portable).

C++ library implementations are allowed to guarantee no *data races* for calling this function.

## Exceptions (C++)

No-throw guarantee: this function never throws exceptions.

### See also

<a href="#">rand</a>	Initialize random number generator (function)
----------------------	---

## /cstdlib/RAND\_MAX

macro

### RAND\_MAX

<cstdlib>

#### Maximum value returned by rand

This macro expands to an integral constant expression whose value is the maximum value returned by the `rand` function.

This value is library-dependent, but is guaranteed to be at least 32767 on any standard library implementation.

## /cstdlib/realloc

function

### realloc

<cstdlib>

`void* realloc (void* ptr, size_t size);`

#### Reallocate memory block

Changes the size of the memory block pointed to by `ptr`.

The function may move the memory block to a new location (whose address is returned by the function).

The content of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new `size` is larger, the value of the newly allocated portion is indeterminate.

In case that `ptr` is a null pointer, the function behaves like `malloc`, assigning a new block of `size` bytes and returning a pointer to its beginning.

Otherwise, if `size` is zero, the memory previously allocated at `ptr` is deallocated as if a call to `free` was made, and a *null pointer* is returned.

If `size` is zero, the return value depends on the particular library implementation: it may either be a *null pointer* or some other location that shall not be dereferenced.

If the function fails to allocate the requested block of memory, a *null pointer* is returned, and the memory block pointed to by argument `ptr` is not deallocated (it is still valid, and with its contents unchanged).

### Parameters

`ptr`

Pointer to a memory block previously allocated with `malloc`, `calloc` or `realloc`. Alternatively, this can be a *null pointer*, in which case a new block is allocated (as if `malloc` was called).

`size`

New size for the memory block, in bytes. `size_t` is an unsigned integral type.

### Return Value

A pointer to the reallocated memory block, which may be either the same as `ptr` or a new location.

The type of this pointer is `void*`, which can be cast to the desired type of data pointer in order to be dereferenceable.

A *null-pointer* indicates either that `size` was zero (and thus `ptr` was deallocated), or that the function did not allocate storage (and thus the block pointed by `ptr` was not modified).

A *null-pointer* indicates that the function failed to allocate storage, and thus the block pointed by `ptr` was not modified.

### Example

```
1 /* realloc example: rememb-o-matic */
2 #include <stdio.h>      /* printf, scanf, puts */
3 #include <stdlib.h>      /* realloc, free, exit, NULL */
4
5 int main ()
6 {
7     int input,n;
8     int count = 0;
9     int* numbers = NULL;
10    int* more_numbers = NULL;
11
12    do {
13        printf ("Enter an integer value (0 to end): ");
14        scanf ("%d", &input);
15        count++;
16
17        more_numbers = (int*) realloc (numbers, count * sizeof(int));
18
19        if (more_numbers!=NULL) {
20            numbers=more_numbers;
21            numbers[count-1]=input;
22        }
23        else {
24            free (numbers);
25        }
26    } while (input != 0);
27
28    free (numbers);
29
30    return 0;
31}
```

```

25     puts ("Error (re)allocating memory");
26     exit (1);
27 }
28 } while (input!=0);
29
30 printf ("Numbers entered: ");
31 for (n=0;n<count;n++) printf ("%d ",numbers[n]);
32 free (numbers);
33
34 return 0;
35 }

```

The program prompts the user for numbers until a zero character is entered. Each time a new value is introduced the memory block pointed by numbers is increased by the size of an `int`.

## Data races

Only the storage referenced by `ptr` and by the returned pointer are modified. No other storage locations are accessed by the call. If the function releases or reuses a unit of storage that is reused or released by another *allocation or deallocation function*, the functions are synchronized in such a way that the deallocation happens entirely before the next allocation.

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

## See also

<code>free</code>	Deallocate memory block (function )
<code>calloc</code>	Allocate and zero-initialize array (function )
<code>malloc</code>	Allocate memory block (function )

## /cstdlib/size\_t

type

## size\_t

`<cstddef> <cstdio> <cstdlib> <cstring> <ctime> <cwchar>`

### Unsigned integral type

Alias of one of the fundamental unsigned integer types.

It is a type able to represent the size of any object in bytes: `size_t` is the type returned by the `sizeof` operator and is widely used in the standard library to represent sizes and counts.

In `<cstdlib>`, it is used as the type of some parameters in the functions `bsearch`, `qsort`, `calloc`, `malloc`, `realloc`, `mblen`, `mbtowc`, `mbstowcs` and `wcstombs`, and in the case of `mbstowcs` and `wcstombs` also as its returning type.

In all cases it is used as a type to represent the length or count in bytes of a specific buffer or string.

## /cstdlib/srand

function

## srand

`<cstdlib>`

`void srand (unsigned int seed);`

### Initialize random number generator

The pseudo-random number generator is initialized using the argument passed as `seed`.

For every different `seed` value used in a call to `srand`, the pseudo-random number generator can be expected to generate a different succession of results in the subsequent calls to `rand`.

Two different initializations with the same `seed` will generate the same succession of results in subsequent calls to `rand`.

If `seed` is set to 1, the generator is reinitialized to its initial value and produces the same values as before any call to `rand` or `srand`.

In order to generate random-like numbers, `srand` is usually initialized to some distinctive runtime value, like the value returned by function `time` (declared in header `<ctime>`). This is distinctive enough for most trivial randomization needs.

## Parameters

`seed`

An integer value to be used as `seed` by the pseudo-random number generator algorithm.

## Return Value

none

## Example

```

1 /* srand example */
2 #include <stdio.h>      /* printf, NULL */
3 #include <stdlib.h>      /* srand, rand */
4 #include <time.h>        /* time */
5
6 int main ()

```

```

7 {
8     printf ("First number: %d\n", rand()%100);
9     srand (time(NULL));
10    printf ("Random number: %d\n", rand()%100);
11    srand (1);
12    printf ("Again the first number: %d\n", rand()%100);
13
14    return 0;
15 }

```

Possible output:

```

First number: 41
Random number: 13
Again the first number: 41

```

## Data races

The function accesses and modifies internal state objects, which may cause data races with concurrent calls to `rand` or `srand`.

Some libraries provide an alternative function of `rand` that explicitly avoids this kind of data race: `rand_r` (non-portable).

C++ library implementations are allowed to guarantee no *data races* for calling this function.

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

## See also

<code>rand</code>	Generate random number (function )
-------------------	------------------------------------

# /cstdlib/strtod

function

## strtod

<cstdlib>

`double strtod (const char* str, char** endptr);`

### Convert string to double

Parses the C-string `str` interpreting its content as a floating point number (according to the current locale) and returns its value as a `double`. If `endptr` is not a *null pointer*, the function also sets the value of `endptr` to point to the first character after the number.

The function first discards as many whitespace characters (as in `isspace`) as necessary until the first non-whitespace character is found. Then, starting from this character, takes as many characters as possible that are valid following a syntax resembling that of floating point literals (see below), and interprets them as a numerical value. A pointer to the rest of the string after the last valid character is stored in the object pointed by `endptr`.

A valid floating point number for `strtod` using the "C" locale is formed by an optional sign character (+ or -), followed by a sequence of digits, optionally containing a decimal-point character (.), optionally followed by an exponent part (an e or E character followed by an optional sign and a sequence of digits).

A valid floating point number for `strtod` using the "C" locale is formed by an optional sign character (+ or -), followed by one of:

- A sequence of digits, optionally containing a decimal-point character (.), optionally followed by an exponent part (an e or E character followed by an optional sign and a sequence of digits).
- A 0x or 0X prefix, then a sequence of hexadecimal digits (as in `isxdigit`) optionally containing a period which separates the whole and fractional number parts. Optionally followed by a power of 2 exponent (a p or P character followed by an optional sign and a sequence of hexadecimal digits).
- INF or INFINITY (ignoring case).
- NAN or NANsequence (ignoring case), where `sequence` is a sequence of characters, where each character is either an alphanumeric character (as in `isalnum`) or the underscore character (\_).

If the first sequence of non-whitespace characters in `str` does not form a valid floating-point number as just described, or if no such sequence exists because either `str` is empty or contains only whitespace characters, no conversion is performed and the function returns a zero value.

## Parameters

`str`

C-string beginning with the representation of a floating-point number.

`endptr`

Reference to an already allocated object of type `char*`, whose value is set by the function to the next character in `str` after the numerical value. This parameter can also be a *null pointer*, in which case it is not used.

## Return Value

On success, the function returns the converted floating point number as a value of type `double`.

If no valid conversion could be performed, the function returns zero (0.0).

If the correct value is out of the range of representable values for the type, a positive or negative `HUGE_VAL` is returned, and `errno` is set to `ERANGE`.

If the correct value would cause underflow, the function returns a value whose magnitude is no greater than the smallest normalized positive number and sets `errno` to `ERANGE`.

If the correct value would cause underflow, the function returns a value whose magnitude is no greater than the smallest normalized positive number (some library implementations may also set `errno` to `ERANGE` in this case).

## Example

```

1 /* strtod example */
2 #include <stdio.h>      /* printf, NULL */
3 #include <stdlib.h>      /* strtod */
4

```

```

5 int main ()
6 {
7     char szOrbits[] = "365.24 29.53";
8     char* pEnd;
9     double d1, d2;
10    d1 = strtod (szOrbits, &pEnd);
11    d2 = strtod (pEnd, NULL);
12    printf ("The moon completes %.2f orbits per Earth year.\n", d1/d2);
13    return 0;
14 }

```

Output:

```
The moon completes 12.37 orbits per Earth year.
```

## Data races

The array pointed by *str* is accessed, and the pointer pointed by *endptr* is modified (if not null).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if *endptr* does not point to a valid pointer object, it causes *undefined behavior*.

## See also

<a href="#">atof</a>	Convert string to double ( <a href="#">function</a> )
<a href="#">strtol</a>	Convert string to long integer ( <a href="#">function</a> )
<a href="#">strtoul</a>	Convert string to unsigned long integer ( <a href="#">function</a> )

## /cstdlib/strtod

function  
**strtod** <cstdlib>

```
float strtod (const char* str, char** endptr);
```

### Convert string to float

Parses the C-string *str* interpreting its content as a floating point number (according to the current locale) and returns its value as a *float*. If *endptr* is not a *null pointer*, the function also sets the value of *endptr* to point to the first character after the number.

The function first discards as many whitespace characters (as in [isspace](#)) as necessary until the first non-whitespace character is found. Then, starting from this character, takes as many characters as possible that are valid following a syntax resembling that of floating point literals (see below), and interprets them as a numerical value. A pointer to the rest of the string after the last valid character is stored in the object pointed by *endptr*.

A valid floating point number for *strtod* using the "c" locale is formed by an optional sign character (+ or -), followed by one of:

- A sequence of digits, optionally containing a decimal-point character (.), optionally followed by an exponent part (an e or E character followed by an optional sign and a sequence of digits).
- A 0x or 0X prefix, then a sequence of hexadecimal digits (as in [isxdigit](#)) optionally containing a period which separates the whole and fractional number parts. Optionally followed by a power of 2 exponent (a p or P character followed by an optional sign and a sequence of hexadecimal digits).
- INF or INFINITY (ignoring case).
- NAN or NANsequence (ignoring case), where *sequence* is a sequence of characters, where each character is either an alphanumeric character (as in [isalnum](#)) or the underscore character (\_).

If the first sequence of non-whitespace characters in *str* does not form a valid floating-point number as just described, or if no such sequence exists because either *str* is empty or contains only whitespace characters, no conversion is performed and the function returns 0.0F.

## Parameters

*str* C-string beginning with the representation of a floating-point number.

*endptr* Reference to an already allocated object of type *char\**, whose value is set by the function to the next character in *str* after the numerical value. This parameter can also be a *null pointer*, in which case it is not used.

## Return Value

On success, the function returns the converted floating point number as a value of type *float*.

If no valid conversion could be performed, the function returns zero (0.0F).

If the correct value is out of the range of representable values for the type, a positive or negative [HUGE\\_VALF](#) is returned, and *errno* is set to [ERANGE](#).

If the correct value would cause underflow, the function returns a value whose magnitude is no greater than the smallest normalized positive number (some library implementations may also set *errno* to [ERANGE](#) in this case).

## Example

```

1 /* strtod example */
2 #include <stdio.h>           /* printf, NULL */
3 #include <stdlib.h>          /* strtod */
4
5 int main ()
6 {
7     char szOrbits[] = "686.97 365.24";
8     char* pEnd;

```

```

9 float f1, f2;
10 f1 = strtod (szOrbits, &pEnd);
11 f2 = strtod (pEnd, NULL);
12 printf ("One martian year takes %.2f Earth years.\n", f1/f2);
13 return 0;
14 }
```

Output:

```
One martian year takes 1.88 Earth years.
```

## Data races

The array pointed by *str* is accessed, and the pointer pointed by *endptr* is modified (if not null).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if *endptr* does not point to a valid pointer object, it causes *undefined behavior*.

## See also

<a href="#">atof</a>	Convert string to double (function )
<a href="#">strtod</a>	Convert string to double (function )
<a href="#">strtol</a>	Convert string to long integer (function )

## /cstdlib/strtol

function  
**strtol** <cstdlib>

```
long int strtol (const char* str, char** endptr, int base);
```

### Convert string to long integer

Parses the C-string *str* interpreting its content as an integral number of the specified *base*, which is returned as a *long int* value. If *endptr* is not a null pointer, the function also sets the value of *endptr* to point to the first character after the number.

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes as many characters as possible that are valid following a syntax that depends on the *base* parameter, and interprets them as a numerical value. Finally, a pointer to the first character following the integer representation in *str* is stored in the object pointed by *endptr*.

If the value of *base* is zero, the syntax expected is similar to that of integer constants, which is formed by a succession of:

- An optional sign character (+ or -)
- An optional prefix indicating octal or hexadecimal base ("0" or "0x"/"0X" respectively)
- A sequence of decimal digits (if no base prefix was specified) or either octal or hexadecimal digits if a specific prefix is present

If the *base* value is between 2 and 36, the format expected for the integral number is a succession of any of the valid digits and/or letters needed to represent integers of the specified radix (starting from '0' and up to 'z'/'z' for radix 36). The sequence may optionally be preceded by a sign (either + or -) and, if *base* is 16, an optional "0x" or "0X" prefix.

If the first sequence of non-whitespace characters in *str* is not a valid integral number as defined above, or if no such sequence exists because either *str* is empty or it contains only whitespace characters, no conversion is performed.

For locales other than the "c" locale, additional subject sequence forms may be accepted.

## Parameters

**str** C-string beginning with the representation of an integral number.

**endptr** Reference to an object of type *char\**, whose value is set by the function to the next character in *str* after the numerical value. This parameter can also be a null pointer, in which case it is not used.

**base** Numerical base (radix) that determines the valid characters and their interpretation. If this is 0, the base used is determined by the format in the sequence (see above).

## Return Value

On success, the function returns the converted integral number as a *long int* value.

If no valid conversion could be performed, a zero value is returned (0L).

If the value read is out of the range of representable values by a *long int*, the function returns *LONG\_MAX* or *LONG\_MIN* (defined in *<climits>*), and *errno* is set to *ERANGE*.

## Example

```

1 /* strtol example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* strtol */
4
5 int main ()
6 {
7     char szNumbers[] = "2001 60c0c0 -1101110100110100100000 0xfffffff";
8     char * pEnd;
```

```

9 long int li1, li2, li3, li4;
10 li1 = strtol (szNumbers,&pEnd,10);
11 li2 = strtol (pEnd,&pEnd,16);
12 li3 = strtol (pEnd,&pEnd,2);
13 li4 = strtol (pEnd,NULL,0);
14 printf ("The decimal equivalents are: %ld, %ld, %ld and %ld.\n", li1, li2, li3, li4);
15 return 0;
16 }

```

Output:

```
The decimal equivalents are: 2001, 6340800, -3624224 and 7340031
```

## Data races

The array pointed by *str* is accessed, and the pointer pointed by *endptr* is modified (if not null).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if *endptr* does not point to a valid pointer object, it causes *undefined behavior*.

## See also

<a href="#">atol</a>	Convert string to long integer (function )
<a href="#">strtoul</a>	Convert string to unsigned long integer (function )
<a href="#">strtod</a>	Convert string to double (function )

## /cstdlib/strtold

function  
**strtold** <cstdlib>

```
long double strtold (const char* str, char** endptr);
```

### Convert string to long double

Parses the C string *str* interpreting its content as a floating point number (according to the current locale) and returns its value as a *long double*. If *endptr* is not a *null pointer*, the function also sets the value of *endptr* to point to the first character after the number.

This function operates like [strtod](#) to interpret the string, but produces numbers of type *long double* (see [strtod](#) for details on the interpretation process).

## Parameters

*str* C string beginning with the representation of a floating-point number.

*endptr* Reference to an already allocated object of type *char\**, whose value is set by the function to the next character in *str* after the numerical value.  
This parameter can also be a *null pointer*, in which case it is not used.

## Return Value

On success, the function returns the converted floating point number as a value of type *long double*.  
If no valid conversion could be performed, the function returns zero (0.0L).

If the correct value is out of the range of representable values for the type, a positive or negative [HUGE\\_VALL](#) is returned, and *errno* is set to [ERANGE](#).

If the correct value would cause underflow, the function returns a value whose magnitude is no greater than the smallest normalized positive number (some library implementations may also set *errno* to [ERANGE](#) in this case).

## Example

```

1 /* strtold example */
2 #include <stdio.h>      /* printf, NULL */
3 #include <stdlib.h>      /* strtold */
4
5 int main ()
6 {
7     char szOrbits[] = "90613.305 365.24";
8     char * pEnd;
9     long double f1, f2;
10    f1 = strtold (szOrbits, &pEnd);
11    f2 = strtold (pEnd, NULL);
12    printf ("Pluto takes %.2Lf years to complete an orbit.\n", f1/f2);
13    return 0;
14 }

```

Output:

```
Pluto takes 248.09 years to complete an orbit.
```

## Data races

The array pointed by *str* is accessed, and the pointer pointed by *endptr* is modified (if not null).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if *endptr* does not point to a valid pointer object, it causes *undefined behavior*.

## See also

<a href="#">atof</a>	Convert string to double (function )
<a href="#">strtod</a>	Convert string to double (function )
<a href="#">strtol</a>	Convert string to long integer (function )

## /cstdlib/strtoll

function

### strtoll

<cstdlib>

```
long long int strtoll (const char* str, char** endptr, int base);
```

#### Convert string to long long integer

Parses the C-string *str* interpreting its content as an integral number of the specified *base*, which is returned as a value of type `long long int`. If *endptr* is not a *null pointer*, the function also sets the value of *endptr* to point to the first character after the number.

This function operates like [strtol](#) to interpret the string, but produces numbers of type `long long int` (see [strtol](#) for details on the interpretation process).

## Parameters

<code>str</code>	C-string beginning with the representation of an integral number.
<code>endptr</code>	Reference to an object of type <code>char*</code> , whose value is set by the function to the next character in <i>str</i> after the numerical value. This parameter can also be a <i>null pointer</i> , in which case it is not used.
<code>base</code>	Numerical base (radix) that determines the valid characters and their interpretation. If this is 0, the base used is determined by the format in the sequence (see <a href="#">strtol</a> for details).

## Return Value

On success, the function returns the converted integral number as a `long int` value.

If no valid conversion could be performed, a zero value is returned (`0LL`).

If the value read is out of the range of representable values by a `long long int`, the function returns `LLONG_MAX` or `LLONG_MIN` (defined in [<climits>](#)), and `errno` is set to `ERANGE`.

## Example

```
1 /* strtoll example */
2 #include <stdio.h>           /* printf, NULL */
3 #include <stdlib.h>          /* strtoll */
4
5 int main ()
6 {
7     char szNumbers[] = "1856892505 17b00a12b -01100011010110000010001101100 0x6fffff";
8     char* pEnd;
9     long long int lli1, lli2, lli3, lli4;
10    lli1 = strtoll (szNumbers, &pEnd, 10);
11    lli2 = strtoll (pEnd, &pEnd, 16);
12    lli3 = strtoll (pEnd, &pEnd, 2);
13    lli4 = strtoll (pEnd, NULL, 0);
14    printf ("The decimal equivalents are: %lld, %lld, %lld and %lld.\n", lli1, lli2, lli3, lli4);
15    return 0;
16 }
```

Possible output:

```
The decimal equivalents are: 1856892505, 6358606123, -208340076 and 7340031
```

## Data races

The array pointed by *str* is accessed, and the pointer pointed by *endptr* is modified (if not *null*).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if *endptr* does not point to a valid pointer object, it causes *undefined behavior*.

## See also

<a href="#">strtol</a>	Convert string to long integer (function )
<a href="#">atol</a>	Convert string to long integer (function )
<a href="#">strtoull</a>	Convert string to unsigned long long integer (function )
<a href="#">strtod</a>	Convert string to double (function )

## /cstdlib/strtoul

function  
**strtoul** <cstdlib>

```
unsigned long int strtoul (const char* str, char** endptr, int base);
```

### Convert string to unsigned long integer

Parses the C-string *str*, interpreting its content as an integral number of the specified *base*, which is returned as an value of type `unsigned long int`.

This function operates like `strtol` to interpret the string, but produces numbers of type `unsigned long int` (see `strtol` for details on the interpretation process).

### Parameters

**str** C-string containing the representation of an integral number.

**endptr** Reference to an object of type `char*`, whose value is set by the function to the next character in *str* after the numerical value.  
This parameter can also be a *null pointer*, in which case it is not used.

**base** Numerical base (radix) that determines the valid characters and their interpretation.  
If this is 0, the base used is determined by the format in the sequence (see `strtol` for details).

### Return Value

On success, the function returns the converted integral number as an `unsigned long int` value.

If no valid conversion could be performed, a zero value is returned.

If the value read is out of the range of representable values by an `unsigned long int`, the function returns `ULONG_MAX` (defined in `<climits>`), and `errno` is set to `ERANGE`.

### Example

```
1 /* strtoul example */
2 #include <stdio.h>      /* printf, NULL */
3 #include <stdlib.h>      /* strtoul */
4
5 int main ()
6 {
7     char buffer [256];
8     unsigned long ul;
9     printf ("Enter an unsigned number: ");
10    fgets (buffer, 256, stdin);
11    ul = strtoul (buffer, NULL, 0);
12    printf ("Value entered: %lu. Its double: %lu\n",ul,ul*2);
13    return 0;
14 }
```

Possible output:

```
Enter an unsigned number: 30003
Value entered: 30003. Its double: 60006
```

For an example with the *endptr* parameter in action see `strtol`.

### Data races

The array pointed by *str* is accessed, and the pointer pointed by *endptr* is modified (if not null).

### Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If *str* does not point to a valid C-string, or if *endptr* does not point to a valid pointer object, it causes *undefined behavior*.

### See also

<a href="#">atoi</a>	Convert string to long integer (function )
<a href="#">strtol</a>	Convert string to long integer (function )
<a href="#">strtod</a>	Convert string to double (function )

## /cstdlib/strtoull

function  
**strtoull** <cstdlib>

```
unsigned long long int strtoull (const char* str, char** endptr, int base);
```

### Convert string to unsigned long long integer

Parses the C-string *str* interpreting its content as an integral number of the specified *base*, which is returned as a value of type `unsigned long long int`. If *endptr* is not a null pointer, the function also sets the value of *endptr* to point to the first character after the number.

This function operates like `strtol` to interpret the string, but produces numbers of type `unsigned long long int` (see `strtol` for details on the interpretation process).

## Parameters

`str` C-string beginning with the representation of an integral number.

`endptr` Reference to an object of type `char*`, whose value is set by the function to the next character in `str` after the numerical value. This parameter can also be a *null pointer*, in which case it is not used.

`base` Numerical base (radix) that determines the valid characters and their interpretation. If this is 0, the base used is determined by the format in the sequence (see `strtol` for details).

## Return Value

On success, the function returns the converted integral number as an `unsigned long long int` value.

If no valid conversion could be performed, a zero value is returned (`ULLL`).

If the value read is out of the range of representable values by an `unsigned long long int`, the function returns `ULLONG_MAX` (defined in `<climits>`), and `errno` is set to `ERANGE`.

## Example

```
1 /* strtoull example */
2 #include <stdio.h>           /* printf, NULL */
3 #include <stdlib.h>          /* strtoull */
4
5 int main ()
6 {
7     char szNumbers[] = "250068492 7b06af00 11000110111101010001100000 0x6fffff";
8     char * pEnd;
9     unsigned long long int ulli1, ulli2, ulli3, ulli4;
10    ulli1 = strtoull (szNumbers, &pEnd, 10);
11    ulli2 = strtoull (pEnd, &pEnd, 16);
12    ulli3 = strtoull (pEnd, &pEnd, 2);
13    ulli4 = strtoull (pEnd, NULL, 0);
14    printf ("The decimal equivalents are: %llu, %llu, %llu and %llu.\n", ulli1, ulli2, ulli3, ulli4);
15    return 0;
16 }
```

Output:

```
The decimal equivalents are: 250068492, 2064035584, 208622688 and 7340031.
```

## Data races

The array pointed by `str` is accessed, and the pointer pointed by `endptr` is modified (if not null).

## Exceptions (C++)

**No-throw guarantee:** this function never throws exceptions.

If `str` does not point to a valid C-string, or if `endptr` does not point to a valid pointer object, it causes *undefined behavior*.

## See also

<a href="#">strtoul</a>	Convert string to unsigned long integer (function )
<a href="#">atol</a>	Convert string to long integer (function )
<a href="#">strtoll</a>	Convert string to long long integer (function )
<a href="#">strtod</a>	Convert string to double (function )

## /cstdlib/system

function  
**system** `<cstdlib>`

```
int system (const char* command);
```

### Execute system command

Invokes the command processor to execute a `command`.

If `command` is a *null pointer*, the function only checks whether a `command processor` is available through this function, without invoking any command.

The effects of invoking a command depend on the system and library implementation, and may cause a program to behave in a non-standard manner or to terminate.

## Parameters

`command` C-string containing the system command to be executed.  
Or, alternatively, a *null pointer*, to check for a command processor.

## Return Value

If *command* is a null pointer, the function returns a non-zero value in case a *command processor* is available and a zero value if it is not.

If *command* is not a null pointer, the value returned depends on the system and library implementations, but it is generally expected to be the status code returned by the called command, if supported.

## Example

```
1 /* system example : DIR */
2 #include <stdio.h>           /* printf */
3 #include <stdlib.h>          /* system, NULL, EXIT_FAILURE */
4
5 int main ()
6 {
7     int i;
8     printf ("Checking if processor is available...");
9     if (system(NULL)) puts ("Ok");
10    else exit (EXIT_FAILURE);
11    printf ("Executing command DIR...\n");
12    i=system ("dir");
13    printf ("The value returned was: %d.\n",i);
14    return 0;
15 }
```

## Data races

The function accesses the array pointed by *command*.

Concurrently calling this function with a *null pointer* as argument is safe. Otherwise, it depends on the system and library implementation.

## Exceptions (C++)

**No-throw guarantee:** this function does not throw exceptions.

If *command* is not a *null pointer*, it causes *undefined behavior*.

## See also

<a href="#">exit</a>	Terminate calling process (function )
<a href="#">getenv</a>	Get environment string (function )

## /cstdlib/wcstombs

function

### wcstombs

<cstdlib>

`size_t wcstombs (char* dest, const wchar_t* src, size_t max);`

#### Convert wide-character string to multibyte string

Translates wide characters from the sequence pointed by *src* to the multibyte equivalent sequence (which is stored at the array pointed by *dest*), up until either *max* bytes have been translated or until a wide characters translates into a *null character*.

If *max* bytes are successfully translated, the resulting string stored in *dest* is not null-terminated.

The resulting multibyte sequence begins in the initial shift state (if any).

The behavior of this function depends on the `LC_CTYPE` category of the selected C locale.

## Parameters

*dest*

Pointer to an array of `char` elements long enough to contain the resulting sequence (at most, *max* bytes).

*src*

C wide string to be translated.

*max*

Maximum number of bytes to be written to *dest*.

`size_t` is an unsigned integral type.

## Return Value

The number of bytes written to *dest*, not including the eventual ending null-character.

If a wide character that does not correspond to a valid multibyte character is encountered, a (`size_t`)-1 value is returned.

Notice that `size_t` is an unsigned integral type, and thus none of the values possibly returned is less than zero.

## Example

```
1 /* wcstombs example */
2 #include <stdio.h>           /* printf */
3 #include <stdlib.h>          /* wcstombs, wchar_t(C) */
4
5 int main() {
6     const wchar_t str[] = L"wcstombs example";
7     char buffer[32];
8     int ret;
9
10    printf ("wchar_t string: %ls \n",str);
```

```

11     ret = wcstombs ( buffer, str, sizeof(buffer) );
12     if (ret==32) buffer[31]='\0';
13     if (ret) printf ("multibyte string: %s \n",buffer);
14
15     return 0;
16 }

```

Output:

```
wchar_t string: wcstombs example
multibyte string: wcstombs example
```

## Data races

The function accesses the array pointed by *src*, and modifies the array pointed by *dest*. The function may also access and modify an internal state object, which may cause data races on concurrent calls to this function if the implementation uses a static object (see `wcsrtombs` for an alternative that can use an external state object). Concurrently changing locale settings may also introduce data races.

## Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If *dest* does not point to an array long enough to contain the translated sequence, or if *src* is either not null-terminated or does not contain enough wide characters to generate a sequence of *max* multibyte characters, it causes *undefined behavior*.

## See also

<code>mblen</code>	Get length of multibyte character ( <a href="#">function</a> )
<code>wctomb</code>	Convert wide character to multibyte sequence ( <a href="#">function</a> )
<code>mbstowcs</code>	Convert multibyte string to wide-character string ( <a href="#">function</a> )

## /cstdlib/wctomb

function  
**wctomb** <cstdlib>

`int wctomb (char* pmb, wchar_t wc);`

### Convert wide character to multibyte sequence

The wide character *wc* is translated to its multibyte equivalent and stored in the array pointed by *pmb*. The function returns the length in bytes of the equivalent multibyte sequence pointed by *pmb* after the call.

*wctomb* has its own internal *shift state*, which is altered as necessary only by calls to this function. A call to the function with a null pointer as *pmb* resets the state (and returns whether multibyte sequences are state-dependent).

The behavior of this function depends on the `LC_CTYPE` category of the selected C locale.

## Parameters

*pmb*

Pointer to an array large enough to hold a multibyte sequence.

The maximum length of a multibyte sequence for a character in the current locale is `MB_CUR_MAX` bytes.

Alternatively, the function may be called with a *null pointer*, in which case the function resets its internal shift state to the initial value and returns whether multibyte sequences use a state-dependent encoding.

*wc*

Wide character of type `wchar_t`.

## Return Value

If the argument passed as *pmb* is not a null pointer, the size in bytes of the character written to *pmb* is returned. If there is no character correspondence, -1 is returned.

If the argument passed as *pmb* is a null pointer, the function returns a nonzero value if multibyte character encodings are state-dependent, and zero otherwise.

## Example

```

1 /* wctomb example */
2 #include <stdio.h>      /* printf */
3 #include <stdlib.h>      /* wctomb, wchar_t(C) */
4
5 int main() {
6     const wchar_t str[] = L"wctomb example";
7     const wchar_t* pt;
8     char buffer [MB_CUR_MAX];
9     int i,length;
10
11    pt = str;
12    while (*pt) {
13        length = wctomb(buffer,*pt);
14        if (length<1) break;
15        for (i=0;i<length;++i) printf ("[%c]",buffer[i]);
16        ++pt;
17    }
18 }
```

```
19 |     return 0;
20 }
```

The example prints the multibyte characters that a wide character string translates to, using the selected locale (in this case, the default "c" locale).

Output:

```
[w][c][t][o][m][b][ ][e][x][a][m][p][l][e]
```

## Data races

The function modifies the array pointed by *pmb*.

The function also accesses and modifies an internal state object, which may cause data races on concurrent calls to this function (see `wcrtombs` for an alternative that may use an external state object).

Concurrently changing locale settings may also introduce data races.

## Exceptions (C++)

**No-throw guarantee:** this function throws no exceptions.

If *pmb* is neither a *null pointer* nor a pointer to an array long enough for the translated character, it causes *undefined behavior*.

## See also

<b>mblen</b>	Get length of multibyte character ( <a href="#">function</a> )
<b>mbtowc</b>	Convert multibyte sequence to wide character ( <a href="#">function</a> )
<b>mbstowcs</b>	Convert multibyte string to wide-character string ( <a href="#">function</a> )
<b>wcstombs</b>	Convert wide-character string to multibyte string ( <a href="#">function</a> )

# /cstring

header

## <cstring> (string.h)

### C Strings

This header file defines several functions to manipulate C strings and arrays.

### Functions

#### Copying:

<b>memcpy</b>	Copy block of memory ( <a href="#">function</a> )
<b>memmove</b>	Move block of memory ( <a href="#">function</a> )
<b>strcpy</b>	Copy string ( <a href="#">function</a> )
<b>strncpy</b>	Copy characters from string ( <a href="#">function</a> )

#### Concatenation:

<b>strcat</b>	Concatenate strings ( <a href="#">function</a> )
<b>strncat</b>	Append characters from string ( <a href="#">function</a> )

#### Comparison:

<b>memcmp</b>	Compare two blocks of memory ( <a href="#">function</a> )
<b>strcmp</b>	Compare two strings ( <a href="#">function</a> )
<b>strcoll</b>	Compare two strings using locale ( <a href="#">function</a> )
<b>strncmp</b>	Compare characters of two strings ( <a href="#">function</a> )
<b>strxfrm</b>	Transform string using locale ( <a href="#">function</a> )

#### Searching:

<b>memchr</b>	Locate character in block of memory ( <a href="#">function</a> )
<b>strchr</b>	Locate first occurrence of character in string ( <a href="#">function</a> )
<b>strcspn</b>	Get span until character in string ( <a href="#">function</a> )
<b>strpbrk</b>	Locate characters in string ( <a href="#">function</a> )
<b>strrchr</b>	Locate last occurrence of character in string ( <a href="#">function</a> )
<b>strspn</b>	Get span of character set in string ( <a href="#">function</a> )
<b>strstr</b>	Locate substring ( <a href="#">function</a> )
<b>strtok</b>	Split string into tokens ( <a href="#">function</a> )

#### Other:

<b>memset</b>	Fill block of memory ( <a href="#">function</a> )
<b>strerror</b>	Get pointer to error message string ( <a href="#">function</a> )
<b>strlen</b>	Get string length ( <a href="#">function</a> )

## Macros

<b>NULL</b>	Null pointer ( <a href="#">macro</a> )
-------------	--

## Types

<a href="#">size_t</a>	Unsigned integral type ( <a href="#">type</a> )
------------------------	---

## /cstring/memchr

function

### memchr

<cstring>

```
const void * memchr ( const void * ptr, int value, size_t num );
void * memchr (      void * ptr, int value, size_t num );
```

#### Locate character in block of memory

Searches within the first *num* bytes of the block of memory pointed by *ptr* for the first occurrence of *value* (interpreted as an *unsigned char*), and returns a pointer to it.

Both *value* and each of the bytes checked on the the *ptr* array are interpreted as *unsigned char* for the comparison.

### Parameters

*ptr*

Pointer to the block of memory where the search is performed.

*value*

Value to be located. The value is passed as an *int*, but the function performs a byte per byte search using the *unsigned char* conversion of this value.

*num*

Number of bytes to be analyzed.

*size\_t* is an unsigned integral type.

### Return Value

A pointer to the first occurrence of *value* in the block of memory pointed by *ptr*.

If the *value* is not found, the function returns a null pointer.

### Portability

In C, this function is only declared as:

```
void * memchr ( const void *, int, size_t );
```

instead of the two overloaded versions provided in C++.

### Example

```
1 /* memchr example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char * pch;
8     char str[] = "Example string";
9     pch = (char*) memchr (str, 'p', strlen(str));
10    if (pch!=NULL)
11        printf ("'p' found at position %d.\n", pch-str+1);
12    else
13        printf ("'p' not found.\n");
14    return 0;
15 }
```

Output:

```
'p' found at position 5.
```

### See also

<a href="#">memcmp</a>	Compare two blocks of memory ( <a href="#">function</a> )
------------------------	---

<a href="#">strchr</a>	Locate first occurrence of character in string ( <a href="#">function</a> )
------------------------	---

<a href="#"> strrchr</a>	Locate last occurrence of character in string ( <a href="#">function</a> )
--------------------------	--

## /cstring/memcmp

function

### memcmp

<cstring>

```
int memcmp ( const void * ptr1, const void * ptr2, size_t num );
```

#### Compare two blocks of memory

Compares the first *num* bytes of the block of memory pointed by *ptr1* to the first *num* bytes pointed by *ptr2*, returning zero if they all match or a value different from zero representing which is greater if they do not.

Notice that, unlike [strcmp](#), the function does not stop comparing after finding a null character.

## Parameters

**ptr1** Pointer to block of memory.  
**ptr2** Pointer to block of memory.  
**num** Number of bytes to compare.

## Return Value

Returns an integral value indicating the relationship between the content of the memory blocks:

return value	indicates
<0	the first byte that does not match in both memory blocks has a lower value in <i>ptr1</i> than in <i>ptr2</i> (if evaluated as <i>unsigned char</i> values)
0	the contents of both memory blocks are equal
>0	the first byte that does not match in both memory blocks has a greater value in <i>ptr1</i> than in <i>ptr2</i> (if evaluated as <i>unsigned char</i> values)

## Example

```
1 /* memcmp example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char buffer1[] = "DWgaOtP12df0";
8     char buffer2[] = "DWGAOTP12DF0";
9
10    int n;
11
12    n=memcmp ( buffer1, buffer2, sizeof(buffer1) );
13
14    if (n>0) printf ("'%s' is greater than '%s'.\n",buffer1,buffer2);
15    else if (n<0) printf ("'%s' is less than '%s'.\n",buffer1,buffer2);
16    else printf ("'%s' is the same as '%s'.\n",buffer1,buffer2);
17
18    return 0;
19 }
```

Output:

```
'DWgaOtP12df0' is greater than 'DWGAOTP12DF0'.
```

DWgaOtP12df0 is greater than DWGAOTP12DF0 because the first non-matching character in both words are 'g' and 'G' respectively, and 'g' (103) evaluates as greater than 'G' (71).

## See also

<b>strcmp</b>	Compare two strings ( <a href="#">function</a> )
<b>memchr</b>	Locate character in block of memory ( <a href="#">function</a> )
<b>memcpy</b>	Copy block of memory ( <a href="#">function</a> )
<b>memset</b>	Fill block of memory ( <a href="#">function</a> )
<b>strncpy</b>	Compare characters of two strings ( <a href="#">function</a> )

## /cstring/memcpy

function

### memcpy

<cstring>

```
void * memcpy ( void * destination, const void * source, size_t num );
```

#### Copy block of memory

Copies the values of *num* bytes from the location pointed to by *source* directly to the memory block pointed to by *destination*.

The underlying type of the objects pointed to by both the *source* and *destination* pointers are irrelevant for this function; The result is a binary copy of the data.

The function does not check for any terminating null character in *source* - it always copies exactly *num* bytes.

To avoid overflows, the size of the arrays pointed to by both the *destination* and *source* parameters, shall be at least *num* bytes, and should not overlap (for overlapping memory blocks, [memmove](#) is a safer approach).

## Parameters

**destination** Pointer to the destination array where the content is to be copied, type-casted to a pointer of type `void*`.  
**source** Pointer to the source of data to be copied, type-casted to a pointer of type `const void*`.  
**num** Number of bytes to copy.  
`size_t` is an unsigned integral type.

## Return Value

*destination* is returned.

## Example

```
1 /* memcpy example */
2 #include <stdio.h>
3 #include <string.h>
4
5 struct {
6     char name[40];
7     int age;
8 } person, person_copy;
9
10 int main ()
11 {
12     char myname[] = "Pierre de Fermat";
13
14     /* using memcpy to copy string: */
15     memcpy ( person.name, myname, strlen(myname)+1 );
16     person.age = 46;
17
18     /* using memcpy to copy structure: */
19     memcpy ( &person_copy, &person, sizeof(person) );
20
21     printf ("person_copy: %s, %d \n", person_copy.name, person_copy.age );
22
23     return 0;
24 }
```

Output:

```
person_copy: Pierre de Fermat, 46
```

## See also

<a href="#">memmove</a>	Move block of memory ( <a href="#">function</a> )
<a href="#">memchr</a>	Locate character in block of memory ( <a href="#">function</a> )
<a href="#">memcmp</a>	Compare two blocks of memory ( <a href="#">function</a> )
<a href="#">memset</a>	Fill block of memory ( <a href="#">function</a> )
<a href="#">strncpy</a>	Copy characters from string ( <a href="#">function</a> )

## /cstring/memmove

function

### memmove

<cstring>

```
void * memmove ( void * destination, const void * source, size_t num );
```

#### Move block of memory

Copies the values of *num* bytes from the location pointed by *source* to the memory block pointed by *destination*. Copying takes place as if an intermediate buffer were used, allowing the *destination* and *source* to overlap.

The underlying type of the objects pointed by both the *source* and *destination* pointers are irrelevant for this function; The result is a binary copy of the data.

The function does not check for any terminating null character in *source* - it always copies exactly *num* bytes.

To avoid overflows, the size of the arrays pointed by both the *destination* and *source* parameters, shall be at least *num* bytes.

## Parameters

### destination

Pointer to the destination array where the content is to be copied, type-casted to a pointer of type `void*`.

### source

Pointer to the source of data to be copied, type-casted to a pointer of type `const void*`.

### num

Number of bytes to copy.

`size_t` is an unsigned integral type.

## Return Value

*destination* is returned.

## Example

```
1 /* memmove example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "memmove can be very useful.....";
8     memmove (str+20,str+15,11);
```

```
9 |     puts (str);
10|     return 0;
11| }
```

Output:

```
memmove can be very very useful.
```

## See also

<a href="#">memcpy</a>	Copy block of memory (function )
<a href="#">memchr</a>	Locate character in block of memory (function )
<a href="#">memcmp</a>	Compare two blocks of memory (function )
<a href="#">memset</a>	Fill block of memory (function )
<a href="#">strncpy</a>	Copy characters from string (function )

## /cstring/memset

function

### memset

<cstring>

```
void * memset ( void * ptr, int value, size_t num );
```

#### Fill block of memory

Sets the first *num* bytes of the block of memory pointed by *ptr* to the specified *value* (interpreted as an *unsigned char*).

## Parameters

*ptr* Pointer to the block of memory to fill.

*value* Value to be set. The value is passed as an *int*, but the function fills the block of memory using the *unsigned char* conversion of this *value*.

*num* Number of bytes to be set to the *value*.  
*size\_t* is an unsigned integral type.

## Return Value

*ptr* is returned.

## Example

```
1 /* memset example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "almost every programmer should know memset!";
8     memset (str,'-',6);
9     puts (str);
10    return 0;
11 }
```

Output:

```
----- every programmer should know memset!
```

## See also

<a href="#">memcpy</a>	Copy block of memory (function )
<a href="#">strncpy</a>	Copy characters from string (function )
<a href="#">memcmp</a>	Compare two blocks of memory (function )

## /cstring/NULL

macro

### NULL

<cstddef> <cstdlib> <cstring> <cwchar> <ctime> <clocale> <cstdio>

#### Null pointer

This macro expands to a *null pointer constant*.

A *null-pointer constant* is an integral constant expression that evaluates to zero (like 0 or 0L), or the cast of such value to type *void\** (like (*void\**)0).

A *null-pointer constant* is an integral constant expression that evaluates to zero (such as 0 or 0L).

A *null-pointer constant* is either an integral constant expression that evaluates to zero (such as 0 or 0L), or a value of type *nullptr\_t* (such as *nullptr*).

A null pointer constant can be converted to any *pointer type* (or *pointer-to-member type*), which acquires a *null pointer value*. This is a special value that

indicates that the pointer is not pointing to any object.

## /cstring/size\_t

type

### size\_t

<cstddef> <cstdio> <stdlib> <cstring> <ctime> <cwchar>

#### Unsigned integral type

Alias of one of the fundamental unsigned integer types.

It is a type able to represent the size of any object in bytes: `size_t` is the type returned by the `sizeof` operator and is widely used in the standard library to represent sizes and counts.

In `<cstring>`, it is used as the type of the parameter `num` in the functions `memchr`, `memcmp`, `memcpy`, `memmove`, `memset`, `strcat`, `strcmp`, `strncpy` and `strxfrm`, which in all cases it is used to specify the maximum number of bytes or characters the function has to affect.

It is also used as the return type for `strcspn`, `strlen`, `strspn` and `strxfrm` to return sizes and lengths.

## /cstring/strcat

function

### strcat

<cstring>

`char * strcat ( char * destination, const char * source );`

#### Concatenate strings

Appends a copy of the `source` string to the `destination` string. The terminating null character in `destination` is overwritten by the first character of `source`, and a null-character is included at the end of the new string formed by the concatenation of both in `destination`.

`destination` and `source` shall not overlap.

#### Parameters

`destination`

Pointer to the destination array, which should contain a C string, and be large enough to contain the concatenated resulting string.

`source`

C string to be appended. This should not overlap `destination`.

#### Return Value

`destination` is returned.

#### Example

```
1 /* strcat example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[80];
8     strcpy (str,"these ");
9     strcat (str,"strings ");
10    strcat (str,"are ");
11    strcat (str,"concatenated.");
12    puts (str);
13    return 0;
14 }
```

Output:

these strings are concatenated.

#### See also

<a href="#">strcat</a>	Append characters from string (function )
<a href="#">strcpy</a>	Copy string (function )
<a href="#">memcpy</a>	Copy block of memory (function )

## /cstring/strchr

function

### strchr

<cstring>

`const char * strchr ( const char * str, int character );`  
`char * strchr ( char * str, int character );`

#### Locate first occurrence of character in string

Returns a pointer to the first occurrence of `character` in the C string `str`.

The terminating null-character is considered part of the C string. Therefore, it can also be located in order to retrieve a pointer to the end of a string.

## Parameters

**str**  
C string.  
**character**  
Character to be located. It is passed as its `int` promotion, but it is internally converted back to `char` for the comparison.

## Return Value

A pointer to the first occurrence of `character` in `str`.  
If the `character` is not found, the function returns a null pointer.

## Portability

In C, this function is only declared as:

```
char * strchr ( const char *, int );
```

instead of the two overloaded versions provided in C++.

## Example

```
1 /* strchr example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "This is a sample string";
8     char * pch;
9     printf ("Looking for the 's' character in \"%s\"...\n",str);
10    pch=strchr(str,'s');
11    while ( pch!=NULL )
12    {
13        printf ("found at %d\n",pch-str+1);
14        pch=strchr(pch+1,'s');
15    }
16    return 0;
17 }
```

Output:

```
Looking for the 's' character in "This is a sample string"...
found at 4
found at 7
found at 11
found at 18
```

## See also

<a href="#">strchr</a>	Locate last occurrence of character in string (function )
<a href="#">memchr</a>	Locate character in block of memory (function )
<a href="#">strupr</a>	Locate characters in string (function )

# /cstring/strcmp

function  
**strcmp** <cstring>

```
int strcmp ( const char * str1, const char * str2 );
```

### Compare two strings

Compares the C string `str1` to the C string `str2`.

This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached.

This function performs a binary comparison of the characters. For a function that takes into account locale-specific rules, see [strcoll](#).

## Parameters

**str1**  
C string to be compared.  
**str2**  
C string to be compared.

## Return Value

Returns an integral value indicating the relationship between the strings:

return value	indicates
<0	the first character that does not match has a lower value in <code>ptr1</code> than in <code>ptr2</code>

0	the contents of both strings are equal
>0	the first character that does not match has a greater value in <i>ptr1</i> than in <i>ptr2</i>

## Example

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main ()
5 {
6     char key[] = "apple";
7     char buffer[80];
8     do {
9         printf ("Guess my favorite fruit? ");
10        fflush (stdout);
11        scanf ("%79s",buffer);
12    } while (strcmp (key,buffer) != 0);
13    puts ("Correct answer!");
14    return 0;
15 }
```

Output:

```

Guess my favourite fruit? orange
Guess my favourite fruit? apple
Correct answer!
```

## See also

<a href="#">strcmp</a>	Compare characters of two strings ( <a href="#">function</a> )
<a href="#">memcmp</a>	Compare two blocks of memory ( <a href="#">function</a> )
<a href="#">strchr</a>	Locate last occurrence of character in string ( <a href="#">function</a> )
<a href="#">strspn</a>	Get span of character set in string ( <a href="#">function</a> )

## /cstring/strcoll

function

### strcoll

<cstring>

```
int strcoll ( const char * str1, const char * str2 );
```

#### Compare two strings using locale

Compares the C string *str1* to the C string *str2*, both interpreted appropriately according to the `LC_COLLATE` category of the [C locale](#) currently selected.

This function starts comparing the first character of each string. If they are equal to each other continues with the following pair until the characters differ or until a null-character signaling the end of a string is reached.

The behavior of this function depends on the `LC_COLLATE` category of the selected [C locale](#).

## Parameters

<code>str1</code>	C string to be compared.
<code>str2</code>	C string to be compared.

## Return Value

Returns an integral value indicating the relationship between the strings:

A zero value indicates that both strings are equal.

A value greater than zero indicates that the first character that does not match has a greater value in *str1* than in *str2*; And a value less than zero indicates the opposite.

## See also

<a href="#">strcmp</a>	Compare two strings ( <a href="#">function</a> )
<a href="#">strcmp</a>	Compare characters of two strings ( <a href="#">function</a> )
<a href="#">memcmp</a>	Compare two blocks of memory ( <a href="#">function</a> )

## /cstring/strcpy

function

### strcpy

<cstring>

```
char * strcpy ( char * destination, const char * source );
```

#### Copy string

Copies the C string pointed by *source* into the array pointed by *destination*, including the terminating null character (and stopping at that point).

To avoid overflows, the size of the array pointed by *destination* shall be long enough to contain the same C string as *source* (including the terminating null character), and should not overlap in memory with *source*.

## Parameters

**destination**  
Pointer to the destination array where the content is to be copied.

**source**  
C string to be copied.

## Return Value

*destination* is returned.

## Example

```
1 /* strcpy example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str1[]="Sample string";
8     char str2[40];
9     char str3[40];
10    strcpy (str2,str1);
11    strcpy (str3,"copy successful");
12    printf ("%s\n%s\n%s\n",str1,str2,str3);
13    return 0;
14 }
```

Output:

```
str1: Sample string
str2: Sample string
str3: copy successful
```

## See also

<a href="#">strncpy</a>	Copy characters from string (function )
<a href="#">memcpy</a>	Copy block of memory (function )
<a href="#">memmove</a>	Move block of memory (function )
<a href="#">memchr</a>	Locate character in block of memory (function )
<a href="#">memcmp</a>	Compare two blocks of memory (function )
<a href="#">memset</a>	Fill block of memory (function )

## /cstring/strcspn

function

### strcspn

<cstring>

```
size_t strcspn ( const char * str1, const char * str2 );
```

#### Get span until character in string

Scans *str1* for the first occurrence of any of the characters that are part of *str2*, returning the number of characters of *str1* read before this first occurrence.

The search includes the terminating null-characters. Therefore, the function will return the length of *str1* if none of the characters of *str2* are found in *str1*.

## Parameters

**str1**  
C string to be scanned.

**str2**  
C string containing the characters to match.

## Return value

The length of the initial part of *str1* **not** containing any of the characters that are part of *str2*. This is the length of *str1* if none of the characters in *str2* are found in *str1*. *size\_t* is an unsigned integral type.

## Example

```
1 /* strcspn example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "fcba73";
8     char keys[] = "1234567890";
9     int i;
10    i = strcspn (str,keys);
11    printf ("The first number in str is at position %d.\n",i+1);
```

```
12 |     return 0;
13 }
```

Output:

```
The first number in str is at position 5
```

## See also

<a href="#">strpbrk</a>	Locate characters in string (function )
<a href="#">strspn</a>	Get span of character set in string (function )
<a href="#">strstr</a>	Locate substring (function )
<a href="#">strcmp</a>	Compare characters of two strings (function )

## /cstring/strerror

function

### strerror

<cstring>

```
char * strerror ( int errnum );
```

#### Get pointer to error message string

Interprets the value of *errnum*, generating a string with a message that describes the error condition as if set to *errno* by a function of the library.

The returned pointer points to a statically allocated string, which shall not be modified by the program. Further calls to this function may overwrite its content (particular library implementations are not required to avoid data races).

The error strings produced by *strerror* may be specific to each system and library implementation.

## Parameters

*errnum*  
Error number.

## Return Value

A pointer to the error string describing error *errnum*.

## Example

```
1 /* strerror example : error list */
2 #include <stdio.h>
3 #include <string.h>
4 #include <errno.h>
5
6 int main ()
7 {
8     FILE * pFile;
9     pFile = fopen ("unexist.ent", "r");
10    if (pFile == NULL)
11        printf ("Error opening file unexist.ent: %s\n", strerror(errno));
12    return 0;
13 }
```

Possible output:

```
Error opening file unexist.ent: No such file or directory
```

## See also

<a href="#">errno</a>	Last error number (macro )
<a href="#"> perror</a>	Print error message (function )

## /cstring/strlen

function

### strlen

<cstring>

```
size_t strlen ( const char * str );
```

#### Get string length

Returns the length of the C string *str*.

The length of a C string is determined by the terminating null-character: A *C string* is as long as the number of characters between the beginning of the string and the terminating null character (without including the terminating null character itself).

This should not be confused with the size of the array that holds the string. For example:

```
char mystr[100] = "test string";
```

defines an array of characters with a size of 100 chars, but the C string with which *mystr* has been initialized has a length of only 11 characters. Therefore, while

`sizeof(mystr)` evaluates to 100, `strlen(mystr)` returns 11.

In C++, `char_traits::length` implements the same behavior.

## Parameters

`str`  
C string.

## Return Value

The length of string.

## Example

```
1 /* strlen example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char szInput[256];
8     printf ("Enter a sentence: ");
9     gets (szInput);
10    printf ("The sentence entered is %u characters long.\n", (unsigned)strlen(szInput));
11    return 0;
12 }
```

Output:

```
Enter sentence: just testing
The sentence entered is 12 characters long.
```

## See also

<a href="#">strcmp</a>	Compare two strings (function )
<a href="#">strchr</a>	Locate first occurrence of character in string (function )
<a href="#"> strrchr</a>	Locate last occurrence of character in string (function )

## /cstring/strncat

function

### strncat

<cstring>

```
char * strncat ( char * destination, const char * source, size_t num );
```

#### Append characters from string

Appends the first `num` characters of `source` to `destination`, plus a terminating null-character.

If the length of the C string in `source` is less than `num`, only the content up to the terminating null-character is copied.

## Parameters

`destination`  
Pointer to the destination array, which should contain a C string, and be large enough to contain the concatenated resulting string, including the additional null-character.

`source`  
C string to be appended.

`num`  
Maximum number of characters to be appended.  
`size_t` is an unsigned integral type.

## Return Value

`destination` is returned.

## Example

```
1 /* strncat example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str1[20];
8     char str2[20];
9     strcpy (str1,"To be ");
10    strcpy (str2,"or not to be");
11    strncat (str1, str2, 6);
12    puts (str1);
13    return 0;
14 }
```

Output:

```
To be or not
```

## See also

<a href="#">strcat</a>	Concatenate strings (function )
<a href="#">strncpy</a>	Copy characters from string (function )
<a href="#">memcpy</a>	Copy block of memory (function )

## /cstring/strcmp

function

### strcmp

<cstring>

```
int strcmp ( const char * str1, const char * str2, size_t num );
```

#### Compare characters of two strings

Compares up to *num* characters of the C string *str1* to those of the C string *str2*.

This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ, until a terminating null-character is reached, or until *num* characters match in both strings, whichever happens first.

## Parameters

*str1* C string to be compared.

*str2* C string to be compared.

*num* Maximum number of characters to compare.  
*size\_t* is an unsigned integral type.

## Return Value

Returns an integral value indicating the relationship between the strings:

return value	indicates
<0	the first character that does not match has a lower value in <i>str1</i> than in <i>str2</i>
0	the contents of both strings are equal
>0	the first character that does not match has a greater value in <i>str1</i> than in <i>str2</i>

## Example

```
1 /* strcmp example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[][5] = { "R2D2" , "C3PO" , "R2A6" };
8     int n;
9     puts ("Looking for R2 astromech droids...");
10    for (n=0 ; n<3 ; n++)
11        if (strcmp (str[n],"R2xx",2) == 0)
12        {
13            printf ("found %s\n",str[n]);
14        }
15    return 0;
16 }
```

Output:

```
Looking for R2 astromech droids...
found R2D2
found R2A6
```

## See also

<a href="#">strcmp</a>	Compare two strings (function )
<a href="#">memcmp</a>	Compare two blocks of memory (function )
<a href="#">strchr</a>	Locate last occurrence of character in string (function )
<a href="#">strspn</a>	Get span of character set in string (function )

## /cstring/strncpy

function

### strncpy

<cstring>

```
char * strncpy ( char * destination, const char * source, size_t num );
```

#### Copy characters from string

Copies the first *num* characters of *source* to *destination*. If the end of the *source* C string (which is signaled by a null-character) is found before *num* characters have been copied, *destination* is padded with zeros until a total of *num* characters have been written to it.

No null-character is implicitly appended at the end of *destination* if *source* is longer than *num*. Thus, in this case, *destination* shall not be considered a null terminated C string (reading it as such would overflow).

*destination* and *source* shall not overlap (see [memmove](#) for a safer alternative when overlapping).

## Parameters

*destination*  
Pointer to the destination array where the content is to be copied.

*source*  
C string to be copied.

*num*  
Maximum number of characters to be copied from *source*.  
*size\_t* is an unsigned integral type.

## Return Value

*destination* is returned.

## Example

```
1 /* strncpy example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str1[] = "To be or not to be";
8     char str2[40];
9     char str3[40];
10
11    /* copy to sized buffer (overflow safe): */
12    strncpy ( str2, str1, sizeof(str2) );
13
14    /* partial copy (only 5 chars): */
15    strncpy ( str3, str2, 5 );
16    str3[5] = '\0'; /* null character manually added */
17
18    puts (str1);
19    puts (str2);
20    puts (str3);
21
22    return 0;
23 }
```

Output:

```
To be or not to be
To be or not to be
To be
```

## See also

<a href="#">strcpy</a>	Copy string (function )
<a href="#">memcpy</a>	Copy block of memory (function )
<a href="#">memmove</a>	Move block of memory (function )
<a href="#">memchr</a>	Locate character in block of memory (function )
<a href="#">memcmp</a>	Compare two blocks of memory (function )
<a href="#">memset</a>	Fill block of memory (function )

## /cstring/strpbrk

function

### strpbrk

<cstring>

```
const char * strpbrk ( const char * str1, const char * str2 );
    char * strpbrk (      char * str1, const char * str2 );
```

#### Locate characters in string

Returns a pointer to the first occurrence in *str1* of any of the characters that are part of *str2*, or a null pointer if there are no matches.

The search does not include the terminating null-characters of either strings, but ends there.

## Parameters

*str1*  
C string to be scanned.

*str2*  
C string containing the characters to match.

## Return Value

A pointer to the first occurrence in *str1* of any of the characters that are part of *str2*, or a null pointer if none of the characters of *str2* is found in *str1* before the terminating null-character.

If none of the characters of *str2* is present in *str1*, a null pointer is returned.

## Portability

In C, this function is only declared as:

```
char * strpbrk ( const char *, const char * );
```

instead of the two overloaded versions provided in C++.

## Example

```
1 /* strpbrk example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "This is a sample string";
8     char key[] = "aeiou";
9     char * pch;
10    printf ("Vowels in '%s': ",str);
11    pch = strpbrk (str, key);
12    while (pch != NULL)
13    {
14        printf ("%c ", *pch);
15        pch = strpbrk (pch+1,key);
16    }
17    printf ("\n");
18    return 0;
19 }
```

Output:

```
Vowels in 'This is a sample string': i i a a e i
```

## See also

<a href="#">strcspn</a>	Get span until character in string ( <a href="#">function</a> )
<a href="#">strchr</a>	Locate first occurrence of character in string ( <a href="#">function</a> )
<a href="#">strrchr</a>	Locate last occurrence of character in string ( <a href="#">function</a> )
<a href="#">memchr</a>	Locate character in block of memory ( <a href="#">function</a> )

# /cstring/strrchr

function

## strrchr

<cstring>

```
const char * strrchr ( const char * str, int character );
char * strrchr ( char * str, int character );
```

### Locate last occurrence of character in string

Returns a pointer to the last occurrence of *character* in the C string *str*.

The terminating null-character is considered part of the C string. Therefore, it can also be located to retrieve a pointer to the end of a string.

## Parameters

*str*  
C string.

*character*  
Character to be located. It is passed as its int promotion, but it is internally converted back to *char*.

## Return Value

A pointer to the last occurrence of *character* in *str*.

If the *character* is not found, the function returns a null pointer.

## Portability

In C, this function is only declared as:

```
char * strrchr ( const char *, int );
```

instead of the two overloaded versions provided in C++.

## Example

```
1 /* strrchr example */
2 #include <stdio.h>
3 #include <string.h>
```

```

4
5 int main ()
6 {
7     char str[] = "This is a sample string";
8     char * pch;
9     pch=strrchr(str,'s');
10    printf ("Last occurrence of 's' found at %d \n",pch-str+1);
11    return 0;
12 }

```

Output:

Last occurrence of 's' found at 18

### See also

<a href="#">strchr</a>	Locate first occurrence of character in string (function )
<a href="#">memchr</a>	Locate character in block of memory (function )
<a href="#">strpbrk</a>	Locate characters in string (function )

## /cstring/strspn

function  
**strspn** <cstring>

`size_t strspn ( const char * str1, const char * str2 );`

### Get span of character set in string

Returns the length of the initial portion of *str1* which consists only of characters that are part of *str2*.

The search does not include the terminating null-characters of either strings, but ends there.

### Parameters

*str1*  
C string to be scanned.

*str2*  
C string containing the characters to match.

### Return value

The length of the initial portion of *str1* containing only characters that appear in *str2*.

Therefore, if all of the characters in *str1* are in *str2*, the function returns the length of the entire *str1* string, and if the first character in *str1* is not in *str2*, the function returns zero.

`size_t` is an unsigned integral type.

### Example

```

1 /* strspn example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     int i;
8     char strtext[] = "129th";
9     char cset[] = "1234567890";
10
11    i = strspn (strtext,cset);
12    printf ("The initial number has %d digits.\n",i);
13    return 0;
14 }
15

```

Output:

The initial number has 3 digits.

### See also

<a href="#">strcspn</a>	Get span until character in string (function )
<a href="#">strstr</a>	Locate substring (function )
<a href="#">strcmp</a>	Compare characters of two strings (function )

## /cstring/strstr

function  
**strstr** <cstring>

`const char * strstr ( const char * str1, const char * str2 );`  
`char * strstr ( char * str1, const char * str2 );`

## Locate substring

Returns a pointer to the first occurrence of *str2* in *str1*, or a null pointer if *str2* is not part of *str1*.

The matching process does not include the terminating null-characters, but it stops there.

### Parameters

*str1*  
C string to be scanned.  
*str2*  
C string containing the sequence of characters to match.

### Return Value

A pointer to the first occurrence in *str1* of the entire sequence of characters specified in *str2*, or a null pointer if the sequence is not present in *str1*.

### Portability

In C, this function is only declared as:

```
char * strstr ( const char *, const char * );
```

instead of the two overloaded versions provided in C++.

### Example

```
1 /* strstr example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] = "This is a simple string";
8     char * pch;
9     pch = strstr (str, "simple");
10    strncpy (pch, "sample", 6);
11    puts (str);
12    return 0;
13 }
```

This example searches for the "simple" substring in *str* and replaces that word for "sample".

Output:

```
This is a sample string
```

### See also

<a href="#">strspn</a>	Get span of character set in string ( <a href="#">function</a> )
<a href="#">strupr</a>	Locate characters in string ( <a href="#">function</a> )
<a href="#">strchr</a>	Locate first occurrence of character in string ( <a href="#">function</a> )

## /cstring/strtok

function

### strtok

<cstring>

```
char * strtok ( char * str, const char * delimiters );
```

#### Split string into tokens

A sequence of calls to this function split *str* into tokens, which are sequences of contiguous characters separated by any of the characters that are part of *delimiters*.

On a first call, the function expects a C string as argument for *str*, whose first character is used as the starting location to scan for tokens. In subsequent calls, the function expects a null pointer and uses the position right after the end of the last token as the new starting location for scanning.

To determine the beginning and the end of a token, the function first scans from the starting location for the first character **not** contained in *delimiters* (which becomes the *beginning of the token*). And then scans starting from this *beginning of the token* for the first character contained in *delimiters*, which becomes the *end of the token*. The scan also stops if the terminating *null character* is found.

This *end of the token* is automatically replaced by a null-character, and the *beginning of the token* is returned by the function.

Once the terminating null character of *str* is found in a call to *strtok*, all subsequent calls to this function (with a null pointer as the first argument) return a null pointer.

The point where the last token was found is kept internally by the function to be used on the next call (particular library implementations are not required to avoid data races).

### Parameters

*str*  
C string to truncate.  
Notice that this string is modified by being broken into smaller strings (tokens).  
Alternatively, a null pointer may be specified, in which case the function continues scanning where a previous successful call to the function ended.  
*delimiters*

C string containing the delimiter characters.  
These can be different from one call to another.

### Return Value

If a token is found, a pointer to the beginning of the token.

Otherwise, a *null pointer*.

A *null pointer* is always returned when the end of the string (i.e., a null character) is reached in the string being scanned.

### Example

```
1 /* strtok example */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main ()
6 {
7     char str[] ="- This, a sample string.";
8     char * pch;
9     printf ("Splitting string \"%s\" into tokens:\n",str);
10    pch = strtok (str," .-");
11    while (pch != NULL)
12    {
13        printf ("%s\n",pch);
14        pch = strtok (NULL, " .-");
15    }
16    return 0;
17 }
```

Output:

```
Splitting string "- This, a sample string." into tokens:
This
a
sample
string
```

### See also

<a href="#">strcspn</a>	Get span until character in string (function )
<a href="#">strpbrk</a>	Locate characters in string (function )

## /cstring/strxfrm

function  
**strxfrm** <cstring>

```
size_t strxfrm ( char * destination, const char * source, size_t num );
```

### Transform string using locale

Transforms the C string pointed by *source* according to the current locale and copies the first *num* characters of the transformed string to *destination*, returning its length.

Alternatively, the function can be used to only retrieve the length, by specifying a null pointer for *destination* and zero for *num*.

*destination* and *source* shall not overlap.

The behavior of this function depends on the `LC_COLLATE` category of the selected C locale.

### Parameters

**destination**  
Pointer to the destination array where the content is to be copied.  
It can be a null pointer if the argument for *num* is zero.

**source**  
C string to be transformed.

**num**  
Maximum number of characters to be copied to *destination*.  
`size_t` is an unsigned integral type.

### Return Value

The length of the transformed string, not including the terminating null-character.  
`size_t` is an unsigned integral type.

### See also

<a href="#">strncpy</a>	Copy characters from string (function )
<a href="#">strcmp</a>	Compare characters of two strings (function )
<a href="#">strcoll</a>	Compare two strings using locale (function )

## /deque

header

## <deque>

### Deque header

Header that defines the `deque` container class:

### Classes

<code>deque</code>	Double ended queue ( <a href="#">class template</a> )
--------------------	---

### Functions

<code>begin</code>	Iterator to beginning ( <a href="#">function template</a> )
<code>end</code>	Iterator to end ( <a href="#">function template</a> )

## /deque/deque

class template

### `std::deque`

<deque>

`template < class T, class Alloc = allocator<T> > class deque;`

#### Double ended queue

`deque` (usually pronounced like "deck") is an irregular acronym of **d**ouble-**e**nded **q**ueue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).

Specific libraries may implement `deques` in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

Therefore, they provide a functionality similar to `vectors`, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike `vectors`, `deques` are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a `deque` by offsetting a pointer to another element causes *undefined behavior*.

Both `vectors` and `deques` provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While `vectors` use a single array that needs to be occasionally reallocated for growth, the elements of a `deque` can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, `deques` are a little more complex internally than `vectors`, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, `deques` perform worse and have less consistent iterators and references than `lists` and `forward lists`.

### Container properties

#### Sequence

Elements in sequence containers are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

#### Dynamic array

Generally implemented as a dynamic array, it allows direct access to any element in the sequence and provides relatively fast addition/removal of elements at the beginning or the end of the sequence.

#### Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

### Template parameters

T

Type of the elements.

Aliased as member type `deque::value_type`.

Alloc

Type of the allocator object used to define the storage allocation model. By default, the `allocator` class template is used, which defines the simplest memory allocation model and is value-independent.

Aliased as member type `deque::allocator_type`.

### Member types

member type	definition	notes
<code>value_type</code>	The first template parameter (T)	
<code>allocator_type</code>	The second template parameter (Alloc)	defaults to: <code>allocator&lt;value_type&gt;</code>
<code>reference</code>	<code>allocator_type::reference</code>	for the default allocator: <code>value_type&amp;</code>
<code>const_reference</code>	<code>allocator_type::const_reference</code>	for the default allocator: <code>const value_type&amp;</code>
<code>pointer</code>	<code>allocator_type::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_type::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a random access iterator to <code>value_type</code>	convertible to <code>const_iterator</code>
<code>const_iterator</code>	a random access iterator to <code>const value_type</code>	
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>
member type	definition	notes

<code>value_type</code>	The first template parameter ( <code>T</code> )	
<code>allocator_type</code>	The second template parameter ( <code>Alloc</code> )	defaults to: <code>allocator&lt;value_type&gt;</code>
<code>reference</code>	<code>value_type&amp;</code>	
<code>const_reference</code>	<code>const value_type&amp;</code>	
<code>pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a random access iterator to <code>value_type</code>	convertible to <code>const_iterator</code>
<code>const_iterator</code>	a random access iterator to <code>const value_type</code>	
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>

## Member functions

<b>(constructor)</b>	Construct deque container (public member function )
<b>(destructor)</b>	Deque destructor (public member function )
<b>operator=</b>	Assign content (public member function )

### Iterators:

<code>begin</code>	Return iterator to beginning (public member function )
<code>end</code>	Return iterator to end (public member function )
<code>rbegin</code>	Return reverse iterator to reverse beginning (public member function )
<code>rend</code>	Return reverse iterator to reverse end (public member function )
<code>cbegin</code>	Return const_iterator to beginning (public member function )
<code>cend</code>	Return const_iterator to end (public member function )
<code>crbegin</code>	Return const_reverse_iterator to reverse beginning (public member function )
<code>crend</code>	Return const_reverse_iterator to reverse end (public member function )

### Capacity:

<code>size</code>	Return size (public member function )
<code>max_size</code>	Return maximum size (public member function )
<code>resize</code>	Change size (public member function )
<code>empty</code>	Test whether container is empty (public member function )
<code>shrink_to_fit</code>	Shrink to fit (public member function )

### Element access:

<b>operator[]</b>	Access element (public member function )
<b>at</b>	Access element (public member function )
<b>front</b>	Access first element (public member function )
<b>back</b>	Access last element (public member function )

### Modifiers:

<code>assign</code>	Assign container content (public member function )
<code>push_back</code>	Add element at the end (public member function )
<code>push_front</code>	Insert element at beginning (public member function )
<code>pop_back</code>	Delete last element (public member function )
<code>pop_front</code>	Delete first element (public member function )
<code>insert</code>	Insert elements (public member function )
<code>erase</code>	Erase elements (public member function )
<code>swap</code>	Swap content (public member function )
<code>clear</code>	Clear content (public member function )
<code>emplace</code>	Construct and insert element (public member function )
<code>emplace_front</code>	Construct and insert element at beginning (public member function )
<code>emplace_back</code>	Construct and insert element at the end (public member function )

### Allocator:

<code>get_allocator</code>	Get allocator (public member function )
----------------------------	---

## Non-member functions overloads

<b>relational operators</b>	Relational operators for deque (function )
<b>swap</b>	Exchanges the contents of two deque containers (function template )

## /deque/deque/assign

public member function

**std::deque::assign**

<deque>

```

range (1) template <class InputIterator>
    void assign (InputIterator first, InputIterator last);
fill (2) void assign (size_type n, const value_type& val);

range (1) template <class InputIterator>
    void assign (InputIterator first, InputIterator last);
fill (2) void assign (size_type n, const value_type& val);
initializer list (3) void assign (initializer_list<value_type> il);

```

### Assign container content

Assigns new contents to the `deque` container, replacing its current contents, and modifying its `size` accordingly.

In the *range version* (1), the new contents are elements constructed from each of the elements in the range between `first` and `last`, in the same order.

In the *fill version* (2), the new contents are  $n$  elements, each initialized to a copy of `val`.

If needed, the container uses its [internal allocator](#) to allocate additional storage.

In the *range version* (1), the new contents are elements constructed from each of the elements in the range between `first` and `last`, in the same order.

In the *fill version* (2), the new contents are  $n$  elements, each initialized to a copy of `val`.

In the *initializer list version* (3), the new contents are copies of the values passed as initializer list, in the same order.

If there are changes in storage, the [internal allocator](#) is used (through its [traits](#)). It is also used to [destroy](#) all existing elements, and to [construct](#) the new ones.

Any elements held in the container before the call are *destroyed* and replaced by newly constructed elements (no assignments of elements take place).

### Parameters

`first, last`

Input iterators to the initial and final positions in a sequence. The range used is `[first, last)`, which includes all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

The function template argument `InputIterator` shall be an [input iterator](#) type that points to elements of a type from which `value_type` objects can be constructed.

`n`

New size for the container.

Member type `size_type` is an unsigned integral type.

`val`

Value to fill the container with. Each of the  $n$  elements in the container will be initialized to a copy of this value.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter ( $\mathbb{T}$ ).

`il`

An [initializer\\_list](#) object. The compiler will automatically construct such objects from *initializer list* declarators.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter ( $\mathbb{T}$ ).

### Return value

none

### Example

```

1 // deque::assign
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> first;
8     std::deque<int> second;
9     std::deque<int> third;
10
11    first.assign (7,100);           // 7 ints with a value of 100
12
13    std::deque<int>::iterator it;
14    it=first.begin()+1;
15
16    second.assign (it,first.end()-1); // the 5 central values of first
17
18    int myints[] = {1776,7,4};
19    third.assign (myints,myints+3);   // assigning from array.
20
21    std::cout << "Size of first: " << int (first.size()) << '\n';
22    std::cout << "Size of second: " << int (second.size()) << '\n';
23    std::cout << "Size of third: " << int (third.size()) << '\n';
24
25 }

```

Output:

```

Size of first: 7
Size of second: 5
Size of third: 3

```

### Complexity

Linear in initial and final `sizes` (destructions, constructions).

### Iterator validity

All iterators, pointers and references related to this container are invalidated.

## Data races

All copied elements are accessed.  
The container is modified.  
All contained elements are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.  
If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

## See also

<code>deque::operator=</code>	Assign content (public member function )
<code>deque::resize</code>	Change size (public member function )

# /deque/deque/at

public member function

## std::deque::at

<deque>

`reference at (size_type n);`  
`const_reference at (size_type n) const;`

### Access element

Returns a reference to the element at position *n* in the `deque` container object.

The function automatically checks whether *n* is within the bounds of valid elements in the container, throwing an `out_of_range` exception if it is not (i.e., if *n* is greater or equal than its `size`). This is in contrast with member `operator[]`, that does not check against bounds.

Returns a reference to the element at position *n* in the `deque` container object.

The difference between this member function and member operator function `operator[]` is that `deque::at` signals if the requested position is out of range by throwing an `out_of_range` exception.

## Parameters

*n*  
Position of an element in the container.  
If this is greater than the `deque size`, an exception of type `out_of_range` is thrown.  
Notice that the first element has a position of 0 (not 1).  
Member type `size_type` is an unsigned integral type.

## Return value

The element at the specified position in the container.

If the `deque` object is `const`-qualified, the function returns a `const_reference`. Otherwise, it returns a `reference`.

Member types `reference` and `const_reference` are the reference types to the elements of the container (see `deque` member types).

## Example

```
1 // deque::at
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<unsigned> mydeque (10);    // 10 zero-initialized unsigneds
8
9     // assign some values:
10    for (unsigned i=0; i<mydeque.size(); i++)
11        mydeque.at(i)=i;
12
13    std::cout << "mydeque contains:";
14    for (unsigned i=0; i<mydeque.size(); i++)
15        std::cout << ' ' << mydeque.at(i);
16
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
mydeque contains: 0 1 2 3 4 5 6 7 8 9
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
Element  $n$  is potentially accessed or modified by the caller. Concurrently accessing or modifying other elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
It throws `out_of_range` if  $n$  is out of bounds.

## See also

<code>deque::operator[]</code>	Access element (public member function )
<code>deque::front</code>	Access first element (public member function )
<code>deque::back</code>	Access last element (public member function )

# /deque/deque/back

public member function

## std::deque::back

<deque>

`reference back();`  
`const_reference back() const;`

### Access last element

Returns a reference to the last element in the container.

Unlike member `deque::end`, which returns an iterator just past this element, this function returns a direct reference.

Calling this function on an `empty` container causes undefined behavior.

## Parameters

none

## Return value

A reference to the last element in the `deque` container.

If the `deque` object is `const`-qualified, the function returns a `const_reference`. Otherwise, it returns a `reference`.

Member types `reference` and `const_reference` are the reference types to the elements of the container (see `deque` member types).

## Example

```
1 // deque::back
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
8
9     mydeque.push_back(10);
10
11    while (mydeque.back() != 0)
12        mydeque.push_back ( mydeque.back() -1 );
13
14    std::cout << "mydeque contains:" ;
15
16    for (std::deque<int>::iterator it = mydeque.begin(); it!=mydeque.end(); ++it)
17        std::cout << ' ' << *it;
18
19    std::cout << '\n' ;
20
21    return 0;
22 }
```

Output:

```
mydeque contains: 10 9 8 7 6 5 4 3 2 1 0
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
The last element is potentially accessed or modified by the caller. Concurrently accessing or modifying other elements is safe.

## Exception safety

If the container is not [empty](#), the function never throws exceptions (no-throw guarantee). Otherwise, it causes [undefined behavior](#).

## See also

<a href="#">deque::front</a>	Access first element ( <a href="#">public member function</a> )
<a href="#">deque::end</a>	Return iterator to end ( <a href="#">public member function</a> )
<a href="#">deque::push_back</a>	Add element at the end ( <a href="#">public member function</a> )
<a href="#">deque::pop_back</a>	Delete last element ( <a href="#">public member function</a> )

# /deque/deque/begin

public member function

## std::deque::begin

<deque>

```
    iterator begin();
const_iterator begin() const;
    iterator begin() noexcept;
const_iterator begin() const noexcept;
```

### Return iterator to beginning

Returns an iterator pointing to the first element in the [deque](#) container.

Notice that, unlike member [deque::front](#), which returns a reference to the first element, this function returns a [random access iterator](#) pointing to it.

If the container is [empty](#), the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

An iterator to the beginning of the sequence container.

If the [deque](#) object is const-qualified, the function returns a [const\\_iterator](#). Otherwise, it returns an [iterator](#).

Member types [iterator](#) and [const\\_iterator](#) are [random access iterator](#) types (pointing to an element and to a const element, respectively).

## Example

```
1 // deque::begin
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
8
9     for (int i=1; i<=5; i++) mydeque.push_back(i);
10
11    std::cout << "mydeque contains:" ;
12
13    std::deque<int>::iterator it = mydeque.begin();
14
15    while (it != mydeque.end())
16        std::cout << ' ' << *it++;
17
18    std::cout << '\n';
19
20    return 0;
21 }
```

Output:

```
mydeque contains: 1 2 3 4 5
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">deque::front</a>	Access first element ( <a href="#">public member function</a> )
<a href="#">deque::end</a>	Return iterator to end ( <a href="#">public member function</a> )
<a href="#">deque::rbegin</a>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<a href="#">deque::rend</a>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )

# /deque/deque/cbegin

public member function

## std::deque::cbegin

<deque>

`const_iterator cbegin() const noexcept;`

**Return const\_iterator to beginning**

Returns a `const_iterator` pointing to the first element in the container.

A `const_iterator` is an iterator that points to `const` content. This iterator can be increased and decreased (unless it is itself `const`), just like the iterator returned by `deque::begin`, but it cannot be used to modify the contents it points to, even if the `deque` object is not itself `const`.

If the container is `empty`, the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

A `const_iterator` to the beginning of the sequence.

Member type `const_iterator` is a [random access iterator](#) type that points to a `const` element.

## Example

```
1 // deque::cbegin/cend
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque = {10,20,30,40,50};
8
9     std::cout << "mydeque contains:";
10
11    for (auto it = mydeque.cbegin(); it != mydeque.cend(); ++it)
12        std::cout << ' ' << *it;
13
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

```
mydeque contains: 10 20 30 40 50
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">deque::begin</a>	Return iterator to beginning ( <a href="#">public member function</a> )
<a href="#">deque::cend</a>	Return <code>const_iterator</code> to end ( <a href="#">public member function</a> )
<a href="#">deque::crbegin</a>	Return <code>const_reverse_iterator</code> to reverse beginning ( <a href="#">public member function</a> )

# /deque/deque/cend

public member function  
**std::deque::cend** <deque>  
const\_iterator cend() const noexcept;

#### Return const\_iterator to end

Returns a const\_iterator pointing to the *past-the-end* element in the container.

A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by `deque::end`, but it cannot be used to modify the contents it points to, even if the `deque` object is not itself const.

If the container is `empty`, this function returns the same as `deque::cbegin`.

The value returned shall not be dereferenced.

#### Parameters

none

#### Return Value

A const\_iterator to the element past the end of the sequence.

Member type `const_iterator` is a `random access iterator` type that points to a const element.

#### Example

```
1 // deque::cbegin/cend
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque = {10,20,30,40,50};
8
9     std::cout << "mydeque contains:";
10
11    for (auto it = mydeque.cbegin(); it != mydeque.cend(); ++it)
12        std::cout << ' ' << *it;
13
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

```
mydeque contains: 10 20 30 40 50
```

#### Complexity

Constant.

#### Iterator validity

No changes.

#### Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

#### See also

<code>deque::end</code>	Return iterator to end ( <a href="#">public member function</a> )
-------------------------	---

<code>deque::cbegin</code>	Return const_iterator to beginning ( <a href="#">public member function</a> )
----------------------------	---

## /deque/deque/clear

public member function

### **std::deque::clear**

<deque>

<code>void clear();</code>
----------------------------

<code>void clear() noexcept;</code>
-------------------------------------

#### Clear content

Removes all elements from the `deque` (which are destroyed), leaving the container with a `size` of 0.

#### Parameters

none

## Return value

none

## Example

```
1 // clearing deques
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     unsigned int i;
8     std::deque<int> mydeque;
9     mydeque.push_back (100);
10    mydeque.push_back (200);
11    mydeque.push_back (300);
12
13    std::cout << "mydeque contains:";
14    for (std::deque<int>::iterator it = mydeque.begin(); it!=mydeque.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    mydeque.clear();
19    mydeque.push_back (1101);
20    mydeque.push_back (2202);
21
22    std::cout << "mydeque contains:";
23    for (std::deque<int>::iterator it = mydeque.begin(); it!=mydeque.end(); ++it)
24        std::cout << ' ' << *it;
25    std::cout << '\n';
26
27    return 0;
28 }
```

Output:

```
mydeque contains: 100 200 300
mydeque contains: 1101 2202
```

## Complexity

Linear in `size` (destructions).

## Iterator validity

All iterators, pointers and references related to this container are invalidated.

## Data races

The container is modified.

All contained elements are modified.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>deque::erase</code>	Erase elements ( <a href="#">public member function</a> )
<code>deque::resize</code>	Change size ( <a href="#">public member function</a> )
<code>deque::pop_back</code>	Delete last element ( <a href="#">public member function</a> )
<code>deque::pop_front</code>	Delete first element ( <a href="#">public member function</a> )
<code>deque::empty</code>	Test whether container is empty ( <a href="#">public member function</a> )

## /deque/deque/crbegin

public member function

### std::deque::crbegin

<deque>

`const_reverse_iterator crbegin() const noexcept;`

#### Return `const_reverse_iterator` to reverse beginning

Returns a `const_reverse_iterator` pointing to the last element in the container (i.e., its *reverse beginning*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse beginning* of the sequence.

Member type `const_reverse_iterator` is a reverse [random access iterator](#) type that points to a `const` element (see [deque member types](#)).

## Example

```
1 // deque::crbegin/crend
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque = {1,2,3,4,5};
8
9     std::cout << "mydeque backwards:";
10    for (auto rit = mydeque.crbegin(); rit != mydeque.crend(); ++rit)
11        std::cout << ' ' << *rit;
12    std::cout << '\n';
13
14    return 0;
15 }
```

Output:

```
mydeque backwards: 5 4 3 2 1
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<b>deque::begin</b>	Return iterator to beginning ( <a href="#">public member function</a> )
<b>deque::crend</b>	Return const_reverse_iterator to reverse end ( <a href="#">public member function</a> )
<b>deque::rbegin</b>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )

# /deque/deque/crend

public member function

## std::deque::crend

<deque>

```
const_reverse_iterator crend() const noexcept;
```

### Return const\_reverse\_iterator to reverse end

Returns a const\_reverse\_iterator pointing to the theoretical element preceding the first element in the container (which is considered its *reverse end*).

## Parameters

none

## Return Value

A const\_reverse\_iterator to the *reverse end* of the sequence.

Member type const\_reverse\_iterator is a reverse [random access iterator](#) type that points to a const element (see [deque member types](#)).

## Example

```
1 // deque::crbegin/crend
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque = {1,2,3,4,5};
8
9     std::cout << "mydeque backwards:";
10    for (auto rit = mydeque.crbegin(); rit != mydeque.crend(); ++rit)
11        std::cout << ' ' << *rit;
12    std::cout << '\n';
13
14    return 0;
15 }
```

Output:

```
mydeque backwards: 5 4 3 2 1
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<b>deque::end</b>	Return iterator to end (public member function )
<b>deque::crbegin</b>	Return const_reverse_iterator to reverse beginning (public member function )
<b>deque::rend</b>	Return reverse iterator to reverse end (public member function )

# /deque/deque/deque

public member function

## std::deque::deque

<deque>

```
default (1) explicit deque (const allocator_type& alloc = allocator_type());
fill (2)     explicit deque (size_type n, const value_type& val = value_type(),
                           const allocator_type& alloc = allocator_type());
template <class InputIterator>
range (3)    deque (InputIterator first, InputIterator last,
                   const allocator_type& alloc = allocator_type());
copy (4)     deque (const deque& x);

default (1) explicit deque (const allocator_type& alloc = allocator_type());
fill (2)     explicit deque (size_type n);
deque (size_type n, const value_type& val,
      const allocator_type& alloc = allocator_type());
range (3)    deque (InputIterator first, InputIterator last,
                   const allocator_type& alloc = allocator_type());
copy (4)     deque (const deque& x);
deque (const deque& x, const allocator_type& alloc);
move (5)     deque (deque&& x);
deque (deque&& x, const allocator_type& alloc);
initializer list (6) deque (initializer_list<value_type> il,
                           const allocator_type& alloc = allocator_type());

default (1) deque();
explicit deque (const allocator_type& alloc);
explicit deque (size_type n, const allocator_type& alloc = allocator_type());
fill (2)     deque (size_type n, const value_type& val,
                   const allocator_type& alloc = allocator_type());
template <class InputIterator>
range (3)    deque (InputIterator first, InputIterator last,
                   const allocator_type& alloc = allocator_type());
copy (4)     deque (const deque& x);
deque (const deque& x, const allocator_type& alloc);
move (5)     deque (deque&& x);
deque (deque&& x, const allocator_type& alloc);
initializer list (6) deque (initializer_list<value_type> il,
                           const allocator_type& alloc = allocator_type());
```

## Construct deque container

Constructs a `deque` container object, initializing its contents depending on the constructor version used:

### (1) empty container constructor (default constructor)

Constructs an `empty` container, with no elements.

### (2) fill constructor

Constructs a container with  $n$  elements. Each element is a copy if  $val$ .

### (3) range constructor

Constructs a container with as many elements as the range  $[first, last]$ , with each element constructed from its corresponding element in that range, in the same order.

### (4) copy constructor

Constructs a container with a copy of each of the elements in  $x$ , in the same order.

The container keeps an internal copy of `alloc`, which is used to allocate storage throughout its lifetime. If no `alloc` argument is passed to the constructor, a default-constructed allocator is used, except in the following case:

- The copy constructor (4) creates a container that keeps and uses a copy of  $x$ 's allocator.

The storage for the elements is allocated using this `internal allocator`.

**(1) empty container constructor (default constructor)**

Constructs an `empty` container, with no elements.

**(2) fill constructor**

Constructs a container with  $n$  elements. Each element is a copy of `val` (if provided).

**(3) range constructor**

Constructs a container with as many elements as the range `[first, last]`, with each element *emplace-constructed* from its corresponding element in that range, in the same order.

**(4) copy constructor (and copying with allocator)**

Constructs a container with a copy of each of the elements in `x`, in the same order.

**(5) move constructor (and moving with allocator)**

Constructs a container that acquires the elements of `x`.

If `alloc` is specified and is different from `x`'s allocator, the elements are moved. Otherwise, no elements are constructed (their ownership is directly transferred).

`x` is left in an unspecified but valid state.

**(6) initializer list constructor**

Constructs a container with a copy of each of the elements in `il`, in the same order.

The container keeps an internal copy of `alloc`, which is used to allocate and deallocate storage for its elements, and to construct and destroy them (as specified by its `allocator_traits`). If no `alloc` argument is passed to the constructor, a default-constructed allocator is used, except in the following cases:

- The copy constructor (4, *first signature*) creates a container that keeps and uses a copy of the allocator returned by calling the appropriate `selected_on_container_copy_construction` trait on `x`'s allocator.

- The move constructor (5, *first signature*) acquires `x`'s allocator.

All elements are *copied*, *moved* or otherwise *constructed* by calling `allocator_traits::construct` with the appropriate arguments.

## Parameters

`alloc`

Allocator object.

The container keeps and uses an internal copy of this allocator.

Member type `allocator_type` is the internal allocator type used by the container, defined in `deque` as an alias of its second template parameter (`Alloc`). If `allocator_type` is an instantiation of the default `allocator` (which has no state), this is not relevant.

`n`

Initial container size (i.e., the number of elements in the container at construction).

Member type `size_type` is an unsigned integral type.

`val`

Value to fill the container with. Each of the  $n$  elements in the container will be initialized to a copy of this value.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter (`T`).

`first, last`

`Input iterators` to the initial and final positions in a range. The range used is `[first, last)`, which includes all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type from which `value_type` objects can be constructed.

`x`

Another `deque` object of the same type (with the same class template arguments `T` and `Alloc`), whose contents are either copied or acquired.

`il`

An `initializer_list` object.

These objects are automatically constructed from `initializer_list` declarators.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter (`T`).

## Example

```

1 // constructing deques
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     unsigned int i;
8
9     // constructors used in the same order as described above:
10    std::deque<int> first;                                // empty deque of ints
11    std::deque<int> second (4,100);                         // four ints with value 100
12    std::deque<int> third (second.begin(),second.end());   // iterating through second
13    std::deque<int> fourth (third);                          // a copy of third
14
15    // the iterator constructor can be used to copy arrays:
16    int myints[] = {16,2,77,29};
17    std::deque<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
18
19    std::cout << "The contents of fifth are:" ;
20    for (std::deque<int>::iterator it = fifth.begin(); it!=fifth.end(); ++it)
21        std::cout << ' ' << *it;
22
23    std::cout << '\n';
24
25    return 0;
26 }
```

Output:

The contents of fifth are: 16 2 77 29

## Complexity

Constant for the *default constructor* (1), and for the *move constructors* (5) (unless `alloc` is different from `x`'s allocator).  
For all other cases, linear in the resulting container `size`.

## Iterator validity

The move constructors (5), invalidate all iterators, pointers and references related to *x* if the elements are moved.

## Data races

All copied elements are accessed.

The move constructors (5) modify *x*.

## Exception safety

**Strong guarantee:** no effects in case an exception is thrown.

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

## See also

<code>deque::operator=</code>	Assign content (public member function )
<code>deque::assign</code>	Assign container content (public member function )
<code>deque::resize</code>	Change size (public member function )
<code>deque::clear</code>	Clear content (public member function )

# /deque/deque/~deque

public member function

## std::deque::~deque

<deque>

`~deque();`

### Deque destructor

Destroys the container object.

This destroys all container elements, and deallocates all the storage capacity allocated by the `deque` container using its `allocator`.

This calls `allocator_traits::destroy` on each of the contained elements, and deallocates all the storage capacity allocated by the `deque` container using its `allocator`.

## Complexity

Linear in `deque::size` (destructors).

## Iterator validity

All iterators, pointers and references are invalidated.

## Data races

The container and all its elements are modified.

## Exception safety

**No-throw guarantee:** never throws exceptions.

# /deque/deque/emplace

public member function

## std::deque::emplace

<deque>

```
template <class... Args>
iterator emplace (const_iterator position, Args&&... args);
```

### Construct and insert element

The container is extended by inserting a new element at *position*. This new element is constructed in place using *args* as the arguments for its construction.

This effectively increases the container `size` by one.

Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in `list` or `forward_list` containers. See `emplace_front` and `emplace_back` for member functions that extend the container directly at the beginning or at the end.

The element is constructed in-place by calling `allocator_traits::construct` with *args* forwarded.

## Parameters

### position

Position in the container where the new element is inserted.

Member type `const_iterator` is a `random access iterator` type that points to a constant element.

### args

Arguments forwarded to construct the new element.

## Return value

An iterator that points to the newly emplaced element.

Member type `iterator` is a [random access iterator](#) type that points to an element.

The storage for the new element is allocated using `allocator_traits<allocator_type>::construct()`, which may throw exceptions on failure (for the default allocator, `bad_alloc` is thrown if the allocation request does not succeed).

## Example

```
1 // deque::emplace
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque = {10,20,30};
8
9     auto it = mydeque.emplace ( mydeque.begin() + 1, 100 );
10    mydeque.emplace ( it, 200 );
11    mydeque.emplace ( mydeque.end(), 300 );
12
13    std::cout << "mydeque contains:";
14    for (auto& x: mydeque)
15        std::cout << ' ' << x;
16    std::cout << '\n';
17
18    return 0;
19 }
```

Output:

```
mydeque contains: 10 200 100 20 30 300
```

## Complexity

Depending on the particular library implementation, up to linear in the number of elements between `position` and one of the ends of the `deque`.

## Iterator validity

If the insertion happens at the beginning or the end of the sequence, all iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.

If the insertion happens anywhere else in the `deque`, all iterators, pointers and references related to this container are invalidated.

## Data races

The container is modified.

If the insertion happens at the beginning or the end of the sequence, no contained elements are accessed (although see *iterator validity* above).

If it happens anywhere else, it is not safe to concurrently access elements.

## Exception safety

If `position` is `begin` or `end`, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If `allocator_traits::construct` is not supported with the appropriate arguments, or if `position` is not valid, it causes *undefined behavior*.

## See also

<code>deque::emplace_front</code>	Construct and insert element at beginning ( <a href="#">public member function</a> )
<code>deque::emplace_back</code>	Construct and insert element at the end ( <a href="#">public member function</a> )
<code>deque::insert</code>	Insert elements ( <a href="#">public member function</a> )
<code>deque::erase</code>	Erase elements ( <a href="#">public member function</a> )
<code>deque::assign</code>	Assign container content ( <a href="#">public member function</a> )

# /deque/deque/emplace\_back

public member function

## std::deque::emplace\_back

<deque>

```
template <class... Args>
void emplace_back (Args&... args);
```

### Construct and insert element at the end

Inserts a new element at the end of the `deque`, right after its current last element. This new element is constructed in place using `args` as the arguments for its construction.

This effectively increases the container `size` by one.

The element is constructed in-place by calling `allocator_traits::construct` with `args` forwarded.

A similar member function exists, `push_back`, which either copies or moves an existing object into the container.

## Parameters

`args`

Arguments forwarded to construct the new element.

## Return value

none.

The storage for the new element is allocated using `allocator_traits<allocator_type>::construct()`, which may throw exceptions on failure (for the default allocator, `bad_alloc` is thrown if the allocation request does not succeed).

## Example

```
1 // deque::emplace_from
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque = {10,20,30};
8
9     mydeque.emplace_back (100);
10    mydeque.emplace_back (200);
11
12    std::cout << "mydeque contains:";
13    for (auto& x: mydeque)
14        std::cout << ' ' << x;
15    std::cout << '\n';
16
17    return 0;
18 }
```

Output:

```
mydeque contains: 10 20 30 100 200
```

## Complexity

Constant.

## Iterator validity

All iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.

## Data races

The container is modified.

No contained elements are accessed by the call: concurrently accessing or modifying them is safe (although see *iterator validity* above).

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<code>deque::emplace_front</code>	Construct and insert element at beginning ( <a href="#">public member function</a> )
<code>deque::push_back</code>	Add element at the end ( <a href="#">public member function</a> )
<code>deque::emplace</code>	Construct and insert element ( <a href="#">public member function</a> )
<code>deque::erase</code>	Erase elements ( <a href="#">public member function</a> )
<code>deque::assign</code>	Assign container content ( <a href="#">public member function</a> )

# /deque/deque/emplace\_front

public member function

## std::deque::emplace\_front

<deque>

```
template <class... Args>
void emplace_front (Args&&... args);
```

### Construct and insert element at beginning

Inserts a new element at the beginning of the `deque`, right before its current first element. This new element is constructed in place using `args` as the arguments for its construction.

This effectively increases the container `size` by one.

The element is constructed in-place by calling `allocator_traits::construct` with `args` forwarded.

A similar member function exists, `push_front`, which either copies or moves an existing object into the container.

## Parameters

`args`

Arguments forwarded to construct the new element.

## Return value

none.

The storage for the new element is allocated using `allocator_traits<allocator_type>::construct()`, which may throw exceptions on failure (for the default `allocator`, `bad_alloc` is thrown if the allocation request does not succeed).

## Example

```
1 // deque::emplace_from
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque = {10,20,30};
8
9     mydeque.emplace_front (111);
10    mydeque.emplace_front (222);
11
12    std::cout << "mydeque contains:";
13    for (auto& x: mydeque)
14        std::cout << ' ' << x;
15    std::cout << '\n';
16
17    return 0;
18 }
```

Output:

```
mydeque contains: 222 111 10 20 30
```

## Complexity

Constant.

## Iterator validity

All iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.

## Data races

The container is modified.

No contained elements are accessed by the call: concurrently accessing or modifying them is safe (although see *iterator validity* above).

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<code>deque::emplace_back</code>	Construct and insert element at the end (public member function )
<code>deque::push_front</code>	Insert element at beginning (public member function )
<code>deque::emplace</code>	Construct and insert element (public member function )
<code>deque::erase</code>	Erase elements (public member function )
<code>deque::assign</code>	Assign container content (public member function )

# /deque/deque/empty

public member function

## std::deque::empty

<deque>

```
bool empty() const;
bool empty() const noexcept;
```

### Test whether container is empty

Returns whether the `deque` container is empty (i.e. whether its `size` is 0).

This function does not modify the container in any way. To clear the content of a `deque` container, see `deque::clear`.

## Parameters

none

## Return Value

true if the container `size` is 0, false otherwise.

## Example

```
1 // deque::empty
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
```

```

8 int sum (0);
9
10 for (int i=1;i<=10;i++) mydeque.push_back(i);
11
12 while (!mydeque.empty())
13 {
14     sum += mydeque.front();
15     mydeque.pop_front();
16 }
17
18 std::cout << "total: " << sum << '\n';
19
20 return 0;
21 }
```

The example initializes the content of the container to a sequence of numbers (form 1 to 10). It then pops the elements one by one until the container is empty and calculates their sum.

Output:

```
total: 55
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">deque::clear</a>	Clear content (public member function )
<a href="#">deque::erase</a>	Erase elements (public member function )
<a href="#">deque::size</a>	Return size (public member function )

# /deque/deque/end

public member function

## std::deque::end

<deque>

```

iterator end();
const_iterator end() const;
iterator end() noexcept;
const_iterator end() const noexcept;
```

### Return iterator to end

Returns an iterator referring to the *past-the-end* element in the [deque](#) container.

The *past-the-end* element is the theoretical element that would follow the last element in the [deque](#) container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with [deque::begin](#) to specify a range including all the elements in the container.

If the container is [empty](#), this function returns the same as [deque::begin](#).

## Parameters

none

## Return Value

An iterator to the element past the end of the sequence.

If the [deque](#) object is [const](#)-qualified, the function returns a [const\\_iterator](#). Otherwise, it returns an [iterator](#).

Member types [iterator](#) and [const\\_iterator](#) are [random access iterator](#) types (pointing to an element and to a [const](#) element, respectively).

## Example

```

1 // deque::end
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
```

```

9  for (int i=1; i<=5; i++) mydeque.insert(mydeque.end(),i);
10 std::cout << "mydeque contains:";
11
12 std::deque<int>::iterator it = mydeque.begin();
13
14 while (it != mydeque.end() )
15     std::cout << ' ' << *it++;
16
17 std::cout << '\n';
18
19 return 0;
20
21 }
```

Output:

```
mydeque contains: 1 2 3 4 5
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>deque::back</code>	Access last element ( <a href="#">public member function</a> )
<code>deque::begin</code>	Return iterator to beginning ( <a href="#">public member function</a> )
<code>deque::rbegin</code>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<code>deque::rend</code>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )

# /deque/deque/erase

public member function

## std::deque::erase

<deque>

```

iterator erase (iterator position);
iterator erase (iterator first, iterator last);
iterator erase (const_iterator position );
iterator erase (const_iterator first, const_iterator last );
```

### Erase elements

Removes from the `deque` container either a single element (*position*) or a range of elements ([*first*,*last*]).

This effectively reduces the container `size` by the number of elements removed, which are destroyed.

Double-ended queues are designed to be efficient removing (and inserting) elements at either the end or the beginning of the sequence. Removals on other positions are usually less efficient than in `list` or `forward_list` containers.

## Parameters

`position`

Iterator pointing to a single element to be removed from the `deque`.

Member types `iterator` and `const_iterator` are [random access iterator](#) types that point to elements.

`first, last`

Iterators specifying a range within the `deque`] to be removed: [*first*,*last*). i.e., the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.

Member types `iterator` and `const_iterator` are [random access iterator](#) types that point to elements.

## Return value

An iterator pointing to the new location of the element that followed the last element erased by the function call. This is the `container end` if the operation erased the last element in the sequence.

Member type `iterator` is a [random access iterator](#) type that points to elements.

## Example

```

1 // erasing from deque
2 #include <iostream>
3 #include <deque>
4
```

```

5 int main ()
6 {
7     std::deque<int> mydeque;
8
9     // set some values (from 1 to 10)
10    for (int i=1; i<=10; i++) mydeque.push_back(i);
11
12    // erase the 6th element
13    mydeque.erase (mydeque.begin() + 5);
14
15    // erase the first 3 elements:
16    mydeque.erase (mydeque.begin(), mydeque.begin() + 3);
17
18    std::cout << "mydeque contains:";
19    for (std::deque<int>::iterator it = mydeque.begin(); it!=mydeque.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }

```

Output:

mydeque contains: 4 5 7 8 9 10

## Complexity

Linear on the number of elements erased (destructions). Plus, depending on the particular library implementation, up to an additional linear time on the number of elements between *position* and one of the ends of the *deque*.

## Iterator validity

If the erasure operation includes the last element in the sequence, the *end iterator* and the iterators, pointers and references referring to the erased elements are invalidated.

If the erasure includes the first element but not the last, only those referring to the erased elements are invalidated.

If it happens anywhere else in the *deque*, all iterators, pointers and references related to the container are invalidated.

## Data races

The container is modified.

If the erasure happens at the beginning or the end of the sequence, only the erased elements are modified (although see *iterator validity* above).

If it happens anywhere else, it is not safe to access or modify elements.

## Exception safety

If the removed elements include the first or the last element in the container, no exceptions are thrown (no-throw guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee): Copying or moving elements while relocating them may throw.

Invalid ranges produce undefined behavior.

## See also

<a href="#">deque::pop_back</a>	Delete last element ( <a href="#">public member function</a> )
<a href="#">deque::insert</a>	Insert elements ( <a href="#">public member function</a> )

# /deque/deque/front

public member function

## std::deque::front

<deque>

`reference front();  
const_reference front() const;`

### Access first element

Returns a reference to the first element in the *deque* container.

Unlike member *deque::begin*, which returns an iterator to this same element, this function returns a direct reference.

Calling this function on an *empty* container causes undefined behavior.

## Parameters

none

## Return value

A reference to the first element in the *deque* container.

If the *deque* object is const-qualified, the function returns a *const\_reference*. Otherwise, it returns a *reference*.

Member types *reference* and *const\_reference* are the reference types to the elements of the container (see *deque member types*).

## Example

```

1 // deque::front
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;

```

```

8   mydeque.push_front(77);
9   mydeque.push_back(20);
10
11  mydeque.front() -= mydeque.back();
12
13  std::cout << "mydeque.front() is now " << mydeque.front() << '\n';
14
15  return 0;
16
17 }
```

Output:

```
mydeque.front() is now 57
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

The first element is potentially accessed or modified by the caller. Concurrently accessing or modifying other elements is safe.

## Exception safety

If the container is not `empty`, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<code>deque::back</code>	Access last element ( <a href="#">public member function</a> )
<code>deque::begin</code>	Return iterator to beginning ( <a href="#">public member function</a> )
<code>deque::push_front</code>	Insert element at beginning ( <a href="#">public member function</a> )
<code>deque::pop_front</code>	Delete first element ( <a href="#">public member function</a> )

# /deque/deque/get\_allocator

public member function

## std::deque::get\_allocator

<deque>

```
allocator_type get_allocator() const;
allocator_type get_allocator() const noexcept;
```

### Get allocator

Returns a copy of the allocator object associated with the `deque` object.

## Parameters

none

## Return Value

The allocator.

Member type `allocator_type` is the type of the allocator used by the container, defined in `deque` as an alias of its second template parameter (`Alloc`).

## Example

```

1 // deque::get_allocator
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
8     int * p;
9     unsigned int i;
10
11    // allocate an array with space for 5 elements using deque's allocator:
12    p = mydeque.get_allocator().allocate(5);
13
14    // construct values in-place on the array:
15    for (i=0; i<5; i++) mydeque.get_allocator().construct(&p[i],i);
16
17    std::cout << "The allocated array contains:";
18    for (i=0; i<5; i++) std::cout << ' ' << p[i];
19    std::cout << '\n';
20
21    // destroy and deallocate:
22    for (i=0; i<5; i++) mydeque.get_allocator().destroy(&p[i]);
23    mydeque.get_allocator().deallocate(p,5);
24 }
```

```
25 |     return 0;
26 }
```

The example shows an elaborate way to allocate memory for an array of `ints` using the same allocator used by the `deque` object. Output:  
The allocated array contains: 0 1 2 3 4

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

Copying any instantiation of the `default_allocator` is also guaranteed to never throw.

## See also

<a href="#">allocator</a>	Default allocator (class template )
---------------------------	-------------------------------------

# /deque/deque/insert

public member function

## std::deque::insert

<deque>

```
single element (1) iterator insert (iterator position, const value_type& val);
fill (2)      void insert (iterator position, size_type n, const value_type& val);
range (3)    template <class InputIterator>
              void insert (iterator position, InputIterator first, InputIterator last);

single element (1) iterator insert (const_iterator position, const value_type& val);
fill (2)      iterator insert (const_iterator position, size_type n, const value_type& val);
range (3)    template <class InputIterator>
              iterator insert (const_iterator position, InputIterator first, InputIterator last);
move (4)      iterator insert (const_iterator position, value_type&& val);
initializer list (5) iterator insert (const_iterator position, initializer_list<value_type> il);
```

### Insert elements

The `deque` container is extended by inserting new elements before the element at the specified `position`.

This effectively increases the container `size` by the amount of elements inserted.

Double-ended queues are designed to be efficient performing insertions (and removals) from either the end or the beginning of the sequence. Insertions on other positions are usually less efficient than in `list` or `forward_list` containers.

The parameters determine how many elements are inserted and to which values they are initialized:

## Parameters

### position

Position in the container where the new elements are inserted.

`iterator` is a member type, defined as a `random access iterator` type that points to elements.

### val

Value to be copied (or moved) to the inserted elements.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter (`T`).

### n

Number of elements to insert. Each element is initialized to a copy of `val`.

Member type `size_type` is an unsigned integral type.

### first, last

Iterators specifying a range of elements. Copies of the elements in the range `[first, last)` are inserted at `position` (in the same order).

Notice that the range includes all the elements between `first` and `last`, including the element pointed by `first` but not the one pointed by `last`.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type from which `value_type` objects can be constructed.

### il

An `initializer_list` object. Copies of these elements are inserted at `position` (in the same order).

These objects are automatically constructed from `initializer_list` declarators.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter (`T`).

## Return value

An iterator that points to the first of the newly inserted elements.

Member type `iterator` is a `random access iterator` type that points to elements.

The storage for the new elements is allocated using the container's allocator, which may throw exceptions on failure (for the default allocator, `bad_alloc` is thrown if the allocation request does not succeed).

## Example

```
1 // inserting into a deque
2 #include <iostream>
3 #include <deque>
4 #include <vector>
5
6 int main ()
7 {
8     std::deque<int> mydeque;
9
10    // set some initial values:
11    for (int i=1; i<6; i++) mydeque.push_back(i); // 1 2 3 4 5
12
13    std::deque<int>::iterator it = mydeque.begin();
14    ++it;
15
16    it = mydeque.insert (it,10); // 1 10 2 3 4 5
17    // "it" now points to the newly inserted 10
18
19    mydeque.insert (it,20); // 1 20 20 10 2 3 4 5
20    // "it" no longer valid!
21
22    it = mydeque.begin() + 2;
23
24    std::vector<int> myvector (2,30);
25    mydeque.insert (it,myvector.begin(),myvector.end()); // 1 20 30 30 20 10 2 3 4 5
26
27
28    std::cout << "mydeque contains:";
29    for (it=mydeque.begin(); it!=mydeque.end(); ++it)
30        std::cout << ' ' << *it;
31    std::cout << '\n';
32
33    return 0;
34 }
```

Output:

```
mydeque contains: 1 20 30 30 20 10 2 3 4 5
```

## Complexity

Linear on the number of elements inserted (copy/move construction). Plus, depending on the particular library implementation, up to an additional linear in the number of elements between `position` and one of the ends of the `deque`.

## Iterator validity

If the insertion happens at the beginning or the end of the sequence, all iterators related to this container are invalidated, but pointers and references remain valid, referring to the same elements they were referring to before the call.

If the insertion happens anywhere else in the `deque`, all iterators, pointers and references related to this container are invalidated.

## Data races

The container is modified.

If the insertion happens at the beginning or the end of the sequence, no contained elements are accessed (although see *iterator validity* above).

If it happens anywhere else, it is not safe to concurrently access elements.

## Exception safety

If the operation inserts a single element at the `begin` or the `end`, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if an invalid `position` or range is specified, it causes *undefined behavior*.

## See also

<code>deque::push_back</code>	Add element at the end ( <a href="#">public member function</a> )
<code>deque::push_front</code>	Insert element at beginning ( <a href="#">public member function</a> )
<code>deque::erase</code>	Erase elements ( <a href="#">public member function</a> )

## /deque/deque/max\_size

public member function

### `std::deque::max_size`

<`deque`>

```
size_type max_size() const;
size_type max_size() const noexcept;
```

#### Return maximum size

Returns the maximum number of elements that the `deque` container can hold.

This is the maximum potential `size` the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size: it can still fail to allocate storage at any point before that size is reached.

## Parameters

none

## Return Value

The maximum number of elements a `deque` container can hold as content.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // deque::max_size
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     unsigned int i;
8     std::deque<int> mydeque;
9
10    std::cout << "Enter number of elements: ";
11    std::cin >> i;
12
13    if (i<mydeque.max_size()) mydeque.resize(i);
14    else std::cout << "That size exceeds the limit.\n";
15
16    return 0;
17 }
```

Here, member `max_size` is used to check beforehand whether the requested size will be allowed by `resize`.

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>deque::size</code>	Return size (public member function )
<code>deque::resize</code>	Change size (public member function )

# /deque/deque/operator=

public member function

## std::deque::operator=

<deque>

```
copy (1) deque& operator= (const deque& x);
copy (1) deque& operator= (const deque& x);
move (2) deque& operator= (deque& x);
initializer list (3) deque& operator= (initializer_list<value_type> il);
```

### Assign content

Assigns new contents to the container, replacing its current contents, and modifying its `size` accordingly.

Copies all the elements from `x` into the container.

The container preserves its `current allocator`.

The `copy assignment` (1) copies all the elements from `x` into the container (with `x` preserving its contents).

The `move assignment` (2) moves the elements of `x` into the container (`x` is left in an unspecified but valid state).

The `initializer list assignment` (3) copies the elements of `il` into the container.

The container preserves its `current allocator`, except if the `allocator traits` indicate `x`'s allocator should `propagate`. This allocator is used (through its `traits`) to `allocate` or `deallocate` if there are changes in storage requirements, and to `construct` or `destroy` elements, if needed.

Any elements held in the container before the call are either `assigned to` or `destroyed`.

## Parameters

`x` A `deque` object of the same type (i.e., with the same template parameters, `T` and `Alloc`).

`il`

An [initializer\\_list](#) object. The compiler will automatically construct such objects from *initializer list* declarators.  
Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter (`T`).

## Return value

`*this`

## Example

```
1 // assignment operator with deques
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> first (3);    // deque with 3 zero-initialized ints
8     std::deque<int> second (5);   // deque with 5 zero-initialized ints
9
10    second = first;
11    first = std::deque<int>();
12
13    std::cout << "Size of first: " << int (first.size()) << '\n';
14    std::cout << "Size of second: " << int (second.size()) << '\n';
15    return 0;
16 }
```

Output:

```
Size of first: 0
Size of second: 3
```

## Complexity

Linear in `size`.

## Iterator validity

All iterators, references and pointers related to this container before the call are invalidated.

In the *move assignment*, iterators, pointers and references referring to elements in `x` are also invalidated.

## Data races

All copied elements are accessed.

The *move assignment* (2) modifies `x`.

The container and all its elements are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if `value_type` is not `copy assignable` (or `move assignable` for (2)), it causes *undefined behavior*.

## See also

<code>deque::assign</code>	Assign container content (public member function )
----------------------------	--

# /deque/deque/operator[]

public member function

## std::deque::operator[]

<deque>

```
reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

### Access element

Returns a reference to the element at position `n` in the `deque` container.

A similar member function, `deque::at`, has the same behavior as this operator function, except that `deque::at` is bound-checked and signals if the requested position is out of range by throwing an `out_of_range` exception.

## Parameters

`n`

Position of an element in the container.  
Notice that the first element has a position of 0 (not 1).  
Member type `size_type` is an unsigned integral type.

## Return value

The element at the specified position in the container.

Member types `reference` and `const_reference` are the reference types to the elements of the container (see `deque` member types).

## Example

```

1 // deque::operator[] example: reversing order
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque (10); // 10 zero-initialized elements
8     std::deque<int>::size_type sz = mydeque.size();
9
10    // assign some values:
11    for (unsigned i=0; i<sz; i++) mydeque[i]=i;
12
13    // reverse order of elements using operator[]:
14    for (unsigned i=0; i<sz/2; i++)
15    {
16        int temp;
17        temp = mydeque[sz-1-i];
18        mydeque[sz-1-i]=mydeque[i];
19        mydeque[i]=temp;
20    }
21
22    // print content:
23    std::cout << "mydeque contains:";
24    for (unsigned i=0; i<sz; i++)
25        std::cout << ' ' << mydeque[i];
26    std::cout << '\n';
27
28    return 0;
29 }

```

Output:

mydeque contains: 9 8 7 6 5 4 3 2 1 0

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Element  $n$  is potentially accessed or modified. Concurrently accessing or modifying other elements is safe.

## Exception safety

If the container `size` is greater than  $n$ , the function never throws exceptions (no-throw guarantee). Otherwise, the behavior is undefined (which may include throwing).

## See also

<code>deque::at</code>	Access element (public member function )
<code>deque::front</code>	Access first element (public member function )
<code>deque::back</code>	Access last element (public member function )

# /deque/deque/operators

function

## std::relational operators (deque)

<deque>

```

(1) template <class T, class Alloc>
    bool operator== (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
(2) template <class T, class Alloc>
    bool operator!= (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
(3) template <class T, class Alloc>
    bool operator< (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
(4) template <class T, class Alloc>
    bool operator<= (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
(5) template <class T, class Alloc>
    bool operator> (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);
(6) template <class T, class Alloc>
    bool operator>= (const deque<T,Alloc>& lhs, const deque<T,Alloc>& rhs);

```

### Relational operators for deque

Performs the appropriate comparison operation between the `deque` containers *lhs* and *rhs*.

The *equality comparison* (`operator==`) is performed by first comparing `sizes`, and if they match, the elements are compared sequentially using `operator==`, stopping at the first mismatch (as if using algorithm `equal`).

The *less-than comparison* (`operator<`) behaves as if using algorithm `lexicographical_compare`, which compares the elements sequentially using `operator<` in a reciprocal manner (i.e., checking both  $a < b$  and  $b < a$ ) and stopping at the first occurrence.

The other operations also use the operators == and < internally to compare the elements, behaving as if the following equivalent operations were performed:

operation	equivalent operation
<code>a!=b</code>	<code>!(a==b)</code>
<code>a&gt;b</code>	<code>b&lt;a</code>

a<=b	!(b<a)
a>=b	!(a<b)

These operators are overloaded in header `<deque>`.

## Parameters

`lhs, rhs`  
`deque` containers (to the left- and right-hand side of the operator, respectively), having both the same template parameters (`T` and `Alloc`).

## Example

```
1 // deque comparisons
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> foo (3,100);    // three ints with a value of 100
8     std::deque<int> bar (2,200);    // two ints with a value of 200
9
10    if (foo==bar) std::cout << "foo and bar are equal\n";
11    if (foo!=bar) std::cout << "foo and bar are not equal\n";
12    if (foo< bar) std::cout << "foo is less than bar\n";
13    if (foo> bar) std::cout << "foo is greater than bar\n";
14    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
15    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
16
17    return 0;
18 }
```

Output:

```
foo and bar are not equal
foo is less than bar
foo is less than or equal to bar
```

## Return Value

true if the condition holds, and false otherwise.

## Complexity

Up to linear in the `size` of `lhs` and `rhs`.

For (1) and (2), constant if the `sizes` of `lhs` and `rhs` differ, and up to linear in that `size` (equality comparisons) otherwise.

For the others, up to linear in the smaller `size` (each representing two comparisons with `operator<`).

## Iterator validity

No changes.

## Data races

Both containers, `lhs` and `rhs`, are accessed.

Up to all of their contained elements may be accessed.

## Exception safety

If the type of the elements supports the appropriate operation with no-throw guarantee, the function never throws exceptions (no-throw guarantee). In any case, the function cannot modify its arguments.

## See also

<code>deque::operator=</code>	Assign content (public member function )
<code>deque::swap</code>	Swap content (public member function )

# /deque/deque/pop\_back

public member function

## `std::deque::pop_back`

`<deque>`

`void pop_back();`

### Delete last element

Removes the last element in the `deque` container, effectively reducing the container `size` by one.

This destroys the removed element.

## Parameters

none

## Return value

none

## Example

```
1 // deque::pop_back
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
8     int sum (0);
9     mydeque.push_back (10);
10    mydeque.push_back (20);
11    mydeque.push_back (30);
12
13    while (!mydeque.empty())
14    {
15        sum+=mydeque.back();
16        mydeque.pop_back();
17    }
18
19    std::cout << "The elements of mydeque add up to " << sum << '\n';
20
21    return 0;
22 }
```

In this example, the elements are popped out from the end of the `deque` after they are added up in the sum. Output:

```
The elements of mydeque add up to 60
```

## Complexity

Constant.

## Iterator validity

The end iterator and any iterator, pointer and reference referring to the removed element are invalidated.

Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.

## Data races

The container is modified.

The last element is modified. Concurrently accessing or modifying other elements is safe (although see *iterator validity* above).

## Exception safety

If the container is not `empty`, the function never throws exceptions (no-throw guarantee).

Otherwise, the behavior is undefined.

## See also

<code>deque::pop_front</code>	Delete first element ( <a href="#">public member function</a> )
<code>deque::push_back</code>	Add element at the end ( <a href="#">public member function</a> )
<code>deque::erase</code>	Erase elements ( <a href="#">public member function</a> )

## /deque/deque/pop\_front

public member function

### std::deque::pop\_front

<deque>

`void pop_front();`

#### Delete first element

Removes the first element in the `deque` container, effectively reducing its `size` by one.

This destroys the removed element.

## Parameters

none

## Return value

none

## Example

```
1 // deque::pop_front
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
8
9     mydeque.push_back (100);
10    mydeque.push_back (200);
```

```

11 mydeque.push_back(300);
12
13 std::cout << "Popping out the elements in mydeque:";
14 while (!mydeque.empty())
15 {
16     std::cout << ' ' << mydeque.front();
17     mydeque.pop_front();
18 }
19
20 std::cout << "\nThe final size of mydeque is " << int(mydeque.size()) << '\n';
21
22 return 0;
23 }
```

Output:

```
Popping out the elements in mydeque: 100 200 300
The final size of mydeque is 0
```

## Complexity

Constant.

## Iterator validity

The iterators, pointers and references referring to the removed element are invalidated.

Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.

## Data races

The container is modified.

The first element is modified. Concurrently accessing or modifying other elements is safe (although see *iterator validity* above).

## Exception safety

If the container is not `empty`, the function never throws exceptions (no-throw guarantee).

Otherwise, the behavior is undefined.

## See also

<code>deque::pop_back</code>	Delete last element ( <a href="#">public member function</a> )
<code>deque::push_front</code>	Insert element at beginning ( <a href="#">public member function</a> )
<code>deque::erase</code>	Erase elements ( <a href="#">public member function</a> )

# /deque/deque/push\_back

public member function

## std::deque::push\_back

<deque>

```
void push_back (const value_type& val);
void push_back (const value_type& val);
void push_back (value_type&& val);
```

### Add element at the end

Adds a new element at the end of the `deque` container, after its current last element. The content of `val` is copied (or moved) to the new element.

This effectively increases the container `size` by one.

## Parameters

`val`

Value to be copied (or moved) to the new element.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter (`T`).

## Return value

none

The storage for the new elements is allocated using the container's allocator, which may throw exceptions on failure (for the default allocator, `bad_alloc` is thrown if the allocation request does not succeed).

## Example

```

1 // deque::push_back
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque;
8     int myint;
9
10    std::cout << "Please enter some integers (enter 0 to end):\n";
11
12    do {
13        std::cin >> myint;
```

```

14     mydeque.push_back (myint);
15 } while (myint);
16
17 std::cout << "mydeque stores " << (int) mydeque.size() << " numbers.\n";
18
19 return 0;
20 }
```

The example uses `push_back` to add a new element to the container each time a new integer is read.

## Complexity

Constant.

## Iterator validity

All iterators related to this container are invalidated. Pointers and references to elements in the container remain valid, referring to the same elements they were referring to before the call.

## Data races

The container is modified.

No existing elements are accessed (although see *iterator validity* above).

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with `val` as argument, it causes *undefined behavior*.

## See also

<code>deque::push_front</code>	Insert element at beginning (public member function )
<code>deque::pop_back</code>	Delete last element (public member function )
<code>deque::insert</code>	Insert elements (public member function )

# /deque/deque/push\_front

public member function

## std::deque::push\_front

<deque>

```

void push_front (const value_type& val);
void push_front (const value_type& val);
void push_front (value_type&& val);
```

### Insert element at beginning

Inserts a new element at the beginning of the `deque` container, right before its current first element. The content of `val` is copied (or moved) to the inserted element.

This effectively increases the container `size` by one.

## Parameters

`val`

Value to be copied (or moved) to the inserted element.

Member type `value_type` is the type of the elements in the container, defined in `deque` as an alias of its first template parameter (`T`).

## Return value

none

The storage for the new elements is allocated using the container's `allocator`, which may throw exceptions on failure (for the default `allocator`, `bad_alloc` is thrown if the allocation request does not succeed).

## Example

```

1 // deque::push_front
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque (2,100);      // two ints with a value of 100
8     mydeque.push_front (200);
9     mydeque.push_front (300);
10
11    std::cout << "mydeque contains:";
12    for (std::deque<int>::iterator it = mydeque.begin(); it != mydeque.end(); ++it)
13        std::cout << ' ' << *it;
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

300 200 100 100
-----------------

## Complexity

Constant.

## Iterator validity

All iterators related to this container are invalidated. Pointers and references to elements in the container remain valid, referring to the same elements they were referring to before the call.

## Data races

The container is modified.

No existing elements are accessed (although see *iterator validity* above).

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with `val` as argument, it causes *undefined behavior*.

## See also

<code>deque::push_back</code>	Add element at the end ( <a href="#">public member function</a> )
<code>deque::pop_front</code>	Delete first element ( <a href="#">public member function</a> )
<code>deque::insert</code>	Insert elements ( <a href="#">public member function</a> )

# /deque/deque/rbegin

public member function

## std::deque::rbegin

<deque>

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
```

### Return reverse iterator to reverse beginning

Returns a *reverse iterator* pointing to the last element in the container (i.e., its *reverse beginning*).

Reverse *iterators* iterate backwards: increasing them moves them towards the beginning of the container.

`rbegin` points to the element right before the one that would be pointed to by member `end`.

Notice that unlike member `deque::back`, which returns a reference to this same element, this function returns a *reverse random access iterator*.

## Parameters

none

## Return Value

A reverse iterator to the *reverse beginning* of the sequence container.

If the `deque` object is `const`-qualified, the function returns a `const_reverse_iterator`. Otherwise, it returns a `reverse_iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are *reverse random access iterator* types (pointing to an element and to a `const` element, respectively). See [deque member types](#).

## Example

```
1 // deque::rbegin/rend
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque (5); // 5 default-constructed ints
8
9     std::deque<int>::reverse_iterator rit = mydeque.rbegin();
10
11    int i=0;
12    for (rit = mydeque.rbegin(); rit!= mydeque.rend(); ++rit)
13        *rit = ++i;
14
15    std::cout << "mydeque contains:";
16    for (std::deque<int>::iterator it = mydeque.begin(); it != mydeque.end(); ++it)
17        std::cout << ' ' << *it;
18    std::cout << '\n';
19
20    return 0;
21 }
```

Output:

```
mydeque contains: 5 4 3 2 1
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">deque::back</a>	Access last element ( <a href="#">public member function</a> )
<a href="#">deque::rend</a>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )
<a href="#">deque::begin</a>	Return iterator to beginning ( <a href="#">public member function</a> )
<a href="#">deque::end</a>	Return iterator to end ( <a href="#">public member function</a> )

# /deque/deque/rend

public member function

## std::deque::rend

<deque>

```
reverse_iterator rend();
const_reverse_iterator rend() const;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

### Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element preceding the first element in the [deque](#) container (which is considered its *reverse end*).

The range between [deque::rbegin](#) and [deque::rend](#) contains all the elements of the [deque](#) container (in reverse order).

## Parameters

none

## Return Value

A reverse iterator to the *reverse end* of the sequence container.

If the [deque](#) object is const-qualified, the function returns a [const\\_reverse\\_iterator](#). Otherwise, it returns a [reverse\\_iterator](#).

Member types [reverse\\_iterator](#) and [const\\_reverse\\_iterator](#) are reverse random access iterator types (pointing to an element and to a const element, respectively). See [deque member types](#).

## Example

```
1 // deque::rbegin/rend
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque (5); // 5 default-constructed ints
8
9     std::deque<int>::reverse_iterator rit = mydeque.rbegin();
10
11    int i=0;
12    for (rit = mydeque.rbegin(); rit!= mydeque.rend(); ++rit)
13        *rit = ++i;
14
15    std::cout << "mydeque contains:";
16    for (std::deque<int>::iterator it = mydeque.begin(); it != mydeque.end(); ++it)
17        std::cout << ' ' << *it;
18    std::cout << '\n';
19
20    return 0;
21 }
```

Output:

```
5 4 3 2 1
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<b>deque::rbegin</b>	Return reverse iterator to reverse beginning (public member function )
<b>deque::front</b>	Access first element (public member function )
<b>deque::begin</b>	Return iterator to beginning (public member function )
<b>deque::end</b>	Return iterator to end (public member function )

# /deque/deque/resize

public member function

## std::deque::resize

<deque>

```
void resize (size_type n, value_type val = value_type());  
void resize (size_type n);  
void resize (size_type n, const value_type& val);
```

### Change size

Resizes the container so that it contains *n* elements.

If *n* is smaller than the current container **size**, the content is reduced to its first *n* elements, removing those beyond (and destroying them).

If *n* is greater than the current container **size**, the content is expanded by inserting at the end as many elements as needed to reach a size of *n*. If *val* is specified, the new elements are initialized as copies of *val*, otherwise, they are value-initialized.

Notice that this function changes the actual content of the container by inserting or erasing elements from it.

## Parameters

*n*

New container size, expressed in number of elements.  
Member type **size\_type** is an unsigned integral type.

*val*

Object whose content is copied to the added elements in case that *n* is greater than the current container **size**.  
If not specified, the default constructor is used instead.  
Member type **value\_type** is the type of the elements in the container, defined in **deque** as an alias of the first template parameter ( $\pi$ ).

## Return Value

none

In case of growth, the storage for the new elements is allocated using the container's **allocator**, which may throw exceptions on failure (for the default **allocator**, **bad\_alloc** is thrown if the allocation request does not succeed).

## Example

```
1 // resizing deque  
2 #include <iostream>  
3 #include <deque>  
4  
5 int main ()  
6 {  
7     std::deque<int> mydeque;  
8     std::deque<int>::iterator it;  
9  
10    // set some initial content:  
11    for (int i=1; i<10; ++i) mydeque.push_back(i);  
12  
13    mydeque.resize(5);  
14    mydeque.resize(8,100);  
15    mydeque.resize(12);  
16  
17    std::cout << "mydeque contains:";  
18    for (std::deque<int>::iterator it = mydeque.begin(); it != mydeque.end(); ++it)  
19        std::cout << ' ' << *it;  
20    std::cout << '\n';  
21  
22    return 0;  
23 }
```

The code sets a sequence of 9 numbers as the initial content for **mydeque**. It then uses **resize** first to set the container size to 5, then to extend its size to 8 with values of 100 for its new elements, and finally it extends its size to 12 with their default values (for **int** elements this is zero). Output:

```
mydeque contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

## Complexity

Linear on the number of elements inserted/erased (constructions/destructions).

## Iterator validity

In case the container shrinks, all iterators, pointers and references to elements that have not been removed remain valid after the resize and refer to the same elements they were referring to before the call.

If the container expands, all iterators are invalidated, but existing pointers and references remain valid, referring to the same elements they were referring to before.

## Data races

The container is modified.

Removed elements are modified (see *iterator validity* above).

## Exception safety

If  $n$  is less than or equal to the [size](#) of the container, the function never throws exceptions (no-throw guarantee).

Otherwise, if an exception is thrown, the container is left with a valid state (basic guarantee): Constructing elements or allocating storage may throw.

## See also

<a href="#">deque::size</a>	Return size (public member function )
<a href="#">deque::clear</a>	Clear content (public member function )
<a href="#">deque::erase</a>	Erase elements (public member function )
<a href="#">deque::max_size</a>	Return maximum size (public member function )

# /deque/deque/shrink\_to\_fit

public member function

## std::deque::shrink\_to\_fit

<deque>

```
void shrink_to_fit();
```

### Shrink to fit

Requests the container to reduce its memory usage to fit its [size](#).

A [deque](#) container may have more memory allocated than needed to hold its current elements: this is because most libraries implement [deque](#) as a dynamic array that can keep the allocated space of removed elements or allocate additional capacity in advance to allow for faster insertion operations.

This function requests that the memory usage is adapted to the current [size](#) of the container, but the request is non-binding, and the container implementation is free to optimize its memory usage otherwise.

Note that this function does not change the [size](#) of the container (for that, see [resize](#) instead).

## Parameters

none

## Return value

none

## Example

```
1 // deque::shrink_to_fit
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> mydeque (100);
8     std::cout << "1. size of mydeque: " << mydeque.size() << '\n';
9
10    mydeque.resize(10);
11    std::cout << "2. size of mydeque: " << mydeque.size() << '\n';
12
13    mydeque.shrink_to_fit();
14
15    return 0;
16 }
```

Output:

```
1. size of mydeque: 100
2. size of mydeque: 10
```

## Complexity

At most, linear in the [container size](#).

## Iterator validity

No changes.

## Data races

The container is modified.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.

## See also

<a href="#">deque::size</a>	Return size (public member function )
<a href="#">deque::resize</a>	Change size (public member function )
<a href="#">deque::clear</a>	Clear content (public member function )

# /deque/deque/size

public member function

## std::deque::size

<deque>

```
size_type size() const;
size_type size() const noexcept;
```

### Return size

Returns the number of elements in the [deque](#) container.

### Parameters

none

### Return Value

The number of elements in the container.

Member type `size_type` is an unsigned integral type.

### Example

```
1 // deque::size
2 #include <iostream>
3 #include <deque>
4
5 int main ()
6 {
7     std::deque<int> myints;
8     std::cout << "0. size: " << myints.size() << '\n';
9
10    for (int i=0; i<5; i++) myints.push_back(i);
11    std::cout << "1. size: " << myints.size() << '\n';
12
13    myints.insert (myints.begin(),5,100);
14    std::cout << "2. size: " << myints.size() << '\n';
15
16    myints.pop_back();
17    std::cout << "3. size: " << myints.size() << '\n';
18
19    return 0;
20 }
```

Output:

```
0. size: 0
1. size: 5
2. size: 10
3. size: 9
```

### Complexity

Constant.

### Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">deque::empty</a>	Test whether container is empty (public member function )
<a href="#">deque::resize</a>	Change size (public member function )
<a href="#">deque::max_size</a>	Return maximum size (public member function )

# /deque/deque/swap

public member function

## std::deque::swap

<deque>

[void swap \(deque& x\);](#)

### Swap content

Exchanges the content of the container by the content of *x*, which is another [deque](#) object containing elements of the same type. Sizes may differ.

After the call to this member function, the elements in this container are those which were in *x* before the call, and the elements of *x* are those which were in this. All iterators, references and pointers remain valid for the swapped objects.

Notice that a non-member function exists with the same name, [swap](#), overloading that algorithm with an optimization that behaves like this member function.

No specifics on [allocators](#). [contradictory specifications]

Whether the container [allocators](#) are also swapped is not defined, unless in the case the appropriate [allocator trait](#) indicates explicitly that they shall [propagate](#).

## Parameters

x

Another [deque](#) container of the same type (i.e., instantiated with the same template parameters, T and Alloc) whose content is swapped with that of this container.

## Return value

none

## Example

```
1 // swap deque
2 #include <iostream>
3 #include <deque>
4
5 main ()
6 {
7     unsigned int i;
8     std::deque<int> foo (3,100);    // three ints with a value of 100
9     std::deque<int> bar (5,200);    // five ints with a value of 200
10
11    foo.swap(bar);
12
13    std::cout << "foo contains:";
14    for (std::deque<int>::iterator it = foo.begin(); it!=foo.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    std::cout << "bar contains:";
19    for (std::deque<int>::iterator it = bar.begin(); it!=bar.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
foo contains: 200 200 200 200 200
bar contains: 100 100 100
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.

Note that the *end iterators* do not refer to elements and may be invalidated.

## Data races

Both the container and *x* are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both containers compare equal, or if their [allocator traits](#) indicate that the allocators shall [propagate](#), the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">swap (deque)</a>	Exchanges the contents of two deque containers (function template )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )

# /deque/deque/swap-free

function template

## std::swap (deque)

<deque>

```
template <class T, class Alloc>
void swap (deque<T,Alloc>& x, deque<T,Alloc>& y);
```

### Exchanges the contents of two deque containers

The contents of container *x* are exchanged with those of *y*. Both container objects must be of the same type (same template parameters), although sizes may differ.

After the call to this member function, the elements in *x* are those which were in *y* before the call, and the elements of *y* are those which were in *x*. All iterators, references and pointers remain valid for the swapped objects.

This is an overload of the generic algorithm [swap](#) that improves its performance by mutually transferring ownership over their assets to the other container (i.e., the containers exchange references to their data, without actually performing any element copy or movement): It behaves as if *x.swap(y)* was called.

## Parameters

*x,y* deque containers of the same type (i.e., having both the same template parameters, *T* and *Alloc*).

## Return value

none

## Example

```
1 // swap (deque overload)
2 #include <iostream>
3 #include <deque>
4
5 main ()
6 {
7     unsigned int i;
8     std::deque<int> foo (3,100);    // three ints with a value of 100
9     std::deque<int> bar (5,200);    // five ints with a value of 200
10
11    swap(foo,bar);
12
13    std::cout << "foo contains:";
14    for (std::deque<int>::iterator it = foo.begin(); it!=foo.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    std::cout << "bar contains:";
19    for (std::deque<int>::iterator it = bar.begin(); it!=bar.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
foo contains: 200 200 200 200 200
bar contains: 100 100 100
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.

Note that the *end* iterators do not refer to elements and may be invalidated.

## Data races

Both containers, *x* and *y*, are modified.

## Exception safety

If the allocators in both compare compare equal, or if their [allocator traits](#) indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">deque::swap</a>	Swap content (public member function )
-----------------------------	--

<a href="#">swap</a>	Exchange values of two objects ( <a href="#">function template</a> )
<a href="#">swap_ranges</a>	Exchange values of two ranges ( <a href="#">function template</a> )

## /ios

header

**<ios>**

### Input-Output base classes

Header providing base classes and types for the IOStream hierarchy of classes:

 click on an element for detailed information

### Types

#### Class templates

<a href="#">basic_ios</a>	Base class for streams (type-dependent components) ( <a href="#">class template</a> )
<a href="#">fpos</a>	Stream position class template ( <a href="#">class template</a> )

#### Classes

<a href="#">ios</a>	Base class for streams (type-dependent components) ( <a href="#">class</a> )
<a href="#">ios_base</a>	Base class for streams ( <a href="#">class</a> )
<a href="#">wios</a>	Base class for wide character streams ( <a href="#">class</a> )

#### Other types

<a href="#">io_errc</a>	Input/output error conditions ( <a href="#">enum class</a> )
<a href="#">streamoff</a>	Stream offset type ( <a href="#">type</a> )
<a href="#">streampos</a>	Stream position type ( <a href="#">type</a> )
<a href="#">streamsize</a>	Stream size type ( <a href="#">type</a> )
<a href="#">wstreampos</a>	Wide stream position type ( <a href="#">type</a> )

### Format flag manipulators (functions)

#### Independent flags (switch on):

<a href="#">boolalpha</a>	Alphanumerical bool values ( <a href="#">function</a> )
<a href="#">showbase</a>	Show numerical base prefixes ( <a href="#">function</a> )
<a href="#">showpoint</a>	Show decimal point ( <a href="#">function</a> )
<a href="#">showpos</a>	Show positive signs ( <a href="#">function</a> )
<a href="#">skipws</a>	Skip whitespaces ( <a href="#">function</a> )
<a href="#">unitbuf</a>	Flush buffer after insertions ( <a href="#">function</a> )
<a href="#">uppercase</a>	Generate upper-case letters ( <a href="#">function</a> )

#### Independent flags (switch off):

<a href="#">noboolalpha</a>	No alphanumerical bool values ( <a href="#">function</a> )
<a href="#">noshowbase</a>	Do not show numerical base prefixes ( <a href="#">function</a> )
<a href="#">noshowpoint</a>	Do not show decimal point ( <a href="#">function</a> )
<a href="#">noshowpos</a>	Do not show positive signs ( <a href="#">function</a> )
<a href="#">noskipws</a>	Do not skip whitespaces ( <a href="#">function</a> )
<a href="#">nounitbuf</a>	Do not force flushes after insertions ( <a href="#">function</a> )
<a href="#">nouppercase</a>	Do not generate upper case letters ( <a href="#">function</a> )

#### Numerical base format flags ("basefield" flags):

<a href="#">dec</a>	Use decimal base ( <a href="#">function</a> )
<a href="#">hex</a>	Use hexadecimal base ( <a href="#">function</a> )
<a href="#">oct</a>	Use octal base ( <a href="#">function</a> )

#### Floating-point format flags ("floatfield" flags):

<a href="#">fixed</a>	Use fixed floating-point notation ( <a href="#">function</a> )
<a href="#">scientific</a>	Use scientific floating-point notation ( <a href="#">function</a> )

#### Adjustment format flags ("adjustfield" flags):

<b>internal</b>	Adjust field by inserting characters at an internal position ( <a href="#">function</a> )
<b>left</b>	Adjust output to the left ( <a href="#">function</a> )
<b>right</b>	Adjust output to the right ( <a href="#">function</a> )

## Other functions

<b>iostream_category</b>	Return iostream category ( <a href="#">function</a> )
--------------------------	---

Notice that not all standard manipulators are defined in this header. Input streams also support `ws`, and output streams `endl`, `ends` and `flush`. Streams also support an additional set of manipulators, which are parametric and defined apart in header `<iomanip>`. These are: `setiosflags`, `resetiosflags`, `setbase`, `setfill`, `setprecision`, `setw`

## /ios/basic\_ios

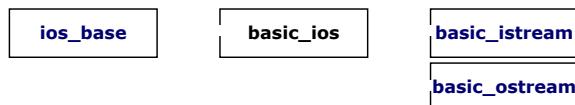
class template

**std::basic\_ios**

`<iostream>`

```
template <class charT, class traits = char_traits<charT> >
class basic_ios;
```

**Base class for streams (type-dependent components)**



Template class to instantiate the base classes for all stream classes.

Both this class template and its parent class, `ios_base`, define the components of streams that do not depend on whether the stream is an input or an output stream. `ios_base` describes the members that are independent of the template parameters, while this one describes the members that are dependent on the template parameters.

The class template adds to the information kept by its inherited `ios_base` component, the following:

	<b>field</b>	<b>member functions</b>	<b>description</b>
<i>Formatting</i>	fill character	<code>fill</code>	Character to pad a formatted field up to the <i>field width</i> ( <code>width</code> ).
<i>State</i>	error state	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	exception mask	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
<i>Other</i>	tied stream	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	stream buffer	<code>rdbuf</code>	Pointer to the associated <code>basic_streambuf</code> object, which is charge of all input/output operations.

## Template parameters

`charT`

Character type.

This shall be a non-array POD type.

Aliased as member type `basic_ios::char_type`.

`traits`

Character traits class that defines essential properties of the characters used by stream objects (see `char_traits`).

`traits::char_type` shall be the same as `charT`.

Aliased as member type `basic_ios::traits_type`.

## Template instantiations

<b>ios</b>	Base class for streams (type-dependent components) ( <a href="#">class</a> )
<b>wios</b>	Base class for wide character streams ( <a href="#">class</a> )

These instantiations are declared in `<iosfwd>`, which is included by reference in `<iostream>` and `<iostream>`.

## Member types

<b>member type</b>	<b>definition</b>	<b>notes</b>
<code>char_type</code>	The first template parameter ( <code>charT</code> )	
<code>traits_type</code>	The second template parameter ( <code>traits</code> )	defaults to: <code>char_traits&lt;charT&gt;</code>
<code>int_type</code>	<code>traits_type::int_type</code>	
<code>pos_type</code>	<code>traits_type::pos_type</code>	generally, the same as <code>streampos</code>
<code>off_type</code>	<code>traits_type::off_type</code>	generally, the same as <code>streamoff</code>

Along with the member types inherited from `ios_base`:

<b>event</b>	Type to indicate event type ( <a href="#">public member type</a> )
<b>event_callback</b>	Event callback function type ( <a href="#">public member type</a> )
<b>failure</b>	Base class for stream exceptions ( <a href="#">public member class</a> )
<b>fmtflags</b>	Type for stream format flags ( <a href="#">public member type</a> )
<b>Init</b>	Initialize standard stream objects ( <a href="#">public member class</a> )
<b>iostate</b>	Type for stream state flags ( <a href="#">public member type</a> )
<b>openmode</b>	Type for stream opening mode flags ( <a href="#">public member type</a> )
<b>seekdir</b>	Type for stream seeking direction flag ( <a href="#">public member type</a> )

## Public member functions

<b>(constructor)</b>	Construct object (public member function )
<b>(destructor)</b>	Destroy object (public member function )

### State flag functions:

<b>good</b>	Check whether state of stream is good (public member function )
<b>eof</b>	Check whether eofbit is set (public member function )
<b>fail</b>	Check whether failbit or badbit is set (public member function )
<b>bad</b>	Check whether badbit is set (public member function )
<b>operator!</b>	Evaluate stream (not) (public member function )
<b>operator bool</b>	Evaluate stream (public member function )
<b>rdstate</b>	Get error state flags (public member function )
<b>setstate</b>	Set error state flag (public member function )
<b>clear</b>	Set error state flags (public member function )

### Formatting:

<b>copyfmt</b>	Copy formatting information (public member function )
<b>fill</b>	Get/set fill character (public member function )

### Other:

<b>exceptions</b>	Get/set exceptions mask (public member function )
<b>imbind</b>	Imbind locale (public member function )
<b>tie</b>	Get/set tied stream (public member function )
<b>rdbuf</b>	Get/set stream buffer (public member function )
<b>narrow</b>	Narrow character (public member function )
<b>widen</b>	Widen character (public member function )

## Protected member functions

<b>init</b>	Initialize object (protected member function )
<b>move</b>	Move internals (protected member function )
<b>swap</b>	Swap internals (protected member function )
<b>set_rdbuf</b>	Set stream buffer (protected member function )

## Public member functions inherited from `ios_base`

<b>flags</b>	Get/set format flags (public member function )
<b>setf</b>	Set specific format flags (public member function )
<b>unsetf</b>	Clear specific format flags (public member function )
<b>precision</b>	Get/Set floating-point decimal precision (public member function )
<b>width</b>	Get/set field width (public member function )
<b>imbind</b>	Imbind locale (public member function )
<b>getloc</b>	Get current locale (public member function )
<b>xalloc</b>	Get new index for extensible array [static] (public static member function )
<b>iword</b>	Get integer element of extensible array (public member function )
<b>pword</b>	Get pointer element of extensible array (public member function )
<b>register_callback</b>	Register event callback function (public member function )
<b>sync_with_stdio</b>	Toggle synchronization with cstdio streams [static] (public static member function )

## /ios/basic\_ios/bad

public member function

### `std::basic_ios::bad`

`<iostream>`

`bool bad() const;`

#### Check whether badbit is set

Returns `true` if the `badbit` error state flag is set for the stream.

This flag is set by operations performed on the stream when an error occurs while read or writing data, generally causing the loss of integrity of the stream.

Notice that this function is not the exact opposite of `good`, which checks whether none of the error flags (eofbit, failbit and badbit) are set, and not only badbit:

<code>iostate</code> value (member constants)	indicates	functions to check state flags				
		<code>good()</code>	<code>eof()</code>	<code>fail()</code>	<code>bad()</code>	<code>rdstate()</code>
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). `goodbit` is zero, indicating that none of the other bits is set.

## Parameters

none

## Return Value

true if the stream's `badbit` error state flag is set.  
false otherwise.

## Data races

Accesses the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<code>basic_ios::good</code>	Check whether state of stream is good (public member function )
<code>basic_ios::fail</code>	Check whether failbit or badbit is set (public member function )
<code>basic_ios::eof</code>	Check whether eofbit is set (public member function )
<code>basic_ios::rdstate</code>	Get error state flags (public member function )
<code>basic_ios::clear</code>	Set error state flags (public member function )

## /ios/basic\_ios/~basic\_ios

public member function

### std::basic\_ios::~basic\_ios

<iostream>

`virtual ~basic_ios();`

## Destroy object

Destroys an object of this class.

Note that this does *not* destroy the associated *stream buffer*.

## Data races

The object is modified.

## Exception safety

**No-throw guarantee:** never throws exceptions.

## See also

## /ios/basic\_ios/basic\_ios

public member function

### std::basic\_ios::basic\_ios

<iostream>

```
initialization (1)    public: explicit basic_ios (basic_streambuf<char_type,traits_type>* sb);
default (2)           protected: basic_ios();
initialization (1)    public: explicit basic_ios (basic_streambuf<char_type,traits_type>* sb);
default (2)           protected: basic_ios();
copy (3)              basic_ios (const basic_ios&) = delete;
copy (3)              basic_ios& operator= (const basic_ios&) = delete;
```

## Construct object

The initialization constructor (1) initializes the stream object by calling `init(sb)`.

If invoked by a derived class using the default constructor (2), it constructs an object leaving its members uninitialized. In this case the object shall be explicitly initialized by calling `init` at some point before its first use or before it is destroyed (if never used).

The *copy constructor* (3) is explicitly deleted (as well as the *copy assignment* overload of `operator=`).

## Parameters

`sb`

pointer to a `basic_streambuf` object with the same template parameters as the `basic_ios` object.

`char_type` and `traits_type` are member types defined as aliases of the first and second class template parameters, respectively (see `basic_ios` types).

## Data races

The object pointed by *sb* may be accessed and/or modified.

## Exception safety

If an exception is thrown, the only side effects may come from accessing/modifying *sb*.

## See also

<a href="#">basic_ios::init</a>	Initialize object (protected member function )
<a href="#">ios_base::ios_base</a>	Construct object (public member function )

# /ios/basic\_ios/clear

public member function

## std::basic\_ios::clear

<iostream>

`void clear (iostate state = goodbit);`

### Set error state flags

Sets a new value for the stream's internal *error state flags*.

The current value of the flags is overwritten: All bits are replaced by those in *state*; If *state* is `goodbit` (which is zero) all error flags are cleared.

In the case that no *stream buffer* is associated with the stream when this function is called, the `badbit` flag is automatically set (no matter the value for that bit passed in argument *state*).

Note that changing the *state* may throw an exception, depending on the latest settings passed to member `exceptions`.

The current state can be obtained with member function `rdstate`.

## Parameters

*state*

An object of type `ios_base::iostate` that can take as value any combination of the following state flag member constants:

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). `goodbit` is zero, indicating that none of the other bits is set.

## Return Value

none

## Example

```
1 // clearing errors
2 #include <iostream>      // std::cout
3 #include <fstream>       // std::fstream
4
5 int main () {
6     char buffer [80];
7     std::fstream myfile;
8
9     myfile.open ("test.txt",std::fstream::in);
10
11    myfile << "test";
12    if (myfile.fail())
13    {
14        std::cout << "Error writing to test.txt\n";
15        myfile.clear();
16    }
17
18    myfile.getline (buffer,80);
19    std::cout << buffer << " successfully read from file.\n";
20
21    return 0;
22 }
```

In this example, `myfile` is open for input operations, but we perform an output operation on it, so `failbit` is set. The example calls then `clear` in order to remove the flag and allow further operations like `getline` to be attempted on `myfile`.

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set throw for that state.

## See also

<a href="#">basic_ios::fail</a>	Check whether failbit or badbit is set (public member function )
<a href="#">basic_ios::good</a>	Check whether state of stream is good (public member function )
<a href="#">basic_ios::bad</a>	Check whether badbit is set (public member function )
<a href="#">basic_ios::eof</a>	Check whether eofbit is set (public member function )
<a href="#">basic_ios::rdstate</a>	Get error state flags (public member function )

# /ios/basic\_ios/copyfmt

public member function

## std::basic\_ios::copyfmt

<iostream>

### Copy formatting information

Copies the values of all the internal members of *rhs* (except the *state flags* and the associated *stream buffer*) to the corresponding members of *\*this*.

After the call, the following member functions return the same for *rhs* and *\*this*:

element	description
<code>flags</code>	format flags
<code>width</code>	field width
<code>precision</code>	precision
<code>getloc</code>	selected locale
<code>iarray</code>	internal extensible array *
<code>parray</code>	internal extensible array *
<code>fill</code>	fill character
<code>tie</code>	tied stream
<code>exceptions</code>	exceptions mask (last to be copied, see below)

\* Each stream object keeps its own copy of the *internal extensible array* (*iword*, *pword*): Its contents are copied, not just a pointer to it.

\* Each stream object keeps its own copy of the *internal extensible array* (*iword*, *pword*): Its contents are copied, not just a pointer to it.

If any of the pointer values to be copied points to objects stored outside *rhs* and those objects are destroyed when *rhs* is destroyed, *\*this* stores instead pointers to newly constructed copies of these objects.

Calling this function invokes all functions registered through member `register_callback` twice: First, before the copying process starts, the function calls each registered callback *fn* using `(*fn)(erase_event, *this, index)`. Then, at the end, right before the `exceptions mask` is copied, the function calls each registered callback *fn* with `(*fn)(copyfmt_event, *this, index)` (this second round can be used, for example, to access and modify the copied *internal extensible array*).

## Parameters

*rhs*

Stream object whose members are copied to *\*this*.

## Return Value

*\*this*

## Example

```
1 // copying formatting information
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::ofstream
4
5 int main () {
6     std::ofstream filestr;
7     filestr.open ("test.txt");
8
9     std::cout.fill ('*');
10    std::cout.width (10);
11    filestr.copyfmt (std::cout);
12
13    std::cout << 40;
14    filestr << 40;
15
16    return 0;
17 }
```

This example outputs a number formatted in the same way to both `cout` and a file called "test.txt":

\*\*\*\*\*40

## Data races

Modifies the stream object (*\*this*), and accesses *rhs*.

Concurrent access to any of the objects may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<b>basic_ios::tie</b>	Get/set tied stream (public member function )
<b>basic_ios::fill</b>	Get/set fill character (public member function )
<b>ios_base::width</b>	Get/set field width (public member function )
<b>ios_base::fmtflags</b>	Type for stream format flags (public member type )

## /ios/basic\_ios/eof

public member function

### std::basic\_ios::eof

<iostream>

`bool eof() const;`

**Check whether eofbit is set**

Returns true if the eofbit error state flag is set for the stream.

This flag is set by all standard input operations when the End-of-File is reached in the sequence associated with the stream.

Note that the value returned by this function depends on the last operation performed on the stream (and not on the next).

Operations that attempt to read at the End-of-File fail, and thus both the eofbit and the failbit end up set. This function can be used to check whether the failure is due to reaching the End-of-File or to some other reason.

#### Parameters

none

#### Return Value

true if the stream's eofbit error state flag is set (which signals that the End-of-File has been reached by the last input operation).  
false otherwise.

#### Example

```

1 // ios::eof example
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::ifstream
4
5 int main () {
6
7   std::ifstream is("example.txt");    // open file
8
9   char c;
10  while (is.get(c))                // loop getting single characters
11    std::cout << c;
12
13  if (is.eof())                   // check for EOF
14    std::cout << "[EOF reached]\n";
15  else
16    std::cout << "[error reading]\n";
17
18  is.close();                     // close file
19
20  return 0;
21 }
```

#### Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

#### See also

<b>basic_ios::good</b>	Check whether state of stream is good (public member function )
<b>basic_ios::fail</b>	Check whether failbit or badbit is set (public member function )
<b>basic_ios::bad</b>	Check whether badbit is set (public member function )
<b>basic_ios::rdstate</b>	Get error state flags (public member function )
<b>basic_ios::clear</b>	Set error state flags (public member function )

## /ios/basic\_ios/exceptions

public member function

### std::basic\_ios::exceptions

<iostream>

```

get(1) iostate exceptions() const;
set(2) void exceptions (iostate except);
```

**Get/set exceptions mask**

The first form (1) returns the current *exception mask* for the stream.

The second form (2) sets a new *exception mask* for the stream and clears the stream's *error state flags* (as if member `clear()` was called).

The *exception mask* is an internal value kept by all stream objects specifying for which state flags an exception of member type `failure` (or some derived type) is thrown when set. This mask is an object of member type `iostate`, which is a value formed by any combination of the following member constants:

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on I/O operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on I/O operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator), so that the stream throws when any of the selected *error state flags* is set.

`goodbit` is zero, indicating that no exceptions shall be thrown when an *error state flags* is set.

All streams have `goodbit` by default (they do not throw exceptions due to *error state flags* being set).

## Parameters

except

A bitmask value of member type `iostate` formed by a combination of error state flag bits to be set (`badbit`, `eofbit` and/or `failbit`), or set to `goodbit` (or zero).

## Return Value

The first form (1) returns a bitmask of member type `iostate` representing the existing exception mask before the call to this member function.

## Example

```
1 // basic_ios::exceptions
2 #include <iostream>           // std::cerr
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream file;
7     file.exceptions ( std::ifstream::failbit | std::ifstream::badbit );
8     try {
9         file.open ("test.txt");
10        while (!file.eof()) file.get();
11    }
12    catch (std::ifstream::failure e) {
13        std::cerr << "Exception opening/reading file";
14    }
15
16    file.close();
17
18    return 0;
19 }
```

## Data races

Accesses (1) or modifies (2) the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

`basic_ios::rdstate` Get error state flags (public member function )

# /ios/basic\_ios/fail

public member function

**std::basic\_ios::fail**

`<iostream>`

`bool fail() const;`

**Check whether failbit or badbit is set**

Returns true if either (or both) the `failbit` or the `badbit` *error state flags* is set for the stream.

At least one of these flags is set when an error occurs during an input operation.

`failbit` is generally set by an operation when the error is related to the internal logic of the operation itself, and further operations on the stream may be possible. While `badbit` is generally set when the error involves the loss of integrity of the stream, which is likely to persist even if a different operation is attempted on the stream. `badbit` can be checked independently by calling member function `bad`:

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on I/O operation	false	false	true	false	failbit

`badbit` |Read/writing error on i/o operation |false |false|true |true |badbit |

eofbit, failbit and badbit are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator).

goodbit is zero, indicating that none of the other bits is set.

Note that failing to read due to reaching the *End-of-File* sets both the eofbit and the failbit.

This function is a synonym of `basic_ios::operator!=`.

## Parameters

none

## Return Value

true if badbit and/or failbit are set.

false otherwise.

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<code>ios_base::iostate</code>	Type for stream state flags ( <a href="#">public member type</a> )
<code>basic_ios::good</code>	Check whether state of stream is good ( <a href="#">public member function</a> )
<code>basic_ios::bad</code>	Check whether badbit is set ( <a href="#">public member function</a> )
<code>basic_ios::eof</code>	Check whether eofbit is set ( <a href="#">public member function</a> )
<code>basic_ios::rdstate</code>	Get error state flags ( <a href="#">public member function</a> )
<code>basic_ios::clear</code>	Set error state flags ( <a href="#">public member function</a> )

## /ios/basic\_ios/fill

public member function

### `std::basic_ios::fill`

`<iostream>`

```
get (1) char_type fill() const;
set (2) char_type fill (char_type fillch);
```

#### Get/set fill character

The first form (1) returns the *fill character*.

The second form (2) sets *fillch* as the new *fill character* and returns the *fill character* used before the call.

The *fill character* is the character used by output insertion functions to fill spaces when padding results to the *field width*.

The parametric manipulator `setfill` can also be used to set the *fill character*.

## Parameters

`fillch`

the new *fill character*.

Member type `char_type` is the type of characters used by the stream (i.e., its first class template parameter, `charT`).

## Return Value

The value of the *fill character* before the call.

Member type `char_type` is the type of characters used by the stream (i.e., its first class template parameter, `charT`).

## Example

```
1 // using the fill character
2 #include <iostream>      // std::cout
3
4 int main () {
5     char prev;
6
7     std::cout.width (10);
8     std::cout << 40 << '\n';
9
10    prev = std::cout.fill ('x');
11    std::cout.width (10);
12    std::cout << 40 << '\n';
13
14    std::cout.fill(prev);
15
16    return 0;
17 }
```

Output:

```
40
xxxxxxx40
```

## Data races

Accesses (1) or modifies (2) the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">setfill</a>	Set fill character ( <a href="#">function</a> )
<a href="#">ios_base::width</a>	Get/set field width ( <a href="#">public member function</a> )

## /ios/basic\_ios/good

public member function

### **std::basic\_ios::good**

<iostream>

```
bool good() const;
```

#### Check whether state of stream is good

Returns true if none of the stream's *error state flags* (eofbit, failbit and badbit) is set.

This function behaves as if defined as:

```
1 bool basic_ios::good() const {
2     return rdstate() == goodbit;
3 }
```

Notice that this function is not the exact opposite of member [bad](#), which only checks whether the [badbit](#) flag is set.

Whether specific error flags are set, can be checked with member functions [eof](#), [fail](#), and [bad](#):

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
goodbit	No errors (zero value <a href="#">iostate</a> )	true	false	false	false	goodbit
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit
failbit	Logical error on i/o operation	false	false	true	false	failbit
badbit	Read/writing error on i/o operation	false	false	true	true	badbit

eofbit, failbit and badbit are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). goodbit is zero, indicating that none of the other bits is set.

## Parameters

none

## Return Value

true if none of the stream's state flags are set.

false if any of the stream's state flags are set (badbit, eofbit or failbit).

## Example

```
1 // error state flags
2 #include <iostream>           // std::cout, std::ios
3 #include <sstream>            // std::stringstream
4
5 void print_state (const std::ios& stream) {
6     std::cout << " good()=" << stream.good();
7     std::cout << " eof()=" << stream.eof();
8     std::cout << " fail()=" << stream.fail();
9     std::cout << " bad()=" << stream.bad();
10 }
11
12 int main () {
13     std::stringstream stream;
14
15     stream.clear (stream.goodbit);
16     std::cout << "goodbit:"; print_state(stream); std::cout << '\n';
17
18     stream.clear (stream.eofbit);
19     std::cout << "eofbit:"; print_state(stream); std::cout << '\n';
20
21     stream.clear (stream.failbit);
22     std::cout << "failbit:"; print_state(stream); std::cout << '\n';
23
24     stream.clear (stream.badbit);
25     std::cout << " badbit:"; print_state(stream); std::cout << '\n';
26
27     return 0;
28 }
```

Output:

```
goodbit: good()=1 eof()=0 fail()=0 bad()=0
eofbit: good()=0 eof()=1 fail()=0 bad()=0
failbit: good()=0 eof()=0 fail()=1 bad()=0
badbit: good()=1 eof()=0 fail()=1 bad()=1
```

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<a href="#">basic_ios::fail</a>	Check whether failbit or badbit is set (public member function )
<a href="#">basic_ios::bad</a>	Check whether badbit is set (public member function )
<a href="#">basic_ios::eof</a>	Check whether eofbit is set (public member function )
<a href="#">basic_ios::rdstate</a>	Get error state flags (public member function )
<a href="#">basic_ios::setstate</a>	Set error state flag (public member function )
<a href="#">basic_ios::clear</a>	Set error state flags (public member function )

## /ios/basic\_ios/imbue

public member function

### std::basic\_ios::imbue

<iostream>

**locale imbue (const locale& loc);**

#### Imbue locale

Associates *loc* to both the stream and its associated *stream buffer* (if any) as the new locale object to be used with locale-sensitive operations.

This function calls its inherited homonym *ios\_base::imbue(loc)* and, if the stream is associated with a *stream buffer*, it also calls *rdbuf() ->pubimbue(loc)*.

All callback functions registered with member *register\_callback* are called by *ios\_base::imbue*.

## Parameters

*loc*

locale object to be imbued as the new locale for the stream.

## Return value

The *locale* object associated with the stream before the call.

## Example

```
1 // imbue example
2 #include <iostream>      // std::cout
3 #include <locale>        // std::locale
4
5 int main()
6 {
7     std::locale mylocale(""); // get global locale
8     std::cout.imbue(mylocale); // imbue global locale
9     std::cout << 3.14159 << '\n';
10    return 0;
11 }
```

This code writes a floating point number using the global locale given by the environment. For example, in a system configured with a Spanish locale as default, this could write the number using a comma decimal separator:

```
3,14159
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">ios_base::imbue</a>	Imbue locale (public member function )
<a href="#">ios_base::getloc</a>	Get current locale (public member function )

## /ios/basic\_ios/init

protected member function

### std::basic\_ios::init

<iostream>

```
protected:  
void init (basic_streambuf<char_type,traits_type*>* sb);
```

#### Initialize object

Initializes the values of the stream's internal flags and member variables.

Derived classes are expected to call this protected member function at some point before its first use or before its destruction (generally, during construction).

The internal state is initialized in such a way that each of these members return the following values:

member function	return value
rdbuf	sb
tie	0
rdstate	goodbit if sb is not a null pointer, badbit otherwise
exceptions	goodbit
flags	skipws   dec
width	0
precision	6
fill	' ' (whitespace)
getloc	a copy of locale()

On initialization, the *internal extensible array* (`iword`, `pword`) is empty.

#### Parameters

sb

Pointer to a `basic_streambuf` object with the same template parameters as the `basic_ios` object.

`char_type` and `traits_type` are member types defined as aliases of the first and second class template parameters, respectively (see `basic_ios` types).

#### Return Value

none

#### Data races

Modifies the stream object. The object pointed by `sb` may be accessed and/or modified.

Concurrent access to the same stream object or stream buffer may cause data races.

#### Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

#### See also

[basic\\_ios::basic\\_ios](#) | Construct object (public member function )

## /ios/basic\_ios/move

protected member function

### std::basic\_ios::move

<iostream>

```
void move (basic_ios& x);  
void move (basic_ios&& x);
```

#### Move internals

Transfers all internal members of `x` to `*this`, except the associated *stream buffer* (`rdbuf` returns a *null pointer* after the call).

`x` is left in an unspecified but valid state, except that it is not *tied* (`tie` returns always a *null pointer*) and its associated *stream buffer* is unchanged (`rdbuf` returns the same as before the call).

Derived classes can call this function to implement *move semantics*.

#### Parameters

x

Stream object whose members are moved to `*this`.

#### Return Value

none

#### Data races

Modifies both stream objects (`*this` and `x`).

Concurrent access to any of these stream objects may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, both streams are in a valid state.

## See also

<a href="#">basic_ios::swap</a>	Swap internals (protected member function )
---------------------------------	---

## /ios/basic\_ios/narrow

public member function

### std::basic\_ios::narrow

<iostream>

`char narrow (char_type wc, char default) const;`

#### Narrow character

Returns the transformation of character `wc` (generally, of a wide character type) to its equivalent of type `char`.

This function returns the result of calling the `ctype::narrow` facet of the `locale` object currently *imbued* in the stream.

## Parameters

`wc`

Wide character to be *narrowed*.

Member type `char_type` is the type of characters used by the stream (i.e., its first class template parameter, `charT`).

`default`

Character returned if `wc` has no standard equivalent.

## Return Value

The `char` equivalent of `c`.

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<a href="#">basic_ios::widen</a>	Widen character (public member function )
----------------------------------	---

<a href="#">ctype::widen</a>	Widen character(s) (public member function )
------------------------------	--

## /ios/basic\_ios/operator\_bool

public member function

### std::basic\_ios::operator bool

<iostream>

`operator void*() const;`  
`explicit operator bool() const;`

#### Evaluate stream

Returns whether an error flag is set (either `failbit` or `badbit`).

Notice that this function does not return the same as member `good`, but the opposite of member `fail`.

The function returns a *null pointer* if at least one of these error flags is set, and some other value otherwise.

The function returns `false` if at least one of these error flags is set, and `true` otherwise.

## Parameters

none

## Return Value

A *null pointer* if at least one of `failbit` or `badbit` is set. Some other value otherwise.

`true` if none of `failbit` or `badbit` is set.

`false` otherwise.

## Example

```
1 // evaluating a stream
2 #include <iostream>           // std::cerr
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream is;
7     is.open ("test.txt");
```

```

8 if (is) {
9     // read file
10 }
11 else {
12     std::cerr << "Error opening 'test.txt'\n";
13 }
14 return 0;
15 }

```

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<b>basic_ios::fail</b>	Check whether failbit or badbit is set (public member function )
<b>basic_ios::operator!</b>	Evaluate stream (not) (public member function )

## /ios/basic\_ios/operator\_not

public member function

### std::basic\_ios::operator!

<iostream>

bool operator!() const;

#### Evaluate stream (not)

Returns true if no error flag is set (either failbit or badbit), and false otherwise.

This is equivalent to calling member `fail`.

## Parameters

none

## Return Value

true if either `failbit` or `badbit` is set.  
false otherwise.

## Example

```

1 // evaluating a stream (not)
2 #include <iostream>           // std::cout
3 #include <fstream>           // std::ifstream
4
5 int main () {
6     std::ifstream is;
7     is.open ("test.txt");
8     if (!is)
9         std::cerr << "Error opening 'test.txt'\n";
10    return 0;
11 }

```

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<b>basic_ios::fail</b>	Check whether failbit or badbit is set (public member function )
<b>basic_ios::good</b>	Check whether state of stream is good (public member function )

## /ios/basic\_ios/rdbuf

public member function

### std::basic\_ios::rdbuf

<iostream>

```

get (1) basic_streambuf<char_type,traits_type>* rdbuf() const;
set (2) basic_streambuf<char_type,traits_type>* rdbuf (basic_streambuf<char_type,traits_type>* sb);

```

**Get/set stream buffer**

The first form (1) returns a pointer to the *stream buffer* object currently associated with the stream.

The second form (2) also sets the object pointed by *sb* as the *stream buffer* associated with the stream and clears the *error state flags*.

If *sb* is a *null pointer*, the function automatically sets the *badbit* *error state flags* (which may throw an exception if member *exceptions* has been passed *badbit*).

Some derived stream classes (such as *string streams* and *file streams*) maintain their own *internal stream buffer*, to which they are *associated* on construction. Calling this function to change the *associated stream buffer* shall have no effect on that *internal stream buffer*: the stream will have an *associated stream buffer* which is different from its *internal stream buffer* (although input/output operations on streams always use the *associated stream buffer*, as returned by this member function).

## Parameters

*sb* Pointer to a *basic\_streambuf* object with the same template parameters as the *basic\_ios* object.

*char\_type* and *traits\_type* are member types defined as aliases of the first and second class template parameters, respectively (see *basic\_ios* types).

## Return Value

A pointer to the *stream buffer* object associated with the stream before the call.

## Example

```
1 // redirecting cout's output through its stream buffer
2 #include <iostream>           // std::streambuf, std::cout
3 #include <fstream>            // std::ofstream
4
5 int main () {
6     std::streambuf *psbuf, *backup;
7     std::ofstream filestr;
8     filestr.open ("test.txt");
9
10    backup = std::cout.rdbuf();      // back up cout's streambuf
11
12    psbuf = filestr.rdbuf();        // get file's streambuf
13    std::cout.rdbuf(psbuf);         // assign streambuf to cout
14
15    std::cout << "This is written to the file";
16
17    std::cout.rdbuf(backup);        // restore cout's original streambuf
18
19    filestr.close();
20
21    return 0;
22 }
```

This example uses both function forms: first to get a pointer to a file's *basic\_streambuf* object and then to assign it to *cout*.

## Data races

Accesses (1) or modifies (2) the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

It throws an exception of member type *failure* if *sb* is a *null pointer* and member *exceptions* was set to throw for *badbit*.

## See also

**streambuf** Base buffer class for streams (class )

# /ios/basic\_ios/rdstate

public member function

## std::basic\_ios::rdstate

<iostream>

*iostate* *rdstate()* const;

### Get error state flags

Returns the current internal *error state flags* of the stream.

The internal *error state flags* are automatically set by calls to input/output functions on the stream to signal certain errors.

## Parameters

none

## Return Value

An object of type *ios\_base::iostate* that can contain any combination of the following state flag member constants:

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
goodbit	No errors (zero value <i>iostate</i> )	true	false	false	false	goodbit

<b>eofbit</b>	End-of-File reached on input operation	false	true	false	false	eofbit
<b>failbit</b>	Logical error on i/o operation	false	false	true	false	failbit
<b>badbit</b>	Read/writing error on i/o operation	false	false	true	true	badbit

eofbit, failbit and badbit are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). goodbit is zero, indicating that none of the other bits is set.

## Example

```

1 // getting the state of stream objects
2 #include <iostream>           // std::cerr
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream is;
7     is.open ("test.txt");
8     if ( (is.rdstate() & std::ifstream::failbit) != 0 )
9         std::cerr << "Error opening 'test.txt'\n";
10    return 0;
11 }
```

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<b>basic_ios::fail</b>	Check whether failbit or badbit is set ( <a href="#">public member function</a> )
<b>basic_ios::good</b>	Check whether state of stream is good ( <a href="#">public member function</a> )
<b>basic_ios::bad</b>	Check whether badbit is set ( <a href="#">public member function</a> )
<b>basic_ios::eof</b>	Check whether eofbit is set ( <a href="#">public member function</a> )
<b>basic_ios::clear</b>	Set error state flags ( <a href="#">public member function</a> )

## /ios/basic\_ios/set\_rdbuf

protected member function

**std::basic\_ios::set\_rdbuf** <iostream>

```
void set_rdbuf (basic_streambuf<char_type,traits_type>* sb);
```

### Set stream buffer

Sets *sb* as the *stream buffer* associated with the stream, without altering the *control state flag* (*rdstate*).

*sb* shall not be a *null pointer*.

Derived classes can call this function to change the *stream buffer*.

## Parameters

*sb*  
Pointer to a *basic\_streambuf* object with the same template parameters as the *basic\_ios* object.  
This shall not be a *null pointer*.  
*char\_type* and *traits\_type* are member types defined as aliases of the first and second class template parameters, respectively (see *basic\_ios* types).

## Return Value

none

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<b>basic_ios::rdbuf</b>	Get/set stream buffer ( <a href="#">public member function</a> )
-------------------------	--

## /ios/basic\_ios/setstate

public member function

**std::basic\_ios::setstate** <iostream>

```
void setstate (iostate state);
```

### Set error state flag

Modifies the current internal *error state flags* by combining the current flags with those in argument *state* (as if performing a bitwise OR operation).

Any error bitflag already set is not cleared. See member [clear](#) for a similar function that does.

In the case that no *stream buffer* is associated with the stream when this function is called, the *badbit* flag is automatically set (no matter the value for that bit passed in argument *state*).

Note that changing the *state* may throw an exception, depending on the latest settings passed to member [exceptions](#).

The current state can be obtained with member function [rdstate](#).

This function behaves as if defined as:

```
1 void basic_ios::setstate (iostate state) {  
2     clear(rdstate()|state);  
3 }
```

## Parameters

### state

An object of type [ios\\_base::iostate](#) that can take as value any combination of the following member constants:

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
goodbit	No errors (zero value <i>iostate</i> )	true	false	false	false	goodbit
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit
failbit	Logical error on i/o operation	false	false	true	false	failbit
badbit	Read/writing error on i/o operation	false	false	true	true	badbit

eofbit, failbit and badbit are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). goodbit is zero, indicating that none of the other bits is set.

## Return Value

none

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

It throws an exception of member type [failure](#) if the resulting *error state flag* is not goodbit and member [exceptions](#) was set throw for that state.

## See also

<a href="#">basic_ios::fail</a>	Check whether failbit or badbit is set ( <a href="#">public member function</a> )
<a href="#">basic_ios::good</a>	Check whether state of stream is good ( <a href="#">public member function</a> )
<a href="#">basic_ios::bad</a>	Check whether badbit is set ( <a href="#">public member function</a> )
<a href="#">basic_ios::eof</a>	Check whether eofbit is set ( <a href="#">public member function</a> )
<a href="#">basic_ios::rdstate</a>	Get error state flags ( <a href="#">public member function</a> )
<a href="#">basic_ios::clear</a>	Set error state flags ( <a href="#">public member function</a> )

## /ios/basic\_ios/swap

protected member function

### std::basic\_ios::swap

<iostream>

```
void swap (basic_ios& x) noexcept;
```

#### Swap internals

Exchanges all internal members between *x* and *\*this*, except the pointers to the associated *stream buffers*: *rdbuf* shall return the same in both objects as before the call.

Derived classes can call this function to implement custom *swap* functions.

## Parameters

x

Another stream object of the same type.

## Return Value

none

## Data races

Modifies both stream objects (*\*this* and *x*).

## Exception safety

No-throw guarantee: this member function never throws exceptions.

## See also

[basic\\_ios::move](#)

Move internals (protected member function )

# /ios/basic\_ios/tie

public member function

## std::basic\_ios::tie

<iostream>

```
get (1) basic_ostream<char_type,traits_type>* tie() const;
set (2) basic_ostream<char_type,traits_type>* tie (basic_ostream<char_type,traits_type>* tiestr);
```

### Get/set tied stream

The first form (1) returns a pointer to the tied output stream.

The second form (2) ties the object to *tiestr* and returns a pointer to the stream tied before the call, if any.

The *tied stream* is an output stream object which is *flushed* before each i/o operation in this stream object.

By default, `cin` is tied to `cout`, and `wcin` to `wcout`. Library implementations may tie other standard streams on initialization.

By default, the standard narrow streams `cin` and `cerr` are tied to `cout`, and their wide character counterparts (`wcin` and `wcerr`) to `wcout`. Library implementations may also tie `clog` and `wclog`.

## Parameters

*tiestr*

An output stream object.

`char_type` and `traits_type` are member types defined as aliases of the first and second class template parameters, respectively (see [basic\\_ios types](#)).

## Return Value

A pointer to the stream object tied before the call, or a *null pointer* in case the stream was not tied.

## Example

```
1 // redefine tied object
2 #include <iostream>           // std::ostream, std::cout, std::cin
3 #include <fstream>            // std::ofstream
4
5 int main () {
6     std::ostream *prevstr;
7     std::ofstream ofs;
8     ofs.open ("test.txt");
9
10    std::cout << "tie example:\n";
11
12    *std::cin.tie() << "This is inserted into cout";
13    prevstr = std::cin.tie (&ofs);
14    *std::cin.tie() << "This is inserted into the file";
15    std::cin.tie (prevstr);
16
17    ofs.close();
18
19    return 0;
20 }
```

Output:

```
tie example:
This is inserted into cout
```

## Data races

Accesses (1) or modifies (2) the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

Basic guarantee: if an exception is thrown, the stream is in a valid state.

[basic\\_ios::rdbuf](#)

Get/set stream buffer (public member function )

# /ios/basic\_ios/widen

public member function

## std::basic\_ios::widen

<iostream>

```
char_type widen (char c) const;
```

### Widen character

Returns the transformation of the narrow character *c* to its equivalent of type `char_type` (generally, a wide character type).

This function returns the result of calling the `ctype::widen` facet of the `locale` object currently *imbued* in the stream.

### Parameters

*c*

Narrow character to be *widened*.

### Return Value

The `char_type` equivalent of *c*.

Member type `char_type` is the type of characters used by the stream (i.e., its first class template parameter, `charT`).

### Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

### See also

<code>basic_ios::narrow</code>	Narrow character ( <a href="#">public member function</a> )
<code>ctype::narrow</code>	Narrow character(s) ( <a href="#">public member function</a> )

## /ios/boolalpha

function  
**std::boolalpha** `<iostream>`

```
ios_base& boolalpha (ios_base& str);
```

### Alphanumerical bool values

Sets the `boolalpha` format flag for the *str* stream.

When the `boolalpha` format flag is set, bool values are inserted/extracted by their textual representation: either `true` or `false`, instead of integral values.

This flag can be unset with the `noboolalpha` manipulator.

For standard streams, the `boolalpha` flag is **not set** on initialization.

### Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

### Return Value

Argument *str*.

### Example

```
1 // modify boolalpha flag
2 #include <iostream>      // std::cout, std::boolalpha, std::noboolalpha
3
4 int main () {
5     bool b = true;
6     std::cout << std::boolalpha << b << '\n';
7     std::cout << std::noboolalpha << b << '\n';
8     return 0;
9 }
```

Output:

```
true
1
```

### Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

### Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">noboolalpha</a>	No alphanumeric bool values (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )
<a href="#">setiosflags</a>	Set format flags (function )

## /ios/dec

function  
**std::dec** <iostream>

`ios_base& dec (ios_base& str);`

### Use decimal base

Sets the basefield format flag for the `str` stream to `dec`.

When `basefield` is set to `dec`, integer values inserted into the stream are expressed in decimal base (i.e., radix 10). For input streams, extracted values are also expected to be expressed in decimal base when this flag is set.

The basefield format flag can take any of the following values (each with its own manipulator):

flag value	effect when set
<code>dec</code>	read/write integer values using decimal base format.
<code>hex</code>	read/write integer values using hexadecimal base format.
<code>oct</code>	read/write integer values using octal base format.

For standard streams, the basefield flag is set to `dec` on initialization.

## Parameters

`str`

Stream object whose basefield *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

Argument `str`.

## Example

```
1 // modify basefield
2 #include <iostream>      // std::cout, std::dec, std::hex, std::oct
3
4 int main () {
5     int n = 70;
6     std::cout << std::dec << n << '\n';
7     std::cout << std::hex << n << '\n';
8     std::cout << std::oct << n << '\n';
9     return 0;
10 }
```

Output:

```
70
46
106
```

## Data races

Modifies `str`. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, `str` is in a valid state.

## See also

<a href="#">hex</a>	Use hexadecimal base (function )
<a href="#">oct</a>	Use octal base (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )

## /ios/defaultfloat

function  
**std::defaultfloat** <iostream>

`ios_base& defaultfloat (ios_base& str);`

## Use default floating-point notation

Sets the floatfield format flag for the *str* stream to defaultfloat.

When floatfield is set to defaultfloat, floating-point values are written using the default notation: the representation uses as many meaningful digits as needed up to the stream's *decimal precision* ([precision](#)), counting both the digits before and after the decimal point (if any).

The floatfield format flag is both a selective and a toggle flag: it can take any of the following values, or none:

flag value	effect when set
<code>fixed</code>	write floating-point values in fixed-point notation.
<code>scientific</code>	write floating-point values in scientific notation.
<code>hexfloat</code>	write floating-point values in hexadecimal format. The value of this is the same as (fixed scientific)
<code>defaultfloat</code>	write floating-point values in default floating-point notation. This is the value by default (same as none, before any other floatfield bit is set).

For standard streams, the floatfield format flag is set to this value (defaultfloat) on initialization.

## Parameters

*str*

Stream object whose floatfield *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // hexfloat floatfield
2 #include <iostream>      // std::cout, std::hexfloat, std::defaultfloat
3
4 int main () {
5     double a = 3.1415926534;
6     double b = 2006.0;
7     double c = 1.0e-10;
8
9     std::cout.precision(5);
10
11    std::cout << "hexfloat:\n" << std::hexfloat;
12    std::cout << a << '\n' << b << '\n' << c << '\n';
13
14    std::cout << '\n';
15
16    std::cout << "defaultfloat:\n" << std::defaultfloat;
17    std::cout << a << '\n' << b << '\n' << c << '\n';
18
19    return 0;
20 }
```

Possible output:

```
hexfloat:
0x1.921fb5p+1
0x1.f58000p+10
0x1.b7cdfep-34

defaultfloat:
3.14159
2006
1e-010
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<code>fixed</code>	Use fixed floating-point notation ( <a href="#">function</a> )
<code>scientific</code>	Use scientific floating-point notation ( <a href="#">function</a> )
<code>hexfloat</code>	Use hexadecimal floating-point format ( <a href="#">function</a> )
<code>ios_base::flags</code>	Get/set format flags ( <a href="#">public member function</a> )
<code>ios_base::setf</code>	Set specific format flags ( <a href="#">public member function</a> )
<code>ios_base::unsetf</code>	Clear specific format flags ( <a href="#">public member function</a> )

## /ios/fixed

function

`std::fixed`

`<iostream>`

```
ios_base& fixed (ios_base& str);
```

### Use fixed floating-point notation

Sets the floatfield format flag for the `str` stream to `fixed`.

When `floatfield` is set to `fixed`, floating-point values are written using fixed-point notation: the value is represented with exactly as many digits in the decimal part as specified by the `precision` field (`precision`) and with no exponent part.

The `floatfield` format flag is both a selective and a toggle flag: it can take one, both or none of the following values:

flag value	effect when set
<code>fixed</code>	write floating-point values in fixed-point notation
<code>scientific</code>	write floating-point values in scientific notation.
<code>(none)</code>	write floating-point values in default floating-point notation.

The default notation `(none)` is a different floatfield value than either `fixed` or `scientific`. The default notation can be selected by calling `str.unsetf(ios_base::floatfield)`.

For standard streams, no `floatfield` is set on initialization (default notation).

The `floatfield` format flag is both a selective and a toggle flag: it can take any of the following values, or none:

flag value	effect when set
<code>fixed</code>	write floating-point values in fixed-point notation.
<code>scientific</code>	write floating-point values in scientific notation.
<code>hexfloat</code>	write floating-point values in hexadecimal format. The value of this is the same as <code>(fixed scientific)</code>
<code>defaultfloat</code>	write floating-point values in default floating-point notation. This is the value by default (same as <code>none</code> , before any other <code>floatfield</code> bit is set).

For standard streams, the `floatfield` format flag is set to `defaultfloat` on initialization.

The `precision` field can be modified using member `precision`.

Notice that the treatment of the `precision` field differs between the default floating-point notation and the fixed and scientific notations (see `precision`). On the default floating-point notation, the `precision` field specifies the maximum number of meaningful digits to display both before and after the decimal point, while in both the fixed and scientific notations, the `precision` field specifies exactly how many digits to display *after* the decimal point, even if they are trailing decimal zeros.

## Parameters

`str`

Stream object whose `floatfield` format flag is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

Argument `str`.

## Example

```
1 // modify floatfield
2 #include <iostream>      // std::cout, std::fixed, std::scientific
3
4 int main () {
5     double a = 3.1415926534;
6     double b = 2006.0;
7     double c = 1.0e-10;
8
9     std::cout.precision(5);
10
11    std::cout << "default:\n";
12    std::cout << a << '\n' << b << '\n' << c << '\n';
13
14    std::cout << '\n';
15
16    std::cout << "fixed:\n" << std::fixed;
17    std::cout << a << '\n' << b << '\n' << c << '\n';
18
19    std::cout << '\n';
20
21    std::cout << "scientific:\n" << std::scientific;
22    std::cout << a << '\n' << b << '\n' << c << '\n';
23    return 0;
24 }
```

Possible output:

```
default:
3.1416
2006
1e-010

fixed:
3.14159
2006.00000
0.00000

scientific:
3.14159e+000
2.00600e+003
1.00000e-010
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">scientific</a>	Use scientific floating-point notation (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/fpos

class template

### std::fpos

<iostream>

`template <class stateT> class fpos;`

#### Stream position class template

Class template used as a template for types to indicate positions in streams. The template depends on the state type *stateT*.

The details of this class are implementation-defined, but has at least two members:

```
1 stateT state() const;
2 void state(stateT);
```

Each either getting or setting the value of the state type (*stateT*) kept internally by the object.

Objects of any *fpos* instanced type support construction and conversion from *int*, and allow consistent conversions to/from values of type *streamoff* (as well as being added or subtracted values of this type).

Two objects of this type can be compared with operators == and !=. They can also be subtracted, which yields a value of type *streamoff*.

The synonym types *streampos* and *wstreampos* are instantiations of this template with *mbstate\_t* as *stateT*.

## See also

<a href="#">streampos</a>	Stream position type (type )
<a href="#">wstreampos</a>	Wide stream position type (type )

## /ios/hex

function

### std::hex

<iostream>

`ios_base& hex (ios_base& str);`

#### Use hexadecimal base

Sets the basefield format flag for the *str* stream to hex.

When basefield is set to hex, integer values inserted into the stream are expressed in hexadecimal base (i.e., radix 16). For input streams, extracted values are also expected to be expressed in hexadecimal base when this flag is set.

The basefield format flag can take any of the following values (each with its own manipulator):

flag value	effect when set
<code>dec</code>	read/write integer values using decimal base format.
<code>hex</code>	read/write integer values using hexadecimal base format.
<code>oct</code>	read/write integer values using octal base format.

Notice that the basefield flag only affects the insertion/extraction of integer values (floating-point values are always interpreted in decimal base).

Notice also that no base prefix is implicitly prepended to the number unless the *showbase* format flag is set.

For standard streams, the basefield flag is set to `dec` on initialization.

## Parameters

*str*

Stream object whose basefield *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify basefield
2 #include <iostream>      // std::cout, std::dec, std::hex, std::oct
3
4 int main () {
5     int n = 70;
6     std::cout << std::dec << n << '\n';
7     std::cout << std::hex << n << '\n';
8     std::cout << std::oct << n << '\n';
9     return 0;
10 }
```

Output:

```
70
46
106
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">dec</a>	Use decimal base (function )
<a href="#">oct</a>	Use octal base (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )

## /ios/hexfloat

function  
**std::hexfloat** <iostream>

```
ios_base& hexfloat (ios_base& str);
```

### Use hexadecimal floating-point format

Sets the *floatfield* format flag for the *str* stream to *hexfloat*.

When *floatfield* is set to *hexfloat*, floating-point values are written using hexadecimal format.

The *floatfield* format flag is both a selective and a toggle flag: it can take any of the following values, or none:

flag value	effect when set
<code>fixed</code>	write floating-point values in fixed-point notation.
<code>scientific</code>	write floating-point values in scientific notation.
<code>hexfloat</code>	write floating-point values in hexadecimal format. The value of this is the same as ( <code>fixed scientific</code> )
<code>defaultfloat</code>	write floating-point values in default floating-point notation. This is the value by default (same as <code>none</code> , before any other <i>floatfield</i> bit is set).

For standard streams, the *floatfield* format flag is set to `defaultfloat` on initialization.

## Parameters

*str*

Stream object whose *floatfield* format flag is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // hexfloat floatfield
2 #include <iostream>      // std::cout, std::hexfloat, std::defaultfloat
3
4 int main () {
5     double a = 3.1415926534;
6     double b = 2006.0;
7     double c = 1.0e-10;
8
9     std::cout.precision(5);
10
11    std::cout << "hexfloat:\n" << std::hexfloat;
12    std::cout << a << '\n' << b << '\n' << c << '\n';
13
14    std::cout << '\n';
15
16    std::cout << "defaultfloat:\n" << std::defaultfloat;
17    std::cout << a << '\n' << b << '\n' << c << '\n';
```

```
18
19     return 0;
20 }
```

Possible output:

```
hexfloat:
0x1.921fb5p+1
0x1.f58000p+10
0x1.b7cdfeep-34

defaultfloat:
3.14159
2006
1e-010
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

### Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

### See also

<b>fixed</b>	Use fixed floating-point notation ( <a href="#">function</a> )
<b>scientific</b>	Use scientific floating-point notation ( <a href="#">function</a> )
<b>defaultfloat</b>	Use default floating-point notation ( <a href="#">function</a> )
<b>ios_base::flags</b>	Get/set format flags ( <a href="#">public member function</a> )
<b>ios_base::setf</b>	Set specific format flags ( <a href="#">public member function</a> )
<b>ios_base::unsetf</b>	Clear specific format flags ( <a href="#">public member function</a> )

## /ios/internal

function  
**std::internal** <iostream>

```
ios_base& internal (ios_base& str);
```

### Adjust field by inserting characters at an internal position

Sets the *adjustfield* format flag for the *str* stream to *internal*.

When *adjustfield* is set to *internal*, the output is padded to the *field width* (*width*) by inserting *fill characters* (*fill*) at a specified internal point, which for numerical values is between the sign and/or numerical base and the number magnitude. For non-numerical values it is equivalent to *right*.

The *adjustfield* format flag can take any of the following values (each with its own manipulator):

flag value	effect when set
<i>internal</i>	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at a specified internal point.
<i>left</i>	the output is padded to the <i>field width</i> appending <i>fill characters</i> at the end.
<i>right</i>	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at the beginning.

For standard streams, the *adjustfield* flag is set to *right* on initialization.

### Parameters

*str*

Stream object whose *adjustfield* *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

### Return Value

Argument *str*.

### Example

```
1 // modify adjustfield using manipulators
2 #include <iostream>      // std::cout, std::internal, std::left, std::right
3
4 int main () {
5     int n = -77;
6     std::cout.width(6); std::cout << std::internal << n << '\n';
7     std::cout.width(6); std::cout << std::left << n << '\n';
8     std::cout.width(6); std::cout << std::right << n << '\n';
9     return 0;
10 }
```

Output:

```
- 77
-77
-77
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">left</a>	Adjust output to the left (function )
<a href="#">right</a>	Adjust output to the right (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/io\_errc

enum class

### std::io\_errc

<iostream>

enum class *io\_errc*;

#### Input/output error conditions

This enum class type defines the *error conditions* of the *iostream category*.

The enum includes at least the following label:

io_errc label	value	description
stream	1	Error in stream

All library implementations define at least this value (*stream*, with a value of 1), but may provide additional values, especially if they require to produce additional *error codes* for the *iostream category*.

Values of the enum type *io\_errc* may be used to create *error\_condition* objects to be compared against the value returned by the *code* member of *ios\_base::failure*.

Although notice that exceptions of type *ios\_base::failure* may also carry error codes from other categories (such as from the *system\_category*).

## Non-member function overloads

[make\\_error\\_code \(io\\_errc\)](#) Make error code (function )

[make\\_error\\_condition \(io\\_errc\)](#) Make error condition (function )

## Non-member class specializations

[is\\_error\\_code\\_enum \(io\\_errc\)](#) error\_code enum flag for *io\_errc* (class )

## Example

```
1 // io_errc example           // std::cin, std::cerr, std::ios,
2 #include <iostream>          // std::make_error_condition, std::ios_errc
3
4 int main () {
5     std::cin.exceptions (std::ios::failbit|std::ios::badbit);
6     try {
7         std::cin.rdbuf(nullptr);    // throws
8     } catch (std::ios::failure& e) {
9         std::cerr << "failure caught: ";
10        if ( e.code() == std::make_error_condition(std::io_errc::stream) )
11            std::cerr << "stream error condition\n";
12        else
13            std::cerr << "some other error condition\n";
14    }
15
16    return 0;
17 }
```

Possible output:

failure caught: stream error condition

## See also

[errc](#) Generic error conditions (enum class )

[iostream\\_category](#) Return iostream category (function )

[make\\_error\\_code \(io\\_errc\)](#) Make error code (function )

[make\\_error\\_condition \(io\\_errc\)](#) Make error condition (function )

## /ios/io\_errc/is\_error\_code\_enum

```

class
std::is_error_code_enum (io_errc)                                     <iostream>
template <*>
struct is_error_code_enum<io_errc> : public true_type {};
error_code enum flag for io_errc

integral_constant   is_error_code_enum

```

This traits specialization identifies `io_errc` as an `error code enum` type. Enabling it to be used to construct or assign values of this type to objects of type `error_code`.

For `io_errc` it inherits from `true_type`.

### Member types

Inherited from `integral_constant` (`is_true` is a typedef of an instantiation of `integral_constant`):

<b>member type</b>	<b>definition</b>
<code>value_type</code>	<code>bool</code>
<code>type</code>	<code>true_type</code>

### Member constants

Inherited from `integral_constant` (`is_true` is a typedef of an instantiation of `integral_constant`):

<b>member constant</b>	<b>definition</b>
<code>value</code>	<code>true</code>

### Member functions

Inherited from `integral_constant` (`is_true` is a typedef of an instantiation of `integral_constant`):

<b>operator bool</b>	Returns value (public member function )
----------------------	---

## /ios/io\_errc/make\_error\_code

function

**std::make\_error\_code (io\_errc)** <iostream>

<code>error_code make_error_code (io_errc e);</code>
<code>error_code make_error_code (io_errc e) noexcept;</code>

### Make error code

Creates an `error_code` object of the `iostream_category` from the `io_errc` enum value `e`.

It returns the same as

<code>error_code(static_cast&lt;int&gt;(e), iostream_category());</code>
--

This function is called by `error_code`'s constructor overload for the `io_errc` type.

### Parameters

`e`  
An enum value of type `io_errc`.

### Return value

An `error_code` object representing the enum value `e`.

### Exception safety

**No-throw guarantee:** this function never throws exceptions.

### See also

<b>io_errc</b>	Input/output error conditions (enum class )
<b>make_error_condition (io_errc)</b>	Make error condition (function )
<b>iostream_category</b>	Return iostream category (function )

## /ios/io\_errc/make\_error\_condition

function

**std::make\_error\_condition (io\_errc)** <iostream>

<code>error_code make_error_condition (io_errc e);</code>
<code>error_code make_error_condition (io_errc e) noexcept;</code>

### Make error condition

Creates an `error_condition` object from the `io_errc` enum value `e` (of the `iostream_category`).

It returns the same as

```
error_condition(static_cast<int>(e),iosstream_category());
```

This overload is called by `error_condition`'s constructor when passed an argument of type `io_errc`.

## Parameters

e

An enum value of type `io_errc`.

## Return value

An `error_condition` object representing the enum value e.

## Exception safety

**No-throw guarantee:** this function never throws exceptions.

## See also

<a href="#">io_errc</a>	Input/output error conditions (enum class )
<a href="#">make_error_code (io_errc)</a>	Make error code (function )
<a href="#">iosstream_category</a>	Return iostream category (function )

# /ios/ios

class		
<b>std::ios</b>		<iostream>
typedef basic_ios<char> ios;		
<b>Base class for streams (type-dependent components)</b>		
<b>ios_base</b>	<b>ios</b>	<b>istream</b>
		<b>ostream</b>

Base class for all stream classes using narrow characters (of type `char`)

This is an instantiation of `basic_ios` with the following template parameters:

template parameter	definition	comments
charT	char	Aliased as member <code>char_type</code>
traits	<code>char_traits&lt;char&gt;</code>	Aliased as member <code>traits_type</code>

Both this class and its parent class, `ios_base`, define the components of streams that do not depend on whether the stream is an input or an output stream. `ios_base` describes the members that are independent of the template parameters, while this one describes the members that are dependent on the template parameters.

The class adds to the information kept by its inherited `ios_base` component, the following:

	field	member functions	description
<i>Formatting</i>	fill character	<code>fill</code>	Character to pad a formatted field up to the <code>field width (width)</code> .
<i>State</i>	error state	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	exception mask	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
<i>Other</i>	tied stream	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	stream buffer	<code>rdbuf</code>	Pointer to the associated <code>streambuf</code> object, which is charge of all input/output operations.

## Member types

member type	definition
<code>char_type</code>	<code>char</code>
<code>traits_type</code>	<code>char_traits&lt;char&gt;</code>
<code>int_type</code>	<code>int</code>
<code>pos_type</code>	<code>streampos</code>
<code>off_type</code>	<code>streamoff</code>

Along with the member types inherited from `ios_base`:

<b>event</b>	Type to indicate event type (public member type )
<b>event_callback</b>	Event callback function type (public member type )
<b>failure</b>	Base class for stream exceptions (public member class )
<b>fmtflags</b>	Type for stream format flags (public member type )
<b>Init</b>	Initialize standard stream objects (public member class )
<b>iostate</b>	Type for stream state flags (public member type )
<b>openmode</b>	Type for stream opening mode flags (public member type )
<b>seekdir</b>	Type for stream seeking direction flag (public member type )

## Public member functions

(constructor)	Construct object (public member function )
(destructor)	Destroy object (public member function )
<b>State flag functions:</b>	
good	Check whether state of stream is good (public member function )
eof	Check whether eofbit is set (public member function )
fail	Check whether either failbit or badbit is set (public member function )
bad	Check whether badbit is set (public member function )
operator!	Evaluate stream (not) (public member function )
operator bool	Evaluate stream (public member function )
rdstate	Get error state flags (public member function )
setstate	Set error state flag (public member function )
clear	Set error state flags (public member function )

## Formatting:

copyfmt	Copy formatting information (public member function )
fill	Get/set fill character (public member function )

## Other:

exceptions	Get/set exceptions mask (public member function )
imbue	Imbue locale (public member function )
tie	Get/set tied stream (public member function )
rdbuf	Get/set stream buffer (public member function )
narrow	Narrow character (public member function )
widen	Widen character (public member function )

## Protected member functions

init	Initialize object (protected member function )
move	Move internals (protected member function )
swap	Swap internals (protected member function )
set_rdbuf	Set stream buffer (protected member function )

## Public member functions inherited from ios\_base

flags	Get/set format flags (public member function )
setf	Set specific format flags (public member function )
unsetf	Clear specific format flags (public member function )
precision	Get/Set floating-point decimal precision (public member function )
width	Get/set field width (public member function )
imbue	Imbue locale (public member function )
getloc	Get current locale (public member function )
xalloc	Get new index for extensible array [static] (public static member function )
iword	Get integer element of extensible array (public member function )
pword	Get pointer element of extensible array (public member function )
register_callback	Register event callback function (public member function )
sync_with_stdio	Toggle synchronization with cstdio streams [static] (public static member function )

## /ios/ios/bad

public member function

### std::ios::bad

<iostream>

bool bad() const;

#### Check whether badbit is set

Returns true if the badbit error state flag is set for the stream.

This flag is set by operations performed on the stream when an error occurs while read or writing data, generally causing the loss of integrity of the stream.

Notice that this function is not the exact opposite of `good`, which checks whether none of the error flags (eofbit, failbit and badbit) are set, and not only badbit:

iostate value (member constants)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
goodbit	No errors (zero value iostate)	true	false	false	false	goodbit
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit
failbit	Logical error on i/o operation	false	false	true	false	failbit
badbit	Read/writing error on i/o operation	false	false	true	true	badbit

eofbit, failbit and badbit are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). goodbit is zero, indicating that none of the other bits is set.

## Parameters

none

## Return Value

true if the stream's badbit error state flag is set.

false otherwise.

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<a href="#">ios::good</a>	Check whether state of stream is good (public member function )
<a href="#">ios::fail</a>	Check whether either failbit or badbit is set (public member function )
<a href="#">ios::eof</a>	Check whether eofbit is set (public member function )
<a href="#">ios::rdstate</a>	Get error state flags (public member function )
<a href="#">ios::clear</a>	Set error state flags (public member function )

## /ios/ios\_base

class	
<a href="#">std::ios_base</a>	<iostream>
class <a href="#">ios_base</a> ;	
<b>Base class for streams</b>	
<a href="#">ios_base</a>	
	<a href="#">basic_ios</a>

Base class for the entire hierarchy of stream classes in the standard input/output library, describing the most basic part of a stream which is common to all stream objects, independently of their character type.

It has no public constructors, and thus no objects of this class can be declared.

Both `ios_base` and its derived class `basic_ios` define the components of streams that do not depend on whether the stream is an input or an output stream: `ios_base` describes the members that are independent of the template parameters (i.e. the character type and traits), while `basic_ios` describes the members that do depend on them.

More specifically, the `ios_base` class maintains the following information of a stream:

	<b>field</b>	<b>member functions</b>	<b>description</b>
Formatting	format flags	<code>flags</code> <code>setf</code> <code>unsetf</code>	A set of internal flags that affect how certain input/output operations are interpreted or generated. See member type <code>fmtflags</code> .
	field width	<code>width</code>	Width of the next formatted element to insert.
	display precision	<code>precision</code>	Decimal precision for the next floating-point value inserted.
	locale	<code>getloc</code> <code>im��ue</code>	The <code>locale</code> object used by the function for formatted input/output operations affected by localization properties.
Other	callback stack	<code>register_callback</code>	Stack of pointers to functions that are called when certain events occur.
	extensible arrays	<code>iword</code> <code>pword</code> <code>xalloc</code>	Internal arrays to store objects of type <code>long</code> and <code>void*</code> .

## Member Functions

<a href="#">(constructor)</a>	Construct object (public member function )
<a href="#">(destructor)</a>	Destruct object (public member function )

### Formatting:

<a href="#">flags</a>	Get/set format flags (public member function )
<a href="#">setf</a>	Set specific format flags (public member function )
<a href="#">unsetf</a>	Clear specific format flags (public member function )
<a href="#">precision</a>	Get/Set floating-point decimal precision (public member function )
<a href="#">width</a>	Get/set field width (public member function )

### Locales:

<a href="#">imbue</a>	Im��ue locale (public member function )
<a href="#">getloc</a>	Get current locale (public member function )

### Internal extensible array:

<a href="#">xalloc</a>	Get new index for extensible array [static] (public static member function )
<a href="#">iword</a>	Get integer element of extensible array (public member function )

<b>pword</b>	Get pointer element of extensible array (public member function )
--------------	---

**Others:**

<b>register_callback</b>	Register event callback function (public member function )
<b>sync_with_stdio</b>	Toggle synchronization with cstdio streams [static] (public static member function )

**Member types**

<b>event</b>	Type to indicate event type (public member type )
<b>event_callback</b>	Event callback function type (public member type )
<b>fmtflags</b>	Type for stream format flags (public member type )
<b>iostate</b>	Type for stream state flags (public member type )
<b>openmode</b>	Type for stream opening mode flags (public member type )
<b>seekdir</b>	Type for stream seeking direction flag (public member type )

**Member classes**

<b>failure</b>	Base class for stream exceptions (public member class )
<b>Init</b>	Initialize standard stream objects (public member class )

**Member constants**

Streams have member constants with the possible values for member types `fmtflags`, `iostate`, `openmode` and `seekdir` (see the description of each type for more info).

## /ios/ios\_base/event

public member type

### std::ios\_base::event

<iostream>

enum event;

#### Type to indicate event type

Enum type used as the first parameter in functions registered with `ios_base::register_callback`. This argument identifies the type of event that triggered the function call.

value	event triggered
<code>copyfmt_event</code>	on a call to <code>ios::copyfmt</code> (at the moment where all format flags have been copied, but before the exception mask is)
<code>erase_event</code>	on a call to the stream destructor (also called at the beginning of <code>ios::copyfmt</code> ).
<code>imbue_event</code>	on a call to <code>ios_base::imbue</code> (just before the function returns).

This member enum type is defined within `ios_base` as:

```
enum event { erase_event, imbue_event, copyfmt_event };
```

**See also**

<b>ios_base::register_callback</b>	Register event callback function (public member function )
------------------------------------	--

## /ios/ios\_base/event\_callback

public member type

### std::ios\_base::event\_callback

<iostream>

#### Event callback function type

Type for *callback functions* registered with member `register_callback`.

It is defined as a member type of `ios_base` as:

```
typedef void (*event_callback) (event ev, ios_base& obj, int index);
```

Therefore it is a function returning no value and having taking three arguments:

`ev`

An object of enum member type `event`. When the *callback function* is called, this is set to one of the three possible values to indicate what type of event triggered the function call.

`obj`

When the *callback function* is called, this is a reference to the stream object on which the even is triggered (`*this`).

`index`

When the *callback function* is called, this is set to the same value used as `index` argument when the function was registered with member `register_callback`.

**See also**

<b>ios_base::register_callback</b>	Register event callback function (public member function )
------------------------------------	--

## /ios/ios\_base/failure

public member class

### std::ios\_base::failure

<iostream>

class failure;

#### Base class for stream exceptions

This embedded class inherits from `exception` and serves as the base class for the *exceptions* thrown by the elements of the standard input/output library.

It is defined as:

```
1 class ios_base::failure : public exception {  
2 public:  
3     explicit failure (const string& msg);  
4     virtual ~failure();  
5     virtual const char* what() const throw();  
6 }
```

Member `what` returns the `msg` with which the exception is constructed.

The specific value for `msg` is entirely implementation-defined.

This embedded class inherits from `system_error` and serves as the base class for the *exceptions* thrown by the elements of the standard input/output library.

It is defined as:

```
1 class ios_base::failure : public system_error {  
2 public:  
3     explicit failure (const string& msg, const error_code& ec = io_errc::stream);  
4     explicit failure (const char* msg, const error_code& ec = io_errc::stream);  
5 }
```

These errors are typically categorized either in the `iostream_category` (if they relate to the operations of the library) or in the `system_category` (if the error arises from the system). Although the specifics are implementation-defined.

The library implementation may use values of type `io_errc` to portably identify *error conditions* of the `iostream_category`.

#### Member functions

##### Inherited from exception

<code>what</code>	Get string identifying exception ( public member function )
-------------------	---

##### Inherited from system\_error

<code>code</code>	Get error code (public member function )
<code>what</code>	Get message associated to exception (public member function )

#### See also

<code>exception</code>	Standard exception class (class )
<code>system_error</code>	System error exception (class )

## /ios/ios\_base/flags

public member function

### std::ios\_base::flags

<iostream>

```
get (1) fmtflags flags() const;  
set (2) fmtflags flags (fmtflags fmtfl);
```

#### Get/set format flags

The first form (1) returns the format flags currently selected in the stream.

The second form (2) sets new format flags for the stream, returning its former value.

The *format flags* of a stream affect the way data is interpreted in certain input functions and how these are written by certain output functions. See `ios_base::fmtflags` for the possible values of this function's argument and the interpretation of its return value.

The second form of this function sets the value for **all** the format flags of the stream, overwriting the existing values and clearing any flag not explicitly set in the argument. To access individual flags, see members `setf` and `unsetf`.

#### Parameters

`fmtfl`

Format flags to be used by the stream.

`ios_base::fmtflags` is a *bitmask type*.

#### Return Value

The format flags selected in the stream before the call.

`ios_base::fmtflags` is a *bitmask type*.

#### Example

```

1 // modify flags
2 #include <iostream>      // std::cout, std::ios
3
4 int main () {
5     std::cout.flags ( std::ios::right | std::ios::hex | std::ios::showbase );
6     std::cout.width (10);
7     std::cout << 100 << '\n';
8     return 0;
9 }
```

This simple example sets some format flags for `cout` that affect the insertion operation by printing the value in hexadecimal (0x64) padded right as in a field ten spaces long:

0x64

## Data races

Accesses (1) or modifies (2) the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<code>ios_base::setf</code>	Set specific format flags ( <a href="#">public member function</a> )
<code>ios_base::unsetf</code>	Clear specific format flags ( <a href="#">public member function</a> )
<code>ios_base::fmtflags</code>	Type for stream format flags ( <a href="#">public member type</a> )
<code>setiosflags</code>	Set format flags ( <a href="#">function</a> )

## /ios/ios\_base/fmtflags

public member type

### std::ios\_base::fmtflags

<iostream>

#### Type for stream format flags

Bitmask type to represent stream *format flags*.

This type is used as its parameter and/or return value by the member functions `flags`, `setf` and `unsetf`.

The values passed and retrieved by these functions can be any valid combination of the following member constants:

field	member constant	effect when set
<i>independent flags</i>	<code>boolalpha</code>	read/write bool elements as alphabetic strings ( <code>true</code> and <code>false</code> ).
	<code>showbase</code>	write integral values preceded by their corresponding numeric base prefix.
	<code>showpoint</code>	write floating-point values including always the decimal point.
	<code>showpos</code>	write non-negative numerical values preceded by a plus sign (+).
	<code>skipws</code>	skip leading whitespaces on certain input operations.
	<code>unitbuf</code>	flush output after each inserting operation.
	<code>uppercase</code>	write uppercase letters replacing lowercase letters in certain insertion operations.
<i>numerical base</i> ( <i>basefield</i> )	<code>dec</code>	read/write integral values using decimal base format.
	<code>hex</code>	read/write integral values using hexadecimal base format.
	<code>oct</code>	read/write integral values using octal base format.
<i>float format</i> ( <i>floatfield</i> )	<code>fixed</code>	write floating point values in fixed-point notation.
	<code>scientific</code>	write floating-point values in scientific notation.
<i>adjustment</i> ( <i>adjustfield</i> )	<code>internal</code>	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at a specified internal point.
	<code>left</code>	the output is padded to the <i>field width</i> appending <i>fill characters</i> at the end.
	<code>right</code>	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at the beginning.

Three additional bitmask constants made of the combination of the values of each of the three groups of selective flags can also be used:

flag value	equivalent to
<code>adjustfield</code>	<code>left</code>   <code>right</code>   <code>internal</code>
<code>basefield</code>	<code>dec</code>   <code>oct</code>   <code>hex</code>
<code>floatfield</code>	<code>scientific</code>   <code>fixed</code>

The values of these constants can be combined into a single `fmtflags` value using the OR bitwise operator (|).

These constants are defined as public members in the `ios_base` class. Therefore, they can be referred to either directly by their name as `ios_base` members (like `ios_base::hex`) or by using any of their inherited classes or instantiated objects, like for example `ios::left` or `cout.oct`.

These values of type `ios_base::fmtflags` should not be confused with the manipulators that have the same name but in the global scope, because they are used in different circumstances. The manipulators cannot be used as values for `ios_base::fmtflags`, as well as these constants shouldn't be used instead of the manipulators. Notice the difference:

```

1 ios_base::skipws      // constant value of type ios_base::fmtflags
2 skipws                // manipulator (global function)
```

Notice that several **manipulators** have the same name as these member constants (but as global functions instead) - see **manipulators**. The behavior of these manipulators generally corresponds to the same as setting or unsetting them with `ios_base::setf` or `ios_base::unsetf`, but they should not be confused! Manipulators are global functions and these constants are member constants. For example, `showbase` is a manipulator, while `ios_base::showbase` is a constant value that can be used as parameter with `ios_base::setf`.

## Example

```
1 // using ios_base::fmtflags
2 #include <iostream>           // std::cout, std::ios_base, std::ios,
3 // std::hex, std::showbase
4 int main () {
5
6     // using fmtflags as class member constants:
7     std::cout.setf (std::ios_base::hex , std::ios_base::basefield);
8     std::cout.setf (std::ios_base::showbase);
9     std::cout << 100 << '\n';
10
11    // using fmtflags as inherited class member constants:
12    std::cout.setf (std::ios::hex , std::ios::basefield);
13    std::cout.setf (std::ios::showbase);
14    std::cout << 100 << '\n';
15
16    // using fmtflags as object member constants:
17    std::cout.setf (std::cout.hex , std::cout.basefield);
18    std::cout.setf (std::cout.showbase);
19    std::cout << 100 << '\n';
20
21    // using fmtflags as a type:
22    std::ios_base::fmtflags ff;
23    ff = std::cout.flags();
24    ff &= ~std::cout.basefield;    // unset basefield bits
25    ff |= std::cout.hex;         // set hex
26    ff |= std::cout.showbase;    // set showbase
27    std::cout.flags(ff);
28    std::cout << 100 << '\n';
29
30    // not using fmtflags, but using manipulators:
31    std::cout << std::hex << std::showbase << 100 << '\n';
32
33    return 0;
34 }
```

The code shows some different ways of printing the same result, using both the `fmtflags` member constants and their homonymous manipulators.

Output:

```
0x64
0x64
0x64
0x64
0x64
```

## See also

<a href="#">ios_base::flags</a>	Get/set format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::unsetf</a>	Clear specific format flags ( <a href="#">public member function</a> )
<a href="#">setiosflags</a>	Set format flags ( <a href="#">function</a> )

## /ios/ios\_base/getloc

public member function

### std::ios\_base::getloc

<iostream>

`locale getloc() const;`

#### Get current locale

Returns the `locale` object currently associated with the stream.

#### Parameters

none

#### Return Value

The `locale` object currently associated with the stream.

#### Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

#### See also

<a href="#">ios_base::imbue</a>	Imbue locale ( <a href="#">public member function</a> )
---------------------------------	---

## /ios/ios\_base/imbue

public member function

### std::ios\_base::imbue

<iostream>

```
locale imbue (const locale& loc);
```

#### Imbue locale

Associates *loc* to the stream as the new locale object to be used with locale-sensitive operations.

Before that, the function calls all functions registered through member `register_callback` with `imbue_event` as first argument.

Standard stream classes do not inherit this member, but inherit `basic_ios::imbue` instead, which calls this function, but also imbues the `locale` to the associated *stream buffer*, if any.

#### Parameters

loc

locale imbued as the new locale for the stream.

#### Return value

The `locale` object associated with the stream before the call.

#### Example

```
1 // imbue example
2 #include <iostream>      // std::cout
3 #include <locale>        // std::locale
4
5 int main()
6 {
7     std::locale mylocale("");    // get global locale
8     std::cout.imbue(mylocale);   // imbue global locale
9     std::cout << 3.14159 << '\n';
10    return 0;
11 }
```

This code writes a floating point number using the global locale given by the environment. For example, in a system configured with a Spanish locale as default, this could write the number using a comma decimal separator:

3,14159

#### Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

#### Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

#### See also

[ios\\_base::getloc](#) Get current locale (public member function)

## /ios/ios\_base/Init

public member class

### std::ios\_base::Init

<iostream>

```
class Init;
```

#### Initialize standard stream objects

The construction of an object of this member type, ensures that the standard stream objects (`cin`, `cout`, `cerr`, `clog`, `wcin`, `wcout`, `wcerr` and `wclog`) are constructed and properly initialized.

The class maintains an internal static counter with the number of existing objects.

```
1 class ios_base::Init {
2     static int init_cnt; // internal static counter (for exposition only)
3 public:
4     Init();
5     ~Init();
6 }
```

#### Member functions

`Init();` (constructor)

Increases the internal static counter by one. If the value of the internal counter was zero, the standard iostream objects are constructed and initialized, if they have not yet been constructed and initialized.

`~Init();` (destructor)

Decreases the internal static counter by one. If the value of the internal counter reaches zero, the standard output streams are *flushed* (as if their respective `flush` members were called).  
Notice that this does not destroy any of the standard objects, whose duration lasts until program termination.

## See also

[<iostream>](#) | Standard Input / Output Streams Library ([header](#))

## /ios/ios\_base/~ios\_base

public member function

**std::ios\_base::~ios\_base**

[<iostream>](#)

`virtual ~ios_base();`

### Destruct object

Before the object is destroyed, all callback functions registered with member `register_callback` are called with `erase_event` as first argument.

### Data races

The object is modified.

### Exception safety

**No-throw guarantee:** never throws exceptions.

## /ios/ios\_base/ios\_base

public member function

**std::ios\_base::ios\_base**

[<iostream>](#)

```
protected: ios_base();
private: ios_base (const ios_base&);
protected: ios_base();
ios_base (const ios_base&) = delete;
```

### Construct object

`ios_base` objects have indeterminate values on construction. Each `ios_base` base object shall be initialized by calling `basic_ios::init`.

The class is meant to be a base class, and thus has no public constructors, preventing objects of this class to be constructed -- only objects of derived classes can be constructed.

`ios_base` also declares a copy assignment member function. Like the copy constructor, this function is also `private`:

```
1 private:
2   ios_base (const ios_base&);
3   ios_base& operator= (const ios_base&);
```

`ios_base` also deletes its copy assignment member function. Like the copy constructor, this function is also deleted:

```
1   ios_base (const ios_base&) = delete;
2   ios_base& operator= (const ios_base&) = delete;
```

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no side effects.

## /ios/ios\_base/iostate

public member type

**std::ios\_base::iostate**

[<iostream>](#)

### Type for stream state flags

Bitmask type to represent stream *error state flags*.

All stream objects keep information on the state of the object internally. This information can be retrieved as an element of this type by calling member function `basic_ios::rdstate` or set by calling `basic_ios::setstate`.

The values passed and retrieved by these functions can be any valid combination (using the boolean OR operator, " | ") of the following member constants:

flag value	indicates
<code>eofbit</code>	End-Of-File reached while performing an extracting operation on an input stream.
<code>failbit</code>	The last input operation failed because of an error related to the internal logic of the operation itself.
<code>badbit</code>	Error due to the failure of an input/output operation on the stream buffer.
<code>goodbit</code>	No error. Represents the absence of all the above (the value zero).

These constants are defined in the `ios_base` class as public members. Therefore, they can be referred to either directly by their name as `ios_base` members (like `ios_base::badbit`) or by using any of their inherited classes or instantiated objects, like for example `ios::eofbit` or `cin.goodbit`.

## See also

<a href="#">ios::rdstate</a>	Get error state flags (public member function )
<a href="#">ios::setstate</a>	Set error state flag (public member function )
<a href="#">ios::good</a>	Check whether state of stream is good (public member function )
<a href="#">ios::bad</a>	Check whether badbit is set (public member function )
<a href="#">ios::fail</a>	Check whether either failbit or badbit is set (public member function )
<a href="#">ios::eof</a>	Check whether eofbit is set (public member function )

## /ios/ios\_base/iword

public member function

### std::ios\_base::iword

<iostream>

`long& iword (int idx);`

#### Get integer element of extensible array

Returns a reference to the object of type `long` which corresponds to index `idx` in the *internal extensible array*.

If `idx` is an index to a new element and the internal extensible array is not long enough (or is not yet allocated), the function extends it (or allocates it) with as many zero-initialized elements as necessary.

The reference returned is guaranteed to be valid at least until another operation is performed on the stream object, including another call to `iword`. Once another operation is performed, the reference may become invalid, although a subsequent call to this same function with the same `idx` argument returns a reference to the same value within the *internal extensible array*.

The *internal extensible array* is a general-purpose array of objects of type `long` (if accessed with member `iword`) or `void*` (if accessed with member `pword`). Libraries may implement this array in diverse ways: `iword` and `pword` may or may not share a unique array, and they may not even be arrays, but some other data structure.

## Parameters

`idx`

An index value for an element of the *internal extensible array*.

Some implementations expect `idx` to be a value previously returned by member `xalloc`.

## Return Value

A reference to the element in the internal extensible array whose index is `idx`.

This value is returned as a reference to an object of type `long`.

On failure, a valid `long&` initialized to `0L` is returned, and (if the stream object inherits from `basic_ios`) the `badbit` *state flag* is set.

## Example

```
1 // internal extensible array
2 #include <iostream>      // std::cout, std::cerr
3
4 // custom manipulator with per-stream static data:
5 std::ostream& Counter (std::ostream& os) {
6     const static int index = os.xalloc();
7     return os << ++os.iword(index);
8 }
9
10 int main()
11 {
12     std::cout << Counter << ": first line\n";
13     std::cout << Counter << ": second line\n";
14     std::cout << Counter << ": third line\n";
15     // cerr has its own count
16     std::cerr << Counter << ": error line\n";
17     return 0;
18 }
```

Possible output:

```
1: first line
2: second line
3: third line
1: error line
```

## Data races

May modify the stream object. The returned value may also be used to modify it.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">ios_base::xalloc</a>	Get new index for extensible array [static] (public static member function )
<a href="#">ios_base::pword</a>	Get pointer element of extensible array (public member function )

## /ios/ios\_base/openmode

public member type

### std::ios\_base::openmode

<iostream>

#### Type for stream opening mode flags

Bitmask type to represent stream *opening mode flags*.

A value of this type can be any valid combination of the following member constants:

member constant	opening mode
app	(append) Set the stream's position indicator to the end of the stream before each output operation.
ate	(at end) Set the stream's position indicator to the end of the stream on opening.
binary	(binary) Consider stream as binary rather than text.
in	(input) Allow input operations on the stream.
out	(output) Allow output operations on the stream.
trunc	(truncate) Any current content is discarded, assuming a length of zero on opening.

These constants are defined in the `ios_base` class as public members. Therefore, they can be referred to either directly by their name as members of `ios_base` (like `ios_base::in`) or by using any of their inherited classes or instantiated objects, like for example `ios::ate` or `cout.out`.

## /ios/ios\_base/precision

public member function

### std::ios\_base::precision

<iostream>

```
get (1) streamsize precision() const;  
set (2) streamsize precision (streamsize prec);
```

#### Get/Set floating-point decimal precision

The first form (1) returns the value of the current floating-point precision field for the stream.

The second form (2) also sets it to a new value.

The *floating-point precision* determines the maximum number of digits to be written on insertion operations to express floating-point values. How this is interpreted depends on whether the `floatfield` `format flag` is set to a specific notation (either `fixed` or `scientific`) or it is unset (using the `default notation`, which is not necessarily equivalent to either `fixed` nor `scientific`).

For the default locale:

- Using the default floating-point notation, the precision field specifies the maximum number of meaningful digits to display in total counting both those before and those after the decimal point. Notice that it is not a minimum, and therefore it does not pad the displayed number with trailing zeros if the number can be displayed with less digits than the *precision*.
- In both the `fixed` and `scientific` notations, the precision field specifies exactly how many digits to display after the decimal point, even if this includes trailing decimal zeros. The digits before the decimal point are not relevant for the *precision* in this case.

This *decimal precision* can also be modified using the parameterized manipulator `setprecision`.

#### Parameters

prec

New value for the floating-point precision.  
`streamsize` is a signed integral value.

#### Return Value

The *precision* selected in the stream before the call.

#### Example

```
1 // modify precision  
2 #include <iostream>      // std::cout, std::ios  
3  
4 int main () {  
5     double f = 3.14159;  
6     std::cout.unsetf ( std::ios::floatfield );           // floatfield not set  
7     std::cout.precision(5);  
8     std::cout << f << '\n';  
9     std::cout.precision(10);  
10    std::cout << f << '\n';  
11    std::cout.setf( std::ios::fixed, std::ios::floatfield ); // floatfield set to fixed  
12    std::cout << f << '\n';  
13    return 0;  
14 }
```

Possible output:

```
3.1416  
3.14159  
3.1415900000
```

Notice how the first number written is just 5 digits long, while the second is 6, but not more, even though the stream's precision is now 10. That is because precision with the default `floatfield` only specifies the *maximum* number of digits to be displayed, but not the minimum.

The third number printed displays 10 digits after the decimal point because the `floatfield` `format flag` is in this case set to `fixed`.

## Data races

Accesses (1) or modifies (2) the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">setprecision</a>	Set decimal precision ( <a href="#">function</a> )
<a href="#">ios_base::width</a>	Get/set field width ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )

## /ios/ios\_base/pword

public member function

### std::ios\_base::pword

<iostream>

`void*& pword (int idx);`

#### Get pointer element of extensible array

Returns a reference to the object of type `void*` which corresponds to index `idx` in the *internal extensible array*.

If `idx` is an index to a new element and the internal extensible array is not long enough (or is not yet allocated), the function extends it (or allocates it) with as many elements initialized to `null pointers` as necessary.

The reference returned is guaranteed to be valid at least until another operation is performed on the stream object, including another call to `pword`. Once another operation is performed, the reference may become invalid, although a subsequent call to this same function with the same `idx` argument returns a reference to the same value within the *internal extensible array*.

The *internal extensible array* is a general-purpose array of objects of type `long` (if accessed with member `iword`) or `void*` (if accessed with member `pword`). Libraries may implement this array in diverse ways: `iword` and `pword` may or may not share a unique array, and they may not even be arrays, but some other data structure.

## Parameters

`idx`

An index value for an element of the *internal extensible array*.

Some implementations expect `idx` to be a value previously returned by member `xalloc`.

## Return Value

A reference to the element in the internal extensible array whose index is `idx`.

This value is returned as a reference to an object of type `void*`.

On failure, a valid `void*&` initialized to 0 is returned, and (if the stream object inherits from `basic_ios`) the `badbit` *state flag* is set.

## Example

```
1 // pword example
2 #include <iostream>      // std::ios, std::cout, std::cerr, std::clog
3
4 const int name_index = std::ios::xalloc();
5
6 // stores pointer in extensible array:
7 void SetStreamName (std::ios& stream, const char* name) {
8     stream.pword(name_index) = const_cast<char*>(name);
9 }
10
11 // custom manipulator that uses stored pointer:
12 std::ostream& StreamName (std::ostream& os) {
13     const char* name = static_cast<const char*>(os.pword(name_index));
14     if (name) os << name;
15     else os << "(unknown)";
16     return os;
17 }
18
19 int main()
20 {
21     SetStreamName(std::cout, "standard output stream");
22     SetStreamName(std::cerr, "standard error stream");
23     std::cout << StreamName << '\n';
24     std::cerr << StreamName << '\n';
25     std::clog << StreamName << '\n';
26     return 0;
27 }
```

Possible output:

```
standard output stream
standard error stream
(unknown)
```

## Data races

May modify the stream object. The returned value may also be used to modify it. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<code>ios_base::xalloc</code>	Get new index for extensible array [static] (public static member function )
<code>ios_base::iword</code>	Get integer element of extensible array (public member function )

## /ios/ios\_base/register\_callback

public member function

### std::ios\_base::register\_callback

<iostream>

`void register_callback (event_callback fn, int index);`

#### Register event callback function

Registers `fn` as a callback function to be called automatically with `index` as argument when a *stream event* occurs.

If more than one callback function is registered, they are all called, in the inverse order of registration.

The callback function shall be of a type convertible to `event_callback`. And it is called by an expression equivalent to:

`(*fn)(ev, stream, index)`

where `index` is the `index` argument passed when the callback function is registered with this function, `stream` is a pointer to the stream object suffering the event, and `ev` is an object of member enum type `event` indicating which event occurred. It can be one of the following member values:

member constant	event triggered
<code>copyfmt_event</code>	on a call to member <code>copyfmt</code> (at the moment where all format flags have been copied, but before the exception mask is)
<code>erase_event</code>	on a call to the stream destructor (also called at the beginning of <code>basic_ios::copyfmt</code> ).
<code>imbue_event</code>	on a call to <code>imbue</code> (just before the function returns).

All registered functions are called on all of the cases above. The function itself can use the `ev` parameter to discern which event triggered the function call.

## Parameters

`fn`

Pointer to the function to be called.

The `event_callback` member type is defined as:

```
typedef void (*event_callback) (event ev, ios_base& ios, int index);
```

`index`

Integer value passed as parameter to the callback function.

## Return Value

none

## Example

```
1 // stream callbacks
2 #include <iostream>           // std::cout, std::ios_base
3 #include <fstream>            // ofstream
4
5 void testfn (std::ios::event ev, std::ios_base& stream, int index)
6 {
7     switch (ev)
8     {
9         case stream.copyfmt_event:
10             std::cout << "copyfmt_event\n"; break;
11         case stream.imbue_event:
12             std::cout << "imbue_event\n"; break;
13         case stream.erase_event:
14             std::cout << "erase_event\n"; break;
15     }
16 }
17
18 int main () {
19     std::ofstream filestr;
20     filestr.register_callback (testfn,0);
21     filestr.imbue (std::cout.getloc());
22     return 0;
23 }
```

Output:

```
imbue_event
erase_event
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">ios_base::imbue</a>	Imbue locale (public member function )
<a href="#">ios::copyfmt</a>	Copy formatting information (public member function )
<a href="#">ios_base::event</a>	Type to indicate event type (public member type )

## /ios/ios\_base/seekdir

public member type

### std::ios\_base::seekdir

<iostream>

#### Type for stream seeking direction flag

Enumerated type to represent the seeking direction of a stream *seeking operation*.

The possible values for this type are one of the following member constants:

member constant	seeking relative to
beg	beginning of sequence.
cur	current position within sequence.
end	end of sequence.

These constants are defined in the `ios_base` class as public members. Therefore, they can be referred to either directly by their name as members of `ios_base` (like `ios_base::beg`) or by using any of their inherited classes or instantiated objects, like for example `ios::end` or `cin.cur`.

## /ios/ios\_base/setf

public member function

### std::ios\_base::setf

<iostream>

```
set(1) fmtflags setf (fmtflags fmtfl);
mask(2) fmtflags setf (fmtflags fmtfl, fmtflags mask);
```

#### Set specific format flags

The first form (1) sets the stream's *format flags* whose bits are set in `fmtfl`, leaving unchanged the rest, as if a call to `flags(fmtfl|flags())`.

The second form (2) sets the stream's *format flags* whose bits are set in both `fmtfl` and `mask`, and clears the *format flags* whose bits are set in `mask` but not in `fmtfl`, as if a call to `flags((fmtfl&mask)|(flags()&~mask))`.

Both return the value of the stream's *format flags* before the call.

The format flags of a stream affect the way data is interpreted in certain input functions and how it is written by certain output functions. See `ios_base::fmtflags` for the possible values of this function's arguments.

The first form of `setf(1)` is generally used to set *independent format flags*: `boolalpha`, `showbase`, `showpoint`, `showpos`, `skipws`, `unitbuf` and `uppercase`, which can also be unset directly with member `unsetf`.

The second form (2) is generally used to set a value for one of the selective flags, using one of the field bitmasks as the `mask` argument:

fmtfl format flag value	mask field bitmask
left, right or internal	adjustfield
dec, oct or hex	basefield
scientific or fixed	floatfield

The parameterized manipulator `setiosflags` behaves in a similar way as the first form of this member function (1).

## Parameters

### fmtfl

Format flags to be set. If the second syntax is used, only the bits set in both `fmtfl` and `mask` are set in the stream's format flags; the flags set in `mask` but not in `fmtfl` are cleared.

### mask

Mask containing the flags to be modified.

Member type `fmtflags` is a bitmask type (see `ios_base::fmtflags`).

## Return Value

The format flags selected in the stream before the call.

## Example

```

1 // modifying flags with setf/unsetf
2 #include <iostream>      // std::cout, std::ios
3
4 int main () {
5     std::cout.setf ( std::ios::hex, std::ios::basefield ); // set hex as the basefield
6     std::cout.setf ( std::ios::showbase );                  // activate showbase
7     std::cout << 100 << '\n';
8     std::cout.unsetf ( std::ios::showbase );                // deactivate showbase
9     std::cout << 100 << '\n';
10    return 0;
11 }

```

Output:

```

0x64
64

```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )
<a href="#">ios_base::fmtflags</a>	Type for stream format flags (public member type )
<a href="#">setiosflags</a>	Set format flags (function )
<a href="#">resetiosflags</a>	Reset format flags (function )

## /ios/ios\_base/sync\_with\_stdio

public static member function

### std::ios\_base::sync\_with\_stdio

<iostream>

`bool sync_with_stdio (bool sync = true);`

#### Toggle synchronization with cstdio streams [static]

Toggles on or off synchronization of all the iostream standard streams with their corresponding standard C streams if it is called before the program performs its first input or output operation.

If called once an input or output operation has occurred, its effects are *implementation-defined*.

By default, `iostream` objects and `cstdio` streams are synchronized (as if this function was called with `true` as argument).

The stream correspondences are:

C stream	iostream object
stdin	cin
	wcin
stdout	cout
	wcout
stderr	cerr
	wcerr
	clog
	wclog

If the streams are synchronized, a program can mix iostream operations with stdio operations, and their observable effects are guaranteed to follow the same order as used in the program.

If the streams are synchronized, a program can mix iostream operations with stdio operations, and their observable effects are guaranteed to follow the same order as used in the thread.

Concurrently accessing *synchronized streams* (i.e., streams for which this function returns `true`) never introduces data races: characters are read/written individually, although with no further guarantees on its order between threads. This may result in interleaved characters between threads unless proper synchronization of entire operations is enforced by the program.

With stdio synchronization turned off, iostream standard stream objects may operate independently of the standard C streams (although they are not required to), and mixing operations may result in unexpectedly interleaved characters.

Notice that this is a static member function, and a call to this function using this member of *any* stream object toggles on or off synchronization for *all* standard iostream objects.

## Parameters

### sync

Boolean parameter indicating whether synchronization is to be turned on or off: A value of `true` requests synchronization to be turned *on*, while a value of `false` requests it to be turned *off*.

## Return Value

Returns the synchronization state before the call.  
It always returns true the first time it is called.

## Data races

May modify the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

# /ios/ios\_base/unsetf

public member function

## std::ios\_base::unsetf

<iostream>

void unsetf (fmtflags mask);

### Clear specific format flags

Clears the format flags selected in *mask*.

The parameterized manipulator `resetiosflags` behaves in a similar way as this member function.

## Parameters

**mask**  
Bitmask specifying the flags to be cleared. The flags are specified as a combination of flags of the `fmtflags` member type.

## Return Value

none

## Example

```
1 // modifying flags with setf/unsetf
2 #include <iostream>      // std::cout, std::ios
3
4 int main () {
5     std::cout.setf ( std::ios::hex, std::ios::basefield );    // set hex as the basefield
6     std::cout.setf ( std::ios::showbase );                      // activate showbase
7     std::cout << 100 << '\n';
8     std::cout.unsetf ( std::ios::showbase );                   // deactivate showbase
9     std::cout << 100 << '\n';
10    return 0;
11 }
```

Output:

```
0x64
64
```

## Data races

Modifies the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">ios_base::flags</a>	Get/set format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::fmtflags</a>	Type for stream format flags ( <a href="#">public member type</a> )
<a href="#">resetiosflags</a>	Reset format flags ( <a href="#">function</a> )

# /ios/ios\_base/width

public member function

## std::ios\_base::width

<iostream>

```
get (1) streamsize width() const;
set (2) streamsize width (streamsize wide);
```

### Get/set field width

The first form (1) returns the current value of the *field width*.  
The second form (2) also sets a new *field width* for the stream.

The *field width* determines the minimum number of characters to be written in some output representations. If the standard width of the representation is

shorter than the field width, the representation is padded with *fill characters* at a point determined by the format flag `adjustfield` (one of `left`, `right` or `internal`).

The *fill character* can be retrieved or changed by calling the member function `fill`.

The format flag `adjustfield` can be modified by calling the member functions `flags` or `setf`, by inserting one of the following manipulators: `left`, `right` and `internal`, or by inserting the parameterized manipulator `setiosflags`.

The *field width* can also be modified using the parameterized manipulator `setw`.

## Parameters

`wide`

New value for the stream's *field width*.  
`streamsize` in signed integral type.

## Return Value

The value of the *field width* before the call.

## Example

```
1 // field width
2 #include <iostream>      // std::cout, std::left
3
4 int main () {
5     std::cout << 100 << '\n';
6     std::cout.width(10);
7     std::cout << 100 << '\n';
8     std::cout.fill('x');
9     std::cout.width(15);
10    std::cout << std::left << 100 << '\n';
11    return 0;
12 }
```

Output:

```
100
     100
100xxxxxxxxxxxx
```

## Data races

Accesses (1) or modifies (2) the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<code>setw</code>	Set field width ( <a href="#">function</a> )
<code>ios_base::flags</code>	Get/set format flags ( <a href="#">public member function</a> )
<code>ios_base::setf</code>	Set specific format flags ( <a href="#">public member function</a> )
<code>ios::fill</code>	Get/set fill character ( <a href="#">public member function</a> )

## /ios/ios\_base/xalloc

public static member function

### `std::ios_base::xalloc`

`<iostream>`

`static int xalloc();`

#### Get new index for extensible array [static]

Returns a new index value to be used with member functions in the internal extensible array.

The *internal extensible array* is a general-purpose array of objects of type `long` (if accessed with member `iword`) or `void*` (if accessed with member `pword`).

This is a static member function.

## Parameters

none

## Return Value

A new index that can be used with either member `iword` or member `pword`.

## Data races

Concurrently calling this function may introduce data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in any stream.

### See also

<a href="#">ios_base::iword</a>	Get integer element of extensible array (public member function )
<a href="#">ios_base::pword</a>	Get pointer element of extensible array (public member function )

## /ios/ios/clear

public member function

### std::ios::clear

<iostream>

```
void clear (iostate state = goodbit);
```

#### Set error state flags

Sets a new value for the stream's internal *error state flags*.

The current value of the flags is overwritten: All bits are replaced by those in *state*; If *state* is *goodbit* (which is zero) all error flags are cleared.

In the case that no *stream buffer* is associated with the stream when this function is called, the *badbit* flag is automatically set (no matter the value for that bit passed in argument *state*).

Note that changing the *state* may throw an exception, depending on the latest settings passed to member [exceptions](#).

The current state can be obtained with member function [rdstate](#).

### Parameters

*state*

An object of type [ios\\_base::iostate](#) that can take as value any combination of the following state flag member constants:

iostate value (member constant)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
<code>goodbit</code>	No errors (zero value <i>iostate</i> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	badbit

*eofbit*, *failbit* and *badbit* are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). *goodbit* is zero, indicating that none of the other bits is set.

### Return Value

none

### Example

```
1 // clearing errors
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::fstream
4
5 int main () {
6     char buffer [80];
7     std::fstream myfile;
8
9     myfile.open ("test.txt",std::fstream::in);
10
11    myfile << "test";
12    if (myfile.fail())
13    {
14        std::cout << "Error writing to test.txt\n";
15        myfile.clear();
16    }
17
18    myfile.getline (buffer,80);
19    std::cout << buffer << " successfully read from file.\n";
20
21    return 0;
22 }
```

In this example, *myfile* is open for input operations, but we perform an output operation on it, so *failbit* is set. The example calls then *clear* in order to remove the flag and allow further operations like *getline* to be attempted on *myfile*.

### Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

### Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

It throws an exception of member type [failure](#) if the resulting *error state flag* is not *goodbit* and member [exceptions](#) was set to *throw* for that state.

### See also

<b>ios::fail</b>	Check whether either failbit or badbit is set ( <a href="#">public member function</a> )
<b>ios::good</b>	Check whether state of stream is good ( <a href="#">public member function</a> )
<b>ios::bad</b>	Check whether badbit is set ( <a href="#">public member function</a> )
<b>ios::eof</b>	Check whether eofbit is set ( <a href="#">public member function</a> )
<b>ios::rdstate</b>	Get error state flags ( <a href="#">public member function</a> )

## /ios/ios/copyfmt

public member function

### std::ios::copyfmt

<ios> <iostream>

`ios::copyfmt (const ios& rhs);`

#### Copy formatting information

Copies the values of all the internal members of *rhs* (except the *state flags* and the associated *stream buffer*) to the corresponding members of *\*this*.

After the call, the following member functions return the same for *rhs* and *\*this*:

element	description
<code>flags</code>	format flags
<code>width</code>	field width
<code>precision</code>	precision
<code>getloc</code>	selected locale
<code>iarray</code>	internal extensible array *
<code>parray</code>	internal extensible array *
<code>fill</code>	fill character
<code>tie</code>	tied stream
<code>exceptions</code>	exceptions mask (last to be copied, see below)

\* Each stream object keeps its own copy of the *internal extensible array* (`iword`, `pword`): Its contents are copied, not just a pointer to it.

\* Each stream object keeps its own copy of the *internal extensible array* (`iword`, `pword`): Its contents are copied, not just a pointer to it.

If any of the pointer values to be copied points to objects stored outside *rhs* and those objects are destroyed when *rhs* is destroyed, *\*this* stores instead pointers to newly constructed copies of these objects.

Calling this function invokes all functions registered through member `register_callback` twice: First, before the copying process starts, the function calls each registered callback *fn* using `(*fn)(erase_event, *this, index)`. Then, at the end, right before the *exceptions mask* is copied, the function calls each registered callback *fn* with `(*fn)(copyfmt_event, *this, index)` (this second round can be used, for example, to access and modify the copied *internal extensible array*).

#### Parameters

`rhs`

Stream object whose members are copied to *\*this*.

#### Return Value

`*this`

#### Example

```
1 // copying formatting information
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::ofstream
4
5 int main () {
6     std::ofstream filestr;
7     filestr.open ("test.txt");
8
9     std::cout.fill ('*');
10    std::cout.width (10);
11    filestr.copyfmt (std::cout);
12
13    std::cout << 40;
14    filestr << 40;
15
16    return 0;
17 }
```

This example outputs a number formatted in the same way to both `cout` and a file called "test.txt":

\*\*\*\*\*40

#### Data races

Modifies the stream object (*\*this*), and accesses *rhs*.

Concurrent access to any of the objects may cause data races.

#### Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

#### See also

<b>ios::tie</b>	Get/set tied stream ( <a href="#">public member function</a> )
-----------------	--

<b>ios::fill</b>	Get/set fill character ( <a href="#">public member function</a> )
<b>ios_base::width</b>	Get/set field width ( <a href="#">public member function</a> )
<b>ios_base::fmtflags</b>	Type for stream format flags ( <a href="#">public member type</a> )

## /ios/ios/eof

public member function

### std::ios::eof

<ios> <iostream>

**bool eof() const;**

#### Check whether eofbit is set

Returns true if the eofbit error state flag is set for the stream.

This flag is set by all standard input operations when the End-of-File is reached in the sequence associated with the stream.

Note that the value returned by this function depends on the last operation performed on the stream (and not on the next).

Operations that attempt to read at the *End-of-File* fail, and thus both the eofbit and the failbit end up set. This function can be used to check whether the failure is due to reaching the *End-of-File* or to some other reason.

#### Parameters

none

#### Return Value

true if the stream's eofbit error state flag is set (which signals that the End-of-File has been reached by the last input operation).  
false otherwise.

#### Example

```

1 // ios::eof example
2 #include <iostream>           // std::cout
3 #include <ifstream>          // std::ifstream
4
5 int main () {
6
7     std::ifstream is("example.txt");    // open file
8
9     char c;                           // loop getting single characters
10    while (is.get(c))                // check for EOF
11        std::cout << c;
12
13    if (is.eof())
14        std::cout << "[EOF reached]\n";
15    else
16        std::cout << "[error reading]\n";
17
18    is.close();                      // close file
19
20    return 0;
21 }
```

#### Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

#### See also

<b>ios::good</b>	Check whether state of stream is good ( <a href="#">public member function</a> )
<b>ios::fail</b>	Check whether either failbit or badbit is set ( <a href="#">public member function</a> )
<b>ios::bad</b>	Check whether badbit is set ( <a href="#">public member function</a> )
<b>ios::rdstate</b>	Get error state flags ( <a href="#">public member function</a> )
<b>ios::clear</b>	Set error state flags ( <a href="#">public member function</a> )

## /ios/ios/exceptions

public member function

### std::ios::exceptions

<ios> <iostream>

```

get (1) iostate exceptions() const;
set (2) void exceptions (iostate except);
```

#### Get/set exceptions mask

The first form (1) returns the current exception mask for the stream.

The second form (2) sets a new exception mask for the stream and clears the stream's *error state flags* (as if member `clear()` was called).

The exception mask is an internal value kept by all stream objects specifying for which state flags an exception of member type `failure` (or some derived type) is thrown when set. This mask is an object of member type `iostate`, which is a value formed by any combination of the following member constants:

<code>iostate</code> value (member constants)	indicates	functions to check state flags				
		<code>good()</code>	<code>eof()</code>	<code>fail()</code>	<code>bad()</code>	<code>rdstate()</code>
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator), so that the stream throws when any of the selected *error state flags* is set.

`goodbit` is zero, indicating that no exceptions shall be thrown when an *error state flags* is set.

All streams have `goodbit` by default (they do not throw exceptions due to *error state flags* being set).

## Parameters

`except`

A bitmask value of member type `iostate` formed by a combination of error state flag bits to be set (`badbit`, `eofbit` and/or `failbit`), or set to `goodbit` (or zero).

## Return Value

The first form (1) returns a bitmask of member type `iostate` representing the existing exception mask before the call to this member function.

## Example

```
1 // ios::exceptions
2 #include <iostream>           // std::cerr
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream file;
7     file.exceptions ( std::ifstream::failbit | std::ifstream::badbit );
8     try {
9         file.open ("test.txt");
10        while (!file.eof()) file.get();
11        file.close();
12    }
13    catch (std::ifstream::failure e) {
14        std::cerr << "Exception opening/reading/closing file\n";
15    }
16
17    return 0;
18 }
```

## Data races

Accesses (1) or modifies (2) the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<code>ios::rdstate</code>	Get error state flags (public member function )
---------------------------	---

## /ios/ios/fail

public member function

### `std::ios::fail`

`<iostream>`

`bool fail() const;`

**Check whether either failbit or badbit is set**

Returns true if either (or both) the `failbit` or the `badbit` *error state flags* is set for the stream.

At least one of these flags is set when an error occurs during an input operation.

`failbit` is generally set by an operation when the error is related to the internal logic of the operation itself; further operations on the stream may be possible. While `badbit` is generally set when the error involves the loss of integrity of the stream, which is likely to persist even if a different operation is attempted on the stream. `badbit` can be checked independently by calling member function `bad`:

<code>iostate</code> value (member constants)	indicates	functions to check state flags				
		<code>good()</code>	<code>eof()</code>	<code>fail()</code>	<code>bad()</code>	<code>rdstate()</code>
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator).

`goodbit` is zero, indicating that none of the other bits is set.

Reaching the *End-of-File* sets the `eofbit`. But note that operations that reach the *End-of-File* may also set the `failbit` if this makes them fail (thus setting both `eofbit` and `failbit`).

This function is a synonym of `ios::operator!.`

## Parameters

none

## Return Value

true if `badbit` and/or `failbit` are set.  
false otherwise.

## Data races

Accesses the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<a href="#">ios_base::iostate</a>	Type for stream state flags (public member type )
<a href="#">ios::good</a>	Check whether state of stream is good (public member function )
<a href="#">ios::bad</a>	Check whether <code>badbit</code> is set (public member function )
<a href="#">ios::eof</a>	Check whether <code>eofbit</code> is set (public member function )
<a href="#">ios::rdstate</a>	Get error state flags (public member function )
<a href="#">ios::clear</a>	Set error state flags (public member function )

# /ios/ios/fill

public member function

## std::ios::fill

<iostream>

```
get (1) char fill() const;
set (2) char fill (char fillch);
```

### Get/set fill character

The first form (1) returns the *fill character*.

The second form (2) sets `fillch` as the new *fill character* and returns the *fill character* used before the call.

The *fill character* is the character used by output insertion functions to fill spaces when padding results to the *field width*.

The parametric manipulator `setfill` can also be used to set the *fill character*.

## Parameters

`fillch`  
the new *fill character*.

## Return Value

The value of the *fill character* before the call.

## Example

```
1 // using the fill character
2 #include <iostream>      // std::cout
3
4 int main () {
5     char prev;
6
7     std::cout.width (10);
8     std::cout << 40 << '\n';
9
10    prev = std::cout.fill ('x');
11    std::cout.width (10);
12    std::cout << 40 << '\n';
13
14    std::cout.fill(prev);
15
16    return 0;
17 }
```

Output:

```
40
xxxxxxxx40
```

## Data races

Accesses (1) or modifies (2) the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">setfill</a>	Set fill character ( <a href="#">function</a> )
<a href="#">ios_base::width</a>	Get/set field width ( <a href="#">public member function</a> )

# /ios/ios/good

public member function

## std::ios::good

<iostream>

`bool good() const;`

### Check whether state of stream is good

Returns true if none of the stream's *error state flags* (eofbit, failbit and badbit) is set.

This function behaves as if defined as:

```
1 bool ios::good() const {
2     return rdstate() == goodbit;
3 }
```

Notice that this function is not the exact opposite of member `bad`, which only checks whether the `badbit` flag is set.

Whether specific error flags are set, can be checked with member functions `eof`, `fail`, and `bad`:

iostate value (member constant)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). `goodbit` is zero, indicating that none of the other bits is set.

## Parameters

none

## Return Value

true if none of the stream's state flags are set.

false if any of the stream's state flags are set (`badbit`, `eofbit` or `failbit`).

## Example

```
1 // error state flags
2 #include <iostream>           // std::cout, std::ios
3 #include <sstream>            // std::stringstream
4
5 void print_state (const std::ios& stream) {
6     std::cout << " good()=" << stream.good();
7     std::cout << " eof()=" << stream.eof();
8     std::cout << " fail()=" << stream.fail();
9     std::cout << " bad()=" << stream.bad();
10 }
11
12 int main () {
13     std::stringstream stream;
14
15     stream.clear (stream.goodbit);
16     std::cout << "goodbit:"; print_state(stream); std::cout << '\n';
17
18     stream.clear (stream.eofbit);
19     std::cout << "eofbit:"; print_state(stream); std::cout << '\n';
20
21     stream.clear (stream.failbit);
22     std::cout << "failbit:"; print_state(stream); std::cout << '\n';
23
24     stream.clear (stream.badbit);
25     std::cout << " badbit:"; print_state(stream); std::cout << '\n';
26
27     return 0;
28 }
```

Output:

```
goodbit: good()=1 eof()=0 fail()=0 bad()=0
eofbit: good()=0 eof()=1 fail()=0 bad()=0
failbit: good()=0 eof()=0 fail()=1 bad()=0
badbit: good()=0 eof()=0 fail()=1 bad()=1
```

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<a href="#">ios::fail</a>	Check whether either failbit or badbit is set (public member function )
<a href="#">ios::bad</a>	Check whether badbit is set (public member function )
<a href="#">ios::eof</a>	Check whether eofbit is set (public member function )
<a href="#">ios::rdstate</a>	Get error state flags (public member function )
<a href="#">ios::setstate</a>	Set error state flag (public member function )
<a href="#">ios::clear</a>	Set error state flags (public member function )

## /ios/ios/imbue

public member function

### std::ios::imbue

<iostream>

`locale imbue (const locale& loc);`

#### Imbue locale

Associates `loc` to both the stream and its associated *stream buffer* (if any) as the new locale object to be used with locale-sensitive operations.

This function calls its inherited homonym `ios_base::imbue(loc)` and, if the stream is associated with a *stream buffer*, it also calls `rdbuf() ->pubimbue(loc)`.

All callback functions registered with member `register_callback` are called by `ios_base::imbue`.

## Parameters

`loc`

`locale` object to be imbued as the new locale for the stream.

## Return value

The `locale` object associated with the stream before the call.

## Example

```
1 // imbue example
2 #include <iostream>      // std::cout
3 #include <locale>        // std::locale
4
5 int main()
6 {
7     std::locale mylocale(""); // get global locale
8     std::cout.imbue(mylocale); // imbue global locale
9     std::cout << 3.14159 << '\n';
10    return 0;
11 }
```

This code writes a floating point number using the global locale given by the environment. For example, in a system configured with a Spanish locale as default, this could write the number using a comma decimal separator:

3,14159

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">ios_base::imbue</a>	Imbue locale (public member function )
<a href="#">ios_base::getloc</a>	Get current locale (public member function )

## /ios/ios/init

protected member function

### std::ios::init

<iostream>

```
protected:  
void init (streambuf* sb);
```

#### Initialize object

Initializes the values of the stream's internal flags and member variables.

Derived classes are expected to call this protected member function at some point before its first use or before its destruction (generally, during construction).

The internal state is initialized in such a way that each of these members return the following values:

member function	return value
rdbuf	sb
tie	0
rdstate	goodbit if <i>sb</i> is not a null pointer, badbit otherwise
exceptions	goodbit
flags	skipws   dec
width	0
precision	6
fill	' ' (whitespace)
getloc	a copy of <code>locale()</code>

On initialization, the *internal extensible array* (*iword*, *pword*) is empty.

#### Parameters

*sb*  
Pointer to a `streambuf` object.

#### Return Value

none

#### Data races

Modifies the stream object. The object pointed by *sb* may be accessed and/or modified.  
Concurrent access to the same stream object or stream buffer may cause data races.

#### Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

#### See also

[ios::ios](#) Construct object (public member function )

## /ios/ios/ios

public member function

### std::ios::ios

<iostream>

```
initialization (1)    public: explicit ios (streambuf* sb);  
default (2)  protected: ios();  
  
initialization (1)    public: explicit ios (streambuf* sb);  
default (2)  protected: ios();  
copy (3)    ios (const ios&) = delete;  
            ios& operator= (const ios&) = delete;
```

#### Construct object

The initialization constructor (1) initializes the stream object by calling `init(sb)`.

If invoked by a derived class using the default constructor (2), it constructs an object leaving its members uninitialized. In this case the object shall be explicitly initialized by calling `init` at some point before its first use or before it is destroyed (if never used).

The *copy constructor* (3) is explicitly deleted (as well as the *copy assignment* overload of `operator=`).

#### Parameters

*sb*  
pointer to an object of type `streambuf`.

#### Data races

The object pointed by *sb* may be accessed and/or modified.

#### Exception safety

If an exception is thrown, the only side effects may come from accessing/modifying *sb*.

#### See also

[ios::init](#) Initialize object (protected member function )

## /ios/ios/~ios

public member function

### **std::ios::~ios**

<iostream>

`virtual ~ios();`

#### Destroy object

Destroys an object of this class.

Note that this does *not* destroy the associated *stream buffer*.

#### Data races

The object is modified.

#### Exception safety

**No-throw guarantee:** never throws exceptions.

#### See also

## /ios/ios/move

protected member function

### **std::ios::move**

<iostream>

`void move (ios& x);`  
`void move (ios&& x);`

#### Move internals

Transfers all internal members of *x* to *\*this*, except the associated *stream buffer* (*rdbuf* returns a *null pointer* after the call).

*x* is left in an unspecified but valid state, except that it is not *tied* (*tie* returns always a *null pointer*) and its associated *stream buffer* is unchanged (*rdbuf* returns the same as before the call).

Derived classes can call this function to implement *move semantics*.

#### Parameters

*x*

Stream object whose members are moved to *\*this*.

#### Return Value

none

#### Data races

Modifies both stream objects (*\*this* and *x*).

Concurrent access to any of these stream objects may cause data races.

#### Exception safety

**Basic guarantee:** if an exception is thrown, both streams are in a valid state.

#### See also

### **ios::swap**

Swap internals (protected member function )

## /ios/ios/narrow

public member function

### **std::ios::narrow**

<iostream>

`char narrow (char c, char default) const;`

#### Narrow character

Returns the transformation of *c* to its equivalent using the *ctype::narrow* facet of the *locale* object currently *imbued* in the stream, if such an equivalence exists, or *default* otherwise.

This function is designed for instantiations of *basic\_ios* that use a different (wider) character type: see *basic\_ios::narrow*

#### Parameters

*c*

Character to be "narrowed".

`default`  
Character returned if `c` has no standard equivalent.

## Return Value

The narrow equivalent of `c`, if any. Otherwise, it returns `default`.

## Data races

Accesses the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<code>ios::widen</code>	Widen character (public member function )
-------------------------	---

# /ios/ios/operator\_bool

public member function

## std::ios::operator bool

`<iostream>`

```
operator void*() const;
explicit operator bool() const;
```

### Evaluate stream

Returns whether an error flag is set (either `failbit` or `badbit`).

Notice that this function does not return the same as member `good`, but the opposite of member `fail`.

The function returns a *null pointer* if at least one of these error flags is set, and some other value otherwise.

The function returns `false` if at least one of these error flags is set, and `true` otherwise.

## Parameters

none

## Return Value

A *null pointer* if at least one of `failbit` or `badbit` is set. Some other value otherwise.

`true` if none of `failbit` or `badbit` is set.  
`false` otherwise.

## Example

```
1 // evaluating a stream
2 #include <iostream>           // std::cerr
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream is;
7     is.open ("test.txt");
8     if (is) {
9         // read file
10    }
11    else {
12        std::cerr << "Error opening 'test.txt'\n";
13    }
14    return 0;
15 }
```

## Data races

Accesses the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<code>ios::fail</code>	Check whether either <code>failbit</code> or <code>badbit</code> is set (public member function )
<code>ios::operator!</code>	Evaluate stream (not) (public member function )

# /ios/ios/operator\_not

public member function

## std::ios::operator!

<iostream>

bool operator() const;

### Evaluate stream (not)

Returns true if either `failbit` or `badbit` is set, and false otherwise.

This is equivalent to calling member `fail`.

### Parameters

none

### Return Value

true if either `failbit` or `badbit` is set.

false otherwise.

### Example

```
1 // evaluating a stream (not)
2 #include <iostream>      // std::cout
3 #include <fstream>       // std::ifstream
4
5 int main () {
6     std::ifstream is;
7     is.open ("test.txt");
8     if (!is)
9         std::cerr << "Error opening 'test.txt'\n";
10    return 0;
11 }
```

### Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

### See also

<b>ios::fail</b>	Check whether either <code>failbit</code> or <code>badbit</code> is set (public member function )
<b>ios::good</b>	Check whether state of stream is good (public member function )

## /ios/ios/rdbuf

public member function

## std::ios::rdbuf

<iostream>

```
get (1) streambuf* rdbuf() const;
set (2) streambuf* rdbuf (streambuf* sb);
```

### Get/set stream buffer

The first form (1) returns a pointer to the *stream buffer* object currently associated with the stream.

The second form (2) also sets the object pointed by *sb* as the *stream buffer* associated with the stream and clears the *error state flags*.

If *sb* is a *null pointer*, the function automatically sets the `badbit` *error state flags* (which may throw an exception if member `exceptions` has been passed `badbit`).

Some derived stream classes (such as `stringstream` and `fstream`) maintain their own *internal stream buffer*, to which they are *associated* on construction. Calling this function to change the *associated stream buffer* shall have no effect on that *internal stream buffer*: the stream will have an *associated stream buffer* which is different from its *internal stream buffer* (although input/output operations on streams always use the *associated stream buffer*, as returned by this member function).

### Parameters

*sb*

Pointer to a `streambuf` object.

### Return Value

A pointer to the *stream buffer* object associated with the stream before the call.

### Example

```
1 // redirecting cout's output through its stream buffer
2 #include <iostream>      // std::streambuf, std::cout
3 #include <fstream>       // std::ofstream
4
5 int main () {
6     std::streambuf *psbuf, *backup;
```

```

7 std::ofstream filestr;
8 filestr.open ("test.txt");
9
10 backup = std::cout.rdbuf(); // back up cout's streambuf
11
12 psbuf = filestr.rdbuf(); // get file's streambuf
13 std::cout.rdbuf(psbuf); // assign streambuf to cout
14
15 std::cout << "This is written to the file";
16
17 std::cout.rdbuf(backup); // restore cout's original streambuf
18
19 filestr.close();
20
21 return 0;
22 }

```

This example uses both function forms: first to get a pointer to a file's `streambuf` object and then to assign it to `cout`.

## Data races

Accesses (1) or modifies (2) the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.  
It throws an exception of member type `failure` if `sb` is a *null pointer* and member `exceptions` was set to throw for `badbit`.

## See also

<code>streambuf</code>	Base buffer class for streams ( <a href="#">class</a> )
------------------------	---

## /ios/ios/rdstate

public member function

### `std::ios::rdstate`

`<iostream>`

`iostate rdstate() const;`

#### Get error state flags

Returns the current internal *error state flags* of the stream.

The internal *error state flags* are automatically set by calls to input/output functions on the stream to signal certain errors.

## Parameters

none

## Return Value

An object of type `ios_base::iostate` that can contain any combination of the following state flag member constants:

<code>iostate</code> value (member constant)	indicates	functions to check state flags				
		<code>good()</code>	<code>eof()</code>	<code>fail()</code>	<code>bad()</code>	<code>rdstate()</code>
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	<code>goodbit</code>
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	<code>eofbit</code>
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	<code>failbit</code>
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	<code>badbit</code>

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). `goodbit` is zero, indicating that none of the other bits is set.

## Example

```

1 // getting the state of stream objects
2 #include <iostream> // std::cerr
3 #include <fstream> // std::ifstream
4
5 int main () {
6   std::ifstream is;
7   is.open ("test.txt");
8   if ( (is.rdstate() & std::ifstream::failbit) != 0 )
9     std::cerr << "Error opening 'test.txt'\n";
10  return 0;
11 }

```

## Data races

Accesses the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<a href="#">ios::fail</a>	Check whether either failbit or badbit is set (public member function )
<a href="#">ios::good</a>	Check whether state of stream is good (public member function )
<a href="#">ios::bad</a>	Check whether badbit is set (public member function )
<a href="#">ios::eof</a>	Check whether eofbit is set (public member function )
<a href="#">ios::clear</a>	Set error state flags (public member function )

## /ios/ios/set\_rdbuf

protected member function

### std::ios::set\_rdbuf

<iostream>

`void set_rdbuf (streambuf* sb);`

#### Set stream buffer

Sets *sb* as the *stream buffer* associated with the stream, without altering the *control state flag* (`rdstate`).

*sb* shall not be a *null pointer*.

Derived classes can call this function to change the *stream buffer*.

## Parameters

*sb*

Pointer to a `streambuf` object.  
This shall not be a *null pointer*.

## Return Value

none

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

## See also

<a href="#">ios::rdbuf</a>	Get/set stream buffer (public member function )
----------------------------	---

## /ios/ios/setstate

public member function

### std::ios::setstate

<iostream>

`void setstate (iostate state);`

#### Set error state flag

Modifies the current internal *error state flags* by combining the current flags with those in argument *state* (as if performing a bitwise OR operation).

Any error bitflag already set is not cleared. See member `clear` for a similar function that does.

In the case that no *stream buffer* is associated with the stream when this function is called, the *badbit* flag is automatically set (no matter the value for that bit passed in argument *state*).

Note that changing the *state* may throw an exception, depending on the latest settings passed to member `exceptions`.

The current state can be obtained with member function `rdstate`.

This function behaves as if defined as:

```
1 void ios::setstate (iostate state) {  
2     clear(rdstate()|state);  
3 }
```

## Parameters

*state*

An object of type `ios_base::iostate` that can take as value any combination of the following member constants:

iostate value (member constant)	indicates	functions to check state flags				
		good()	eof()	fail()	bad()	rdstate()
<code>goodbit</code>	No errors (zero value <code>iostate</code> )	true	false	false	false	goodbit
<code>eofbit</code>	End-of-File reached on input operation	false	true	false	false	eofbit
<code>failbit</code>	Logical error on i/o operation	false	false	true	false	failbit
<code>badbit</code>	Read/writing error on i/o operation	false	false	true	true	badbit

`eofbit`, `failbit` and `badbit` are member constants with implementation-defined values that can be combined (as if with the bitwise OR operator). `goodbit` is zero, indicating that none of the other bits is set.

## Return Value

none

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set throw for that state.

## See also

<code>ios::fail</code>	Check whether either failbit or badbit is set (public member function )
<code>ios::good</code>	Check whether state of stream is good (public member function )
<code>ios::bad</code>	Check whether badbit is set (public member function )
<code>ios::eof</code>	Check whether eofbit is set (public member function )
<code>ios::rdstate</code>	Get error state flags (public member function )
<code>ios::clear</code>	Set error state flags (public member function )

## /ios/ios/swap

protected member function

`<iostream>`

### std::ios::swap

`void swap (ios& x) noexcept;`

#### Swap internals

Exchanges all internal members between `x` and `*this`, except the pointers to the associated *stream buffers*: `rdbuf` shall return the same in both objects as before the call.

Derived classes can call this function to implement custom `swap` functions.

## Parameters

`x`

Another stream object of the same type.

## Return Value

none

## Data races

Modifies both stream objects (`*this` and `x`).

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>ios::move</code>	Move internals (protected member function )
------------------------	---

## /ios/ios/tie

public member function

`<iostream>`

### std::ios::tie

`get (1) ostream* tie() const;`  
`set (2) ostream* tie (ostream* tiestr);`

#### Get/set tied stream

The first form (1) returns a pointer to the tied output stream.

The second form (2) ties the object to `tiestr` and returns a pointer to the stream tied before the call, if any.

The *tied stream* is an output stream object which is `flushed` before each i/o operation in this stream object.

By default, `cin` is tied to `cout`, and `wcin` to `wcout`. Library implementations may tie other standard streams on initialization.

By default, the standard narrow streams `cin` and `cerr` are tied to `cout`, and their wide character counterparts (`wcin` and `wcerr`) to `wcout`. Library implementations may also tie `clog` and `wclog`.

## Parameters

`tiestr`  
An output stream object.

## Return Value

A pointer to the stream object tied before the call, or a *null pointer* in case the stream was not tied.

## Example

```
1 // redefine tied object
2 #include <iostream>           // std::ostream, std::cout, std::cin
3 #include <fstream>            // std::ofstream
4
5 int main () {
6     std::ostream *prevstr;
7     std::ofstream ofs;
8     ofs.open ("test.txt");
9
10    std::cout << "tie example:\n";
11
12    *std::cin.tie() << "This is inserted into cout";
13    prevstr = std::cin.tie (&ofs);
14    *std::cin.tie() << "This is inserted into the file";
15    std::cin.tie (prevstr);
16
17    ofs.close();
18
19    return 0;
20 }
```

Output:

```
tie example:
This is inserted into cout
```

## Data races

Accesses (1) or modifies (2) the stream object.  
Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the stream is in a valid state.

<code>ios::rdbuf</code>	Get/set stream buffer ( <a href="#">public member function</a> )
-------------------------	--

# /ios/iostream\_category

function  
**std::iostream\_category** <iost> <iostream>  
const error\_category& iostream\_category();  
const error\_category& iostream\_category() noexcept;

## Return iostream category

Returns a reference to the static object of the `error_category` type that has the following characteristics:

- Its `name` member function returns a pointer to the character sequence "iostream"
- Its `equivalent` and `default_error_condition` member functions behave as specified for the base `error_category` class.

`error_condition` objects describing errors that correspond to the enum type `io_errc` are associated to this category. What constitutes one of these correspondences depends on the operating system and the particular library implementation.

## Parameters

none

## Return value

A reference to the `iostream error_category` object.

## Example

```
1 // io_errc example
2 // iostream_category example
3 #include <iostream>           // std::cin, std::cerr, std::ios,
4                           // std::iostream_category
5 int main () {
6     std::cin.exceptions (std::ios::failbit|std::ios::badbit);
7     try {
8         std::cin.rdbuf(nullptr);    // throws
9     } catch (std::ios::failure& e) {
10        std::cerr << "failure caught: ";
11        if ( e.code().category() == std::iostream_category() )
```

```

13     std::cerr << "error code of the iostream category\n";
14
15     else
16         std::cerr << "error code of some other category\n";
17 }
18 return 0;
}

```

Possible output:

```
failure caught: error code of the iostream category
```

### Exception safety

**No-throw guarantee:** this function never throws exceptions.

### See also

<a href="#">io_errc</a>	Input/output error conditions (enum class )
-------------------------	---

## /ios/ios/widen

public member function

### std::ios::widen

<iostream>

`char widen (char c) const;`

#### Widen character

Returns the transformation of `c` to its equivalent using the `ctype::widen` facet of the `locale` object currently *imbued* in the stream.

This function is designed for instantiations of `basic_ios` that use a different (wider) character type: see `basic_ios::widen`

### Parameters

`c`  
Character to be "widened".

### Return Value

The wide equivalent of `c`.

### Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

### See also

<a href="#">ios::narrow</a>	Narrow character (public member function )
-----------------------------	--

## /ios/left

function  
**std::left** <iostream>

`ios_base& left (ios_base& str);`

#### Adjust output to the left

Sets the `adjustfield` format flag for the `str` stream to `left`.

When `adjustfield` is set to `left`, the output is padded to the *field width* (`width`) by inserting *fill characters* (`fill`) at the end, effectively adjusting the field to the left.

The `adjustfield` format flag can take any of the following values (each with its own manipulator):

flag value	effect when set
<code>internal</code>	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at a specified internal point.
<code>left</code>	the output is padded to the <i>field width</i> appending <i>fill characters</i> at the end.
<code>right</code>	the output is padded to the <i>field width</i> by inserting <i>fill characters</i> at the beginning.

For standard streams, the `adjustfield` flag is set to `right` on initialization.

### Parameters

`str`

Stream object whose `adjustfield` format flag is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify adjustfield using manipulators
2 #include <iostream>      // std::cout, std::internal, std::left, std::right
3
4 int main () {
5     int n = -77;
6     std::cout.width(6); std::cout << std::internal << n << '\n';
7     std::cout.width(6); std::cout << std::left << n << '\n';
8     std::cout.width(6); std::cout << std::right << n << '\n';
9     return 0;
10 }
```

Output:

```
- 77
-77
-77
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<b>internal</b>	Adjust field by inserting characters at an internal position ( <a href="#">function</a> )
<b>right</b>	Adjust output to the right ( <a href="#">function</a> )
<b>ios_base::flags</b>	Get/set format flags ( <a href="#">public member function</a> )
<b>ios_base::setf</b>	Set specific format flags ( <a href="#">public member function</a> )
<b>ios_base::unsetf</b>	Clear specific format flags ( <a href="#">public member function</a> )

## /ios/noboolalpha

function  
**std::noboolalpha** <iostream>

*ios\_base& noboolalpha (ios\_base& str);*

### No alphanumerical bool values

Clears the *boolalpha* format flag for the *str* stream.

When the *boolalpha* format flag is *not set*, *bool* values are inserted/extracted as integral values (0 and 1) instead of their textual representations: *true* and *false*.

This flag can be set with the *boolalpha* manipulator.

For standard streams, the *boolalpha* flag is **not set** on initialization.

## Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<<) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify boolalpha flag
2 #include <iostream>      // std::cout, std::boolalpha, std::noboolalpha
3
4 int main () {
5     bool b = true;
6     std::cout << std::boolalpha << b << '\n';
7     std::cout << std::noboolalpha << b << '\n';
8     return 0;
9 }
```

Output:

```
true
1
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">boolalpha</a>	Alphanumerical bool values (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/noshowbase

function  
**std::noshowbase** <iostream>

`ios_base& noshowbase (ios_base& str);`

### Do not show numerical base prefixes

Clears the `showbase` format flag for the *str* stream.

When the `showbase` format flag is not set, numerical values are inserted into the stream without prefixing them with any numerical base prefix (i.e., `0x` for hexadecimal values, `0` for octal values and no prefix for decimal-base values).

This flag can be set with the `showbase` manipulator, which forces the prefixing of numerical integer values with their respective numerical base prefix.

For standard streams, the `showbase` flag is **not set** on initialization.

## Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify showbase flag
2 #include <iostream>      // std::cout, std::showbase, std::noshowbase
3
4 int main () {
5     int n = 20;
6     std::cout << std::hex << std::showbase << n << '\n';
7     std::cout << std::hex << std::noshowbase << n << '\n';
8     return 0;
9 }
```

Output:

```
0x14
14
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">showbase</a>	Show numerical base prefixes (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/noshowpoint

function  
**std::noshowpoint** <iostream>

`ios_base& noshowpoint (ios_base& str);`

## Do not show decimal point

Clears the `showpoint` format flag for the `str` stream.

When the `showpoint` format flag is not set, the decimal point is only written when necessary for floating-point values inserted into the stream: when their decimal part is not zero.

This flag can be set with the `showpoint` manipulator. When the flag is set, the decimal point is always written for floating point values inserted into the stream, even for those whose decimal part is zero.

For standard streams, the `showpoint` flag is **not set** on initialization.

### Parameters

`str`

Stream object where to apply.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the insertion (`<<`) and extraction (`>>`) operations on streams (see example below).

### Parameters

`str`

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

### Return Value

Argument `str`.

### Example

```
1 // modify showpoint flag
2 #include <iostream>      // std::cout, std::showpoint, std::noshowpoint
3
4 int main () {
5     double a = 30;
6     double b = 10000.0;
7     double pi = 3.1416;
8     std::cout.precision (5);
9     std::cout << std::showpoint << a << '\t' << b << '\t' << pi << '\n';
10    std::cout << std::noshowpoint << a << '\t' << b << '\t' << pi << '\n';
11    return 0;
12 }
```

Possible output:

```
30.000 10000. 3.1416
30      10000   3.1416
```

### Data races

Modifies `str`. Concurrent access to the same stream object may cause data races.

### Exception safety

**Basic guarantee:** if an exception is thrown, `str` is in a valid state.

### See also

<a href="#">showpoint</a>	Show decimal point (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/noshowpos

function

**std::noshowpos**

`<iostream>`

```
ios_base& noshowpos (ios_base& str);
```

### Do not show positive signs

Clears the `showpos` format flag for the `str` stream.

When the `showpos` format flag is not set, no plus signs precede positive values inserted in `str`.

This flag can be set with the `showpos` manipulator, which forces the writing of a plus sign (+) before every non-negative numerical value inserted into the stream (including zeros).

For standard streams, the `showpos` flag is **not set** on initialization.

### Parameters

`str`

Stream object whose *format flag* is affected.  
Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

Argument `str`.

## Example

```
1 // modify showpos flag
2 #include <iostream>      // std::cout, std::showpos, std::noshowpos
3
4 int main () {
5     int p = 1;
6     int z = 0;
7     int n = -1;
8     std::cout << std::showpos << p << '\t' << z << '\t' << n << '\n';
9     std::cout << std::noshowpos << p << '\t' << z << '\t' << n << '\n';
10    return 0;
11 }
```

Possible output:

+1	+0	-1
1	0	-1

## Data races

Modifies `str`. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, `str` is in a valid state.

## See also

<a href="#">showpos</a>	Show positive signs (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/noskipws

function  
**std::noskipws** <iostream>

```
ios_base& noskipws (ios_base& str);
```

### Do not skip whitespaces

Clears the `skipws` format flag for the `str` stream.

When the `skipws` format flag is not set, all operations on the stream consider initial whitespace characters as valid content to be extracted.

Tab spaces, carriage returns and blank spaces are all considered whitespaces (see [isspace](#)).

This flag can be set with the `skipws` manipulator. When set, as many initial whitespace characters as necessary are read and discarded from the stream until a non-whitespace character is found. This would apply to every formatted input operation performed with `operator>>` on the stream.

Notice that many extraction operations consider the whitespaces themselves as the terminating character, therefore, with the `skipws` flag disabled, some extraction operations may extract no characters at all from the stream.

For standard streams, the `skipws` flag is set on initialization.

## Parameters

`str`

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

Argument `str`.

## Example

```
1 // skipws flag example
2 #include <iostream>      // std::cout, std::skipws, std::noskipws
3 #include <sstream>        // std::istringstream
4
5 int main () {
6     char a, b, c;
7     std::istringstream iss (" 123");
```

```

9 iss >> std::skipws >> a >> b >> c;
10 std::cout << a << b << c << '\n';
11
12 iss.seekg(0);
13 iss >> std::noskipws >> a >> b >> c;
14 std::cout << a << b << c << '\n';
15 return 0;
16 }

```

Output:

```
123
1
```

Notice that in the first set of extractions, the leading spaces were ignored, while in the second they were extracted as valid characters.

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">skipws</a>	Skip whitespaces ( <a href="#">function</a> )
<a href="#">ios_base::flags</a>	Get/set format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::unsetf</a>	Clear specific format flags ( <a href="#">public member function</a> )

## /ios/nounitbuf

function  
**std::nounitbuf** <iostream>

```
ios_base& nounitbuf (ios_base& str);
```

### Do not force flushes after insertions

Clears the *unitbuf* "format" flag for the *str* stream.

When the *unitbuf* flag is not set, the associated buffer is not forced to be flushed after every insertion operation.

This flag can be set with the *unitbuf* manipulator, forcing flushes after every insertion.

For standard streams, the *unitbuf* flag is **not set** on initialization.

## Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<<) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```

1 // modify unibuf flag
2 #include <iostream>           // std::unitbuf
3 #include <fstream>            // std::ofstream
4
5 int main () {
6     std::ofstream outfile ("test.txt");
7     outfile << std::unitbuf << "Test " << "file" << '\n'; // flushed three times
8     outfile.close();
9     return 0;
10 }
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">unitbuf</a>	Flush buffer after insertions ( <a href="#">function</a> )
<a href="#">ios_base::flags</a>	Get/set format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )

## /ios/nouppercase

function  
**std::nouppercase** <iostream>

```
ios_base& nouppercase (ios_base& str);
```

**Do not generate upper case letters**

Clears the uppercase format flag for the *str* stream.

When the uppercase format flag is not set, the letters automatically generated by the stream for certain representations (like some hexadecimal representations and numerical base prefixes) are not forced to be displayed using uppercase letters (generally using lowercase letters instead).

This flag can be set with the `uppercase` manipulator, forcing the use of uppercase for generated letters.

For standard streams, the uppercase flag is **not set** on initialization.

### Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<<) and *extraction* (>>) operations on streams (see example below).

### Return Value

Argument *str*.

### Example

```
1 // modify uppercase flag
2 #include <iostream>      // std::cout, std::showbase, std::hex
3                         // std::uppercase, std::nouppercase
4 int main () {
5     std::cout << std::showbase << std::hex;
6     std::cout << std::uppercase << 77 << '\n';
7     std::cout << std::nouppercase << 77 << '\n';
8     return 0;
9 }
```

Possible output:

```
0x4D
0xd
```

### Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

### Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

### See also

<a href="#">uppercase</a>	Generate upper-case letters ( <a href="#">function</a> )
<a href="#">ios_base::flags</a>	Get/set format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::unsetf</a>	Clear specific format flags ( <a href="#">public member function</a> )

## /ios/oct

function  
**std::oct** <iostream>

```
ios_base& oct (ios_base& str);
```

**Use octal base**

Sets the basefield format flag for the *str* stream to oct.

When basefield is set to oct, integer values inserted into the stream are expressed in octal base (i.e., radix 8). For input streams, extracted values are also expected to be expressed in octal base when this flag is set.

The basefield format flag can take any of the following values (each with its own manipulator):

flag value	effect when set
dec	read/write integer values using decimal base format.
hex	read/write integer values using hexadecimal base format.
oct	read/write integer values using octal base format.

Notice that the `basefield` flag only affects the insertion/extraction of integer values (floating-point values are always interpreted in decimal base).

Notice also that no base prefix is implicitly prepended to the number unless the `showbase` format flag is set.

For standard streams, the `basefield` flag is set to `dec` on initialization.

## Parameters

---

`str`

Stream object whose `basefield` *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

---

Argument `str`.

## Example

---

```
1 // modify basefield
2 #include <iostream>      // std::cout, std::dec, std::hex, std::oct
3
4 int main () {
5     int n = 70;
6     std::cout << std::dec << n << '\n';
7     std::cout << std::hex << n << '\n';
8     std::cout << std::oct << n << '\n';
9     return 0;
10 }
```

Output:

```
70
46
106
```

## Data races

---

Modifies `str`. Concurrent access to the same stream object may cause data races.

## Exception safety

---

**Basic guarantee:** if an exception is thrown, `str` is in a valid state.

## See also

---

<code>dec</code>	Use decimal base ( <a href="#">function</a> )
<code>hex</code>	Use hexadecimal base ( <a href="#">function</a> )
<code>ios_base::flags</code>	Get/set format flags ( <a href="#">public member function</a> )
<code>ios_base::setf</code>	Set specific format flags ( <a href="#">public member function</a> )

## /ios/right

function  
**std::right** <iostream>

```
ios_base& right (ios_base& str);
```

### Adjust output to the right

Sets the `adjustfield` format flag for the `str` stream to `right`.

When `adjustfield` is set to `right`, the output is padded to the `field width` (`width`) by inserting `fill characters` (`fill`) at the beginning, effectively adjusting the field to the right.

The `adjustfield` format flag can take any of the following values (each with its own manipulator):

flag value	effect when set
<code>internal</code>	the output is padded to the <code>field width</code> by inserting <code>fill characters</code> at a specified internal point.
<code>left</code>	the output is padded to the <code>field width</code> appending <code>fill characters</code> at the end.
<code>right</code>	the output is padded to the <code>field width</code> by inserting <code>fill characters</code> at the beginning.

For standard streams, the `adjustfield` flag is set to this value (`right`) on initialization.

## Parameters

---

`str`

Stream object whose `adjustfield` *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

---

Argument `str`.

## Example

```
1 // modify adjustfield using manipulators
2 #include <iostream>      // std::cout, std::internal, std::left, std::right
3
4 int main () {
5     int n = -77;
6     std::cout.width(6); std::cout << std::internal << n << '\n';
7     std::cout.width(6); std::cout << std::left << n << '\n';
8     std::cout.width(6); std::cout << std::right << n << '\n';
9     return 0;
10 }
```

Output:

```
- 77
-77
-77
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">internal</a>	Adjust field by inserting characters at an internal position ( <a href="#">function</a> )
<a href="#">left</a>	Adjust output to the left ( <a href="#">function</a> )
<a href="#">ios_base::flags</a>	Get/set format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::unsetf</a>	Clear specific format flags ( <a href="#">public member function</a> )

## /ios/scientific

function  
**std::scientific** <iostream>

```
ios_base& scientific (ios_base& str);
```

### Use scientific floating-point notation

Sets the *floatfield* format flag for the *str* stream to **scientific**.

When *floatfield* is set to **scientific**, floating-point values are written using scientific notation: the value is represented always with only one digit before the decimal point, followed by the decimal point and as many decimal digits as the *precision field* (*precision*). Finally, this notation always includes an exponential part consisting on the letter **e** followed by an optional sign and three exponential digits.

The *floatfield* format flag is both a selective and a toggle flag: it can take one or more of the following values:

flag value	effect when set
<b>fixed</b>	write floating-point values in fixed-point notation
<b>scientific</b>	write floating-point values in scientific notation.
<b>(none)</b>	write floating-point values in default floating-point notation.

The default notation **(none)** is a different *floatfield* value than either **fixed** or **scientific**. The default notation can be selected by calling `str.unsetf(ios_base::floatfield)`.

For standard streams, no *floatfield* is set on initialization (default notation).

The *floatfield* format flag is both a selective and a toggle flag: it can take any of the following values, or none:

flag value	effect when set
<b>fixed</b>	write floating-point values in fixed-point notation.
<b>scientific</b>	write floating-point values in scientific notation.
<b>hexfloat</b>	write floating-point values in hexadecimal format. The value of this is the same as <b>(fixed scientific)</b>
<b>defaultfloat</b>	write floating-point values in default floating-point notation. This is the value by default (same as <b>none</b> , before any other <i>floatfield</i> bit is set).

For standard streams, the *floatfield* format flag is set to **defaultfloat** on initialization.

The *precision field* can be modified using member **precision**.

Notice that the treatment of the *precision field* differs between the default floating-point notation and the fixed and scientific notations (see [precision](#)). On the default floating-point notation, the *precision field* specifies the maximum number of meaningful digits to display both before and after the decimal point, while in both the fixed and scientific notations, the *precision field* specifies exactly how many digits to display *after* the decimal point, even if they are trailing decimal zeros.

## Parameters

**str**

Stream object whose *floatfield format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify floatfield
2 #include <iostream>      // std::cout, std::fixed, std::scientific
3
4 int main () {
5     double a = 3.1415926534;
6     double b = 2006.0;
7     double c = 1.0e-10;
8
9     std::cout.precision(5);
10
11    std::cout << "default:\n";
12    std::cout << a << '\n' << b << '\n' << c << '\n';
13
14    std::cout << '\n';
15
16    std::cout << "fixed:\n" << std::fixed;
17    std::cout << a << '\n' << b << '\n' << c << '\n';
18
19    std::cout << '\n';
20
21    std::cout << "scientific:\n" << std::scientific;
22    std::cout << a << '\n' << b << '\n' << c << '\n';
23
24 }
```

Possible output:

```
default:
3.1416
2006
1e-010

fixed:
3.14159
2006.00000
0.00000

scientific:
3.14159e+000
2.00600e+003
1.00000e-010
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">scientific</a>	Use scientific floating-point notation ( <a href="#">function</a> )
<a href="#">ios_base::flags</a>	Get/set format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::setf</a>	Set specific format flags ( <a href="#">public member function</a> )
<a href="#">ios_base::unsetf</a>	Clear specific format flags ( <a href="#">public member function</a> )

## /ios/showbase

function  
**std::showbase** <iostream>

```
ios_base& showbase (ios_base& str);
```

### Show numerical base prefixes

Sets the showbase format flag for the *str* stream.

When the showbase format flag is set, numerical integer values inserted into output streams are prefixed with the same prefixes used by C++ literal constants: `0x` for hexadecimal values (see [hex](#)), `0` for octal values (see [oct](#)) and no prefix for decimal-base values (see [dec](#)).

This option can be unset with the [noshowbase](#) manipulator. When not set, all numerical values are inserted without base prefixes.

For standard streams, the showbase flag is **not set** on initialization.

## Parameters

**str**

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify showbase flag
2 #include <iostream>      // std::cout, std::showbase, std::noshowbase
3
4 int main () {
5     int n = 20;
6     std::cout << std::hex << std::showbase << n << '\n';
7     std::cout << std::hex << std::noshowbase << n << '\n';
8     return 0;
9 }
```

Output:

```
0x14
14
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">noshowbase</a>	Do not show numerical base prefixes (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/showpoint

function  
**std::showpoint** <iostream>

```
ios_base& showpoint (ios_base& str);
```

### Show decimal point

Sets the *showpoint* format flag for the *str* stream.

When the *showpoint* format flag is set, the decimal point is always written for floating point values inserted into the stream (even for those whose decimal part is zero). Following the decimal point, as many digits as necessary are written to match the *precision* set for the stream (if any).

This flag can be unset with the *noshowpoint* manipulator. When not set, the decimal point is only written for numbers whose decimal part is not zero.

The precision setting can be modified using the *precision* member function.

For standard streams, the *showpoint* flag is **not set** on initialization.

## Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify showpoint flag
2 #include <iostream>      // std::cout, std::showpoint, std::noshowpoint
3
4 int main () {
5     double a = 30;
6     double b = 10000.0;
7     double pi = 3.1416;
8     std::cout.precision (5);
9     std::cout << std::showpoint << a << '\t' << b << '\t' << pi << '\n';
10    std::cout << std::noshowpoint << a << '\t' << b << '\t' << pi << '\n';
11    return 0;
12 }
```

Possible output:

```
30.000 10000. 3.1416
30      10000   3.1416
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">noshowpoint</a>	Do not show decimal point (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/showpos

function

### std::showpos

<iostream>

`ios_base& showpos (ios_base& str);`

#### Show positive signs

Sets the `showpos` format flag for the *str* stream.

When the `showpos` format flag is set, a plus sign (+) precedes every non-negative numerical value inserted into the stream (including zeros).

This flag can be unset with the `noshowpos` manipulator.

For standard streams, the `showpos` flag is **not set** on initialization.

## Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify showpos flag
2 #include <iostream>      // std::cout, std::showpos, std::noshowpos
3
4 int main () {
5     int p = 1;
6     int z = 0;
7     int n = -1;
8     std::cout << std::showpos << p << '\t' << z << '\t' << n << '\n';
9     std::cout << std::noshowpos << p << '\t' << z << '\t' << n << '\n';
10    return 0;
11 }
```

Possible output:

+1	+0	-1
1	0	-1

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">noshowpos</a>	Do not show positive signs (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/skipws

function

### std::skipws

<iostream>

```
ios_base& skipws (ios_base& str);
```

### Skip whitespaces

Sets the `skipws` format flag for the `str` stream.

When the `skipws` format flag is set, as many whitespace characters as necessary are read and discarded from the stream until a non-whitespace character is found before. This applies to every formatted input operation performed with `operator>>` on the stream.

Tab spaces, carriage returns and blank spaces are all considered whitespaces (see `isspace`).

This flag can be unset with the `noskipws` manipulator, forcing extraction operations to consider leading whitepaces as part of the content to be extracted.

For standard streams, the `skipws` flag is set on initialization.

### Parameters

`str`

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (`<<`) and *extraction* (`>>`) operations on streams (see example below).

### Return Value

Argument `str`.

### Example

```
1 // skipws flag example
2 #include <iostream>           // std::cout, std::skipws, std::noskipws
3 #include <sstream>            // std::istringstream
4
5 int main () {
6     char a, b, c;
7
8     std::istringstream iss (" 123");
9     iss >> std::skipws >> a >> b >> c;
10    std::cout << a << b << c << '\n';
11
12    iss.seekg(0);
13    iss >> std::noskipws >> a >> b >> c;
14    std::cout << a << b << c << '\n';
15    return 0;
16 }
```

Output:

```
123
1
```

Notice that in the first set of extractions, the leading spaces were ignored, while in the second they were extracted as valid characters.

### Data races

Modifies `str`. Concurrent access to the same stream object may cause data races.

### Exception safety

**Basic guarantee:** if an exception is thrown, `str` is in a valid state.

### See also

<a href="#">noskipws</a>	Do not skip whitespaces (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/streamoff

type

**std::streamoff**

`<iostream>`

### Stream offset type

Type to represent position offsets in a stream.

It is the type returned by subtracting two stream position of an `fpos` type, and can be converted to and from this type.

The underlying type depends on the particular library implementation.

It is a `typedef` of one the fundamental signed [integral types](#) large enough to represent the maximum possible file size supported by the system.

### See also

<a href="#">streamsize</a>	Stream size type (type )
<a href="#">fpos</a>	Stream position class template (class template )
<a href="#">streampos</a>	Stream position type (type )

## /ios/streampos

type

### std::streampos

<iostream>

```
typedef fpos<mbstate_t> streampos;
```

#### Stream position type

Instantiation of `fpos` used to represent positions in narrow-oriented streams.

Objects of this class support construction and conversion from `int`, and allow consistent conversions to/from `streamoff` values (as well as being added or subtracted a value of this type).

Two objects of this type can be compared with operators `==` and `!=`. They can also be subtracted, which yields a value of type `streamoff`.

`streampos` and `wstreampos` are synonyms: both are `typedefs` of `fpos<mbstate_t>`.

#### See also

<b>fpos</b>	Stream position class template ( <a href="#">class template</a> )
<b>streamoff</b>	Stream offset type ( <a href="#">type</a> )
<b>streamsize</b>	Stream size type ( <a href="#">type</a> )

## /ios/streamsize

type

### std::streamsize

<iostream>

#### Stream size type

Type to represent sizes and character counts in streams.

It is a `typedef` of one the fundamental signed `integral types`.

It is convertible to/from `streamoff`.

#### See also

<b>streamoff</b>	Stream offset type ( <a href="#">type</a> )
------------------	---

## /ios/unitbuf

function

### std::unitbuf

<iostream>

```
ios_base& unitbuf (ios_base& str);
```

#### Flush buffer after insertions

Sets the `unitbuf` "format" flag for the `str` stream.

When the `unitbuf` flag is set, the associated buffer is flushed after each insertion operation.

This flag can be unset with the `nounitbuf` manipulator, not forcing flushes after every insertion.

For standard streams, the `unitbuf` flag is **not set** on initialization.

#### Parameters

`str`

Stream object whose `format flag` is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the `insertion (<<)` and `extraction (>>)` operations on streams (see example below).

#### Return Value

Argument `str`.

#### Example

```
1 // modify unitbuf flag
2 #include <iostream>           // std::unitbuf
3 #include <fstream>           // std::ofstream
4
5 int main () {
6     std::ofstream outfile ("test.txt");
7     outfile << std::unitbuf << "Test " << "file" << '\n'; // flushed three times
8     outfile.close();
9     return 0;
10 }
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">nouitbuf</a>	Do not force flushes after insertions (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/uppercase

function

### std::uppercase

<iostream>

`ios_base& uppercase (ios_base& str);`

#### Generate upper-case letters

Sets the uppercase format flag for the *str* stream.

When the uppercase format flag is set, uppercase (capital) letters are used instead of lowercase for representations on output operations involving stream-generated letters, like some hexadecimal representations and numerical base prefixes.

This flag can be unset with the `nouppercase` manipulator, not forcing the use of uppercase for generated letters.

For standard streams, the uppercase flag is **not set** on initialization.

## Parameters

*str*

Stream object whose *format flag* is affected.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the *insertion* (<>) and *extraction* (>>) operations on streams (see example below).

## Return Value

Argument *str*.

## Example

```
1 // modify uppercase flag
2 #include <iostream>           // std::cout, std::showbase, std::hex
3                         // std::uppercase, std::nouppercase
4 int main () {
5     std::cout << std::showbase << std::hex;
6     std::cout << std::uppercase << 77 << '\n';
7     std::cout << std::nouppercase << 77 << '\n';
8     return 0;
9 }
```

Possible output:

```
0X4D
0x4d
```

## Data races

Modifies *str*. Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, *str* is in a valid state.

## See also

<a href="#">nouppercase</a>	Do not generate upper case letters (function )
<a href="#">ios_base::flags</a>	Get/set format flags (public member function )
<a href="#">ios_base::setf</a>	Set specific format flags (public member function )
<a href="#">ios_base::unsetf</a>	Clear specific format flags (public member function )

## /ios/wios

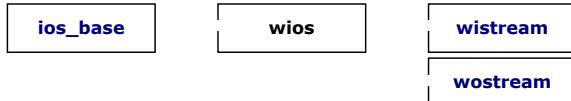
class

### std::wios

<iostream>

`typedef basic_ios<wchar_t> ios;`

## Base class for wide character streams



This class is an instantiation of `basic_ios` designed to serve as base class for all wide stream classes, with `wchar_t` as *character type* (see `basic_ios` for more info on the template).

It instantiates `basic_ios` with the following template parameters:

template parameter	definition	comments
<code>charT</code>	<code>wchar_t</code>	Aliased as member <code>char_type</code>
<code>traits</code>	<code>char_traits&lt;wchar_t&gt;</code>	Aliased as member <code>traits_type</code>

### Member types

member type	definition
<code>char_type</code>	<code>wchar_t</code>
<code>traits_type</code>	<code>char_traits&lt;wchar_t&gt;</code>
<code>int_type</code>	<code>wint_t</code>
<code>pos_type</code>	<code>wstreampos</code>
<code>off_type</code>	<code>streamoff</code>

Along with the member types inherited from `ios_base`:

<code>event</code>	Type to indicate event type (public member type )
<code>event_callback</code>	Event callback function type (public member type )
<code>failure</code>	Base class for stream exceptions (public member class )
<code>fmtflags</code>	Type for stream format flags (public member type )
<code>Init</code>	Initialize standard stream objects (public member class )
<code>iostate</code>	Type for stream state flags (public member type )
<code>openmode</code>	Type for stream opening mode flags (public member type )
<code>seekdir</code>	Type for stream seeking direction flag (public member type )

### Public member functions

Note: This section links to the references for members of its basic template (`basic_ios`).

<code>(constructor)</code>	Construct object (public member function )
<code>(destructor)</code>	Destroy object (public member function )

#### State flag functions:

<code>good</code>	Check whether state of stream is good (public member function )
<code>eof</code>	Check whether eofbit is set (public member function )
<code>fail</code>	Check whether failbit or badbit is set (public member function )
<code>bad</code>	Check whether badbit is set (public member function )
<code>operator!</code>	Evaluate stream (not) (public member function )
<code>operator bool</code>	Evaluate stream (public member function )
<code>rdstate</code>	Get error state flags (public member function )
<code>setstate</code>	Set error state flag (public member function )
<code>clear</code>	Set error state flags (public member function )

#### Formatting:

<code>copyfmt</code>	Copy formatting information (public member function )
<code>fill</code>	Get/set fill character (public member function )

#### Other:

<code>exceptions</code>	Get/set exceptions mask (public member function )
<code>imbue</code>	Imbue locale (public member function )
<code>tie</code>	Get/set tied stream (public member function )
<code>rdbuf</code>	Get/set stream buffer (public member function )
<code>narrow</code>	Narrow character (public member function )
<code>widen</code>	Widen character (public member function )

### Protected member functions

<code>init</code>	Initialize object (protected member function )
<code>move</code>	Move internals (protected member function )
<code>swap</code>	Swap internals (protected member function )
<code>set_rdbuf</code>	Set stream buffer (protected member function )

### Public member functions inherited from `ios_base`

<code>flags</code>	Get/set format flags (public member function )
<code>setf</code>	Set specific format flags (public member function )

<b>unsetf</b>	Clear specific format flags (public member function )
<b>precision</b>	Get/Set floating-point decimal precision (public member function )
<b>width</b>	Get/set field width (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>getloc</b>	Get current locale (public member function )
<b>xalloc</b>	Get new index for extensible array [static] (public static member function )
<b>iword</b>	Get integer element of extensible array (public member function )
<b>pword</b>	Get pointer element of extensible array (public member function )
<b>register_callback</b>	Register event callback function (public member function )
<b>sync_with_stdio</b>	Toggle synchronization with cstdio streams [static] (public static member function )

## /ios/wstreampos

type

### std::wstreampos

<iostream>

`typedef fpos<mbstate_t> wstreampos;`

#### Wide stream position type

Instantiation of `fpos` used to represent positions in wide-oriented streams.

Objects of this class support construction and conversion from `int`, and allow consistent conversions to/from `streamoff` values (as well as being added or subtracted a value of this type).

Two objects of this type can be compared with operators `==` and `!=`. They can also be subtracted, which yields a value of type `streamoff`.

`streampos` and `wstreampos` are synonyms: both are `typedefs` of `fpos<mbstate_t>`.

#### See also

<b>fpos</b>	Stream position class template (class template )
<b>streamoff</b>	Stream offset type (type )
<b>streamsize</b>	Stream size type (type )

## /istream

header

### <iostream>

#### Input stream

Header providing the standard input and combined input/output stream classes:

#### Class templates

<b>basic_istream</b>	Input stream (class template )
<b>basic_iostream</b>	Input/output stream (class template )

#### Classes

<b>istream</b>	Input stream (class )
<b>iostream</b>	Input/output stream (class )
<b>wistream</b>	Input stream (wide) (class )
<b>wiostream</b>	Input/output stream (wide) (class )

#### Input manipulators (functions)

<b>ws</b>	Extract whitespaces (function )
-----------	---------------------------------

## /istream/basic\_iostream

class template

### std::basic\_iostream

<iostream> <iostream>

`template <class charT, class traits = char_traits<charT> >`  
`class basic_iostream;`

#### Input/output stream



This class inherits all members from its two `basic_istream` and `basic_ostringstream` (using *virtual inheritance*), thus being able to perform both input and output operations.

The class relies on a single `basic_streambuf` object for both the input and output operations.

Objects of these classes keep a set of internal fields inherited from `ios_base`, `basic_ios` and `basic_istream`:

	<b>field</b>	<b>member functions</b>	<b>description</b>
Formatting	format flags	<code>flags</code> <code>setf</code> <code>unsetf</code>	A set of internal flags that affect how certain input/output operations are interpreted or generated. See member type <code>fmtflags</code> .
	field width	<code>width</code>	Width of the next formatted element to insert.
	display precision	<code>precision</code>	Decimal precision for the next floating-point value inserted.
	locale	<code>getloc</code> <code>imbe</code>	The <code>locale</code> object used by the function for formatted input/output operations affected by localization properties.
	fill character	<code>fill</code>	Character to pad a formatted field up to the <i>field width</i> ( <code>width</code> ).
State	error state	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	exception mask	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
Other	callback stack	<code>register_callback</code>	Stack of pointers to functions that are called when certain events occur.
	extensible arrays	<code>iword</code> <code>pword</code> <code>xalloc</code>	Internal arrays to store objects of type <code>long</code> and <code>void*</code> .
	tied stream	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	stream buffer	<code>rdbuf</code>	Pointer to the associated <code>basic_streambuf</code> object, which is charge of all input/output operations.
	character count	<code>gcount</code>	Count of characters read by last unformatted input operation (input streams only).

## Template parameters

### charT

Character type.  
This shall be a non-array POD type.  
Aliased as member type `basic_iostream::char_type`.

### traits

Character traits class that defines essential properties of the characters used by stream objects (see `char_traits`).  
`traits::char_type` shall be the same as `charT`.  
Aliased as member type `basic_iostream::traits_type`.

## Template instantiations

<code>iostream</code>	Input/output stream (class )
<code>wiostream</code>	Input/output stream (wide) (class )

These instantiations are declared in `<iostream>`, which is included by reference in `<iostream>`.

## Member types

Member types `char_type`, `traits_type`, `int_type`, `pos_type` and `off_type` are ambiguous (multiple inheritance).

The class declares the following member types:

<b>member type</b>	<b>definition</b>	<b>notes</b>
<code>char_type</code>	The first template parameter ( <code>charT</code> )	
<code>traits_type</code>	The second template parameter ( <code>traits</code> )	defaults to: <code>char_traits&lt;charT&gt;</code>
<code>int_type</code>	<code>traits_type::int_type</code>	
<code>pos_type</code>	<code>traits_type::pos_type</code>	generally, the same as <code>streampos</code>
<code>off_type</code>	<code>traits_type::off_type</code>	generally, the same as <code>streamoff</code>

These member types are inherited from its base classes (`basic_istream`, `basic_oiostream` and `ios_base`):

<code>event</code>	Type to indicate event type (public member type )
<code>event_callback</code>	Event callback function type (public member type )
<code>failure</code>	Base class for stream exceptions (public member class )
<code>fmtflags</code>	Type for stream format flags (public member type )
<code>Init</code>	Initialize standard stream objects (public member class )
<code>iostate</code>	Type for stream state flags (public member type )
<code>openmode</code>	Type for stream opening mode flags (public member type )
<code>seekdir</code>	Type for stream seeking direction flag (public member type )
<code>basic_istream::sentry</code>	Prepare stream for input (public member class )
<code>basic_oiostream::sentry</code>	Prepare stream for output (public member class )

## Public member functions

<code>(constructor)</code>	Construct object (public member function )
<code>(destructor)</code>	Destroy object (public member function )

## Protected member functions

<code>operator=</code>	Move assignment (protected member function )
<code>swap</code>	Swap internals (protected member function )

### Public member functions inherited from `basic_istream`

<code>operator&gt;&gt;</code>	Extract formatted input (public member function )
<code>gcount</code>	Get character count (public member function )
<code>get</code>	Get characters (public member function )
<code>getline</code>	Get line (public member function )
<code>ignore</code>	Extract and discard characters (public member function )
<code>peek</code>	Peek next character (public member function )
<code>read</code>	Read block of data (public member function )
<code>readsome</code>	Read data available in buffer (public member function )
<code>putback</code>	Put character back (public member function )
<code>unget</code>	Unget character (public member function )
<code>tellg</code>	Get position in input sequence (public member function )
<code>seekg</code>	Set position in input sequence (public member function )
<code>sync</code>	Synchronize input buffer (public member function )

### Public member functions inherited from `basic_ostream`

<code>operator&lt;&lt;</code>	Insert formatted output (public member function )
<code>put</code>	Put character (public member function )
<code>write</code>	Write block of data (public member function )
<code>tellp</code>	Get position in output sequence (public member function )
<code>seekp</code>	Set position in output sequence (public member function )
<code>flush</code>	Flush output stream buffer (public member function )

### Public member functions inherited from `basic_ios`

<code>good</code>	Check whether state of stream is good (public member function )
<code>eof</code>	Check whether eofbit is set (public member function )
<code>fail</code>	Check whether failbit or badbit is set (public member function )
<code>bad</code>	Check whether badbit is set (public member function )
<code>operator!</code>	Evaluate stream (not) (public member function )
<code>operator bool</code>	Evaluate stream (public member function )
<code>rdstate</code>	Get error state flags (public member function )
<code>setstate</code>	Set error state flag (public member function )
<code>clear</code>	Set error state flags (public member function )
<code>copyfmt</code>	Copy formatting information (public member function )
<code>fill</code>	Get/set fill character (public member function )
<code>exceptions</code>	Get/set exceptions mask (public member function )
<code>imbue</code>	Imbue locale (public member function )
<code>tie</code>	Get/set tied stream (public member function )
<code>rdbuf</code>	Get/set stream buffer (public member function )
<code>narrow</code>	Narrow character (public member function )
<code>widen</code>	Widen character (public member function )

### Public member functions inherited from `ios_base`

<code>flags</code>	Get/set format flags (public member function )
<code>setf</code>	Set specific format flags (public member function )
<code>unsetf</code>	Clear specific format flags (public member function )
<code>precision</code>	Get/Set floating-point decimal precision (public member function )
<code>width</code>	Get/set field width (public member function )
<code>imbue</code>	Imbue locale (public member function )
<code>getloc</code>	Get current locale (public member function )
<code>xalloc</code>	Get new index for extensible array [static] (public static member function )
<code>iword</code>	Get integer element of extensible array (public member function )
<code>pword</code>	Get pointer element of extensible array (public member function )
<code>register_callback</code>	Register event callback function (public member function )
<code>sync_with_stdio</code>	Toggle synchronization with cstdio streams [static] (public static member function )

## /istream/basic\_iostream/~basic\_iostream

public member function

`std::basic_iostream::~basic_iostream`

`virtual ~basic_iostream();`

Destroy object

`<iostream> <iostream>`

Destroys an object of this class.

Note that this does *not* destroy nor performs any operations on the associated *stream buffer object*.

## Data races

The object is modified.

## Exception safety

No-throw guarantee: never throws exceptions.

# /istream/basic\_iostream/basic\_iostream

public member function

## std::basic\_iostream::basic\_iostream

<iostream> <iostream>

```
initialization (1) explicit basic_istream (basic_streambuf<char_type,traits_type>* sb);
initialization (1) explicit basic_istream (basic_streambuf<char_type,traits_type>* sb);
copy (2) basic_istream& (const basic_istream&) = delete;
move (3) protected: basic_istream& (basic_istream&& x);
```

### Construct object

Constructs a `basic_iostream` object.

#### (1) initialization constructor

Assigns initial values to the components of its base classes by calling the constructors its base classes `basic_istream` and `basic_ostream` with `sb` as argument.

Notice that this calls member `basic_ios::init` twice.

#### (2) copy constructor (deleted)

Deleted: no copy constructor.

#### (3) move constructor (protected)

Acquires the contents of `x`, except its associated *stream buffer*: It calls `basic_istream`'s constructor with `move(x)` as argument, transferring some of `x`'s internal components to the object: `x` is left with a `gcount` value of zero, not *tied*, and with its associated *stream buffer* unchanged (all other components of `x` are in an unspecified but valid state after the call).

## Parameters

`sb`

pointer to a `basic_streambuf` object with the same template parameters as the `basic_iostream` object.  
`char_type` and `traits_type` are member types defined as aliases of the first and second class template parameters, respectively (see `basic_iostream` types).

`x`

Another `basic_iostream` of the same type (with the same class template arguments `charT` and `traits`).

## Data races

The object pointed by `sb` may be accessed and/or modified.

## Exception safety

If an exception is thrown, the only side effects may come from accessing/modifying `sb`.

## See also

`basic_istream::basic_istream` Construct object (public member function )

`basic_ios::init` Initialize object (protected member function )

# /istream/basic\_iostream/operator=

protected member function

## std::basic\_iostream::operator=

<iostream> <iostream>

```
copy (1) basic_istream& operator= (const basic_istream&) = delete;
move (2) basic_istream& operator= (basic_istream&& rhs);
```

### Move assignment

Exchanges all internal members between `rhs` and `*this`, except the pointers to the associated *stream buffers*: `rdbuf` shall return the same in both objects as before the call.

This is the same behavior as calling member `basic_iostream::swap`.

Derived classes can call this function to implement move semantics.

## Parameters

`rhs`

Another `basic_iostream` object with the same template parameters (`charT` and `traits`).

## Return Value

\*this

## Data races

Modifies both stream objects (\*this and rhs).

## Exception safety

No-throw guarantee: this member function never throws exceptions.

## See also

[basic\\_iostream::swap](#) Swap internals (protected member function )

# /istream/basic\_iostream/swap

protected member function

## std::basic\_iostream::swap

<iostream> <iostream>

`void swap (basic_iostream& x);`

### Swap internals

Exchanges all internal members between x and \*this, except the pointer to the associated *stream buffers*: rdbuf shall return the same in both objects as before the call.

Internally, the function calls `basic_istream::swap`.

Derived classes can call this function to implement custom `swap` functions.

## Parameters

x

Another stream object of the same type.

## Return Value

none

## Data races

Modifies both stream objects (\*this and x).

## Exception safety

No-throw guarantee: this member function never throws exceptions.

## See also

[basic\\_istream::swap](#) Swap internals (protected member function )

# /istream/basic\_istream

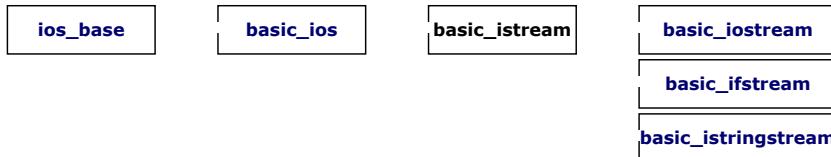
class template

## std::basic\_istream

<iostream> <iostream>

`template <class charT, class traits = char_traits<charT> >`  
`class basic_istream;`

### Input stream



Input stream objects can read and interpret input from sequences of characters. Specific members are provided to perform these input operations (see [functions below](#)).

The standard objects `cin` and `wcin` use particular instantiations of this class template.

Objects of these classes keep a set of internal fields inherited from `ios_base` and `basic_ios`:

	field	member functions	description
Formatting	format flags	flags setf unsetf	A set of internal flags that affect how certain input/output operations are interpreted or generated. See member type <code>fmtflags</code> .
	field width	width	Width of the next formatted element to insert.
	display precision	precision	Decimal precision for the next floating-point value inserted.

	<code>locale</code>	<code>getloc</code> <code>imbue</code>	The <code>locale</code> object used by the function for formatted input/output operations affected by localization properties.
	<code>fill character</code>	<code>fill</code>	Character to pad a formatted field up to the <code>field width</code> ( <code>width</code> ).
<b>State</b>	<code>error state</code>	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	<code>exception mask</code>	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
<b>Other</b>	<code>callback stack</code>	<code>register_callback</code>	Stack of pointers to functions that are called when certain events occur.
	<code>extensible arrays</code>	<code>iword</code> <code>pword</code> <code>xalloc</code>	Internal arrays to store objects of type <code>long</code> and <code>void*</code> .
	<code>tied stream</code>	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	<code>stream buffer</code>	<code>rdbuf</code>	Pointer to the associated <code>basic_streambuf</code> object, which is charge of all input/output operations.

To these, `basic_istream` adds the *character count* (accessible using member `gcount`).

## Template parameters

`charT`

Character type.  
This shall be a non-array POD type.  
Aliased as member type `basic_istream::char_type`.

`traits`

Character traits class that defines essential properties of the characters used by stream objects (see `char_traits`).  
`traits::char_type` shall be the same as `charT`.  
Aliased as member type `basic_istream::traits_type`.

## Template instantiations

<code>istream</code>	Input stream (class)
----------------------	----------------------

<code>wistream</code>	Input stream (wide) (class)
-----------------------	-----------------------------

These instantiations are declared in `<iostream>`, which is included by reference in `<iostream>`.

## Member types

The class contains the following member class:

<code>sentry</code>	Prepare stream for input (public member class)
---------------------	--

Along with the following member types:

member type	definition	notes
<code>char_type</code>	The first template parameter ( <code>charT</code> )	
<code>traits_type</code>	The second template parameter ( <code>traits</code> )	defaults to: <code>char_traits&lt;charT&gt;</code>
<code>int_type</code>	<code>traits_type::int_type</code>	
<code>pos_type</code>	<code>traits_type::pos_type</code>	generally, the same as <code>streampos</code>
<code>off_type</code>	<code>traits_type::off_type</code>	generally, the same as <code>streamoff</code>

And these member types inherited from `ios_base` through `basic_ios`:

<code>event</code>	Type to indicate event type (public member type)
<code>event_callback</code>	Event callback function type (public member type)
<code>failure</code>	Base class for stream exceptions (public member class)
<code>fmtflags</code>	Type for stream format flags (public member type)
<code>Init</code>	Initialize standard stream objects (public member class)
<code>iostate</code>	Type for stream state flags (public member type)
<code>openmode</code>	Type for stream opening mode flags (public member type)
<code>seekdir</code>	Type for stream seeking direction flag (public member type)

## Public member functions

<code>(constructor)</code>	Construct object (public member function)
----------------------------	---

<code>(destructor)</code>	Destroy object (public member function)
---------------------------	---

**Formatted input:**

<code>operator&gt;&gt;</code>	Extract formatted input (public member function)
-------------------------------	--

**Unformatted input:**

<code>gcount</code>	Get character count (public member function)
<code>get</code>	Get characters (public member function)
<code>getline</code>	Get line (public member function)
<code>ignore</code>	Extract and discard characters (public member function)
<code>peek</code>	Peek next character (public member function)
<code>read</code>	Read block of data (public member function)
<code>readsome</code>	Read data available in buffer (public member function)
<code>putback</code>	Put character back (public member function)
<code>unget</code>	Unget character (public member function)

**Positioning:**

<code>tellg</code>	Get position in input sequence (public member function)
--------------------	---

<code>seekg</code>	Set position in input sequence (public member function )
--------------------	--

#### Synchronization:

<code>sync</code>	Synchronize input buffer (public member function )
-------------------	--

#### Protected member functions

<code>operator=</code>	Move assignment (protected member function )
<code>swap</code>	Swap internals (protected member function )

#### Public member functions inherited from `basic_ios`

<code>good</code>	Check whether state of stream is good (public member function )
<code>eof</code>	Check whether eofbit is set (public member function )
<code>fail</code>	Check whether failbit or badbit is set (public member function )
<code>bad</code>	Check whether badbit is set (public member function )
<code>operator!</code>	Evaluate stream (not) (public member function )
<code>operator bool</code>	Evaluate stream (public member function )
<code>rdstate</code>	Get error state flags (public member function )
<code>setstate</code>	Set error state flag (public member function )
<code>clear</code>	Set error state flags (public member function )
<code>copyfmt</code>	Copy formatting information (public member function )
<code>fill</code>	Get/set fill character (public member function )
<code>exceptions</code>	Get/set exceptions mask (public member function )
<code>imbue</code>	Imbue locale (public member function )
<code>tie</code>	Get/set tied stream (public member function )
<code>rdbuf</code>	Get/set stream buffer (public member function )
<code>narrow</code>	Narrow character (public member function )
<code>widen</code>	Widen character (public member function )

#### Public member functions inherited from `ios_base`

<code>flags</code>	Get/set format flags (public member function )
<code>setf</code>	Set specific format flags (public member function )
<code>unsetf</code>	Clear specific format flags (public member function )
<code>precision</code>	Get/Set floating-point decimal precision (public member function )
<code>width</code>	Get/set field width (public member function )
<code>imbue</code>	Imbue locale (public member function )
<code>getloc</code>	Get current locale (public member function )
<code>xalloc</code>	Get new index for extensible array [static] (public static member function )
<code>iword</code>	Get integer element of extensible array (public member function )
<code>pword</code>	Get pointer element of extensible array (public member function )
<code>register_callback</code>	Register event callback function (public member function )
<code>sync_with_stdio</code>	Toggle synchronization with cstdio streams [static] (public static member function )

## /istream/basic\_istream/~basic\_istream

public member function

### `std::basic_istream::~basic_istream`

`<iostream> <iostream>`

`virtual ~basic_istream();`

#### Destroy object

Destroys an object of this class.

Note that this does *not* destroy nor performs any operations on the associated *stream buffer object*.

#### Data races

The object is modified.

#### Exception safety

**No-throw guarantee:** never throws exceptions.

#### See also

## /istream/basic\_istream/basic\_istream

public member function

## std::basic\_istream::basic\_istream

<iostream> <iostream>

```
initialization (1) explicit basic_istream (basic_streambuf<char_type,traits_type>* sb);
initialization (1) explicit basic_istream (basic_streambuf<char_type,traits_type>* sb);
copy (2) basic_istream& (const basic_istream&) = delete;
move (3) protected: basic_istream& (basic_istream&& x);
```

### Construct object

Constructs a `basic_istream` object.

#### (1) initialization constructor

Assigns initial values to the components of its base classes by calling the inherited member `basic_ios::init` with `sb` as argument.

#### (2) copy constructor (deleted)

Deleted: no copy constructor.

#### (3) move constructor (protected)

Acquires the contents of `x`, except its associated *stream buffer*: It copies `x`'s `gcount` value and then calls `basic_ios::move` to transfer `x`'s `basic_ios` components. `x` is left with a `gcount` value of zero, not *tied*, and with its associated *stream buffer* unchanged (all other components of `x` are in an unspecified but valid state after the call).

### Parameters

`sb`  
pointer to a `basic_streambuf` object with the same template parameters as the `basic_istream` object.  
`char_type` and `traits_type` are member types defined as aliases of the first and second class template parameters, respectively (see `basic_istream` types).

`x`  
Another `basic_istream` of the same type (with the same class template arguments `charT` and `traits`).

### Example

```
1 // istream constructor
2 #include <iostream>      // std::ios, std::istream, std::cout
3 #include <fstream>       // std::filebuf
4
5 int main () {
6     std::filebuf fb;
7     if (fb.open ("test.txt",std::ios::in))
8     {
9         std::istream is(&fb);
10        while (is)
11            std::cout << char(is.get());
12        fb.close();
13    }
14    return 0;
15 }
```

This example code uses a `basic_filebuf` object (derived from `basic_streambuf`) to open a file called `test.txt`. The buffer is passed as parameter to the constructor of the `basic_istream` object `is`, associating it to the stream. Then, the program uses the input stream to print its contents to `cout`.

Objects of `basic_istream` classes are seldom constructed directly. Generally some derived class is used (like the standard `basic_ifstream` or `basic_istringstream`).

### Data races

The object pointed by `sb` may be accessed and/or modified.

### Exception safety

If an exception is thrown, the only side effects may come from accessing/modifying `sb`.

### See also

`basic_ios::init` Initialize object (protected member function )

## /istream/basic\_istream/gcount

public member function

## std::basic\_istream::gcount

<iostream> <iostream>

`streamsize gcount() const;`

### Get character count

Returns the number of characters extracted by the last *unformatted input operation* performed on the object.

The *unformatted input operations* that modify the value returned by this function are: `get`, `getline`, `ignore`, `peek`, `read`, `readsome`, `putback` and `unget`.

Notice though, that `peek`, `putback` and `unget` do not actually extract any characters, and thus `gcount` will always return zero after calling any of them.

### Parameters

none

## Return Value

The number of characters extracted by the last unformatted input operation.  
`streamsize` is a signed integral type.

## Example

```
1 // cin.gcount example
2 #include <iostream>      // std::cin, std::cout
3
4 int main () {
5     char str[20];
6
7     std::cout << "Please, enter a word: ";
8     std::cin.getline(str,20);
9     std::cout << std::cin.gcount() << " characters read: " << str << '\n';
10
11    return 0;
12 }
```

Possible output:

```
Please, enter a word: simplify
9 characteres read: simplify
```

## Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

## See also

<a href="#">basic_istream::get</a>	Get characters (public member function )
<a href="#">basic_istream::getline</a>	Get line (public member function )
<a href="#">basic_istream::ignore</a>	Extract and discard characters (public member function )
<a href="#">basic_istream::read</a>	Read block of data (public member function )
<a href="#">basic_istream::readsome</a>	Read data available in buffer (public member function )

# /istream/basic\_istream/get

public member function

## std::basic\_istream::get

`<iostream> <iostream>`

```
single character (1) int_type get();
basic_istream& get (char_type& c);

c-string (2) basic_istream& get (char_type* s, streamsize n);
basic_istream& get (char_type* s, streamsize n, char_type delim);

stream buffer (3) basic_istream& get (basic_streambuf<char_type,traits_type>& sb);
basic_istream& get (basic_streambuf<char_type,traits_type>& sb, char_type delim);
```

### Get characters

Extracts characters from the stream, as *unformatted input*:

#### (1) single character

Extracts a single character from the stream.

The character is either returned (first signature), or set as the value of its argument (second signature).

#### (2) c-string

Extracts characters from the stream and stores them in `s` as a c-string, until either (`n-1`) characters have been extracted or the *delimiting character* is encountered: the *delimiting character* being either the *newline character* (`widen('\n')`) or `delim` (if this argument is specified).

The *delimiting character* is **not** extracted from the input sequence if found, and remains there as the next character to be extracted from the stream (see `getline` for an alternative that *does* discard the *delimiting character*).

A *null character* (`char_type()`) is automatically appended to the written sequence if `n` is greater than zero, even if an empty string is extracted.

#### (3) stream buffer

Extracts characters from the stream and inserts them into the output sequence controlled by the *stream buffer* object `sb`, stopping either as soon as such an insertion fails or as soon as the *delimiting character* is encountered in the input sequence (the *delimiting character* being either the *newline character* or `delim`, if this argument is specified).

Only the characters successfully inserted into `sb` are extracted from the stream: Neither the *delimiting character*, nor eventually the character that failed to be inserted at `sb`, are extracted from the input sequence and remain there as the next character to be extracted from the stream.

The function also stops extracting characters if the *end-of-file* is reached. If this is reached prematurely (before meeting the conditions described above), the function sets the `eofbit` flag.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated *stream buffer* object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member `gcount`.

## Parameters

c

The reference to a character where the extracted value is stored.

<b>s</b>	Pointer to an array of characters where extracted characters are stored as a c-string. If the function does not extract any characters (or if the first character extracted is the <i>delimiter character</i> ) and <i>n</i> is greater than zero, this is set to an empty c-string.
<b>n</b>	Maximum number of characters to write to <i>s</i> (including the terminating null character). If this is less than 2, the function does not extract any characters and sets <b>failbit</b> . <b>streamsize</b> is a signed integral type.
<b>delim</b>	Explicit <i>delimiting character</i> : The operation of extracting successive characters stops as soon as the next character to extract compares equal to this (using <b>traits_type::eq</b> ).
<b>sb</b>	A <b>basic_streambuf</b> object on whose controlled output sequence the characters are copied.

Member type **char\_type** is the type of characters used by the stream (i.e., its first class template parameter, **charT**).

## Return Value

The first signature returns the character read, or the *end-of-file* value (**traits\_type::eof()**) if no characters are available in the stream (note that in this case, the **failbit** flag is also set).

Member type **int\_type** is an integral type able to represent any character value or the special *end-of-file* value.

All other signatures always return **\*this**. Note that this return value can be checked for the state of the stream (see *casting a stream to bool* for more info).

Errors are signaled by modifying the *internal state flags*:

flag	error
<b>eofbit</b>	The function stopped extracting characters because the input sequence has no more characters available ( <i>end-of-file</i> reached).
<b>failbit</b>	Either no characters were written or an empty c-string was stored in <i>s</i> .
<b>badbit</b>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member **exceptions**, the function throws an exception of member type **failure**.

## Example

```

1 // istream::get example
2 #include <iostream>      // std::cin, std::cout
3 #include <fstream>        // std::ifstream
4
5 int main () {
6     char str[256];
7
8     std::cout << "Enter the name of an existing text file: ";
9     std::cin.get(str,256);    // get c-string
10
11    std::ifstream is(str);   // open file
12
13    char c;
14    while (is.get(c))       // loop getting single characters
15        std::cout << c;
16
17    is.close();              // close file
18
19    return 0;
20 }
```

This example prompts for the name of an existing text file and prints its content on the screen, using **cin.get** both to get individual characters and c-strings.

## Data races

Modifies *c*, *sb* or the elements in the array pointed by *s*.

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects **cin** and **wcin** when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type **failure** if the resulting *error state flag* is not **goodbit** and member **exceptions** was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting **badbit**. If **badbit** was set on the last call to **exceptions**, the function rethrows the caught exception.

## See also

<b>basic_istream::getline</b>	Get line ( <a href="#">public member function</a> )
<b>basic_istream::ignore</b>	Extract and discard characters ( <a href="#">public member function</a> )
<b>basic_istream::gcount</b>	Get character count ( <a href="#">public member function</a> )

# /istream/basic\_istream/getline

public member function

**std::basic\_istream::getline**

<iostream> <iostream>

```
basic_istream& getline (char_type* s, streamsize n );
basic_istream& getline (char_type* s, streamsize n, char_type delim);
```

### Get line

Extracts characters from the stream as *unformatted input* and stores them into *s* as a c-string, until either the extracted character is the *delimiting character*, or *n* characters have been written to *s* (including the terminating null character).

The *delimiting character* is the *newline character* (`widen('\n')`) for the first form, and *delim* for the second: when found in the input sequence, it is extracted from the input sequence, but discarded and not written to *s*.

The function will also stop extracting characters if the *end-of-file* is reached. If this is reached prematurely (before either writing *n* characters or finding *delim*), the function sets the `eofbit` flag.

The `failbit` flag is set if the function extracts no characters, or if the *delimiting character* is not found once  $(n-1)$  characters have already been written to *s*. Note that if the character that follows those  $(n-1)$  characters in the input sequence is precisely the *delimiting character*, it is also extracted and the `failbit` flag is not set (the extracted sequence was exactly *n* characters long).

A *null character* (`char_type()`) is automatically appended to the written sequence if *n* is greater than zero, even if an empty string is extracted.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member `gcount`.

This function is overloaded for `basic_string` objects in header `<string>`: See `getline(basic_string)`.

## Parameters

<code>s</code>	Pointer to an array of characters where extracted characters are stored as a c-string.
<code>n</code>	Maximum number of characters to write to <i>s</i> (including the terminating null character). If the function stops reading because this limit is reached without finding the <i>delimiting character</i> , the <code>failbit</code> internal flag is set. <code>streamsize</code> is a signed integral type.
<code>delim</code>	Explicit <i>delimiting character</i> : The operation of extracting successive characters stops as soon as the next character to extract compares equal to this (using <code>traits_type::eq</code> ).

Member type `char_type` is the type of characters used by the stream (i.e., its first class template parameter, `charT`).

## Return Value

The `basic_istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence has no more characters available ( <i>end-of-file</i> reached).
<code>failbit</code>	Either the <i>delimiting character</i> was not found or no characters were extracted at all (because the <i>end-of-file</i> was before the first character or because the construction of <code>sentry</code> failed).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // istream::getline example
2 #include <iostream>           // std::cin, std::cout
3
4 int main () {
5     char name[256], title[256];
6
7     std::cout << "Please, enter your name: ";
8     std::cin.getline (name,256);
9
10    std::cout << "Please, enter your favourite movie: ";
11    std::cin.getline (title,256);
12
13    std::cout << name << "'s favourite movie is " << title;
14
15    return 0;
16 }
```

This example illustrates how to get lines from the standard input stream (`cin`).

## Data races

Modifies the elements in the array pointed by *s* and the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

---

## See also

# /istream/basic\_istream/ignore

public member function

## std::basic\_istream::ignore

<iostream> <iostream>

```
basic_istream& ignore (streamsize n = 1, int_type delim = traits_type::eof());
```

### Extract and discard characters

Extracts characters from the input sequence and discards them, until either *n* characters have been extracted, or one compares equal to *delim*.

The function also stops extracting characters if the *end-of-file* is reached. If this is reached prematurely (before either extracting *n* characters or finding *delim*), the function sets the `eofbit` flag.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

---

## Parameters

*n*

Maximum number of characters to extract (and ignore).

If this is exactly `numeric_limits<streamsize>::max()`, there is no limit: As many characters are extracted as needed until *delim* (or the *end-of-file*) is found.

`streamsize` is a signed integral type.

*delim*

Delimiting character: The function stops extracting characters as soon as an extracted character compares equal to this (using `traits_type::eq`).

If this is the *end-of-file* value (`traits_type::eof()`), no character will compare equal, and thus exactly *n* characters will be discarded (unless the function fails or the *end-of-file* is reached).

Member type `int_type` is an integral type able to represent any valid character value or the special *end-of-file* value.

---

## Return Value

The `basic_istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence has no more characters available ( <i>end-of-file</i> reached).
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

---

## Example

```
1 // istream::ignore example
2 #include <iostream>      // std::cin, std::cout
3
4 int main () {
5     char first, last;
6
7     std::cout << "Please, enter your first name followed by your surname: ";
8
9     first = std::cin.get();    // get one character
10    std::cin.ignore(256, ' '); // ignore until space
11
12    last = std::cin.get();    // get one character
13
14    std::cout << "Your initials are " << first << last << '\n';
15
16    return 0;
17 }
```

Possible output:

```
Please, enter your first name followed by your surname: John Smith
Your initials are JS
```

---

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

---

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state.

Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

---

## See also

`basic_istream::peek`    Peek next character (public member function )

<b>basic_istream::get</b>	Get characters (public member function )
<b>basic_istream::getline</b>	Get line (public member function )
<b>basic_istream::read</b>	Read block of data (public member function )
<b>basic_istream::readsome</b>	Read data available in buffer (public member function )

## /istream/basic\_istream/operator=

protected member function

### std::basic\_istream::operator=

<iostream> <iostream>

```
copy (1) basic_istream& operator= (const basic_istream&) = delete;
move (2) basic_istream& operator= (basic_istream&& rhs);
```

#### Move assignment

Exchanges all internal members between *rhs* and *\*this*, except the pointers to the associated *stream buffers*: *rdbuf* shall return the same in both objects as before the call.

This is the same behavior as calling member `basic_istream::swap`.

Derived classes can call this function to implement move semantics.

#### Parameters

*rhs*

Another `basic_istream` object with the same template parameters (`charT` and `traits`).

#### Return Value

*\*this*

#### Data races

Modifies both stream objects (*\*this* and *rhs*).

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

#### See also

<b>basic_istream::swap</b>	Swap internals (protected member function )
----------------------------	---

## /istream/basic\_istream/operator>>

public member function

### std::basic\_istream::operator>>

<iostream> <iostream>

```
arithmetic types (1) basic_istream& operator>> (bool& val);
                     basic_istream& operator>> (short& val);
                     basic_istream& operator>> (unsigned short& val);
                     basic_istream& operator>> (int& val);
                     basic_istream& operator>> (unsigned int& val);
                     basic_istream& operator>> (long& val);
                     basic_istream& operator>> (unsigned long& val);
                     basic_istream& operator>> (float& val);
                     basic_istream& operator>> (double& val);
                     basic_istream& operator>> (long double& val);
                     basic_istream& operator>> (void*& val);

stream buffers (2)  basic_istream& operator>> (basic_streambuf<char_type,traits_type>* sb );
                     basic_istream& operator>> (basic_istream& (*pf)(basic_istream&));
                     basic_istream& operator>> (
                         basic_ios<char_type,traits_type>& (*pf)(basic_ios<char_type,traits_type>&));
                     basic_istream& operator>> (ios_base& (*pf)(ios_base&));

manipulators (3)   basic_istream& operator>> (bool& val);
                     basic_istream& operator>> (short& val);
                     basic_istream& operator>> (unsigned short& val);
                     basic_istream& operator>> (int& val);
                     basic_istream& operator>> (unsigned int& val);
                     basic_istream& operator>> (long& val);
                     basic_istream& operator>> (unsigned long& val);
                     basic_istream& operator>> (long long& val);
                     basic_istream& operator>> (unsigned long long& val);
                     basic_istream& operator>> (float& val);
                     basic_istream& operator>> (double& val);
                     basic_istream& operator>> (long double& val);
                     basic_istream& operator>> (void*& val);

arithmetic types (1) basic_istream& operator>> (bool& val);
                     basic_istream& operator>> (short& val);
                     basic_istream& operator>> (unsigned short& val);
                     basic_istream& operator>> (int& val);
                     basic_istream& operator>> (unsigned int& val);
                     basic_istream& operator>> (long& val);
                     basic_istream& operator>> (unsigned long& val);
                     basic_istream& operator>> (long long& val);
                     basic_istream& operator>> (unsigned long long& val);
                     basic_istream& operator>> (float& val);
                     basic_istream& operator>> (double& val);
                     basic_istream& operator>> (long double& val);
                     basic_istream& operator>> (void*& val);

stream buffers (2)  basic_istream& operator>> (basic_streambuf<char_type,traits_type>* sb );
                     basic_istream& operator>> (basic_istream& (*pf)(basic_istream&));
                     basic_istream& operator>> (
                         basic_ios<char_type,traits_type>& (*pf)(basic_ios<char_type,traits_type>&));
                     basic_istream& operator>> (ios_base& (*pf)(ios_base&));
```

**Extract formatted input**

This operator (`>>`) applied to an input stream is known as *extraction operator*:

### (1) arithmetic types

Extracts and parses characters sequentially from the stream to interpret them as the representation of a value of the proper type, which is stored as the value of `val`.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `false`). Then (if `good`), it calls `num_get::get` (using the stream's `selected locale`) to perform both the extraction and the parsing operations, adjusting the `internal state flags` accordingly. Finally, it destroys the `sentry` object before returning.

### (2) stream buffers

Extracts as many characters as possible from the stream and inserts them into the output sequence controlled by the `stream buffer` object pointed by `sb` (if any), until either the input sequence is exhausted or the function fails to insert into the object pointed by `sb`.

The function is considered to perform *formatted input*: Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `false`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

The function is considered to perform *unformatted input*: Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member `gcount`.

### (3) manipulators

Calls `pf(*this)`, where `pf` may be a *manipulator*.

*Manipulators* are functions specifically designed to be called when used with this operator.

This operation has no effect on the input sequence and extracts no characters (unless the manipulator itself does, like `ws`).

See `operator>>` for other overloads that extract characters.

Except where stated otherwise, calling this function does not alter the value returned by member `gcount`.

## Parameters

`val`

Object where the value that the extracted characters represent is stored.

Notice that the type of this argument (along with the stream's `format flags`) influences what constitutes a valid representation.

`sb`

Pointer to a `basic_streambuf` object on whose controlled output sequence the characters are copied.

`pf`

A function that takes and returns a stream object. It generally is a *manipulator function*.

The standard manipulators which have an effect when used on standard `basic_istream` objects are:

manipulator	Effect
<code>ws</code>	Extracts whitespaces.
<code>boolalpha/noboolalpha</code>	Activates/deactivates the extraction of alphanumerical representations of values of type <code>bool</code> .
<code>skipws/noskipws</code>	Activates/deactivates whether leading whitespaces are discarded before formatted input operations.
<code>dec/hex/oct</code>	Sets that base used to interpret integral numerical values.

The following extended manipulators can also be applied to `basic_istream` objects (these take additional arguments and require the explicit inclusion of the `<iomanip>` header):

manipulator	Effect
<code>setbase</code>	Sets the numerical base used to interpret integral numerical values.
<code>setiosflags/resetiosflags</code>	Set/reset format flags.

`char_type` and `traits_type` are member types defined as aliases of the first and second class template parameters, respectively (see `basic_istream` types).

## Return Value

The `basic_istream` object (`*this`).

The extracted value or sequence is not returned, but directly stored in the variable passed as argument.

Errors are signaled by modifying the `internal state flags`, except for (3), that never sets any flags (but the particular manipulator applied may):

flag	error
<code>eofbit</code>	The input sequence has no more characters available ( <i>end-of-file</i> reached).
<code>failbit</code>	Either no characters were extracted, or the characters extracted could not be interpreted as a valid value of the appropriate type. For (2), it is set when no characters are inserted in the object pointed by <code>sb</code> , or when <code>sb</code> is a <i>null pointer</i> .
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an `internal state flag` that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // example on extraction
2 #include <iostream>      // std::cin, std::cout, std::hex
3
4 int main () {
5     int n;
6
7     std::cout << "Enter a number: ";
8     std::cin >> n;
9     std::cout << "You have entered: " << n << '\n';
10
11    std::cout << "Enter a hexadecimal number: ";
12    std::cin >> std::hex >> n;           // manipulator
13    std::cout << "Its decimal equivalent is: " << n << '\n';
14
15    return 0;
16 }
```

This example demonstrates the use of some of the overloaded `>>` functions shown above using the standard `basic_istream` object `cin`.

## Data races

Modifies `val` or the object pointed by `sb`.  
Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>basic_istream::get</code>	Get characters (public member function )
<code>basic_istream::getline</code>	Get line (public member function )
<code>basic_ostream::operator&lt;&lt;</code>	Insert formatted output (public member function )
<code>basic_istream::sentry</code>	Prepare stream for input (public member class )

# /istream/basic\_istream/operator-free

public member function

## `std::operator>> (basic_istream)`

`<iostream> <iostream>`

<code>single character (1)</code>	<code>template&lt;class charT, class traits&gt;</code> <code>basic_istream&lt;charT,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;charT,traits&gt;&amp; is, charT&amp; c);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, signed char&amp; c);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, unsigned char&amp; c);</code> <code>template&lt;class charT, class traits&gt;</code> <code>basic_istream&lt;charT,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;charT,traits&gt;&amp; is, charT* s);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, signed char* s);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, unsigned char* s);</code>
<code>character sequence (2)</code>	<code>single character (1)</code>
<code>character sequence (2)</code>	<code>single character (1)</code>
<code>rvalue extraction (3)</code>	<code>template&lt;class charT, class traits&gt;</code> <code>basic_istream&lt;charT,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;charT,traits&gt;&amp; is, charT&amp; c);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, signed char&amp; c);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, unsigned char&amp; c);</code> <code>template&lt;class charT, class traits&gt;</code> <code>basic_istream&lt;charT,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;charT,traits&gt;&amp; is, charT* s);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, signed char* s);</code> <code>template&lt;class traits&gt;</code> <code>basic_istream&lt;char,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;char,traits&gt;&amp; is, unsigned char* s);</code> <code>template&lt;class charT, class traits, class T&gt;</code> <code>basic_istream&lt;charT,traits&gt;&amp; operator&gt;&gt; (basic_istream&lt;charT,traits&gt;&amp;&amp; is, T&amp; val);</code>

### Extract characters

This operator (`>>`) applied to an input stream is known as *extraction operator*, and performs *formatted input*:

#### (1) single character

Extracts the next character from `is` and stores it as the value of `c`.

#### (2) character sequence

Extracts characters from `is` and stores them in `s` as a c-string, stopping as soon as either a `whitespace character` is encountered or (`width()`-1) characters have been extracted (if `width` is not zero).

A `null character` (`charT()`) is automatically appended to the written sequence.  
The function then resets `width` to zero.

#### (3) rvalue extraction

Allows extracting from rvalue `basic_istream` objects, with the same effect as from lvalues: It effectively calls: `is>>val`.

Internally, the function accesses the input sequence of `is` by first constructing a `sentry` with `noskipws` set to `false`: this may flush its *tied stream* and/or discard leading whitespaces (see `basic_istream::sentry`). Then (if `good`), it extracts characters from `is`'s associated `stream buffer` object (as if calling its member functions `sbumpc` or `sgetc`), and finally destroys the `sentry` object before returning.

Notice that if this function extracts the last character of a stream when extracting a single character (1), it does not set its `eofbit` flag, but attempting to extract beyond it does.

Calling this function does not alter the value returned by `gcount` on `is`.

## Parameters

<code>is</code>	Input stream object from which characters are extracted.
<code>c</code>	Object where the extracted character is stored.

**s**  
Pointer to an array of characters where extracted characters are stored as a c-string.  
*is*'s member function `width` may be used to specify a limit on the number of characters to write.

**val**  
Object where content is stored.  
*T* shall be a type supported as right-hand side argument either by this function or by *is*'s member function `operator>>`.

`charT` and `traits` are the class template parameters of *is* (see [basic\\_istream](#)).

## Return Value

The [basic\\_istream](#) object (*is*).

The extracted value or sequence is not returned, but directly stored in the variable passed as argument.

Errors are signaled by modifying the [internal state flags](#):

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence controlled by <i>is</i> has no more characters available (end-of-file reached).
<code>failbit</code>	Either no characters were extracted, or these could not be interpreted as a valid value of the appropriate type.
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an [internal state flag](#) that was registered with *is*'s member `exceptions`, the function throws an exception of *is*'s `failure` member type.

## Example

```
1 // example on extraction
2 #include <iostream>           // std::cin, std::cout
3
4 int main () {
5     char str[10];
6
7     std::cout << "Enter a word: ";
8     std::cin.width (10);          // limit width
9     std::cin >> str;
10    std::cout << "The first 9 chars of your word are: " << str << '\n';
11
12    return 0;
13 }
```

This example demonstrates the use of some of the overloaded `operator>>` functions shown above using the standard [basic\\_istream](#) object `cin`.

## Data races

Modifies *c* or the elements of the array pointed by *s*.  
Modifies *is*.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are [synchronized with stdio](#) (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, all objects involved are in valid states.

It throws an exception of member type `failure` if the resulting *error state flag* for *is* is not `goodbit` and member `exceptions` was set to throw for that state in *is*.

Any exception thrown by an internal operation is caught and handled by the function, setting *is*'s `badbit` flag. If `badbit` was set on the last call to `exceptions` for *is*, the function rethrows the caught exception.

## See also

<a href="#">basic_istream::get</a>	Get characters (public member function )
<a href="#">basic_istream::getline</a>	Get line (public member function )
<a href="#">basic_ostream::operator&lt;&lt;</a>	Insert formatted output (public member function )
<a href="#">basic_istream::sentry</a>	Prepare stream for input (public member class )

# /istream/basic\_istream/peek

public member function

## `std::basic_istream::peek`

`<iostream> <iostream>`

### **Peek next character**

Returns the next character in the input sequence, without extracting it: The character is left as the next character to be extracted from the stream.

If any [internal state flags](#) is already set before the call or is set during the call, the function returns the [end-of-file](#) value (`traits_type::eof()`).

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it reads one character from its associated `stream buffer` object by calling its member function `sgetc`, and finally destroys the `sentry` object before returning.

Calling this function sets the value returned by `gcount` to zero.

## Parameters

none

## Return Value

The next character in the input sequence, converted to a value of type `int_type` using member `traits_type::to_int_type`. Member type `int_type` is an integral type able to represent any character value or the special `end-of-file` value.

If there are no more characters to read in the input sequence, or if any `internal state flags` is set, the function returns the `end-of-file` value (`traits_type::eof()`), and leaves the proper `internal state flags` set:

flag	error
<code>eofbit</code>	No character could be peeked because the input sequence has no characters available (end-of-file reached).
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an `internal state flag` that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // basic_istream::peek example
2 #include <iostream>           // std::cin, std::cout
3 #include <string>             // std::string
4 #include <cctype>              // std::isdigit
5
6 int main () {
7
8     std::cout << "Please, enter a number or a word: ";
9     std::cout.flush();      // ensure output is written
10
11    std::cin >> std::ws;    // eat up any leading white spaces
12    std::istream::int_type c;
13    c = std::cin.peek();   // peek character
14
15    if ( c == std::char_traits<char>::eof() ) return 1;
16    if ( std::isdigit(c) )
17    {
18        int n;
19        std::cin >> n;
20        std::cout << "You entered the number: " << n << '\n';
21    }
22    else
23    {
24        std::string str;
25        std::cin >> str;
26        std::cout << "You entered the word: " << str << '\n';
27    }
28
29    return 0;
30 }
```

Possible output:

```
Please, enter a number or a word: foobar
You entered the word: foobar
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which read characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

`basic_istream::get` Get characters (public member function)

`basic_istream::operator>>` Extract formatted input (public member function)

## /istream/basic\_istream/putback

public member function

### `std::basic_istream::putback`

`<iostream> <iostream>`

`basic_istream& putback (char_type c);`

#### Put character back

Attempts to decrease the current location in the stream by one character, making the last character extracted from the stream once again available to be extracted by input operations.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it calls `sputbackc(c)` on its associated `stream buffer` object (if any). Finally, it destroys the `sentry` object before returning.

If the `eofbit` flag is set before the call, the function fails (sets `failbit` and returns).

The function clears the `eofbit` flag, if set before the call.

If the call to `sputbackc` fails, the function sets the `badbit` flag. Note that this may happen even if `c` was indeed the last character extracted from the stream (depending on the internals of the associated *stream buffer* object).

Calling this function sets the value returned by `gcount` to zero.

## Parameters

`c`

Character to be put back.

If this does not match the character at the put back position, the behavior depends on the particular *stream buffer* object associated to the stream:

- In *string buffers*, the value is overwritten for output stream buffers, but the function fails on input buffers.

Other types of *stream buffer* may either fail, be ignored, or overwrite the character at that position.

Member type `char_type` is the type of characters used by the stream (i.e., its first class template parameter, `charT`).

## Return Value

The `basic_istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	-
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <i>stream state</i> was not <code>good</code> before the call).
<code>badbit</code>	Either the internal call to <code>sputbackc</code> failed, or another error occurred on the stream (such as when the function catches an exception thrown by an internal operation, or when no <i>stream buffer</i> is associated with the stream). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // istream::putback example
2 #include <iostream>           // std::cin, std::cout
3 #include <string>             // std::string
4
5 int main () {
6     std::cout << "Please, enter a number or a word: ";
7     char c = std::cin.get();
8
9     if ( (c >= '0') && (c <= '9') )
10    {
11        int n;
12        std::cin.putback (c);
13        std::cin >> n;
14        std::cout << "You entered a number: " << n << '\n';
15    }
16    else
17    {
18        std::string str;
19        std::cin.putback (c);
20        getline (std::cin,str);
21        std::cout << "You entered a word: " << str << '\n';
22    }
23    return 0;
24 }
```

Possible output:

```
Please, enter a number or a word: pocket
You entered a word: pocket
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>basic_istream::get</code>	Get characters (public member function )
<code>basic_istream::unget</code>	Unget character (public member function )

# /istream/basic\_istream/read

public member function

## `std::basic_istream::read`

```
basic_istream& read (char_type* s, streamsize n);
```

`<iostream> <iostream>`

## Read block of data

Extracts  $n$  characters from the stream and stores them in the array pointed by  $s$ .

This function simply copies a block of data, without checking its contents nor appending a *null character* at the end.

If the input sequence runs out of characters to extract (i.e., the end-of-file is reached) before  $n$  characters have been successfully read, the array pointed by  $s$  contains all the characters read until that point, and both the `eofbit` and `failbit` flags are set for the stream.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member `gcount`.

### Parameters

<code>s</code>	Pointer to an array where the extracted characters are stored. Member type <code>char_type</code> is the type of characters used by the stream (i.e., its first class template parameter, <code>charT</code> ).
<code>n</code>	Number of characters to extract. <code>streamsize</code> is a signed integral type.

### Return Value

The `basic_istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence has no more characters available (end-of-file reached).
<code>failbit</code>	Either the function could not extract $n$ characters or the construction of <code>sentry</code> failed.
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

### Example

```
1 // read a file into memory
2 #include <iostream>           // std::cout
3 #include <ifstream>          // std::ifstream
4
5 int main () {
6
7     std::ifstream is ("test.txt", std::ifstream::binary);
8     if (is) {
9         // get length of file:
10        is.seekg (0, is.end);
11        int length = is.tellg();
12        is.seekg (0, is.beg);
13
14        char * buffer = new char [length];
15
16        std::cout << "Reading " << length << " characters... ";
17        // read data as a block:
18        is.read (buffer,length);
19
20        if (is)
21            std::cout << "all characters read successfully.";
22        else
23            std::cout << "error: only " << is.gcount() << " could be read";
24        is.close();
25
26        // ...buffer contains the entire file...
27
28        delete[] buffer;
29    }
30    return 0;
31 }
```

Possible output:

```
Reading 640 characters... all characters read successfully.
```

### Data races

Modifies the elements in the array pointed by  $s$  and the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

### Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

### See also

[basic\\_istream::get](#) Get characters (public member function )

**basic\_istream::readsome** Read data available in buffer (public member function )

**basic\_istream::operator>>** Extract formatted input (public member function )

## /istream/basic\_istream/readsome

public member function

### std::basic\_istream::readsome

<iostream> <iostream>

streamsize readsome (char\_type\* s, streamsize n);

#### Read data available in buffer

Extracts up to *n* characters from the stream and stores them in the array pointed by *s*, stopping as soon as the internal buffer kept by the *associated stream buffer object* (if any) runs out of characters, even if the *end-of-file* has not yet been reached.

The function is meant to be used to read data from certain types of asynchronous sources that may eventually wait for more characters, since it stops extracting characters as soon as the internal buffer is exhausted, avoiding potential delays.

Note that this function relies on internals of the particular *stream buffer* object associated to the stream whose behavior is mostly implementation-defined for standard classes.

Internally, the function accesses the input sequence by first constructing a *sentry* object (with *noskipws* set to *true*). Then (if *good*), it checks how many characters are currently available at the associated *stream buffer* object by calling its member function *in\_avail* and extracts up to that many characters by calling *sbumpc* (or *sgetc*). Finally, it destroys the *sentry* object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member *gcount*.

#### Parameters

*s*

Pointer to an array where the extracted characters are stored.  
Member type *char\_type* is the type of characters used by the stream (i.e., its first class template parameter, *charT*).

*n*

Maximum number of characters to extract.  
*streamsize* is a signed integral type.

#### Return Value

The number of characters stored.

*streamsize* is a signed integral type.

Errors are signaled by modifying the *internal state flags*:

flag	error
<i>eofbit</i>	The input sequence has no characters available (as reported by <i>rdbuf()-&gt;in_avail()</i> returning -1).
<i>failbit</i>	The construction of <i>sentry</i> failed (such as when the <i>stream state</i> was not <i>good</i> before the call).
<i>badbit</i>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member *exceptions*, the function throws an exception of member type *failure*.

#### Data races

Modifies the elements in the array pointed by *s* and the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects *cin* and *wcin* when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

#### Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type *failure* if the resulting *error state flag* is not *goodbit* and member *exceptions* was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting *badbit*. If *badbit* was set on the last call to *exceptions*, the function rethrows the caught exception.

#### See also

**basic\_istream::read** Read block of data (public member function )

**basic\_ostream::write** Write block of data (public member function )

**basic\_istream::operator>>** Extract formatted input (public member function )

## /istream/basic\_istream/seekg

public member function

### std::basic\_istream::seekg

<iostream> <iostream>

(1) *basic\_istream& seekg (pos\_type pos);*  
(2) *basic\_istream& seekg (off\_type off, ios\_base::seekdir way);*

#### Set position in input sequence

Sets the position of the next character to be extracted from the input stream.

Internally, the function accesses the input sequence by first constructing a *sentry* object (with *noskipws* set to *true*). Then (if *good*), it calls either *pubseekpos*

(1) or `pubseekoff` (2) on its associated `stream buffer` object (if any). Finally, it destroys the `sentry` object before returning.

If the `eofbit` flag is set before the call, the function fails (sets `failbit` and returns).

The function clears the `eofbit` flag, if set before the call.

Calling this function does not alter the value returned by `gcount`.

## Parameters

`pos`

New absolute position within the stream (relative to the beginning).

Member type `pos_type` is determined by the `character traits`: generally, it is an `fpos` type (such as `streampos`) that can be converted to/from integral types.

`off`

Offset value, relative to the `way` parameter.

Member type `off_type` is determined by the `character traits`: generally, it is an alias of the signed integral type `streamoff`.

`way`

Object of type `ios_base::seekdir`. It may take any of the following constant values:

value	offset is relative to...
<code>ios_base::beg</code>	beginning of the stream
<code>ios_base::cur</code>	current position in the stream
<code>ios_base::end</code>	end of the stream

## Return Value

The `basic_istream` object (`*this`).

Errors are signaled by modifying the `internal state flags`:

flag	error
<code>eofbit</code>	-
<code>failbit</code>	Either the construction of <code>sentry</code> failed, or the internal call to <code>pubseekpos</code> (1) or <code>pubseekoff</code> (2) failed (i.e., either function returned <code>-1</code> ).
<code>badbit</code>	Another error occurred on the stream (such as when the function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an `internal state flag` that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // read a file into memory
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream is ("test.txt", std::ifstream::binary);
7     if (is) {
8         // get length of file:
9         is.seekg (0, is.end);
10        int length = is.tellg();
11        is.seekg (0, is.beg);
12
13        // allocate memory:
14        char * buffer = new char [length];
15
16        // read data as a block:
17        is.read (buffer,length);
18
19        is.close();
20
21        // print content:
22        std::cout.write (buffer,length);
23
24        delete[] buffer;
25    }
26
27    return 0;
28 }
```

In this example, `seekg` is used to move the position to the end of the file, and then back to the beginning.

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set to throw for that state.

Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

`basic_istream::tellg` | Get position in input sequence ([public member function](#))

`basic_ostream::seekp` | Set position in output sequence ([public member function](#))

# /istream/basic\_istream/sentry

public member class

## std::basic\_istream::sentry

<iostream> <iostream>

class sentry;

### Prepare stream for input

Member class that performs a series of operations before and after each input operation:

Its constructor performs the following operations on the stream object passed as its argument (in the same order):

- If any of its *internal error flags* is set, the function sets its `failbit` flag and returns.
- If it is a *tied stream*, the function `flushes` the stream it is tied to (if its output buffer is not empty). The class may implement ways for library functions to defer this flush until the next call to `overflow` by its *associated stream buffer*.
- If its `skipws` format flag is set, and the constructor is not passed `true` as second argument (`noskipws`), all leading *whitespace characters* (locale-specific) are extracted and discarded. If this operation exhausts the source of characters, the function sets both the `failbit` and `eofbit` *internal state flags*.

In case of failure during construction, it may set the stream's `failbit` flag.

There are no required operations to be performed by its destructor. But implementations may use the construction and destruction of `sentry` objects to perform additional initialization or cleanup operations on the stream common to all input operations.

All member functions that perform an input operation automatically construct an object of this class and then evaluate it (which returns `true` if no `state flag` was set). Only if this object evaluates to `true`, the function attempts the input operation (otherwise, it returns without performing it). Before returning, the function destroys the `sentry` object.

The `operator>>` formatted input operations construct the `sentry` object by passing `false` as second argument (which skips leading whitespaces). All other member functions that construct a `sentry` object pass `true` as second argument (which does not skip leading whitespaces).

The structure of this class is:

```
1 class sentry {
2 public:
3     explicit sentry (basic_istream& is, bool noskipws = false);
4     ~sentry();
5     operator bool() const;
6 private:
7     sentry (const sentry&);           // not defined
8     sentry& operator= (const sentry&); // not defined
9 };
```

```
1 class sentry {
2 public:
3     explicit sentry (basic_istream& is, bool noskipws = false);
4     ~sentry();
5     explicit operator bool() const;
6     sentry (const sentry&) = delete;
7     sentry& operator= (const sentry&) = delete;
8 };
```

## Members

`explicit sentry (basic_istream& is, bool noskipws = false);`

Prepares the output stream for an output operation, performing the actions described above.

`~sentry();`

Performs no operations (implementation-defined).

`explicit operator bool() const;`

When the object is evaluated, it returns a `bool` value indicating whether the `sentry` constructor successfully performed all its tasks: If at some point of the construction process, an *internal error flags* was set, this function always returns `false` for that object.

## Example

```
1 // istream::sentry example
2 #include <iostream>          // std::istream, std::cout
3 #include <string>            // std::string
4 #include <sstream>           // std::stringstream
5 #include <locale>             // std::isspace, std::isdigit
6
7 struct Phone {
8     std::string digits;
9 };
10
11 // custom extractor for objects of type Phone
12 std::istream& operator>>(std::istream& is, Phone& tel)
13 {
14     std::istream::sentry s(is);
15     if (s) while (is.good()) {
16         char c = is.get();
17         if (std::isspace(c,is.getloc())) break;
18         if (std::isdigit(c,is.getloc())) tel.digits+=c;
19     }
20     return is;
21 }
22
23 int main () {
24     std::stringstream parseme (" (555)2326");
25     Phone myphone;
26     parseme >> myphone;
```

```
27     std::cout << "digits parsed: " << myphone.digits << '\n';
28     return 0;
29 }
```

Output:

```
digits parsed: 5552326
```

### See also

[basic\\_istream::operator>>](#) Extract formatted input (public member function )

## /istream/basic\_istream/swap

protected member function

### std::basic\_istream::swap

<iostream> <iostream>

```
void swap (basic_istream& x);
```

#### Swap internals

Exchanges all internal members between *x* and *\*this*, except the pointer to the associated *stream buffers*: *rdbuf* shall return the same in both objects as before the call.

Internally, the function calls [basic\\_ios::swap](#) and then exchanges the values returned by *gcount*.

Derived classes can call this function to implement custom *swap* functions.

### Parameters

*x*  
Another [basic\\_istream](#) object with the same template parameters (*charT* and *traits*).

### Return Value

none

### Data races

Modifies both stream objects (*\*this* and *x*).

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

### See also

[basic\\_istream::operator=](#) Move assignment (protected member function )

[basic\\_ios::swap](#) Swap internals (protected member function )

## /istream/basic\_istream/sync

public member function

### std::basic\_istream::sync

<iostream> <iostream>

```
int sync();
```

#### Synchronize input buffer

Synchronizes the associated *stream buffer* with its controlled input sequence.

Specifics of the operation depend on the particular implementation of the *stream buffer* object associated to the stream.

Internally, the function accesses the input sequence by first constructing a *sentry* object (with *noskipws* set to true). Then (if *good*), it calls *pubsync* on its associated *stream buffer* object (if *rdbuf* is null, the function returns -1 instead). Finally, it destroys the *sentry* object before returning.

If the call to *pubsync* fails (i.e., it returns -1), the function sets the *badbit* flag, and returns -1. Otherwise it returns zero, indicating success.

Notice that the called function may succeed when no action is performed, if that is the behavior defined for the *stream buffer* object on synchronization.

Calling this function does not alter the value returned by *gcount*.

### Parameters

none

### Return Value

If the function fails, either because no *stream buffer* object is associated to the stream (*rdbuf* is null), or because the call to its *pubsync* member fails, it returns -1.

Otherwise, it returns zero, indicating success.

Errors are signaled by modifying the *internal state flags*:

flag	error
------	-------

<code>eofbit</code>	-
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Either the internal call to <code>pubsync</code> returned -1, or some other error occurred on the stream (such as when the function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```

1 // syncing input stream
2 #include <iostream>      // std::cin, std::cout
3
4 int main () {
5     char first, second;
6
7     std::cout << "Please, enter a word: ";
8     first = std::cin.get();
9     std::cin.sync();
10
11    std::cout << "Please, enter another word: ";
12    second = std::cin.get();
13
14    std::cout << "The first word began by " << first << '\n';
15    std::cout << "The second word began by " << second << '\n';
16
17    return 0;
18 }
```

This example demonstrates how `sync` behaves on certain implementations of `cin`, removing any unread character from the standard input queue of characters.

Possible output:

```
Please, enter a word: test
Please enter another word: text
The first word began by t
The second word began by t
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which characters are extracted or synchronized between threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>basic_ostream::flush</code>	Flush output stream buffer (public member function )
<code>basic_streambuf::pubsync</code>	Synchronize stream buffer (public member function )

# /istream/basic\_istream/tellg

public member function

## std::basic\_istream::tellg

`<iostream> <iostream>`

`pos_type tellg();`

### Get position in input sequence

Returns the position of the current character in the input stream.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`) without evaluating it. Then, if member `fail` returns `true`, the function returns -1.

Otherwise, returns `rdbuf()>pubseekoff(0, cur, in)`. Finally, it destroys the `sentry` object before returning.

Notice that the function will work even if the `eofbit` flag is set before the call.

Calling this function does not alter the value returned by `gcount`.

## Parameters

none

## Return Value

The current position in the stream.

If either the `stream buffer` associated to the stream does not support the operation, or if it fails, the function returns -1.

Member type `pos_type` is determined by the `character traits`: generally, it is an `fpos` type (such as `streampos`) that can be converted to/from integral types.

Errors are signaled by modifying the *internal state flags*:

flag	error

<b>eofbit</b>	-
<b>failbit</b>	The construction of <b>sentry</b> failed (such as when the <b>stream state</b> was not <b>good</b> before the call).
<b>badbit</b>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member **exceptions**, the function throws an exception of member type **failure**.

## Example

```

1 // read a file into memory
2 #include <iostream>           // std::cout
3 #include <ifstream>          // std::ifstream
4
5 int main () {
6     std::ifstream is ("test.txt", std::ifstream::binary);
7     if (is) {
8         // get length of file:
9         is.seekg (0, is.end);
10        int length = is.tellg();
11        is.seekg (0, is.beg);
12
13        // allocate memory:
14        char * buffer = new char [length];
15
16        // read data as a block:
17        is.read (buffer,length);
18
19        is.close();
20
21        // print content:
22        std::cout.write (buffer,length);
23
24        delete[] buffer;
25    }
26
27    return 0;
28 }
```

In this example, **tellg** is used to get the position in the stream after it has been moved with **seekg** to the end of the stream, therefore determining the size of the file.

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type **failure** if the resulting **error state flag** is not **goodbit** and member **exceptions** was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting **badbit**. If **badbit** was set on the last call to **exceptions**, the function rethrows the caught exception.

## See also

<b>basic_istream::seekg</b>	Set position in input sequence ( <a href="#">public member function</a> )
<b>basic_ostream::tello</b>	Get position in output sequence ( <a href="#">public member function</a> )

## /istream/basic\_istream/unget

public member function

### std::basic\_istream::unget

<iostream> <iostream>

**basic\_istream& unget();**

#### Unget character

Attempts to decrease the current location in the stream by one character, making the last character extracted from the stream once again available to be extracted by input operations.

Internally, the function accesses the input sequence by first constructing a **sentry** object (with **noskipws** set to **true**). Then (if **good**), it calls **sungetc** on its associated **stream buffer** object (if any). Finally, it destroys the **sentry** object before returning.

If the <b>eofbit</b> flag is set before the call, the function fails (sets <b>failbit</b> and returns).
---

The function clears the <b>eofbit</b> flag, if set before the call.
---

If the call to <b>sungetc</b> fails, the function sets the <b>badbit</b> flag. Note that this may happen even if the function is called right after a character has been extracted from the stream (depending on the internals of the associated <b>stream buffer</b> object).
--

Calling this function sets the value returned by **gcount** to zero.

## Parameters

none

## Return Value

The `basic_istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	-
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Either the internal call to <code>sungetc</code> failed, or another error occurred on the stream (such as when the function catches an exception thrown by an internal operation, or when no <code>stream buffer</code> is associated with the stream). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // istream::unget example
2 #include <iostream>           // std::cin, std::cout
3 #include <string>             // std::string
4
5 int main () {
6     std::cout << "Please, enter a number or a word: ";
7     char c = std::cin.get();
8
9     if ( (c >= '0') && (c <= '9') )
10    {
11        int n;
12        std::cin.unget();
13        std::cin >> n;
14        std::cout << "You entered a number: " << n << '\n';
15    }
16    else
17    {
18        std::string str;
19        std::cin.unget();
20        getline (std::cin,str);
21        std::cout << "You entered a word: " << str << '\n';
22    }
23    return 0;
24 }
```

Possible output:

```
Please, enter a number or a word: 7791
You entered a number: 7791
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>basic_istream::get</code>	Get characters (public member function )
<code>basic_istream::putback</code>	Put character back (public member function )

# /istream/iostream

class `std::ios` `<iostream> <istream>`

`typedef basic_iostream<char> iostream;`

**Input/output stream**

ios\_base

ios

istream

iostream

fstream

ostringstream

[NOTE: This page describes the `iostream` class, for a description of the `iostream` library, see [Input/Output library](#).]

This is an instantiation of `basic_iostream` with the following template parameters:

template parameter	definition	comments
<code>charT</code>	<code>char</code>	Aliased as member <code>char_type</code>
<code>traits</code>	<code>char_traits&lt;char&gt;</code>	Aliased as member <code>traits_type</code>

This class inherits all members from its two parent classes `istream` and `ostream`, thus being able to perform both input and output operations.

The class relies on a single `streambuf` object for both the input and output operations.

Objects of these classes keep a set of internal fields inherited from `ios_base`, `ios` and `istream`:

field	member functions	description

Formatting	format flags	<code>flags</code> <code>setf</code> <code>unsetf</code>	A set of internal flags that affect how certain input/output operations are interpreted or generated. See member type <code>fmtflags</code> .
	field width	<code>width</code>	Width of the next formatted element to insert.
	display precision	<code>precision</code>	Decimal precision for the next floating-point value inserted.
	locale	<code>getloc</code> <code>imbuf</code>	The <code>locale</code> object used by the function for formatted input/output operations affected by localization properties.
	fill character	<code>fill</code>	Character to pad a formatted field up to the <i>field width</i> ( <code>width</code> ).
State	error state	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	exception mask	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
Other	callback stack	<code>register_callback</code>	Stack of pointers to functions that are called when certain events occur.
	extensible arrays	<code>iword</code> <code>pword</code> <code>xalloc</code>	Internal arrays to store objects of type <code>long</code> and <code>void*</code> .
	tied stream	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	stream buffer	<code>rdbuf</code>	Pointer to the associated <code>streambuf</code> object, which is charge of all input/output operations.
	character count	<code>gcount</code>	Count of characters read by last unformatted input operation (input streams only).

## Member types

Member types `char_type`, `traits_type`, `int_type`, `pos_type` and `off_type` are ambiguous (multiple inheritance).

The class declares the following member types:

member type	definition
<code>char_type</code>	<code>char</code>
<code>traits_type</code>	<code>char_traits&lt;char&gt;</code>
<code>int_type</code>	<code>int</code>
<code>pos_type</code>	<code>streampos</code>
<code>off_type</code>	<code>streamoff</code>

These member types inherited from its base classes (`istream`, `ostream` and `ios_base`):

<code>event</code>	Type to indicate event type (public member type )
<code>event_callback</code>	Event callback function type (public member type )
<code>failure</code>	Base class for stream exceptions (public member class )
<code>fmtflags</code>	Type for stream format flags (public member type )
<code>Init</code>	Initialize standard stream objects (public member class )
<code>iostate</code>	Type for stream state flags (public member type )
<code>openmode</code>	Type for stream opening mode flags (public member type )
<code>seekdir</code>	Type for stream seeking direction flag (public member type )
<code>istream::sentry</code>	Prepare stream for input (public member class )
<code>ostream::sentry</code>	Prepare stream for output (public member class )

## Public member functions

<code>(constructor)</code>	Construct object (public member function )
<code>(destructor)</code>	Destroy object (public member function )

## Protected member functions

<code>operator=</code>	Move assignment (protected member function )
<code>swap</code>	Swap internals (protected member function )

## Public member functions inherited from `istream`

<code>operator&gt;&gt;</code>	Extract formatted input (public member function )
<code>gcount</code>	Get character count (public member function )
<code>get</code>	Get characters (public member function )
<code>getline</code>	Get line (public member function )
<code>ignore</code>	Extract and discard characters (public member function )
<code>peek</code>	Peek next character (public member function )
<code>read</code>	Read block of data (public member function )
<code>readsome</code>	Read data available in buffer (public member function )
<code>putback</code>	Put character back (public member function )
<code>unget</code>	Unget character (public member function )
<code>tellg</code>	Get position in input sequence (public member function )
<code>seekg</code>	Set position in input sequence (public member function )
<code>sync</code>	Synchronize input buffer (public member function )

## Public member functions inherited from `ostream`

<code>operator&lt;&lt;</code>	Insert formatted output (public member function )
<code>put</code>	Put character (public member function )

<b>write</b>	Write block of data (public member function )
<b>tellp</b>	Get position in output sequence (public member function )
<b>seekp</b>	Set position in output sequence (public member function )
<b>flush</b>	Flush output stream buffer (public member function )

### Public member functions inherited from `ios`

<b>good</b>	Check whether state of stream is good (public member function )
<b>eof</b>	Check whether eofbit is set (public member function )
<b>fail</b>	Check whether either failbit or badbit is set (public member function )
<b>bad</b>	Check whether badbit is set (public member function )
<b>operator!</b>	Evaluate stream (not) (public member function )
<b>operator bool</b>	Evaluate stream (public member function )
<b>rdstate</b>	Get error state flags (public member function )
<b>setstate</b>	Set error state flag (public member function )
<b>clear</b>	Set error state flags (public member function )
<b>copyfmt</b>	Copy formatting information (public member function )
<b>fill</b>	Get/set fill character (public member function )
<b>exceptions</b>	Get/set exceptions mask (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>tie</b>	Get/set tied stream (public member function )
<b>rdbuf</b>	Get/set stream buffer (public member function )
<b>narrow</b>	Narrow character (public member function )
<b>widen</b>	Widen character (public member function )

### Public member functions inherited from `ios_base`

<b>flags</b>	Get/set format flags (public member function )
<b>setf</b>	Set specific format flags (public member function )
<b>unsetf</b>	Clear specific format flags (public member function )
<b>precision</b>	Get/Set floating-point decimal precision (public member function )
<b>width</b>	Get/set field width (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>getloc</b>	Get current locale (public member function )
<b>xalloc</b>	Get new index for extensible array [static] (public static member function )
<b>iword</b>	Get integer element of extensible array (public member function )
<b>pword</b>	Get pointer element of extensible array (public member function )
<b>register_callback</b>	Register event callback function (public member function )
<b>sync_with_stdio</b>	Toggle synchronization with cstdio streams [static] (public static member function )

## /istream/iostream/iostream

public member function

### std::iostream::iostream

<istream> <iostream>

```
initialization (1) explicit iostream (streambuf* sb);
initialization (1) explicit iostream (streambuf* sb);
copy (2) iostream& (const iostream&) = delete;
move (3) protected: iostream& (iostream& x);
```

#### Construct object

Constructs a `iostream` object.

##### (1) initialization constructor

Assigns initial values to the components of its base classes by calling the constructors its base classes `istream` and `ostream` with `sb` as argument.  
Notice that this calls member `ios::init` twice.

##### (2) copy constructor (deleted)

Deleted: no copy constructor.

##### (3) move constructor (protected)

Acquires the contents of `x`, except its associated `stream buffer`: It calls `istream`'s constructor with `move(x)` as argument, transferring some of `x`'s internal components to the object: `x` is left with a `gcount` value of zero, not `ties`, and with its associated `stream buffer` unchanged (all other components of `x` are in an unspecified but valid state after the call).

### Parameters

`sb`

pointer to a `streambuf` object with the same template parameters as the `iostream` object.  
`char_type` and `traits_type` are member types defined as aliases of the first and second class template parameters, respectively (see `iostream types`).

`x`

Another `iostream` of the same type (with the same class template arguments `charT` and `traits`).

## Data races

The object pointed by *sb* may be accessed and/or modified.

## Exception safety

If an exception is thrown, the only side effects may come from accessing/modifying *sb*.

## See also

<a href="#">istream::istream</a>	Construct object (public member function )
<a href="#">ios::init</a>	Initialize object (protected member function )

# /istream/iostream/~iostream

public member function

## std::iostream::~iostream

<iostream> <iostream>

`virtual ~iostream();`

### Destroy object

Destroys an object of this class.

Note that this does *not* destroy nor performs any operations on the associated *stream buffer object*.

## Data races

The object is modified.

## Exception safety

No-throw guarantee: never throws exceptions.

# /istream/iostream/operator=

protected member function

## std::iostream::operator=

<iostream> <iostream>

`copy (1) iostream& operator= (const iostream&) = delete;`  
`move (2) iostream& operator= (iostream&& rhs);`

### Move assignment

Exchanges all internal members between *rhs* and *\*this*, except the pointer to the associated *stream buffers*: *rdbuf* shall return the same in both objects as before the call.

This is the same behavior as calling member [iostream::swap](#).

Derived classes can call this function to implement move semantics.

## Parameters

*rhs*

Another [istream](#) object.

## Return Value

*\*this*

## Data races

Modifies both stream objects (*\*this* and *rhs*).

## Exception safety

No-throw guarantee: this member function never throws exceptions.

## See also

<a href="#">iostream::swap</a>	Swap internals (protected member function )
--------------------------------	---

# /istream/iostream/swap

protected member function

## std::iostream::swap

<iostream> <iostream>

`void swap (iostream& x);`

### Swap internals

Exchanges all internal members between `x` and `*this`, except the pointer to the associated *stream buffers*: `rdbuf` shall return the same in both objects as before the call.

Internally, the function calls `istream::swap`.

Derived classes can call this function to implement custom `swap` functions.

## Parameters

`x`  
Another `istream` object.

## Return Value

none

## Data races

Modifies both stream objects (`*this` and `x`).

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

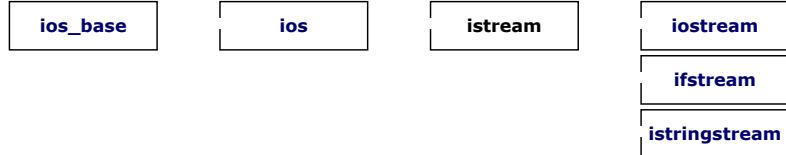
`istream::swap` | Swap internals (protected member function)

# /istream/istream

class  
**std::istream** <iostream> <iostream>

`typedef basic_istream<char> istream;`

**Input stream**



Input stream objects can read and interpret input from sequences of characters. Specific members are provided to perform these input operations (see [functions below](#)).

The standard object `cin` is an object of this type.

This is an instantiation of `basic_istream` with the following template parameters:

template parameter	definition	comments
<code>charT</code>	<code>char</code>	Aliased as member <code>char_type</code>
<code>traits</code>	<code>char_traits&lt;char&gt;</code>	Aliased as member <code>traits_type</code>

Objects of these classes keep a set of internal fields inherited from `ios_base` and `ios`:

	field	member functions	description
Formatting	format flags	<code>flags</code> <code>setf</code> <code>unsetf</code>	A set of internal flags that affect how certain input/output operations are interpreted or generated. See member type <code>fmtflags</code> .
	field width	<code>width</code>	Width of the next formatted element to insert.
	display precision	<code>precision</code>	Decimal precision for the next floating-point value inserted.
	locale	<code>getloc</code> <code>imbuf</code>	The <code>locale</code> object used by the function for formatted input/output operations affected by localization properties.
	fill character	<code>fill</code>	Character to pad a formatted field up to the <i>field width</i> ( <code>width</code> ).
State	error state	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	exception mask	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
Other	callback stack	<code>register_callback</code>	Stack of pointers to functions that are called when certain events occur.
	extensible arrays	<code>iword</code> <code>pword</code> <code>xalloc</code>	Internal arrays to store objects of type <code>long</code> and <code>void*</code> .
	tied stream	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	stream buffer	<code>rdbuf</code>	Pointer to the associated <code>streambuf</code> object, which is charge of all input/output operations.

To these, `istream` adds the *character count* (accessible using member `gcount`).

## Member types

The class contains the following member class:

`sentry` | Prepare stream for input (public member class)

Along with the following member types:

member type	definition
char_type	char
traits_type	char_traits<char>
int_type	int
pos_type	streampos
off_type	streamoff

And these member types inherited from `ios_base` through `ios`:

<b>event</b>	Type to indicate event type (public member type )
<b>event_callback</b>	Event callback function type (public member type )
<b>failure</b>	Base class for stream exceptions (public member class )
<b>fmtflags</b>	Type for stream format flags (public member type )
<b>Init</b>	Initialize standard stream objects (public member class )
<b>iostate</b>	Type for stream state flags (public member type )
<b>openmode</b>	Type for stream opening mode flags (public member type )
<b>seekdir</b>	Type for stream seeking direction flag (public member type )

## Public member functions

<b>(constructor)</b>	Construct object (public member function )
<b>(destructor)</b>	Destroy object (public member function )

### Formatted input:

<b>operator&gt;&gt;</b>	Extract formatted input (public member function )
-------------------------	---

### Unformatted input:

<b>gcount</b>	Get character count (public member function )
<b>get</b>	Get characters (public member function )
<b>getline</b>	Get line (public member function )
<b>ignore</b>	Extract and discard characters (public member function )
<b>peek</b>	Peek next character (public member function )
<b>read</b>	Read block of data (public member function )
<b>readsome</b>	Read data available in buffer (public member function )
<b>putback</b>	Put character back (public member function )
<b>unget</b>	Unget character (public member function )

### Positioning:

<b>tell</b>	Get position in input sequence (public member function )
<b>seekg</b>	Set position in input sequence (public member function )

### Synchronization:

<b>sync</b>	Synchronize input buffer (public member function )
-------------	--

## Protected member functions

<b>operator=</b>	Move assignment (protected member function )
<b>swap</b>	Swap internals (protected member function )

## Public member functions inherited from `ios`

<b>good</b>	Check whether state of stream is good (public member function )
<b>eof</b>	Check whether eofbit is set (public member function )
<b>fail</b>	Check whether either failbit or badbit is set (public member function )
<b>bad</b>	Check whether badbit is set (public member function )
<b>operator!</b>	Evaluate stream (not) (public member function )
<b>operator bool</b>	Evaluate stream (public member function )
<b>rdbuf</b>	Get error state flags (public member function )
<b>setstate</b>	Set error state flag (public member function )
<b>clear</b>	Set error state flags (public member function )
<b>copyfmt</b>	Copy formatting information (public member function )
<b>fill</b>	Get/set fill character (public member function )
<b>exceptions</b>	Get/set exceptions mask (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>tie</b>	Get/set tied stream (public member function )
<b>rdbuf</b>	Get/set stream buffer (public member function )
<b>narrow</b>	Narrow character (public member function )
<b>widen</b>	Widen character (public member function )

## Public member functions inherited from `ios_base`

<b>flags</b>	Get/set format flags (public member function )
<b>setf</b>	Set specific format flags (public member function )

<b>unsetf</b>	Clear specific format flags (public member function )
<b>precision</b>	Get/Set floating-point decimal precision (public member function )
<b>width</b>	Get/set field width (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>getloc</b>	Get current locale (public member function )
<b>xalloc</b>	Get new index for extensible array [static] (public static member function )
<b>iword</b>	Get integer element of extensible array (public member function )
<b>pword</b>	Get pointer element of extensible array (public member function )
<b>register_callback</b>	Register event callback function (public member function )
<b>sync_with_stdio</b>	Toggle synchronization with cstdio streams [static] (public static member function )

## /istream/istream/gcount

public member function

### std::istream::gcount

<iostream> <iostream>

**streamsize gcount() const;**

#### Get character count

Returns the number of characters extracted by the last *unformatted input operation* performed on the object.

The *unformatted input operations* that modify the value returned by this function are: `get`, `getline`, `ignore`, `peek`, `read`, `readsome`, `putback` and `unget`.

Notice though, that `peek`, `putback` and `unget` do not actually extract any characters, and thus `gcount` will always return zero after calling any of them.

#### Parameters

none

#### Return Value

The number of characters extracted by the last unformatted input operation.

`streamsize` is a signed integral type.

#### Example

```
1 // cin.gcount example
2 #include <iostream>      // std::cin, std::cout
3
4 int main () {
5     char str[20];
6
7     std::cout << "Please, enter a word: ";
8     std::cin.getline(str,20);
9     std::cout << std::cin.gcount() << " characters read: " << str << '\n';
10
11    return 0;
12 }
```

Possible output:

```
Please, enter a word: simplify
9 characters read: simplify
```

#### Data races

Accesses the stream object.

Concurrent access to the same stream object may cause data races.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the stream.

#### See also

<b>istream::get</b>	Get characters (public member function )
<b>istream::getline</b>	Get line (public member function )
<b>istream::ignore</b>	Extract and discard characters (public member function )
<b>istream::read</b>	Read block of data (public member function )
<b>istream::readsome</b>	Read data available in buffer (public member function )

## /istream/istream/get

public member function

### std::istream::get

<iostream> <iostream>

```
single character (1) istream& get (char& c);
c-string (2) istream& get (char* s, streamsize n);
                           istream& get (char* s, streamsize n, char delim);
```

```
stream buffer (3) istream& get (streambuf& sb);  
istream& get (streambuf& sb, char delim);
```

## Get characters

Extracts characters from the stream, as *unformatted input*:

### (1) single character

Extracts a single character from the stream.

The character is either returned (first signature), or set as the value of its argument (second signature).

### (2) c-string

Extracts characters from the stream and stores them in *s* as a c-string, until either (*n*-1) characters have been extracted or the *delimiting character* is encountered: the *delimiting character* being either the *newline character* ('\n') or *delim* (if this argument is specified).

The *delimiting character* is **not** extracted from the input sequence if found, and remains there as the next character to be extracted from the stream (see [getline](#) for an alternative that *does* discard the *delimiting character*).

A *null character* ('\0') is automatically appended to the written sequence if *n* is greater than zero, even if an empty string is extracted.

### (3) stream buffer

Extracts characters from the stream and inserts them into the output sequence controlled by the *stream buffer* object *sb*, stopping either as soon as such an insertion fails or as soon as the *delimiting character* is encountered in the input sequence (the *delimiting character* being either the *newline character*, '\n', or *delim*, if this argument is specified).

Only the characters successfully inserted into *sb* are extracted from the stream: Neither the *delimiting character*, nor eventually the character that failed to be inserted at *sb*, are extracted from the input sequence and remain there as the next character to be extracted from the stream.

The function also stops extracting characters if the *end-of-file* is reached. If this is reached prematurely (before meeting the conditions described above), the function sets the [eofbit](#) flag.

Internally, the function accesses the input sequence by first constructing a *sentry* object (with *noskipws* set to true). Then (if *good*), it extracts characters from its associated *stream buffer* object as if calling its member functions [sbumpc](#) or [sgetc](#), and finally destroys the *sentry* object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member [gcount](#).

## Parameters

c	The reference to a character where the extracted value is stored.
s	Pointer to an array of characters where extracted characters are stored as a c-string. If the function does not extract any characters (or if the first character extracted is the <i>delimiter character</i> ) and <i>n</i> is greater than zero, this is set to an empty c-string.
n	Maximum number of characters to write to <i>s</i> (including the terminating null character). If this is less than 2, the function does not extract any characters and sets <a href="#">failbit</a> . <i>streamsize</i> is a signed integral type.
delim	Explicit <i>delimiting character</i> : The operation of extracting successive characters stops as soon as the next character to extract compares equal to this.
sb	A <i>streambuf</i> object on whose controlled output sequence the characters are copied.

## Return Value

The first signature returns the character read, or the *end-of-file* value ([EOF](#)) if no characters are available in the stream (note that in this case, the [failbit](#) flag is also set).

All other signatures always return *\*this*. Note that this return value can be checked for the state of the stream (see [casting a stream to bool](#) for more info).

Errors are signaled by modifying the *internal state flags*:

flag	error
<a href="#">eofbit</a>	The function stopped extracting characters because the input sequence has no more characters available ( <i>end-of-file</i> reached).
<a href="#">failbit</a>	Either no characters were written or an empty c-string was stored in <i>s</i> .
<a href="#">badbit</a>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member [exceptions](#), the function throws an exception of member type [failure](#).

## Example

```
1 // istream::get example  
2 #include <iostream> // std::cin, std::cout  
3 #include <fstream> // std::ifstream  
4  
5 int main () {  
6     char str[256];  
7  
8     std::cout << "Enter the name of an existing text file: ";  
9     std::cin.get (str,256); // get c-string  
10  
11    std::ifstream is(str); // open file  
12  
13    char c;  
14    while (is.get(c)) // loop getting single characters  
15        std::cout << c;  
16  
17    is.close(); // close file  
18  
19    return 0;  
20 }
```

This example prompts for the name of an existing text file and prints its content on the screen, using `cin.get` both to get individual characters and c-strings.

## Data races

Modifies `c`, `sb` or the elements in the array pointed by `s`.

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state.

Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::getline</code>	Get line (public member function)
<code>istream::ignore</code>	Extract and discard characters (public member function)
<code>istream::gcount</code>	Get character count (public member function)

# /istream/istream/getline

public member function

## std::istream::getline

`<iostream> <iostream>`

```
istream& getline (char* s, streamsize n );
istream& getline (char* s, streamsize n, char delim );
```

### Get line

Extracts characters from the stream as *unformatted input* and stores them into `s` as a c-string, until either the extracted character is the *delimiting character*, or `n` characters have been written to `s` (including the terminating null character).

The *delimiting character* is the *newline character* ('`\n`') for the first form, and `delim` for the second: when found in the input sequence, it is extracted from the input sequence, but discarded and not written to `s`.

The function will also stop extracting characters if the *end-of-file* is reached. If this is reached prematurely (before either writing `n` characters or finding `delim`), the function sets the `eofbit` flag.

The `failbit` flag is set if the function extracts no characters, or if the *delimiting character* is not found once (`n-1`) characters have already been written to `s`. Note that if the character that follows those (`n-1`) characters in the input sequence is precisely the *delimiting character*, it is also extracted and the `failbit` flag is not set (the extracted sequence was exactly `n` characters long).

A *null character* ('`\0`') is automatically appended to the written sequence if `n` is greater than zero, even if an empty string is extracted.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member `gcount`.

This function is overloaded for `string` objects in header `<string>`: See `getline(string)`.

## Parameters

`s` Pointer to an array of characters where extracted characters are stored as a c-string.

`n` Maximum number of characters to write to `s` (including the terminating null character). If the function stops reading because this limit is reached without finding the *delimiting character*, the `failbit` internal flag is set. `streamsize` is a signed integral type.

`delim` Explicit *delimiting character*: The operation of extracting successive characters stops as soon as the next character to extract compares equal to this.

## Return Value

The `istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence has no more characters available ( <i>end-of-file</i> reached).
<code>failbit</code>	Either the <i>delimiting character</i> was not found or no characters were extracted at all (because the <i>end-of-file</i> was before the first character or because the construction of <code>sentry</code> failed).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // istream::getline example
2 #include <iostream>      // std::cin, std::cout
3
4 int main () {
```

```

5  char name[256], title[256];
6
7  std::cout << "Please, enter your name: ";
8  std::getline (name,256);
9
10 std::cout << "Please, enter your favourite movie: ";
11 std::getline (title,256);
12
13 std::cout << name << "'s favourite movie is " << title;
14
15 return 0;
16 }
```

This example illustrates how to get lines from the standard input stream (`cin`).

## Data races

Modifies the elements in the array pointed by `s` and the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

# /istream/istream/ignore

public member function

## std::istream::ignore

`<iostream> <iostream>`

`istream& ignore (streamsize n = 1, int delim = EOF);`

### Extract and discard characters

Extracts characters from the input sequence and discards them, until either `n` characters have been extracted, or one compares equal to `delim`.

The function also stops extracting characters if the *end-of-file* is reached. If this is reached prematurely (before either extracting `n` characters or finding `delim`), the function sets the `eofbit` flag.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbufmpc` or `sgetc`, and finally destroys the `sentry` object before returning.

## Parameters

`n`

Maximum number of characters to extract (and ignore).

If this is exactly `numeric_limits<streamsize>::max()`, there is no limit: As many characters are extracted as needed until `delim` (or the *end-of-file*) is found.

`streamsize` is a signed integral type.

`delim`

Delimiting character: The function stops extracting characters as soon as an extracted character compares equal to this.

Note that the *delimiting character* is extracted, and thus the next input operation will continue on the character that follows it (if any).

If this is the *end-of-file* value (`EOF`), no character will compare equal, and thus exactly `n` characters will be discarded (unless the function fails or the *end-of-file* is reached).

## Return Value

The `istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence has no more characters available ( <i>end-of-file</i> reached).
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```

1 // istream::ignore example
2 #include <iostream>      // std::cin, std::cout
3
4 int main () {
5     char first, last;
6
7     std::cout << "Please, enter your first name followed by your surname: ";
8
9     first = std::cin.get();    // get one character
10    std::cin.ignore(256, ' '); // ignore until space
11 }
```

```

12     last = std::cin.get();      // get one character
13
14     std::cout << "Your initials are " << first << last << '\n';
15
16     return 0;
17 }
```

Possible output:

```
Please, enter your first name followed by your surname: John Smith
Your initials are JS
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::peek</code>	Peek next character ( <a href="#">public member function</a> )
<code>istream::get</code>	Get characters ( <a href="#">public member function</a> )
<code>istream::getline</code>	Get line ( <a href="#">public member function</a> )
<code>istream::read</code>	Read block of data ( <a href="#">public member function</a> )
<code>istream::readsome</code>	Read data available in buffer ( <a href="#">public member function</a> )

## /istream/istream/istream

public member function

### std::istream::istream

`<iostream> <iostream>`

```

initialization (1) explicit istream (streambuf* sb);
initialization (1) explicit istream (streambuf* sb);
copy (2) istream& (const istream&) = delete;
move (3) protected: istream& (istream&& x);
```

#### Construct object

Constructs an `istream` object.

##### (1) initialization constructor

Assigns initial values to the components of its base classes by calling the inherited member `ios::init` with `sb` as argument.

##### (2) copy constructor (deleted)

Deleted: no copy constructor.

##### (3) move constructor (protected)

Acquires the contents of `x`, except its associated *stream buffer*: It copies `x`'s `gcount` value and then calls `ios::move` to transfer `x`'s `ios` components. `x` is left with a `gcount` value of zero, not *tied*, and with its associated *stream buffer* unchanged (all other components of `x` are in an unspecified but valid state after the call).

## Parameters

`sb`  
pointer to a `streambuf` object.

`x`  
Another `istream` object.

## Example

```

1 // istream constructor
2 #include <iostream>      // std::ios, std::istream, std::cout
3 #include <fstream>       // std::filebuf
4
5 int main () {
6     std::filebuf fb;
7     if (fb.open ("test.txt",std::ios::in))
8     {
9         std::istream is(&fb);
10        while (is)
11            std::cout << char(is.get());
12        fb.close();
13    }
14    return 0;
15 }
```

This example code uses a `filebuf` object (derived from `streambuf`) to open a file called `test.txt`. The buffer is passed as parameter to the constructor of the

`istream` object *is*, associating it to the stream. Then, the program uses the input stream to print its contents to `cout`.

Objects of `istream` classes are seldom constructed directly. Generally some derived class is used (like the standard `ifstream` or `istringstream`).

### Data races

The object pointed by *sb* may be accessed and/or modified.

### Exception safety

If an exception is thrown, the only side effects may come from accessing/modifying *sb*.

### See also

<code>ios::init</code>	Initialize object (protected member function )
------------------------	--

## /istream/istream/~istream

public member function

### std::istream::~istream

<iostream> <iostream>

`virtual ~istream();`

#### Destroy object

Destroys an object of this class.

Note that this does *not* destroy nor performs any operations on the associated *stream buffer object*.

### Data races

The object is modified.

### Exception safety

No-throw guarantee: never throws exceptions.

### See also

## /istream/istream/operator=

protected member function

### std::istream::operator=

<iostream> <iostream>

`copy (1) basic_istream& operator= (const basic_istream&) = delete;`  
`move (2) basic_istream& operator= (basic_istream&& rhs);`

#### Move assignment

Exchanges all internal members between *rhs* and *\*this*, except the pointers to the associated *stream buffers*: `rdbuf` shall return the same in both objects as before the call.

This is the same behavior as calling member `istream::swap`.

Derived classes can call this function to implement move semantics.

### Parameters

`rhs`

Another `istream` object.

### Return Value

`*this`

### Data races

Modifies both stream objects (*\*this* and *rhs*).

### Exception safety

No-throw guarantee: this member function never throws exceptions.

### See also

<code>istream::swap</code>	Swap internals (protected member function )
----------------------------	---

## /istream/istream/operator>>

public member function

<iostream> <iostream>

## std::istream::operator>>

```
        istream& operator>> (bool& val);
        istream& operator>> (short& val);
        istream& operator>> (unsigned short& val);
        istream& operator>> (int& val);
        istream& operator>> (unsigned int& val);
arithmetic types (1)    istream& operator>> (long& val);
                        istream& operator>> (unsigned long& val);
                        istream& operator>> (float& val);
                        istream& operator>> (double& val);
                        istream& operator>> (long double& val);
                        istream& operator>> (void*& val);
stream buffers (2)     istream& operator>> (streambuf* sb );
                        istream& operator>> (istream& (*pf)(istream&));
manipulators (3)       istream& operator>> (ios& (*pf)(ios&));
                        istream& operator>> (ios_base& (*pf)(ios_base&));

        istream& operator>> (bool& val);
        istream& operator>> (short& val);
        istream& operator>> (unsigned short& val);
        istream& operator>> (int& val);
        istream& operator>> (unsigned int& val);
        istream& operator>> (long& val);
        istream& operator>> (unsigned long& val);
        istream& operator>> (long long& val);
        istream& operator>> (unsigned long long& val);
        istream& operator>> (float& val);
        istream& operator>> (double& val);
        istream& operator>> (long double& val);
        istream& operator>> (void*& val);
stream buffers (2)     istream& operator>> (streambuf* sb );
                        istream& operator>> (istream& (*pf)(istream&));
manipulators (3)       istream& operator>> (ios& (*pf)(ios&));
                        istream& operator>> (ios_base& (*pf)(ios_base&));
```

### Extract formatted input

This operator (>>) applied to an input stream is known as *extraction operator*. It is overloaded as a member function for:

#### (1) arithmetic types

Extracts and parses characters sequentially from the stream to interpret them as the representation of a value of the proper type, which is stored as the value of *val*.

Internally, the function accesses the input sequence by first constructing a *sentry* object (with *noskipws* set to *false*). Then (if *good*), it calls *num\_get::get* (using the stream's *selected locale*) to perform both the extraction and the parsing operations, adjusting the stream's *internal state flags* accordingly. Finally, it destroys the *sentry* object before returning.

#### (2) stream buffers

Extracts as many characters as possible from the stream and inserts them into the output sequence controlled by the *stream buffer* object pointed by *sb* (if any), until either the input sequence is exhausted or the function fails to insert into the object pointed by *sb*.

The function is considered to perform *formatted input*: Internally, the function accesses the input sequence by first constructing a *sentry* object (with *noskipws* set to *false*). Then (if *good*), it extracts characters from its associated *stream buffer* object as if calling its member functions *sbumpc* or *sgetc*, and finally destroys the *sentry* object before returning.

The function is considered to perform *unformatted input*: Internally, the function accesses the input sequence by first constructing a *sentry* object (with *noskipws* set to *true*). Then (if *good*), it extracts characters from its associated *stream buffer* object as if calling its member functions *sbumpc* or *sgetc*, and finally destroys the *sentry* object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member *gcount*.

#### (3) manipulators

Calls *pf(\*this)*, where *pf* may be a *manipulator*.

*Manipulators* are functions specifically designed to be called when used with this operator.

This operation has no effect on the input sequence and extracts no characters (unless the manipulator itself does, like *ws*).

See *operator>>* for additional overloads (as non-member functions) of this operator.

Except where stated otherwise, calling this function does not alter the value returned by member *gcount*.

### Parameters

*val*

Object where the value that the extracted characters represent is stored.

Notice that the type of this argument (along with the stream's *format flags*) influences what constitutes a valid representation.

*sb*

Pointer to a *streambuf* object on whose controlled output sequence the characters are copied.

*pf*

A function that takes and returns a stream object. It generally is a *manipulator function*.

The standard manipulators which have an effect when used on standard *istream* objects are:

manipulator	Effect
<i>ws</i>	Extracts whitespaces.
<i>boolalpha/noboolalpha</i>	Activates/deactivates the extraction of alphanumerical representations of values of type <i>bool</i> .
<i>skipws/noskipws</i>	Activates/deactivates whether leading whitespaces are discarded before formatted input operations.
<i>dec/hex/oct</i>	Sets that base used to interpret integral numerical values.

The following extended manipulators can also be applied to *istream* objects (these take additional arguments and require the explicit inclusion of the *<iomanip>* header):

manipulator	Effect
<i>setbase</i>	Sets the numerical base used to interpret integral numerical values.
<i>setiosflags/resetiosflags</i>	Set/reset format flags.

### Return Value

The `istream` object (`*this`).

The extracted value or sequence is not returned, but directly stored in the variable passed as argument.

Errors are signaled by modifying the *internal state flags*, except for (3), that never sets any flags (but the particular manipulator applied may):

flag	error
<code>eofbit</code>	The input sequence has no more characters available ( <i>end-of-file</i> reached).
<code>failbit</code>	Either no characters were extracted, or the characters extracted could not be interpreted as a valid value of the appropriate type. For (2), it is set when no characters are inserted in the object pointed by <code>sb</code> , or when <code>sb</code> is a <i>null pointer</i> .
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // example on extraction
2 #include <iostream>      // std::cin, std::cout, std::hex
3
4 int main () {
5     int n;
6
7     std::cout << "Enter a number: ";
8     std::cin >> n;
9     std::cout << "You have entered: " << n << '\n';
10
11    std::cout << "Enter a hexadecimal number: ";
12    std::cin >> std::hex >> n;           // manipulator
13    std::cout << "Its decimal equivalent is: " << n << '\n';
14
15    return 0;
16 }
```

This example demonstrates the use of some of the overloaded `operator>>` functions shown above using the standard `istream` object `cin`.

## Data races

Modifies `val` or the object pointed by `sb`.

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::get</code>	Get characters (public member function )
<code>istream::getline</code>	Get line (public member function )
<code>ostream::operator&lt;&lt;</code>	Insert formatted output (public member function )
<code>istream::sentry</code>	Prepare stream for input (public member class )

## /istream/istream/operator-free

public member function

### std::operator>> (istream)

`<iostream> <iostream>`

*single character (1)*    `istream& operator>> (istream& is, char& c);`  
`istream& operator>> (istream& is, signed char& c);`  
`istream& operator>> (istream& is, unsigned char& c);`

*character sequence (2)*    `istream& operator>> (istream& is, char* s);`  
`istream& operator>> (istream& is, signed char* s);`  
`istream& operator>> (istream& is, unsigned char* s);`

*single character (1)*    `istream& operator>> (istream& is, char& c);`  
`istream& operator>> (istream& is, signed char& c);`  
`istream& operator>> (istream& is, unsigned char& c);`

*character sequence (2)*    `istream& operator>> (istream& is, char* s);`  
`istream& operator>> (istream& is, signed char* s);`  
`istream& operator>> (istream& is, unsigned char* s);`

*rvalue extraction (3)*    `template<class charT, class traits, class T>`  
`basic_istream<charT,traits>&`  
`operator>> (basic_istream<charT,traits>&& is, T& val);`

## Extract characters

This operator (`>>`) applied to an input stream is known as *extraction operator*, and performs *formatted input*:

### (1) single character

Extracts the next character from `is` and stores it as the value of `c`.

### (2) character sequence

Extracts characters from *is* and stores them in *s* as a c-string, stopping as soon as either a *whitespace character* is encountered or (*width()*-1) characters have been extracted (if *width* is not zero).

A *null character* (*chart`()*) is automatically appended to the written sequence.

The function then resets *width* to zero.

### (3) rvalue extraction

Allows extracting from rvalue *istream* objects, with the same effect as from lvalues: It effectively calls: *is>>val*.

Internally, the function accesses the input sequence of *is* by first constructing a *sentry* with *noskipws* set to *false*: this may *flush* its *tied stream* and/or discard leading whitespaces (see *istream::sentry*). Then (if *good*), it extracts characters from *is*'s associated *stream buffer* object (as if calling its member functions *sbumpc* or *sgetc*), and finally destroys the *sentry* object before returning.

Notice that if this function extracts the last character of a stream when extracting a single character (1), it does not set its *eofbit* flag, but attempting to extract beyond it does.

Calling this function does not alter the value returned by *gcount* on *is*.

## Parameters

<i>is</i>	Input stream object from which characters are extracted.
<i>c</i>	Object where the extracted character is stored.
<i>s</i>	Pointer to an array of characters where extracted characters are stored as a c-string. <i>is</i> 's member function <i>width</i> may be used to specify a limit on the number of characters to write.
<i>val</i>	Object where content is stored. T shall be a type supported as right-hand side argument either by this function or by <i>is</i> 's member function <i>operator&gt;&gt;</i> . <i>charT</i> and <i>traits</i> are the class template parameters of <i>istream</i> (see <i>basic_istream</i> ).

## Return Value

The *istream* object (*is*).

The extracted value or sequence is not returned, but directly stored in the variable passed as argument.

Errors are signaled by modifying the *internal state flags*:

flag	error
<i>eofbit</i>	The function stopped extracting characters because the input sequence controlled by <i>is</i> has no more characters available (end-of-file reached).
<i>failbit</i>	Either no characters were extracted, or these could not be interpreted as a valid value of the appropriate type.
<i>badbit</i>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with *is*'s member *exceptions*, the function throws an exception of its *failure* member type.

## Example

```
1 // example on extraction
2 #include <iostream>           // std::cin, std::cout
3
4 int main () {
5     char str[10];
6
7     std::cout << "Enter a word: ";
8     std::cin.width (10);          // limit width
9     std::cin >> str;
10    std::cout << "The first 9 chars of your word are: " << str << '\n';
11
12    return 0;
13 }
```

This example demonstrates the use of some of the overloaded *operator>>* functions shown above using the standard *istream* object *cin*.

## Data races

Modifies *c* or the elements of the array pointed by *s*.

Modifies *is*.

Concurrent access to the same stream object may cause data races, except for the standard stream object *cin* when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, all objects involved are in valid states.

It throws an exception of member type *failure* if the resulting *error state flag* for *is* is not *goodbit* and member *exceptions* was set to throw for that state in *is*.

Any exception thrown by an internal operation is caught and handled by the function, setting *is*'s *badbit* flag. If *badbit* was set on the last call to *exceptions* for *is*, the function rethrows the caught exception.

## See also

<a href="#">istream::get</a>	Get characters (public member function )
<a href="#">istream::getline</a>	Get line (public member function )
<a href="#">ostream::operator&lt;&lt;</a>	Insert formatted output (public member function )
<a href="#">istream::sentry</a>	Prepare stream for input (public member class )

# /istream/istream/peek

public member function

## std::istream::peek

<iostream> <iostream>

int peek();

### Peek next character

Returns the next character in the input sequence, without extracting it: The character is left as the next character to be extracted from the stream.

If any *internal state flags* is already set before the call or is set during the call, the function returns the *end-of-file* value (`EOF`).

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it reads one character from its associated `stream buffer` object by calling its member function `sgetc`, and finally destroys the `sentry` object before returning.

Calling this function sets the value returned by `gcount` to zero.

### Parameters

none

### Return Value

The next character in the input sequence, as a value of type `int`.

If there are no more characters to read in the input sequence, or if any *internal state flags* is set, the function returns the *end-of-file* value (`EOF`), and leaves the proper *internal state flags* set:

flag	error
<code>eofbit</code>	No character could be peeked because the input sequence has no characters available (end-of-file reached).
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

### Example

```
1 // istream::peek example
2 #include <iostream>      // std::cin, std::cout
3 #include <string>        // std::string
4 #include <cctype>        // std::isdigit
5
6 int main () {
7
8     std::cout << "Please, enter a number or a word: ";
9     std::cout.flush();    // ensure output is written
10
11    std::cin >> std::ws; // eat up any leading white spaces
12    int c = std::cin.peek(); // peek character
13
14    if ( c == EOF ) return 1;
15    if ( std::isdigit(c) )
16    {
17        int n;
18        std::cin >> n;
19        std::cout << "You entered the number: " << n << '\n';
20    }
21    else
22    {
23        std::string str;
24        std::cin >> str;
25        std::cout << "You entered the word: " << str << '\n';
26    }
27
28    return 0;
29 }
```

Possible output:

```
Please, enter a number or a word: foobar
You entered the word: foobar
```

### Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which read characters are attributed to threads).

### Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

### See also

`istream::get`

Get characters (public member function )

## /istream/istream/putback

public member function

### std::istream::putback

<iostream> <iostream>

`istream& putback (char c);`

#### Put character back

Attempts to decrease the current location in the stream by one character, making the last character extracted from the stream once again available to be extracted by input operations.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it calls `sputbackc(c)` on its associated `stream buffer` object (if any). Finally, it destroys the `sentry` object before returning.

If the `eofbit` flag is set before the call, the function fails (sets `failbit` and returns).

The function clears the `eofbit` flag, if set before the call.

If the call to `sputbackc` fails, the function sets the `badbit` flag. Note that this may happen even if `c` was indeed the last character extracted from the stream (depending on the internals of the associated `stream buffer` object).

Calling this function sets the value returned by `gcount` to zero.

#### Parameters

`c`

Character to be put back.

If this does not match the character at the put back position, the behavior depends on the particular `stream buffer` object associated to the stream:

- In `string buffers`, the value is overwritten for output stream buffers, but the function fails on input buffers.
- In `file buffers`, the value is overwritten on the intermediate buffer (if supported): reading the character again will produce `c`, but the associated input sequence is not modified.

Other types of `stream buffer` may either fail, be ignored, or overwrite the character at that position.

#### Return Value

The `istream` object (`*this`).

Errors are signaled by modifying the `internal state flags`:

flag	error
<code>eofbit</code>	-
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Either the internal call to <code>sputbackc</code> failed, or another error occurred on the stream (such as when the function catches an exception thrown by an internal operation, or when no <code>stream buffer</code> is associated with the stream). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an `internal state flag` that was registered with member `exceptions`, the function throws an exception of member type `failure`.

#### Example

```

1 // istream::putback example
2 #include <iostream>           // std::cin, std::cout
3 #include <iomanip>            // std::string
4
5 int main () {
6     std::cout << "Please, enter a number or a word: ";
7     char c = std::cin.get();
8
9     if ( (c >= '0') && (c <= '9') )
10    {
11        int n;
12        std::cin.putback (c);
13        std::cin >> n;
14        std::cout << "You entered a number: " << n << '\n';
15    }
16    else
17    {
18        std::string str;
19        std::cin.putback (c);
20        getline (std::cin,str);
21        std::cout << "You entered a word: " << str << '\n';
22    }
23    return 0;
24 }
```

Possible output:

```
Please, enter a number or a word: pocket
You entered a word: pocket
```

#### Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::get</code>	Get characters (public member function )
<code>istream::unget</code>	Unget character (public member function )

# /istream/istream/read

public member function

## std::istream::read

`<iostream> <iostream>`

`istream& read (char* s, streamsize n);`

### Read block of data

Extracts *n* characters from the stream and stores them in the array pointed to by *s*.

This function simply copies a block of data, without checking its contents nor appending a *null character* at the end.

If the input sequence runs out of characters to extract (i.e., the end-of-file is reached) before *n* characters have been successfully read, the array pointed to by *s* contains all the characters read until that point, and both the `eofbit` and `failbit` flags are set for the stream.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from its associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member `gcount`.

## Parameters

<code>s</code>	Pointer to an array where the extracted characters are stored.
<code>n</code>	Number of characters to extract. <code>streamsize</code> is a signed integral type.

## Return Value

The `istream` object (`*this`).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence has no more characters available (end-of-file reached).
<code>failbit</code>	Either the function could not extract <i>n</i> characters or the construction of <code>sentry</code> failed.
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // read a file into memory
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::ifstream
4
5 int main () {
6
7     std::ifstream is ("test.txt", std::ifstream::binary);
8     if (is) {
9         // get length of file:
10        is.seekg (0, is.end);
11        int length = is.tellg();
12        is.seekg (0, is.beg);
13
14        char * buffer = new char [length];
15
16        std::cout << "Reading " << length << " characters... ";
17        // read data as a block:
18        is.read (buffer,length);
19
20        if (is)
21            std::cout << "all characters read successfully.";
22        else
23            std::cout << "error: only " << is.gcount() << " could be read";
24        is.close();
25
26        // ...buffer contains the entire file...
27
28        delete[] buffer;
29    }
30    return 0;
31 }
```

Possible output:

```
Reading 640 characters... all characters read successfully.
```

## Data races

Modifies the elements in the array pointed to by *s* and the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state.

Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::get</code>	Get characters (public member function )
<code>istream::readsome</code>	Read data available in buffer (public member function )
<code>istream::operator&gt;&gt;</code>	Extract formatted input (public member function )

# /istream/istream/readsome

public member function

## std::istream::readsome

`<iostream> <iostream>`

```
streamsize readsome (char* s, streamsize n);
```

### Read data available in buffer

Extracts up to *n* characters from the stream and stores them in the array pointed by *s*, stopping as soon as the internal buffer kept by the *associated stream buffer object* (if any) runs out of characters, even if the *end-of-file* has not yet been reached.

The function is meant to be used to read data from certain types of asynchronous sources that may eventually wait for more characters, since it stops extracting characters as soon as the internal buffer is exhausted, avoiding potential delays.

Note that this function relies on internals of the particular *stream buffer* object associated to the stream whose behavior is mostly implementation-defined for standard classes.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it checks how many characters are currently available at the associated *stream buffer* object by calling its member function `in_avail` and extracts up to that many characters by calling `sbumpc` (or `sgetc`). Finally, it destroys the `sentry` object before returning.

The number of characters successfully read and stored by this function can be accessed by calling member `gcount`.

## Parameters

*s* Pointer to an array where the extracted characters are stored.

*n* Maximum number of characters to extract.  
`streamsize` is a signed integral type.

## Return Value

The number of characters stored.

`streamsize` is a signed integral type.

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	The input sequence has no characters available (as reported by <code>rdbuf()-&gt;in_avail()</code> returning -1).
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <i>stream state</i> was not <code>good</code> before the call).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Data races

Modifies the elements in the array pointed by *s* and the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state.

Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::read</code>	Read block of data (public member function )
----------------------------	--

<b>ostream::write</b>	Write block of data (public member function )
<b>istream::operator&gt;&gt;</b>	Extract formatted input (public member function )

## /istream/istream/seekg

public member function

### std::istream::seekg

<iostream> <iostream>

```
(1) istream& seekg (streampos pos);
(2) istream& seekg (streamoff off, ios_base::seekdir way);
```

#### Set position in input sequence

Sets the position of the next character to be extracted from the input stream.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it calls either `pubseekpos` (1) or `pubseekoff` (2) on its associated `stream buffer` object (if any). Finally, it destroys the `sentry` object before returning.

If the `eofbit` flag is set before the call, the function fails (sets `failbit` and returns).

The function clears the `eofbit` flag, if set before the call.

Calling this function does not alter the value returned by `gcount`.

#### Parameters

`pos`

New absolute position within the stream (relative to the beginning).  
`streampos` is an `fpos` type (it can be converted to/from integral types).

`off`

Offset value, relative to the `way` parameter.  
`streamoff` is an offset type (generally, a signed integral type).

`way`

Object of type `ios_base::seekdir`. It may take any of the following constant values:

value	offset is relative to...
<code>ios_base::beg</code>	beginning of the stream
<code>ios_base::cur</code>	current position in the stream
<code>ios_base::end</code>	end of the stream

#### Return Value

The `istream` object (`*this`).

Errors are signaled by modifying the `internal state flags`:

flag	error
<code>eofbit</code>	-
<code>failbit</code>	Either the construction of <code>sentry</code> failed, or the internal call to <code>pubseekpos</code> (1) or <code>pubseekoff</code> (2) failed (i.e., either function returned -1).
<code>badbit</code>	Another error occurred on the stream (such as when the function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an `internal state flag` that was registered with member `exceptions`, the function throws an exception of member type `failure`.

#### Example

```
1 // read a file into memory
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream is ("test.txt", std::ifstream::binary);
7     if (is) {
8         // get length of file:
9         is.seekg (0, is.end);
10        int length = is.tellg();
11        is.seekg (0, is.beg);
12
13        // allocate memory:
14        char * buffer = new char [length];
15
16        // read data as a block:
17        is.read (buffer,length);
18
19        is.close();
20
21        // print content:
22        std::cout.write (buffer,length);
23
24        delete[] buffer;
25    }
26
27    return 0;
28 }
```

In this example, `seekg` is used to move the position to the end of the file, and then back to the beginning.

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting `error state flag` is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::tellg</code>	Get position in input sequence (public member function )
<code>ostream::seekp</code>	Set position in output sequence (public member function )

# /istream/istream/sentry

public member class

## std::istream::sentry

<iostream> <iostream>

`class sentry;`

### Prepare stream for input

Member class that performs a series of operations before and after each input operation:

Its constructor performs the following operations on the stream object passed as its argument (in the same order):

- If any of its `internal error flags` is set, the function sets its `failbit` flag and returns.
- If it is a `tied stream`, the function `flushes` the stream it is tied to (if its output buffer is not empty). The class may implement ways for library functions to defer this flush until the next call to `overflow` by its `associated stream buffer`.
- If its `skipws` format flag is set, and the constructor is not passed `true` as second argument (`noskipws`), all leading `whitespace characters` (locale-specific) are extracted and discarded. If this operation exhausts the source of characters, the function sets both the `failbit` and `eofbit` `internal state flags`

In case of failure during construction, it may set the stream's `failbit` flag.

There are no required operations to be performed by its destructor. But implementations may use the construction and destruction of `sentry` objects to perform additional initialization or cleanup operations on the stream common to all input operations.

All member functions that perform an input operation automatically construct an object of this class and then evaluate it (which returns `true` if no `state flag` was set). Only if this object evaluates to `true`, the function attempts the input operation (otherwise, it returns without performing it). Before returning, the function destroys the `sentry` object.

The `operator>>` formatted input operations construct the `sentry` object by passing `false` as second argument (which skips leading whitespaces). All other member functions that construct a `sentry` object pass `true` as second argument (which does not skip leading whitespaces).

The structure of this class is:

```
1 class sentry {  
2 public:  
3     explicit sentry (istream& is, bool noskipws = false);  
4     ~sentry();  
5     operator bool() const;  
6 private:  
7     sentry (const sentry&); // not defined  
8     sentry& operator= (const sentry&); // not defined  
9 };
```

```
1 class sentry {  
2 public:  
3     explicit sentry (istream& is, bool noskipws = false);  
4     ~sentry();  
5     explicit operator bool() const;  
6     sentry (const sentry&) = delete;  
7     sentry& operator= (const sentry&) = delete;  
8 };
```

## Members

`explicit sentry (istream& is, bool noskipws = false);`

Prepares the output stream for an output operation, performing the actions described above.

`~sentry();`

Performs no operations (implementation-defined).

`explicit operator bool() const;`

When the object is evaluated, it returns a `bool` value indicating whether the `sentry` constructor successfully performed all its tasks: If at some point of the construction process, an `internal error flags` was set, this function always returns `false` for that object.

## Example

```
1 // istream::sentry example  
2 #include <iostream> // std::istream, std::cout  
3 #include <string> // std::string  
4 #include <sstream> // std::stringstream  
5 #include <locale> // std::isspace, std::isdigit  
6  
7 struct Phone {
```

```

8     std::string digits;
9 };
10
11 // custom extractor for objects of type Phone
12 std::istream& operator>>(std::istream& is, Phone& tel)
13 {
14     std::istream::sentry s(is);
15     if (s) while (is.good()) {
16         char c = is.get();
17         if (std::isspace(c, is.getloc())) break;
18         if (std::isdigit(c, is.getloc())) tel.digits+=c;
19     }
20     return is;
21 }
22
23 int main () {
24     std::istringstream parseme (" (555)2326");
25     Phone myphone;
26     parseme >> myphone;
27     std::cout << "digits parsed: " << myphone.digits << '\n';
28     return 0;
29 }
```

Output:

```
digits parsed: 5552326
```

## See also

<a href="#">istream::operator&gt;&gt;</a>	Extract formatted input (public member function )
---	---

## /istream/istream/swap

protected member function

### std::istream::swap

<iostream> <iostream>

[void swap \(istream& x\);](#)

#### Swap internals

Exchanges all internal members between *x* and *\*this*, except the pointer to the associated *stream buffers*: *rdbuf* shall return the same in both objects as before the call.

Internally, the function calls [ios::swap](#) and then exchanges the values returned by [gcount](#).

Derived classes can call this function to implement custom [swap](#) functions.

## Parameters

- x* Another [istream](#) object.

## Return Value

none

## Data races

Modifies both stream objects (*\*this* and *x*).

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">istream::operator=</a>	Move assignment (protected member function )
<a href="#">ios::swap</a>	Swap internals (protected member function )

## /istream/istream/sync

public member function

### std::istream::sync

<iostream> <iostream>

[int sync\(\);](#)

#### Synchronize input buffer

Synchronizes the associated *stream buffer* with its controlled input sequence.

Specifics of the operation depend on the particular implementation of the *stream buffer* object associated to the stream.

Internally, the function accesses the input sequence by first constructing a [sentry](#) object (with *noskipws* set to *true*). Then (if *good*), it calls [pubsync](#) on its associated *stream buffer* object (if *rdbuf* is null, the function returns -1 instead). Finally, it destroys the [sentry](#) object before returning.

If the call to [pubsync](#) fails (i.e., it returns -1), the function sets the [badbit](#) flag, and returns -1. Otherwise it returns zero, indicating success.

Notice that the called function may succeed when no action is performed, if that is the behavior defined for the *stream buffer* object on synchronization.

Calling this function does not alter the value returned by `gcount`.

## Parameters

none

## Return Value

If the function fails, either because no *stream buffer* object is associated to the stream (`rdbuf` is null), or because the call to its `pubsync` member fails, it returns -1.

Otherwise, it returns zero, indicating success.

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	-
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <i>stream state</i> was not <code>good</code> before the call).
<code>badbit</code>	Either the internal call to <code>pubsync</code> returned -1, or some other error occurred on the stream (such as when the function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // syncing input stream
2 #include <iostream>      // std::cin, std::cout
3
4 int main () {
5     char first, second;
6
7     std::cout << "Please, enter a word: ";
8     first = std::cin.get();
9     std::cin.sync();
10
11    std::cout << "Please, enter another word: ";
12    second = std::cin.get();
13
14    std::cout << "The first word began by " << first << '\n';
15    std::cout << "The second word began by " << second << '\n';
16
17    return 0;
18 }
```

This example demonstrates how `sync` behaves on certain implementations of `cin`, removing any unread character from the standard input queue of characters.

Possible output:

```
Please, enter a word: test
Please enter another word: text
The first word began by t
The second word began by t
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races, except for the standard stream object `cin` when this is *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which characters are extracted or synchronized between threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>ostream::flush</code>	Flush output stream buffer (public member function )
<code>streambuf::pubsync</code>	Synchronize stream buffer (public member function )

# /istream/istream/tellg

public member function

## `std::istream::tellg`

`<iostream> <iostream>`

`streampos tellg();`

### Get position in input sequence

Returns the position of the current character in the input stream.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`) without evaluating it. Then, if member `fail` returns `true`, the function returns -1.

Otherwise, returns `rdbuf()>pubseekoff(0, cur, in)`. Finally, it destroys the `sentry` object before returning.

Notice that the function will work even if the `eofbit` flag is set before the call.

Calling this function does not alter the value returned by `gcount`.

## Parameters

none

## Return Value

The current position in the stream.

If either the `stream buffer` associated to the stream does not support the operation, or if it fails, the function returns `-1`.

`streampos` is an `fpos` type (it can be converted to/from integral types).

Errors are signaled by modifying the *internal state flags*:

flag	error
<code>eofbit</code>	-
<code>failbit</code>	The construction of <code>sentry</code> failed (such as when the <code>stream state</code> was not <code>good</code> before the call).
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // read a file into memory
2 #include <iostream>           // std::cout
3 #include <fstream>            // std::ifstream
4
5 int main () {
6     std::ifstream is ("test.txt", std::ifstream::binary);
7     if (is) {
8         // get length of file:
9         is.seekg (0, is.end);
10        int length = is.tellg();
11        is.seekg (0, is.beg);
12
13        // allocate memory:
14        char * buffer = new char [length];
15
16        // read data as a block:
17        is.read (buffer,length);
18
19        is.close();
20
21        // print content:
22        std::cout.write (buffer,length);
23
24        delete[] buffer;
25    }
26
27    return 0;
28 }
```

In this example, `tellg` is used to get the position in the stream after it has been moved with `seekg` to the end of the stream, therefore determining the size of the file.

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* is not `goodbit` and member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting `badbit`. If `badbit` was set on the last call to `exceptions`, the function rethrows the caught exception.

## See also

<code>istream::seekg</code>	Set position in input sequence ( <a href="#">public member function</a> )
<code>ostream::tellp</code>	Get position in output sequence ( <a href="#">public member function</a> )

## /istream/istream/unget

public member function

### `std::istream::unget`

`<iostream> <iostream>`

`istream& unget();`

#### Unget character

Attempts to decrease the current location in the stream by one character, making the last character extracted from the stream once again available to be extracted by input operations.

Internally, the function accesses the input sequence by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it calls `sungetc` on its

associated *stream buffer* object (if any). Finally, it destroys the *sentry* object before returning.

If the *eofbit* flag is set before the call, the function fails (sets *failbit* and returns).

The function clears the *eofbit* flag, if set before the call.

If the call to *sungetc* fails, the function sets the *badbit* flag. Note that this may happen even if the function is called right after a character has been extracted from the stream (depending on the internals of the associated *stream buffer* object).

Calling this function sets the value returned by *gcount* to zero.

## Parameters

none

## Return Value

The *istream* object (*\*this*).

Errors are signaled by modifying the *internal state flags*:

flag	error
<i>eofbit</i>	-
<i>failbit</i>	The construction of <i>sentry</i> failed (such as when the <i>stream state</i> was not <i>good</i> before the call).
<i>badbit</i>	Either the internal call to <i>sungetc</i> failed, or another error occurred on the stream (such as when the function catches an exception thrown by an internal operation, or when no <i>stream buffer</i> is associated with the stream). When set, the integrity of the stream may have been affected.

Multiple flags may be set by a single operation.

If the operation sets an *internal state flag* that was registered with member *exceptions*, the function throws an exception of member type *failure*.

## Example

```
1 // istream::unget example
2 #include <iostream>           // std::cin, std::cout
3 #include <string>             // std::string
4
5 int main () {
6     std::cout << "Please, enter a number or a word: ";
7     char c = std::cin.get();
8
9     if ( (c >= '0') && (c <= '9') )
10    {
11        int n;
12        std::cin.unget();
13        std::cin >> n;
14        std::cout << "You entered a number: " << n << '\n';
15    }
16    else
17    {
18        std::string str;
19        std::cin.unget();
20        getline (std::cin,str);
21        std::cout << "You entered a word: " << str << '\n';
22    }
23    return 0;
24 }
```

Possible output:

```
Please, enter a number or a word: 7791
You entered a number: 7791
```

## Data races

Modifies the stream object.

Concurrent access to the same stream object may cause data races.

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type *failure* if the resulting *error state flag* is not *goodbit* and member *exceptions* was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting *badbit*. If *badbit* was set on the last call to *exceptions*, the function rethrows the caught exception.

## See also

<a href="#">istream::get</a>	Get characters (public member function )
<a href="#">istream::putback</a>	Put character back (public member function )

# /istream/wiostream

class

**std::wiostream**

<iostream> <iostream>

`typedef basic_iostream<wchar_t> wiostream;`

**Input/output stream (wide)**

[ios\\_base](#)

[wios](#)

[wistream](#)

[wiostream](#)

[wfstream](#)

## wostream

## wstringstream

This is an instantiation of `basic_iostream` with the following template parameters:

template parameter	definition	comments
<code>charT</code>	<code>wchar_t</code>	Aliased as member <code>char_type</code>
<code>traits</code>	<code>char_traits&lt;wchar_t&gt;</code>	Aliased as member <code>traits_type</code>

This class inherits all members from its two parent classes `wistream` and `wostream`, thus being able to perform both input and output operations.

The class relies on a single `wstreambuf` object for both the input and output operations.

Objects of these classes keep a set of internal fields inherited from `ios_base` and `wios`:

	field	member functions	description
Formatting	format flags	<code>flags</code> <code>setf</code> <code>unsetf</code>	A set of internal flags that affect how certain input/output operations are interpreted or generated. See member type <code>fmtflags</code> .
	field width	<code>width</code>	Width of the next formatted element to insert.
	display precision	<code>precision</code>	Decimal precision for the next floating-point value inserted.
	locale	<code>getloc</code> <code>imbue</code>	The <code>locale</code> object used by the function for formatted input/output operations affected by localization properties.
State	fill character	<code>fill</code>	Character to pad a formatted field up to the <i>field width</i> ( <code>width</code> ).
	error state	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	exception mask	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
Other	callback stack	<code>register_callback</code>	Stack of pointers to functions that are called when certain events occur.
	extensible arrays	<code>iword</code> <code>pword</code> <code>xalloc</code>	Internal arrays to store objects of type <code>long</code> and <code>void*</code> .
	tied stream	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	stream buffer	<code>rdbuf</code>	Pointer to the associated <code>wstreambuf</code> object, which is charge of all input/output operations.

## Member types

Member types `char_type`, `traits_type`, `int_type`, `pos_type` and `off_type` are ambiguous (multiple inheritance).

The class declares the following member types:

member type	definition
<code>char_type</code>	<code>wchar_t</code>
<code>traits_type</code>	<code>char_traits&lt;wchar_t&gt;</code>
<code>int_type</code>	<code>wint_t</code>
<code>pos_type</code>	<code>wstreampos</code>
<code>off_type</code>	<code>streamoff</code>

These member types inherited from its base classes (`wistream`, `wostream` and `ios_base`):

<code>event</code>	Type to indicate event type (public member type )
<code>event_callback</code>	Event callback function type (public member type )
<code>failure</code>	Base class for stream exceptions (public member class )
<code>fmtflags</code>	Type for stream format flags (public member type )
<code>Init</code>	Initialize standard stream objects (public member class )
<code>iostate</code>	Type for stream state flags (public member type )
<code>openmode</code>	Type for stream opening mode flags (public member type )
<code>seekdir</code>	Type for stream seeking direction flag (public member type )
<code>wistream::sentry</code>	Prepare stream for input (public member class )
<code>wostream::sentry</code>	Prepare stream for output (public member class )

## Public member functions

Note: This section links to the references for members of its basic template (`basic_iostream`).

<code>(constructor)</code>	Construct object (public member function )
<code>(destructor)</code>	Destroy object (public member function )

## Protected member functions

<code>operator=</code>	Move assignment (protected member function )
<code>swap</code>	Swap internals (protected member function )

## Public member functions inherited from `wistream`

Formatted input:

<code>operator&gt;&gt;</code>	Extract formatted input (public member function )
<code>gcount</code>	Get character count (public member function )
<code>get</code>	Get characters (public member function )
<code>getline</code>	Get line (public member function )
<code>ignore</code>	Extract and discard characters (public member function )

<b>peek</b>	Peek next character (public member function )
<b>read</b>	Read block of data (public member function )
<b>readsome</b>	Read data available in buffer (public member function )
<b>putback</b>	Put character back (public member function )
<b>unget</b>	Unget character (public member function )
<b>tellg</b>	Get position in input sequence (public member function )
<b>seekg</b>	Set position in input sequence (public member function )
<b>sync</b>	Synchronize input buffer (public member function )

#### Public member functions inherited from **wostream**

Formatted input:

<b>operator&lt;&lt;</b>	Insert formatted output (public member function )
<b>put</b>	Put character (public member function )
<b>write</b>	Write block of data (public member function )
<b>tellp</b>	Get position in output sequence (public member function )
<b>seekp</b>	Set position in output sequence (public member function )
<b>flush</b>	Flush output stream buffer (public member function )

#### Public member functions inherited from **wios**

<b>good</b>	Check whether state of stream is good (public member function )
<b>eof</b>	Check whether eofbit is set (public member function )
<b>fail</b>	Check whether failbit or badbit is set (public member function )
<b>bad</b>	Check whether badbit is set (public member function )
<b>operator!</b>	Evaluate stream (not) (public member function )
<b>operator bool</b>	Evaluate stream (public member function )
<b>rdstate</b>	Get error state flags (public member function )
<b>setstate</b>	Set error state flag (public member function )
<b>clear</b>	Set error state flags (public member function )
<b>copyfmt</b>	Copy formatting information (public member function )
<b>fill</b>	Get/set fill character (public member function )
<b>exceptions</b>	Get/set exceptions mask (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>tie</b>	Get/set tied stream (public member function )
<b>rdbuf</b>	Get/set stream buffer (public member function )
<b>narrow</b>	Narrow character (public member function )
<b>widen</b>	Widen character (public member function )

#### Public member functions inherited from **ios\_base**

<b>flags</b>	Get/set format flags (public member function )
<b>setf</b>	Set specific format flags (public member function )
<b>unsetf</b>	Clear specific format flags (public member function )
<b>precision</b>	Get/Set floating-point decimal precision (public member function )
<b>width</b>	Get/set field width (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>getloc</b>	Get current locale (public member function )
<b>xalloc</b>	Get new index for extensible array [static] (public static member function )
<b>iword</b>	Get integer element of extensible array (public member function )
<b>pword</b>	Get pointer element of extensible array (public member function )
<b>register_callback</b>	Register event callback function (public member function )
<b>sync_with_stdio</b>	Toggle synchronization with cstdio streams [static] (public static member function )

## /istream/wistream

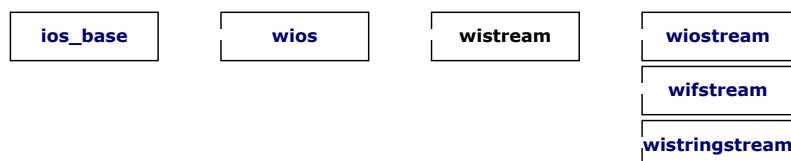
class

**std::wistream**

<iostream> <iostream>

**typedef basic\_istream<wchar\_t> wistream;**

**Input stream (wide)**



Input stream objects can read and interpret input from sequences of characters. Specific members are provided to perform these input operations (see [functions](#) below).

The standard object `wcin` is an object of this type.

This is an instantiation of `basic_istream` with the following template parameters:

template parameter	definition	comments
<code>charT</code>	<code>wchar_t</code>	Aliased as member <code>char_type</code>
<code>traits</code>	<code>char_traits&lt;wchar_t&gt;</code>	Aliased as member <code>traits_type</code>

Objects of these classes keep a set of internal fields inherited from `ios_base` and `wios`:

	field	member functions	description
Formatting	format flags	<code>flags</code> <code>setf</code> <code>unsetf</code>	A set of internal flags that affect how certain input/output operations are interpreted or generated. See member type <code>fmtflags</code> .
	field width	<code>width</code>	Width of the next formatted element to insert.
	display precision	<code>precision</code>	Decimal precision for the next floating-point value inserted.
	locale	<code>getloc</code> <code>imbue</code>	The <code>locale</code> object used by the function for formatted input/output operations affected by localization properties.
	fill character	<code>fill</code>	Character to pad a formatted field up to the <i>field width</i> ( <code>width</code> ).
State	error state	<code>rdstate</code> <code>setstate</code> <code>clear</code>	The current error state of the stream. Individual values may be obtained by calling <code>good</code> , <code>eof</code> , <code>fail</code> and <code>bad</code> . See member type <code>iostate</code> .
	exception mask	<code>exceptions</code>	The state flags for which a <code>failure</code> exception is thrown. See member type <code>iostate</code> .
Other	callback stack	<code>register_callback</code>	Stack of pointers to functions that are called when certain events occur.
	extensible arrays	<code>iword</code> <code>pword</code> <code>xalloc</code>	Internal arrays to store objects of type <code>long</code> and <code>void*</code> .
	tied stream	<code>tie</code>	Pointer to output stream that is flushed before each i/o operation on this stream.
	stream buffer	<code>rdbuf</code>	Pointer to the associated <code>wstreambuf</code> object, which is charge of all input/output operations.

## Member types

The class contains the following member class:

<b>sentry</b>	Prepare stream for input (public member class )
---------------	---

Along with the following member types:

member type	definition
<code>char_type</code>	<code>wchar_t</code>
<code>traits_type</code>	<code>char_traits&lt;wchar_t&gt;</code>
<code>int_type</code>	<code>wint_t</code>
<code>pos_type</code>	<code>wstreampos</code>
<code>off_type</code>	<code>streamoff</code>

And these member types inherited from `ios_base` through `wios`:

<b>event</b>	Type to indicate event type (public member type )
<b>event_callback</b>	Event callback function type (public member type )
<b>failure</b>	Base class for stream exceptions (public member class )
<b>fmtflags</b>	Type for stream format flags (public member type )
<b>Init</b>	Initialize standard stream objects (public member class )
<b>iostate</b>	Type for stream state flags (public member type )
<b>openmode</b>	Type for stream opening mode flags (public member type )
<b>seekdir</b>	Type for stream seeking direction flag (public member type )

## Public member functions

Note: This section links to the references for members of its basic template (`basic_istream`).

<b>(constructor)</b>	Construct object (public member function )
<b>(destructor)</b>	Destroy object (public member function )

**Formatted input:**

<b>operator&gt;&gt;</b>	Extract formatted input (public member function )
-------------------------	---

**Unformatted input:**

<b>gcount</b>	Get character count (public member function )
<b>get</b>	Get characters (public member function )
<b>getline</b>	Get line (public member function )
<b>ignore</b>	Extract and discard characters (public member function )
<b>peek</b>	Peek next character (public member function )
<b>read</b>	Read block of data (public member function )
<b>readsome</b>	Read data available in buffer (public member function )
<b>putback</b>	Put character back (public member function )
<b>unget</b>	Unget character (public member function )

**Positioning:**

<b>tellg</b>	Get position in input sequence (public member function )
--------------	--

<b>seekg</b>	Set position in input sequence (public member function )
--------------	--

**Synchronization:**

<b>sync</b>	Synchronize input buffer (public member function )
-------------	--

**Protected member functions**

<b>operator=</b>	Move assignment (protected member function )
<b>swap</b>	Swap internals (protected member function )

**Public member functions inherited from wios**

<b>good</b>	Check whether state of stream is good (public member function )
<b>eof</b>	Check whether eofbit is set (public member function )
<b>fail</b>	Check whether failbit or badbit is set (public member function )
<b>bad</b>	Check whether badbit is set (public member function )
<b>operator!</b>	Evaluate stream (not) (public member function )
<b>operator bool</b>	Evaluate stream (public member function )
<b>rdstate</b>	Get error state flags (public member function )
<b>setstate</b>	Set error state flag (public member function )
<b>clear</b>	Set error state flags (public member function )
<b>copyfmt</b>	Copy formatting information (public member function )
<b>fill</b>	Get/set fill character (public member function )
<b>exceptions</b>	Get/set exceptions mask (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>tie</b>	Get/set tied stream (public member function )
<b>rdbuf</b>	Get/set stream buffer (public member function )
<b>narrow</b>	Narrow character (public member function )
<b>widen</b>	Widen character (public member function )

**Public member functions inherited from ios\_base**

<b>flags</b>	Get/set format flags (public member function )
<b>setf</b>	Set specific format flags (public member function )
<b>unsetf</b>	Clear specific format flags (public member function )
<b>precision</b>	Get/Set floating-point decimal precision (public member function )
<b>width</b>	Get/set field width (public member function )
<b>imbue</b>	Imbue locale (public member function )
<b>getloc</b>	Get current locale (public member function )
<b>xalloc</b>	Get new index for extensible array [static] (public static member function )
<b>iword</b>	Get integer element of extensible array (public member function )
<b>pword</b>	Get pointer element of extensible array (public member function )
<b>register_callback</b>	Register event callback function (public member function )
<b>sync_with_stdio</b>	Toggle synchronization with cstdio streams [static] (public static member function )

## /istream/ws

function

**std::WS**

<iostream> <iostream>

```
for istream is : istream& ws (istream& is);
basic template template <class charT, class traits>
basic_istream<charT,traits>& ws (basic_istream<charT,traits>& is);
```

**Extract whitespaces**

Extracts as many whitespace characters as possible from the current position in the input sequence. The extraction stops as soon as a non-whitespace character is found. These extracted whitespace characters are discarded.

Notice that `basic_istream` objects have the `skipws` flag set by default: This applies a similar effect before the formatted extraction operations (see `operator>>`).

No specifications on the internal operations performed by this function.

Internally, the function accesses the input sequence `is` by first constructing a `sentry` object (with `noskipws` set to `true`). Then (if `good`), it extracts characters from `is`'s associated `stream buffer` object as if calling its member functions `sbumpc` or `sgetc`, and finally destroys the `sentry` object before returning.

Calling this function does not alter the value returned by `gcount`.

**Parameters**

is

Input stream object from where whitespaces are extracted.

Because this function is a manipulator, it is designed to be used alone with no arguments in conjunction with the `extraction (>>)` operations on input streams (see example below).

## Return Value

Argument *is*.

Errors are signaled by modifying the *internal state flags* of *is*:

flag	error
<code>eofbit</code>	The function stopped extracting characters because the input sequence has no more characters available ( <i>end-of-file</i> reached).
<code>failbit</code>	The <i>stream state</i> of <i>is</i> was not <i>good</i> before the call (applies to C++11 and other implementations constructing a <i>sentry</i> object)
<code>badbit</code>	Error on stream (such as when this function catches an exception thrown by an internal operation). When set, the integrity of the stream may have been affected.

Multiple flags may be set on *is* by a single operation.

If the operation sets an *internal state flag* of *is* that was registered using its member `exceptions`, the function throws an exception of member type `failure`.

## Example

```
1 // ws manipulator example
2 #include <iostream>      // std::cout, std::noskipws
3 #include <sstream>       // std::istringstream, std::ws
4
5 int main () {
6     char a[10], b[10];
7
8     std::istringstream iss ("one \n \t two");
9     iss >> std::noskipws;
10    iss >> a >> std::ws >> b;
11    std::cout << a << ", " << b << '\n';
12
13    return 0;
14 }
```

Output:

```
one, two
```

## Data races

Modifies the stream object *is*.

Concurrent access to the same stream object may cause data races, except for the standard stream objects `cin` and `wcin` when these are *synchronized with stdio* (in this case, no data races are initiated, although no guarantees are given on the order in which extracted characters are attributed to threads).

## Exception safety

**Basic guarantee:** if an exception is thrown, the object is in a valid state.

It throws an exception of member type `failure` if the resulting *error state flag* of *is* is not `goodbit` and its member `exceptions` was set to throw for that state. Any exception thrown by an internal operation is caught and handled by the function, setting *is*'s `badbit` flag. If `badbit` was set on the last call to `exceptions` for *is*, the function rethrows the caught exception.

## See also

<code>skipws</code>	Skip whitespaces (function )
<code>noskipws</code>	Do not skip whitespaces (function )

# /map

header

## <map>

### Map header

Header that defines the `map` and `multimap` container classes:

## Classes

<code>map</code>	Map (class template )
<code>multimap</code>	Multiple-key map (class template )

## Functions

<code>begin</code>	Iterator to beginning (function template )
<code>end</code>	Iterator to end (function template )

# /map/map

class template

## `std::map`

```
template < class Key,
          class T,
          class Compare = less<Key>,
          class Allocator = allocator<pair<const Key, T> >      // map::key_type
          > class map;                                              // map::mapped_type
                                                               // map::key_compare
                                                               // map::allocator_type
```

<map>

## Map

Maps are associative containers that store elements formed by a *key value* and a *mapped value*, following a specific order.

In a `map`, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type `value_type`, which is a `pair` type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a `map` are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal `comparison object` (of type `Compare`).

`map` containers are generally slower than `unordered_map` containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

The mapped values in a `map` can be accessed directly by their corresponding key using the *bracket operator* (`(operator[])`).

Maps are typically implemented as *binary search trees*.

## Container properties

### Associative

Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

### Ordered

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

### Map

Each element associates a *key* to a *mapped value*: Keys are meant to identify the elements whose main content is the *mapped value*.

### Unique keys

No two elements in the container can have equivalent *keys*.

### Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

## Template parameters

### Key

Type of the *keys*. Each element in a `map` is uniquely identified by its *key value*.

Aliased as member type `map::key_type`.

### T

Type of the mapped value. Each element in a `map` stores some data as its mapped value.

Aliased as member type `map::mapped_type`.

### Compare

A binary predicate that takes two element keys as arguments and returns a `bool`. The expression `comp(a,b)`, where `comp` is an object of this type and `a` and `b` are key values, shall return `true` if `a` is considered to go before `b` in the *strict weak ordering* the function defines.

The `map` object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent (by comparing them reflexively: they are equivalent if `!comp(a,b) && !comp(b,a)`). No two elements in a `map` container can have equivalent keys.

This can be a function pointer or a function object (see `constructor` for an example). This defaults to `less<T>`, which returns the same as applying the *less-than operator* (`a < b`).

Aliased as member type `map::key_compare`.

### Alloc

Type of the allocator object used to define the storage allocation model. By default, the `allocator` class template is used, which defines the simplest memory allocation model and is value-independent.

Aliased as member type `map::allocator_type`.

## Member types

member type	definition	notes
<code>key_type</code>	The first template parameter ( <code>Key</code> )	
<code>mapped_type</code>	The second template parameter ( <code>T</code> )	
<code>value_type</code>	<code>pair&lt;const key_type,mapped_type&gt;</code>	
<code>key_compare</code>	The third template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;key_type&gt;</code>
<code>value_compare</code>	<i>Nested function class to compare elements</i>	see <code>value_comp</code>
<code>allocator_type</code>	The fourth template parameter ( <code>Alloc</code> )	defaults to: <code>allocator&lt;value_type&gt;</code>
<code>reference</code>	<code>allocator_type::reference</code>	for the default allocator: <code>value_type&amp;</code>
<code>const_reference</code>	<code>allocator_type::const_reference</code>	for the default allocator: <code>const value_type&amp;</code>
<code>pointer</code>	<code>allocator_type::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_type::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a bidirectional <code>iterator</code> to <code>value_type</code>	convertible to <code>const_iterator</code>
<code>const_iterator</code>	a bidirectional <code>iterator</code> to <code>const value_type</code>	
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>
member type	definition	notes
<code>key_type</code>	The first template parameter ( <code>Key</code> )	
<code>mapped_type</code>	The second template parameter ( <code>T</code> )	
<code>value_type</code>	<code>pair&lt;const key_type,mapped_type&gt;</code>	
<code>key_compare</code>	The third template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;key_type&gt;</code>
<code>value_compare</code>	<i>Nested function class to compare elements</i>	see <code>value_comp</code>
<code>allocator_type</code>	The fourth template parameter ( <code>Alloc</code> )	defaults to: <code>allocator&lt;value_type&gt;</code>

<code>reference</code>	<code>value_type&amp;</code>	
<code>const_reference</code>	<code>const value_type&amp;</code>	
<code>pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a bidirectional iterator to <code>value_type</code>	convertible to <code>const_iterator</code>
<code>const_iterator</code>	a bidirectional iterator to <code>const value_type</code>	
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>

## Member functions

<b>(constructor)</b>	Construct map (public member function )
<b>(destructor)</b>	Map destructor (public member function )
<b>operator=</b>	Copy container content (public member function )

### Iterators:

<code>begin</code>	Return iterator to beginning (public member function )
<code>end</code>	Return iterator to end (public member function )
<code>rbegin</code>	Return reverse iterator to reverse beginning (public member function )
<code>rend</code>	Return reverse iterator to reverse end (public member function )
<code>cbegin</code>	Return <code>const_iterator</code> to beginning (public member function )
<code>cend</code>	Return <code>const_iterator</code> to end (public member function )
<code>crbegin</code>	Return <code>const_reverse_iterator</code> to reverse beginning (public member function )
<code>crend</code>	Return <code>const_reverse_iterator</code> to reverse end (public member function )

### Capacity:

<code>empty</code>	Test whether container is empty (public member function )
<code>size</code>	Return container size (public member function )
<code>max_size</code>	Return maximum size (public member function )

### Element access:

<b>operator[]</b>	Access element (public member function )
<b>at</b>	Access element (public member function )

### Modifiers:

<code>insert</code>	Insert elements (public member function )
<code>erase</code>	Erase elements (public member function )
<code>swap</code>	Swap content (public member function )
<code>clear</code>	Clear content (public member function )
<code>emplace</code>	Construct and insert element (public member function )
<code>emplace_hint</code>	Construct and insert element with hint (public member function )

### Observers:

<code>key_comp</code>	Return key comparison object (public member function )
<code>value_comp</code>	Return value comparison object (public member function )

### Operations:

<code>find</code>	Get iterator to element (public member function )
<code>count</code>	Count elements with a specific key (public member function )
<code>lower_bound</code>	Return iterator to lower bound (public member function )
<code>upper_bound</code>	Return iterator to upper bound (public member function )
<code>equal_range</code>	Get range of equal elements (public member function )

### Allocator:

<b>get_allocator</b>	Get allocator (public member function )
----------------------	---

## /map/map/at

public member function

### std::map::at

<map>

```
mapped_type& at (const key_type& k);
const mapped_type& at (const key_type& k) const;
```

### Access element

Returns a reference to the mapped value of the element identified with key *k*.

If *k* does not match the key of any element in the container, the function throws an `out_of_range` exception.

## Parameters

---

k

Key value of the element whose mapped value is accessed.  
Member type `key_type` is the type of the keys for the elements in the container, defined in `map` as an alias of its first template parameter (`key`).

## Return value

---

A reference to the mapped value of the element with a key value equivalent to k.

If the `map` object is const-qualified, the function returns a reference to `const mapped_type`. Otherwise, it returns a reference to `mapped_type`.

Member type `mapped_type` is the type to the mapped values in the container (see [map member types](#)). In `map` this is an alias of its second template parameter (`T`).

## Example

---

```
1 // map::at
2 #include <iostream>
3 #include <string>
4 #include <map>
5
6 int main ()
7 {
8     std::map<std::string,int> mymap = {
9         { "alpha", 0 },
10        { "beta", 0 },
11        { "gamma", 0 } };
12
13    mymap.at("alpha") = 10;
14    mymap.at("beta") = 20;
15    mymap.at("gamma") = 30;
16
17    for (auto& x: mymap) {
18        std::cout << x.first << ":" << x.second << '\n';
19    }
20
21    return 0;
22 }
```

Possible output:

```
alpha: 10
beta: 20
gamma: 30
```

## Complexity

---

Logarithmic in `size`.

## Iterator validity

---

No changes.

## Data races

---

The container is accessed (neither the const nor the non-const versions modify the container).

The mapped value that is accessed may be modified by the caller. Concurrently accessing or modifying other elements is safe.

## Exception safety

---

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

It throws `out_of_range` if k is not the key of an element in the `map`.

## See also

---

<a href="#">map::operator[]</a>	Access element (public member function )
<a href="#">map::find</a>	Get iterator to element (public member function )

# /map/map/begin

public member function

## std::map::begin

<map>

```
    iterator begin();
const_iterator begin() const;
    iterator begin() noexcept;
const_iterator begin() const noexcept;
```

### Return iterator to beginning

Returns an iterator referring to the first element in the `map` container.

Because `map` containers keep their elements ordered at all times, `begin` points to the element that goes first following the container's `sorting` criterion.

If the container is `empty`, the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

An iterator to the first element in the container.

If the `map` object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements (of type `value_type`). Notice that `value_type` in `map` containers is an alias of `pair<const key_type, mapped_type>`.

## Example

```
1 // map::begin/end
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['b'] = 100;
10    mymap['a'] = 200;
11    mymap['c'] = 300;
12
13 // show content:
14 for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
15     std::cout << it->first << " => " << it->second << '\n';
16
17 return 0;
18 }
```

Output:

```
a => 200
b => 100
c => 300
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>map::end</code>	Return iterator to end ( <a href="#">public member function</a> )
<code>map::rbegin</code>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<code>map::rend</code>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )

# /map/map/cbegin

public member function

## std::map::cbegin

`<map>`

```
const_iterator cbegin() const noexcept;
```

### Return const\_iterator to beginning

Returns a `const_iterator` pointing to the first element in the container.

A `const_iterator` is an iterator that points to `const` content. This iterator can be increased and decreased (unless it is itself `const`), just like the `iterator` returned by `map::begin`, but it cannot be used to modify the contents it points to, even if the `map` object is not itself `const`.

If the container is `empty`, the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

A `const_iterator` to the beginning of the sequence.

Member type `const_iterator` is a [bidirectional iterator](#) type that points to const elements (of type `const value_type`). Notice that `value_type` in `map` containers is an alias of `pair<const key_type, mapped_type>`.

## Example

```
1 // map::cbegin/cend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['b'] = 100;
10    mymap['a'] = 200;
11    mymap['c'] = 300;
12
13    // print content:
14    std::cout << "mymap contains:";
15    for (auto it = mymap.cbegin(); it != mymap.cend(); ++it)
16        std::cout << "[" << (*it).first << ':' << (*it).second << ']';
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
mymap contains: [a:200] [b:100] [c:300]
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>map::cend</code>	Return <code>const_iterator</code> to end ( <a href="#">public member function</a> )
<code>map::begin</code>	Return iterator to beginning ( <a href="#">public member function</a> )
<code>map::crbegin</code>	Return <code>const_reverse_iterator</code> to reverse beginning ( <a href="#">public member function</a> )

# /map/map/cend

public member function

## std::map::cend

`<map>`

```
const_iterator cend() const noexcept;
```

### Return `const_iterator` to end

Returns a `const_iterator` pointing to the *past-the-end* element in the container.

A `const_iterator` is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also `const`), just like the iterator returned by `map::end`, but it cannot be used to modify the contents it points to, even if the `map` object is not itself `const`.

If the container is `empty`, this function returns the same as `map::cbegin`.

The value returned shall not be dereferenced.

## Parameters

none

## Return Value

A `const_iterator` to the element past the end of the sequence.

Member type `const_iterator` is a [bidirectional iterator](#) type that points to const elements.

## Example

```
1 // map::cbegin/cend
2 #include <iostream>
3 #include <map>
```

```

4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['b'] = 100;
10    mymap['a'] = 200;
11    mymap['c'] = 300;
12
13 // print content:
14 std::cout << "mymap contains:";
15 for (auto it = mymap.cbegin(); it != mymap.cend(); ++it)
16     std::cout << "[" << (*it).first << ":" << (*it).second << "]";
17 std::cout << '\n';
18
19 return 0;
20 }

```

Output:

mymap contains: [a:200] [b:100] [c:300]
---

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">map::end</a>	Return iterator to end (public member function )
<a href="#">map::cbegin</a>	Return const_iterator to beginning (public member function )

# /map/map/clear

public member function

## std::map::clear

<map>

void clear();
void clear() noexcept;

### Clear content

Removes all elements from the `map` container (which are destroyed), leaving the container with a `size` of 0.

## Parameters

none

## Return value

none

## Example

```

1 // map::clear
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['x']=100;
10    mymap['y']=200;
11    mymap['z']=300;
12
13    std::cout << "mymap contains:\n";
14    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
15        std::cout << it->first << " => " << it->second << "\n";
16
17    mymap.clear();
18    mymap['a']=1101;
19    mymap['b']=2202;
20
21    std::cout << "mymap contains:\n";
22    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)

```

```

23     std::cout << it->first << " => " << it->second << '\n';
24
25     return 0;
26 }
```

Output:

```

mymap contains:
x => 100
y => 200
z => 300
mymap contains:
a => 1101
b => 2202
```

## Complexity

Linear in `size` (destructions).

## Iterator validity

All iterators, pointers and references related to this container are invalidated.

## Data races

The container is modified.

All contained elements are modified.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>map::erase</code>	Erase elements ( <a href="#">public member function</a> )
<code>map::size</code>	Return container size ( <a href="#">public member function</a> )
<code>map::empty</code>	Test whether container is empty ( <a href="#">public member function</a> )

# /map/map/count

public member function

## std::map::count

<map>

```
size_type count (const key_type& k) const;
```

### Count elements with a specific key

Searches the container for elements with a key equivalent to `k` and returns the number of matches.

Because all elements in a `map` container are unique, the function can only return 1 (if the element is found) or zero (otherwise).

Two `keys` are considered equivalent if the container's `comparison object` returns `false` reflexively (i.e., no matter the order in which the keys are passed as arguments).

## Parameters

`k`

Key to search for.

Member type `key_type` is the type of the element keys in the container, defined in `map` as an alias of its first template parameter (`key`).

## Return value

1 if the container contains an element whose key is equivalent to `k`, or zero otherwise.

Member type `size_type` is an unsigned integral type.

## Example

```

1 // map::count
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     char c;
9
10    mymap [ 'a']=101;
11    mymap [ 'c']=202;
12    mymap [ 'f']=303;
13
14    for (c='a'; c<'h'; c++)
15    {
16        std::cout << c;
17        if (mymap.count(c)>0)
18            std::cout << " is an element of mymap.\n";
19        else
20            std::cout << " is not an element of mymap.\n";
```

```
21 }
22 return 0;
23 }
```

Output:

```
a is an element of mymap.
b is not an element of mymap.
c is an element of mymap.
d is not an element of mymap.
e is not an element of mymap.
f is an element of mymap.
g is not an element of mymap.
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed.

No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>map::find</code>	Get iterator to element (public member function )
<code>map::size</code>	Return container size (public member function )
<code>map::equal_range</code>	Get range of equal elements (public member function )

## /map/map/crbegin

public member function

### std::map::crbegin

<map>

```
const_reverse_iterator crbegin() const noexcept;
```

#### Return `const_reverse_iterator` to reverse beginning

Returns a `const_reverse_iterator` pointing to the last element in the container (i.e., its *reverse beginning*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse beginning* of the sequence.

Member type `const_reverse_iterator` is a reverse `bidirectional iterator` type that points to a `const` element (see [map member types](#)).

## Example

```
1 // map::crbegin/crend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['b'] = 100;
10    mymap['a'] = 200;
11    mymap['c'] = 300;
12
13    std::cout << "mymap backwards:" ;
14    for (auto rit = mymap.crbegin(); rit != mymap.crend(); ++rit)
15        std::cout << "[" << rit->first << ':' << rit->second << ']';
16    std::cout << '\n';
17
18    return 0;
19 }
```

Output:

```
mymap backwards: [c:300] [b:100] [a:200]
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">map::crend</a>	Return const_reverse_iterator to reverse end ( <a href="#">public member function</a> )
<a href="#">map::begin</a>	Return iterator to beginning ( <a href="#">public member function</a> )
<a href="#">map::rbegin</a>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )

# /map/map/crend

public member function

## std::map::crend

<map>

`const_reverse_iterator crend() const noexcept;`

### Return const\_reverse\_iterator to reverse end

Returns a `const_reverse_iterator` pointing to the theoretical element preceding the first element in the container (which is considered its *reverse end*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse end* of the sequence.

Member type `const_reverse_iterator` is a reverse `bidirectional iterator` type that points to a `const` element (see [map member types](#)).

## Example

```
1 // map::crbegin/crend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['b'] = 100;
10    mymap['a'] = 200;
11    mymap['c'] = 300;
12
13    std::cout << "mymap backwards:" ;
14    for (auto rit = mymap.crbegin(); rit != mymap.crend(); ++rit)
15        std::cout << " [" << rit->first << ':' << rit->second << ']';
16    std::cout << '\n';
17
18    return 0;
19 }
```

Output:

```
mymap backwards: [c:300] [b:100] [a:200]
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">map::end</a>	Return iterator to end (public member function )
<a href="#">map::crbegin</a>	Return const_reverse_iterator to reverse beginning (public member function )
<a href="#">map::rend</a>	Return reverse iterator to reverse end (public member function )

## /map/map/emplace

public member function

### std::map::emplace

<map>

```
template <class... Args>
pair<iterator,bool> emplace (Args&&... args);
```

#### Construct and insert element

Inserts a new element in the [map](#) if its key is unique. This new element is constructed in place using [args](#) as the arguments for the construction of a [value\\_type](#) (which is an object of a [pair](#) type).

The insertion only takes place if no other element in the container has a key equivalent to the one being emplaced (keys in a [map](#) container are unique).

If inserted, this effectively increases the container [size](#) by one.

Internally, [map](#) containers keep all their elements sorted by their key following the criterion specified by its [comparison object](#). The element is always inserted in its respective position following this ordering.

The element is constructed in-place by calling [allocator\\_traits::construct](#) with [args](#) forwarded.

A similar member function exists, [insert](#), which either copies or moves existing objects into the container.

## Parameters

[args](#)

Arguments used to construct a new object of the *mapped type* for the inserted element.

Arguments forwarded to construct the new element (of type [pair<const key\\_type, mapped\\_type>](#)).

This can be one of:

- Two arguments: one for the *key*, the other for the *mapped value*.
- A single argument of a [pair](#) type with a value for the *key* as first member, and a value for the *mapped value* as second.
- [piecewise\\_construct](#) as first argument, and two additional arguments with [tuples](#) to be forwarded as arguments for the *key value* and for the *mapped value* respectively.

See [pair::pair](#) for more info.

## Return value

If the function successfully inserts the element (because no equivalent element existed already in the [map](#)), the function returns a [pair](#) of an iterator to the newly inserted element and a value of [true](#).

Otherwise, it returns an iterator to the equivalent element within the container and a value of [false](#).

Member type [iterator](#) is a [bidirectional iterator](#) type that points to an element.

[pair](#) is a class template declared in [<utility>](#) (see [pair](#)).

## Example

```
1 // map::emplace
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap.emplace('x',100);
10    mymap.emplace('y',200);
11    mymap.emplace('z',100);
12
13    std::cout << "mymap contains:" ;
14    for (auto& x: mymap)
15        std::cout << " [" << x.first << ':' << x.second << "] ";
16    std::cout << '\n';
17
18    return 0;
19 }
```

Output:

```
mymap contains: [x:100] [y:200] [z:100]
```

## Complexity

Logarithmic in the container size.

## Iterator validity

No changes.

## Data races

The container is modified.  
Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
If allocator\_traits::construct is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<a href="#">map::emplace_hint</a>	Construct and insert element with hint (public member function )
<a href="#">map::insert</a>	Insert elements (public member function )
<a href="#">map::erase</a>	Erase elements (public member function )

# /map/map/emplace\_hint

public member function

## std::map::emplace\_hint

<map>

```
template <class... Args>
iterator emplace_hint (const_iterator position, Args&&... args);
```

### Construct and insert element with hint

Inserts a new element in the `map` if its key is unique, with a hint on the insertion *position*. This new element is constructed in place using *args* as the arguments for the construction of a `value_type` (which is an object of a `pair` type).

The insertion only takes place if no other element in the container has a key equivalent to the one being emplaced (elements in a `map` container are unique).

If inserted, this effectively increases the container `size` by one.

The value in *position* is used as a hint on the insertion point. The element will nevertheless be inserted at its corresponding position following the order described by its internal `comparison object`, but this hint is used by the function to begin its search for the insertion point, speeding up the process considerably when the actual insertion point is either *position* or close to it.

The element is constructed in-place by calling `allocator_traits::construct` with *args* forwarded.

## Parameters

position

Hint for the position where the element can be inserted.

The function optimizes its insertion time if *position* points to the element that will follow the inserted element (or to the `end`, if it would be the last).

Notice that this does not force the new element to be in that position within the `map` container (the elements in a `map` always follow a specific order).

`const_iterator` is a member type, defined as a `bidirectional iterator` type that points to elements.

args

Arguments used to construct a new object of the `mapped type` for the inserted element.

Arguments forwarded to construct the new element (of type `pair<const key_type, mapped_type>`).

This can be one of:

- Two arguments: one for the `key`, the other for the `mapped value`.

- A single argument of a `pair` type with a value for the `key` as first member, and a value for the `mapped value` as second.

- `piecewise_construct` as first argument, and two additional arguments with `tuples` to be forwarded as arguments for the `key value` and for the `mapped value` respectively.

See `pair::pair` for more info.

## Return value

If the function successfully inserts the element (because no equivalent element existed already in the `map`), the function returns an iterator to the newly inserted element.

Otherwise, it returns an iterator to the equivalent element within the container.

Member type `iterator` is a `bidirectional iterator` type that points to an element.

## Example

```
1 // map::emplace_hint
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     auto it = mymap.end();
9
10    it = mymap.emplace_hint(it,'b',10);
11    mymap.emplace_hint(it,'a',12);
12    mymap.emplace_hint(mymap.end(),'c',14);
13
14    std::cout << "mymap contains:" ;
15    for (auto& x: mymap)
16        std::cout << "[" << x.first << ':' << x.second << ']';
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
mymap contains: [a:12] [b:10] [c:14]
```

## Complexity

Generally, logarithmic in the container `size`.  
Amortized constant if the insertion point for the element is `position`.

## Iterator validity

No changes.

## Data races

The container is modified.  
Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
If `allocator_traits::construct` is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<code>map::emplace</code>	Construct and insert element (public member function )
<code>map::insert</code>	Insert elements (public member function )
<code>map::erase</code>	Erase elements (public member function )

# /map/map/empty

public member function

## std::map::empty

<map>

```
bool empty() const;
bool empty() const noexcept;
```

### Test whether container is empty

Returns whether the `map` container is empty (i.e. whether its `size` is 0).

This function does not modify the container in any way. To clear the content of a `map` container, see `map::clear`.

## Parameters

none

## Return Value

true if the container `size` is 0, false otherwise.

## Example

```
1 // map::empty
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['a']=10;
10    mymap['b']=20;
11    mymap['c']=30;
12
13    while (!mymap.empty())
14    {
15        std::cout << mymap.begin()->first << " => " << mymap.begin()->second << '\n';
16        mymap.erase(mymap.begin());
17    }
18
19    return 0;
20 }
```

Output:

```
a => 10
b => 20
c => 30
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">map::clear</a>	Clear content (public member function )
<a href="#">map::erase</a>	Erase elements (public member function )
<a href="#">map::size</a>	Return container size (public member function )

# /map/map/end

public member function

## std::map::end

<map>

```
iterator end();
const_iterator end() const;
iterator end() noexcept;
const_iterator end() const noexcept;
```

### Return iterator to end

Returns an iterator referring to the *past-the-end* element in the [map](#) container.

The *past-the-end* element is the theoretical element that would follow the last element in the [map](#) container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with [map::begin](#) to specify a range including all the elements in the container.

If the container is [empty](#), this function returns the same as [map::begin](#).

## Parameters

none

## Return Value

An iterator to the *past-the-end* element in the container.

If the [map](#) object is const-qualified, the function returns a [const\\_iterator](#). Otherwise, it returns an [iterator](#).

Member types [iterator](#) and [const\\_iterator](#) are [bidirectional iterator](#) types pointing to elements.

## Example

```
1 // map::begin/end
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['b'] = 100;
10    mymap['a'] = 200;
11    mymap['c'] = 300;
12
13 // show content:
14 for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
15     std::cout << it->first << " => " << it->second << '\n';
16
17 return 0;
18 }
```

Output:

```
a => 200
b => 100
c => 300
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">map::begin</a>	Return iterator to beginning (public member function )
<a href="#">map::rbegin</a>	Return reverse iterator to reverse beginning (public member function )
<a href="#">map::rend</a>	Return reverse iterator to reverse end (public member function )

# /map/map/equal\_range

public member function

## std::map::equal\_range

<map>

```
pair<const_iterator,const_iterator> equal_range (const key_type& k) const;
pair<iterator,iterator> equal_range (const key_type& k);
```

### Get range of equal elements

Returns the bounds of a range that includes all the elements in the container which have a *key* equivalent to *k*.

Because the elements in a [map](#) container have unique keys, the range returned will contain a single element at most.

If no matches are found, the range returned has a length of zero, with both iterators pointing to the first element that has a key considered to go after *k* according to the container's [internal comparison object](#) (*key\_comp*).

Two *keys* are considered equivalent if the container's [comparison object](#) returns `false` reflexively (i.e., no matter the order in which the keys are passed as arguments).

## Parameters

*k*

Key to search for.

Member type *key\_type* is the type of the elements in the container, defined in [map](#) as an alias of its first template parameter (*key*).

## Return value

The function returns a [pair](#), whose member *pair::first* is the lower bound of the range (the same as *lower\_bound*), and *pair::second* is the upper bound (the same as *upper\_bound*).

If the [map](#) object is *const*-qualified, the function returns a [pair](#) of *const\_iterator*. Otherwise, it returns a [pair](#) of *iterator*.

Member types *iterator* and *const\_iterator* are [bidirectional iterator](#) types pointing to elements (of type *value\_type*). Notice that *value\_type* in [map](#) containers is itself also a [pair](#) type: *pair<const key\_type, mapped\_type>*.

## Example

```
1 // map::equal_range
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['a']=10;
10    mymap['b']=20;
11    mymap['c']=30;
12
13    std::pair<std::map<char,int>::iterator,std::map<char,int>::iterator> ret;
14    ret = mymap.equal_range('b');
15
16    std::cout << "lower bound points to: ";
17    std::cout << ret.first->first << " => " << ret.first->second << '\n';
18
19    std::cout << "upper bound points to: ";
20    std::cout << ret.second->first << " => " << ret.second->second << '\n';
21
22    return 0;
23 }
```

```
lower bound points to: 'b' => 20
upper bound points to: 'c' => 30
```

## Complexity

Logarithmic in *size*.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<b>map::count</b>	Count elements with a specific key ( <a href="#">public member function</a> )
<b>map::lower_bound</b>	Return iterator to lower bound ( <a href="#">public member function</a> )
<b>map::upper_bound</b>	Return iterator to upper bound ( <a href="#">public member function</a> )
<b>map::operator[]</b>	Access element ( <a href="#">public member function</a> )
<b>map::find</b>	Get iterator to element ( <a href="#">public member function</a> )

# /map/map/erase

public member function

## std::map::erase

<map>

```
(1)     void erase (iterator position);
(2) size_type erase (const key_type& k);
(3)     void erase (iterator first, iterator last);

(1) iterator erase (const_iterator position);
(2) size_type erase (const key_type& k);
(3) iterator erase (const_iterator first, const_iterator last);
```

### Erase elements

Removes from the `map` container either a single element or a range of elements (`[first, last]`).

This effectively reduces the container `size` by the number of elements removed, which are destroyed.

## Parameters

### position

Iterator pointing to a single element to be removed from the `map`.

This shall point to a valid and dereferenceable element.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types that point to elements.

### k

Key of the element to be removed from the `map`.

Member type `key_type` is the type of the elements in the container, defined in `map` as an alias of its first template parameter (`key`).

### first, last

Iterators specifying a range within the `map` container to be removed: `[first, last]`. i.e., the range includes all the elements between `first` and `last`, including the element pointed by `first` but not the one pointed by `last`.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types that point to elements.

## Return value

For the key-based version (2), the function returns the number of elements erased, which in `map` containers is at most 1.

Member type `size_type` is an unsigned integral type.

The other versions return no value.

The other versions return an iterator to the element that follows the last element removed (or `map::end`, if the last element was removed).

Member type `iterator` is a [bidirectional iterator](#) type that points to an element.

## Example

```
1 // erasing from map
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     std::map<char,int>::iterator it;
9
10    // insert some values:
11    mymap['a']=10;
12    mymap['b']=20;
13    mymap['c']=30;
14    mymap['d']=40;
15    mymap['e']=50;
16    mymap['f']=60;
17
18    it=mymap.find('b');           // erasing by iterator
19    mymap.erase (it);
20
21    mymap.erase ('c');           // erasing by key
22
23    it=mymap.find ('e');
24    mymap.erase ( it, mymap.end() ); // erasing by range
25
```

```

26 // show content:
27 for (it=mymap.begin(); it!=mymap.end(); ++it)
28     std::cout << it->first << " => " << it->second << '\n';
29
30 return 0;
31 }

```

## Output:

```
a => 10
d => 40
```

## Complexity

For the first version (`erase(position)`), amortized constant.

For the second version (`erase(val)`), logarithmic in container `size`.

For the last version (`erase(first,last)`), linear in the distance between `first` and `last`.

## Iterator validity

Iterators, pointers and references referring to elements removed by the function are invalidated.

All other iterators, pointers and references keep their validity.

## Data races

The container is modified.

The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the container is not.

## Exception safety

Unless the container's `comparison object` throws, this function never throws exceptions (no-throw guarantee).

Otherwise, if a single element is to be removed, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If an invalid `position` or range is specified, it causes *undefined behavior*.

## See also

<a href="#">map::clear</a>	Clear content (public member function )
<a href="#">map::insert</a>	Insert elements (public member function )
<a href="#">map::find</a>	Get iterator to element (public member function )

# /map/map/find

public member function

## std::map::find

<map>

```

iterator find (const key_type& k);
const_iterator find (const key_type& k) const;

```

### Get iterator to element

Searches the container for an element with a `key` equivalent to `k` and returns an iterator to it if found, otherwise it returns an iterator to `map::end`.

Two keys are considered equivalent if the container's `comparison object` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

Another member function, `map::count`, can be used to just check whether a particular key exists.

## Parameters

`k`

Key to be searched for.

Member type `key_type` is the type of the keys for the elements in the container, defined in `map` as an alias of its first template parameter (`key`).

## Return value

An iterator to the element, if an element with specified key is found, or `map::end` otherwise.

If the `map` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are `bidirectional iterator` types pointing to elements (of type `value_type`).

Notice that `value_type` in `map` containers is an alias of `pair<const key_type, mapped_type>`.

## Example

```

1 // map::find
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     std::map<char,int>::iterator it;
9
10    mymap['a']=50;
11    mymap['b']=100;
12    mymap['c']=150;
13    mymap['d']=200;
14

```

```

15     it = mymap.find('b');
16     if (it != mymap.end())
17         mymap.erase (it);
18
19 // print content:
20 std::cout << "elements in mymap:" << '\n';
21 std::cout << "a => " << mymap.find('a')->second << '\n';
22 std::cout << "c => " << mymap.find('c')->second << '\n';
23 std::cout << "d => " << mymap.find('d')->second << '\n';
24
25     return 0;
26 }

```

Output:

```

elements in mymap:
a => 50
c => 150
d => 200

```

## Complexity

Logarithmic in [size](#).

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<a href="#">map::operator[]</a>	Access element ( <a href="#">public member function</a> )
<a href="#">map::count</a>	Count elements with a specific key ( <a href="#">public member function</a> )
<a href="#">map::lower_bound</a>	Return iterator to lower bound ( <a href="#">public member function</a> )
<a href="#">map::upper_bound</a>	Return iterator to upper bound ( <a href="#">public member function</a> )

## /map/map/get\_allocator

public member function

### std::map::get\_allocator

<map>

```

allocator_type get_allocator() const;
allocator_type get_allocator() const noexcept;

```

#### Get allocator

Returns a copy of the allocator object associated with the [map](#).

#### Parameters

none

#### Return Value

The allocator.

Member type `allocator_type` is the type of the allocator used by the container, defined in [map](#) as an alias of its fourth template parameter (`Alloc`).

## Example

```

1 // map::get_allocator
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     int psize;
8     std::map<char,int> mymap;
9     std::pair<const char,int>* p;
10
11    // allocate an array of 5 elements using mymap's allocator:
12    p=mymap.get_allocator().allocate(5);
13
14    // assign some values to array
15    psize = sizeof(std::map<char,int>::value_type)*5;
16
17    std::cout << "The allocated array has a size of " << psize << " bytes.\n";
18
19    mymap.get_allocator().deallocate(p,5);
20

```

```
21 |     return 0;
22 }
```

The example shows an elaborate way to allocate memory for an array of pairs using the same allocator used by the container.  
A possible output is:

```
The allocated array has a size of 40 bytes.
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

Copying any instantiation of the [default allocator](#) is also guaranteed to never throw.

## See also

<a href="#">allocator</a>	Default allocator (class template )
---------------------------	-------------------------------------

# /map/map/insert

public member function

## std::map::insert

<map>

```
single element (1) pair<iterator,bool> insert (const value_type& val);
with hint (2) iterator insert (iterator position, const value_type& val);
range (3) template <class InputIterator>
           void insert (InputIterator first, InputIterator last);

single element (1) pair<iterator,bool> insert (const value_type& val);
template <class P> pair<iterator,bool> insert (P&& val);
with hint (2) iterator insert (const_iterator position, const value_type& val);
template <class P> iterator insert (const_iterator position, P&& val);
range (3) void insert (InputIterator first, InputIterator last);
initializer list (4) void insert (initializer_list<value_type> il);
```

### Insert elements

Extends the container by inserting new elements, effectively increasing the container [size](#) by the number of elements inserted.

Because element keys in a [map](#) are unique, the insertion operation checks whether each inserted element has a key equivalent to the one of an element already in the container, and if so, the element is not inserted, returning an iterator to this existing element (if the function returns a value).

For a similar container allowing for duplicate elements, see [multimap](#).

An alternative way to insert elements in a [map](#) is by using member function [map::operator\[\]](#).

Internally, [map](#) containers keep all their elements sorted by their key following the criterion specified by its [comparison object](#). The elements are always inserted in its respective position following this ordering.

The parameters determine how many elements are inserted and to which values they are initialized:

## Parameters

### val

Value to be copied to (or moved as) the inserted element.

Member type [value\\_type](#) is the type of the elements in the container, defined in [map](#) as [pair<const key\\_type,mapped\\_type>](#) (see [map member types](#)).

The template parameter [P](#) shall be a type convertible to [value\\_type](#).

The signatures taking an argument of type [P&&](#) are only called if [std::is\\_constructible<value\\_type,P&&>](#) is true.

If [P](#) is instantiated as a reference type, the argument is copied.

### position

Hint for the position where the element can be inserted.

The function optimizes its insertion time if [position](#) points to the element that will [precede](#) the inserted element.

The function optimizes its insertion time if [position](#) points to the element that will [follow](#) the inserted element (or to the [end](#), if it would be the last).

Notice that this is just a hint and does not force the new element to be inserted at that position within the [map](#) container (the elements in a [map](#) always follow a specific order depending on their key).

Member types [iterator](#) and [const\\_iterator](#) are defined in [map](#) as [bidirectional iterator](#) types that point to elements.

### first, last

Iterators specifying a range of elements. Copies of the elements in the range [\[first, last\)](#) are inserted in the container.

Notice that the range includes all the elements between [first](#) and [last](#), including the element pointed by [first](#) but not the one pointed by [last](#).

The function template argument [InputIterator](#) shall be an [input iterator](#) type that points to elements of a type from which [value\\_type](#) objects can be constructed.

### il

An `initializer_list` object. Copies of these elements are inserted.

These objects are automatically constructed from `initializer_list` declarators.

Member type `value_type` is the type of the elements contained in the container, defined in `map` as `pair<const key_type, mapped_type>` (see `map member types`).

## Return value

The single element versions (1) return a `pair`, with its member `pair::first` set to an iterator pointing to either the newly inserted element or to the element with an equivalent key in the `map`. The `pair::second` element in the `pair` is set to `true` if a new element was inserted or `false` if an equivalent key already existed.

The versions with a hint (2) return an iterator pointing to either the newly inserted element or to the element that already had an equivalent key in the `map`.

Member type `iterator` is a `bidirectional iterator` type that points to elements.

`pair` is a class template declared in `<utility>` (see `pair`).

## Example

```
1 // map::insert (C++98)
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     // first insert function version (single parameter):
10    mymap.insert ( std::pair<char,int>('a',100) );
11    mymap.insert ( std::pair<char,int>('z',200) );
12
13    std::pair<std::map<char,int>::iterator,bool> ret;
14    ret = mymap.insert ( std::pair<char,int>('z',500) );
15    if (ret.second==false) {
16        std::cout << "element 'z' already existed";
17        std::cout << " with a value of " << ret.first->second << '\n';
18    }
19
20    // second insert function version (with hint position):
21    std::map<char,int>::iterator it = mymap.begin();
22    mymap.insert (it, std::pair<char,int>('b',300)); // max efficiency inserting
23    mymap.insert (it, std::pair<char,int>('c',400)); // no max efficiency inserting
24
25    // third insert function version (range insertion):
26    std::map<char,int> anothermap;
27    anothermap.insert(mymap.begin(),mymap.end());
28
29    // showing contents:
30    std::cout << "mymap contains:\n";
31    for (it=mymap.begin(); it!=mymap.end(); ++it)
32        std::cout << it->first << " => " << it->second << '\n';
33
34    std::cout << "anothermap contains:\n";
35    for (it=anothermap.begin(); it!=anothermap.end(); ++it)
36        std::cout << it->first << " => " << it->second << '\n';
37
38    return 0;
39 }
```

Output:

```
element 'z' already existed with a value of 200
mymap contains:
a => 100
b => 300
c => 400
z => 200
anothermap contains:
a => 100
b => 300
```

## Complexity

If a single element is inserted, logarithmic in `size` in general, but amortized constant if a hint is given and the `position` given is the optimal.

If `N` elements are inserted, `Nlog(size+N)` in general, but linear in `size+N` if the elements are already sorted according to the same ordering criterion used by the container.

If `N` elements are inserted, `Nlog(size+N)`.

Implementations may optimize if the range is already sorted.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

If a single element is to be inserted, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if an invalid `position` is specified, it causes `undefined behavior`.

## See also

<a href="#">map::operator[]</a>	Access element (public member function )
<a href="#">map::find</a>	Get iterator to element (public member function )
<a href="#">map::erase</a>	Erase elements (public member function )

## /map/map/key\_comp

public member function

### std::map::key\_comp

<map>

`key_compare key_comp() const;`

#### Return key comparison object

Returns a copy of the *comparison object* used by the container to compare *keys*.

The *comparison object* of a `map` object is set on *construction*. Its type (member `key_compare`) is the third template parameter of the `map` template. By default, this is a `less` object, which returns the same as `operator<`.

This object determines the order of the elements in the container: it is a function pointer or a function object that takes two arguments of the same type as the element keys, and returns `true` if the first argument is considered to go before the second in the *strict weak ordering* it defines, and `false` otherwise.

Two keys are considered equivalent if `key_comp` returns `false` reflexively (i.e., no matter the order in which the keys are passed as arguments).

## Parameters

none

## Return value

The comparison object.

Member type `key_compare` is the type of the *comparison object* associated to the container, defined in `map` as an alias of its third template parameter (`Compare`).

## Example

```
1 // map::key_comp
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     std::map<char,int>::key_compare mycomp = mymap.key_comp();
10
11    mymap['a']=100;
12    mymap['b']=200;
13    mymap['c']=300;
14
15    std::cout << "mymap contains:\n";
16
17    char highest = mymap.rbegin()->first;      // key value of last element
18
19    std::map<char,int>::iterator it = mymap.begin();
20    do {
21        std::cout << it->first << " => " << it->second << '\n';
22    } while ( mycomp((*it++).first, highest) );
23
24    std::cout << '\n';
25
26    return 0;
27 }
```

Output:

```
mymap contains:
a => 100
b => 200
c => 300
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<a href="#">map::value_comp</a>	Return value comparison object ( <a href="#">public member function</a> )
<a href="#">map::find</a>	Get iterator to element ( <a href="#">public member function</a> )
<a href="#">map::count</a>	Count elements with a specific key ( <a href="#">public member function</a> )
<a href="#">map::lower_bound</a>	Return iterator to lower bound ( <a href="#">public member function</a> )
<a href="#">map::upper_bound</a>	Return iterator to upper bound ( <a href="#">public member function</a> )

## /map/map/lower\_bound

public member function

### std::map::lower\_bound

<map>

```
iterator lower_bound (const key_type& k);
const_iterator lower_bound (const key_type& k) const;
```

#### Return iterator to lower bound

Returns an iterator pointing to the first element in the container whose key is not considered to go before *k* (i.e., either it is equivalent or goes after).

The function uses its internal [comparison object \(key\\_comp\)](#) to determine this, returning an iterator to the first element for which `key_comp(element_key, k)` would return `false`.

If the `map` class is instantiated with the default comparison type ([less](#)), the function returns an iterator to the first element whose key is not less than *k*.

A similar member function, `upper_bound`, has the same behavior as `lower_bound`, except in the case that the `map` contains an element with a key equivalent to *k*: In this case, `lower_bound` returns an iterator pointing to that element, whereas `upper_bound` returns an iterator pointing to the next element.

## Parameters

*k*

Key to search for.

Member type `key_type` is the type of the elements in the container, defined in `map` as an alias of its first template parameter (`key`).

## Return value

An iterator to the the first element in the container whose key is not considered to go before *k*, or `map::end` if all keys are considered to go before *k*.

If the `map` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements (of type `value_type`).

Notice that `value_type` in `map` containers is itself also a `pair` type: `pair<const key_type, mapped_type>`.

## Example

```
1 // map::lower_bound/upper_bound
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     std::map<char,int>::iterator itlow,itup;
9
10    mymap['a']=20;
11    mymap['b']=40;
12    mymap['c']=60;
13    mymap['d']=80;
14    mymap['e']=100;
15
16    itlow=mymap.lower_bound ('b'); // itlow points to b
17    itup=mymap.upper_bound ('d'); // itup points to e (not d!)
18
19    mymap.erase(itlow,itup);      // erases [itlow,itup)
20
21    // print content:
22    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
23        std::cout << it->first << " => " << it->second << '\n';
24
25    return 0;
26 }
```

```
a => 20
e => 100
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

### See also

<code>map::upper_bound</code>	Return iterator to upper bound (public member function )
<code>map::equal_range</code>	Get range of equal elements (public member function )
<code>map::find</code>	Get iterator to element (public member function )
<code>map::count</code>	Count elements with a specific key (public member function )

## /map/map/map

public member function

### std::map::map

<map>

<code>empty (1)</code>	<code>explicit map (const key_compare&amp; comp = key_compare(), const allocator_type&amp; alloc = allocator_type());</code>
<code>range (2)</code>	<code>template &lt;class InputIterator&gt; map (InputIterator first, InputIterator last, const key_compare&amp; comp = key_compare(), const allocator_type&amp; alloc = allocator_type());</code>
<code>copy (3)</code>	<code>map (const map&amp; x);</code>
<code>empty (1)</code>	<code>explicit map (const key_compare&amp; comp = key_compare(), const allocator_type&amp; alloc = allocator_type());</code>
<code>range (2)</code>	<code>template &lt;class InputIterator&gt; explicit map (const allocator_type&amp; alloc);</code>
<code>copy (3)</code>	<code>map (InputIterator first, InputIterator last, const key_compare&amp; comp = key_compare(), const allocator_type&amp; alloc = allocator_type());</code>
<code>move (4)</code>	<code>map (const map&amp; x, const allocator_type&amp; alloc);</code>
<code>initializer list (5)</code>	<code>map (initializer_list&lt;value_type&gt; il, const key_compare&amp; comp = key_compare(), const allocator_type&amp; alloc = allocator_type());</code>
<code>empty (1)</code>	<code>map();</code>
<code>range (2)</code>	<code>explicit map (const key_compare&amp; comp, const allocator_type&amp; alloc = allocator_type());</code>
<code>copy (3)</code>	<code>explicit map (const allocator_type&amp; alloc);</code>
<code>move (4)</code>	<code>template &lt;class InputIterator&gt; map (InputIterator first, InputIterator last, const key_compare&amp; comp = key_compare(), const allocator_type&amp; alloc = allocator_type());</code>
<code>initializer list (5)</code>	<code>map (const allocator_type&amp; alloc);</code>
<code>empty (1)</code>	<code>map();</code>
<code>range (2)</code>	<code>explicit map (const key_compare&amp; comp, const allocator_type&amp; alloc = allocator_type());</code>
<code>copy (3)</code>	<code>explicit map (const allocator_type&amp; alloc);</code>
<code>move (4)</code>	<code>template &lt;class InputIterator&gt; map (InputIterator first, InputIterator last, const key_compare&amp; comp = key_compare(), const allocator_type&amp; alloc = allocator_type());</code>
<code>initializer list (5)</code>	<code>map (const allocator_type&amp; alloc);</code>

### Construct map

Constructs a `map` container object, initializing its contents depending on the constructor version used:

#### (1) empty container constructor (default constructor)

Constructs an `empty` container, with no elements.

#### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element constructed from its corresponding element in that range.

#### (3) copy constructor

Constructs a container with a copy of each of the elements in `x`.

The container keeps an internal copy of `alloc` and `comp`, which are used to allocate storage and to sort the elements throughout its lifetime.  
The copy constructor (3) creates a container that keeps and uses copies of `x`'s `allocator` and `comparison object`.

The storage for the elements is allocated using this `internal allocator`.

#### (1) empty container constructors (default constructor)

Constructs an `empty` container, with no elements.

#### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element `emplace-constructed` from its corresponding element in that range.

#### (3) copy constructor (and copying with allocator)

Constructs a container with a copy of each of the elements in `x`.

#### (4) move constructor (and moving with allocator)

Constructs a container that acquires the elements of *x*.  
If *alloc* is specified and is different from *x*'s allocator, the elements are moved. Otherwise, no elements are constructed (their ownership is directly transferred).  
*x* is left in an unspecified but valid state.

#### (5) initializer\_list constructor

Constructs a container with a copy of each of the elements in *il*.

The container keeps an internal copy of *alloc*, which is used to allocate and deallocate storage for its elements, and to construct and destroy them (as specified by its [allocator\\_traits](#)). If no *alloc* argument is passed to the constructor, a default-constructed allocator is used, except in the following cases:

- The copy constructor (3, *first signature*) creates a container that keeps and uses a copy of the allocator returned by calling the appropriate [selected\\_on\\_container\\_copy\\_construction](#) trait on *x*'s allocator.
- The move constructor (4, *first signature*) acquires *x*'s allocator.

The container also keeps an internal copy of *comp* (or *x*'s [comparison object](#)), which is used to establish the order of the elements in the container and to check for elements with equivalent keys.

All elements are *copied*, *moved* or otherwise *constructed* by calling [allocator\\_traits::construct](#) with the appropriate arguments.

The elements are sorted according to the [comparison object](#). If more than one element with equivalent keys is passed to the constructor, only the first one is preserved.

## Parameters

*comp*

Binary predicate that, taking two *element keys* as argument, returns `true` if the first argument goes before the second argument in the *strict weak ordering* it defines, and `false` otherwise.  
This shall be a function pointer or a function object.  
Member type `key_compare` is the internal comparison object type used by the container, defined in `map` as an alias of its third template parameter (`Compare`).  
If `key_compare` uses the default `less` (which has no state), this parameter is not relevant.

*alloc*

Allocator object.  
The container keeps and uses an internal copy of this allocator.  
Member type `allocator_type` is the internal allocator type used by the container, defined in `map` as an alias of its fourth template parameter (`Alloc`).  
If `allocator_type` is an instantiation of the default `allocator` (which has no state), this parameter is not relevant.

*first*, *last*

[Input iterators](#) to the initial and final positions in a range. The range used is `[first, last)`, which includes all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
The function template argument `InputIterator` shall be an [input iterator](#) type that points to elements of a type from which `value_type` objects can be constructed (in `map`, `value_type` is an alias of `pair<const key_type, mapped_type>`)

*x*

Another `map` object of the same type (with the same class template arguments `Key`, `T`, `Compare` and `Alloc`), whose contents are either copied or acquired.

*il*

An [initializer\\_list](#) object.  
These objects are automatically constructed from *initializer list* declarators.  
Member type `value_type` is the type of the elements in the container, defined in `map` as an alias of `pair<const key_type, mapped_type>` (see [map types](#)).

## Example

```
1 // constructing maps
2 #include <iostream>
3 #include <map>
4
5 bool fncomp (char lhs, char rhs) {return lhs<rhs;}
6
7 struct classcomp {
8     bool operator() (const char& lhs, const char& rhs) const
9     {return lhs<rhs;}
10 };
11
12 int main ()
13 {
14     std::map<char,int> first;
15
16     first['a']=10;
17     first['b']=30;
18     first['c']=50;
19     first['d']=70;
20
21     std::map<char,int> second (first.begin(),first.end());
22
23     std::map<char,int> third (second);
24
25     std::map<char,int,classcomp> fourth;           // class as Compare
26
27     bool(*fn_pt)(char,char) = fncomp;
28     std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare
29
30     return 0;
31 }
```

The code does not produce any output, but demonstrates some ways in which a `map` container can be constructed.

## Complexity

Constant for the *empty constructors* (1), and for the *move constructors* (4) (unless *alloc* is different from *x*'s allocator).

For all other cases, linear in the distance between the iterators (copy constructions) if the elements are already sorted according to the same criterion. For unsorted sequences, linearithmic ( $N \log N$ ) in that distance (sorting,copy constructions).

## Iterator validity

The move constructors (4), invalidate all iterators, pointers and references related to *x* if the elements are moved.

## Data races

All copied elements are accessed.

The move constructors (4) modify *x*.

## Exception safety

**Strong guarantee:** no effects in case an exception is thrown.

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

## See also

<code>map::operator=</code>	Copy container content (public member function )
<code>map::insert</code>	Insert elements (public member function )

# /map/map/~map

public member function

## std::map::~map

<map>

`~map();`

### Map destructor

Destroys the container object.

This destroys all container elements, and deallocates all the storage capacity allocated by the `map` container using its allocator.

This calls `allocator_traits::destroy` on each of the contained elements, and deallocates all the storage capacity allocated by the `map` container using its allocator.

## Complexity

Linear in `map::size` (destructors).

## Iterator validity

All iterators, pointers and references are invalidated.

## Data races

The container and all its elements are modified.

## Exception safety

**No-throw guarantee:** never throws exceptions.

# /map/map/max\_size

public member function

## std::map::max\_size

<map>

```
size_type max_size() const;
size_type max_size() const noexcept;
```

### Return maximum size

Returns the maximum number of elements that the `map` container can hold.

This is the maximum potential `size` the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size: it can still fail to allocate storage at any point before that size is reached.

## Parameters

none

## Return Value

The maximum number of elements a `map` container can hold as content.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // map::max_size
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
```

```

7 int i;
8 std::map<int,int> mymap;
9
10 if (mymap.max_size()>1000)
11 {
12     for (i=0; i<1000; i++) mymap[i]=0;
13     std::cout << "The map contains 1000 elements.\n";
14 }
15 else std::cout << "The map could not hold 1000 elements.\n";
16
17 return 0;
18 }
```

Here, member `max_size` is used to check beforehand whether the map will allow for 1000 elements to be inserted.

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>map::size</code>	Return container size (public member function )
------------------------	---

# /map/map/operator=

public member function

## std::map::operator=

<map>

<code>copy (1) map&amp; operator= (const map&amp; x);</code>
<code>copy (1) map&amp; operator= (const map&amp; x);</code>
<code>move (2) map&amp; operator= (map&amp;&amp; x);</code>
<code>initializer list (3) map&amp; operator= (initializer_list&lt;value_type&gt; il);</code>

### Copy container content

Assigns new contents to the container, replacing its current content.

Copies all the elements from `x` into the container, changing its `size` accordingly.

The container preserves its `current allocator`, which is used to allocate additional storage if needed.

The `copy assignment` (1) copies all the elements from `x` into the container (with `x` preserving its contents).

The `move assignment` (2) moves the elements of `x` into the container (`x` is left in an unspecified but valid state).

The `initializer list assignment` (3) copies the elements of `il` into the container.

The new container `size` is the same as the `size` of `x` (or `il`) before the call.

The container preserves its `current allocator`, except if the `allocator traits` indicate `x`'s allocator should `propagate`. This allocator is used (through its `traits`) to `allocate` or `deallocate` if there are changes in storage requirements, and to `construct` or `destroy` elements, if needed.

The elements stored in the container before the call are either assigned to or destroyed.

## Parameters

`x`

A `map` object of the same type (i.e., with the same template parameters, `key`, `T`, `Compare` and `Alloc`).

`il`

An `initializer_list` object. The compiler will automatically construct such objects from `initializer_list` declarators.

Member type `value_type` is the type of the elements in the container, defined in `map` as an alias of `pair<const key_type, mapped_type>` (see `map member types`).

## Return value

`*this`

## Example

```

1 // assignment operator with maps
2 #include <iostream>
3 #include <map>
4
```

```

5 int main ()
6 {
7     std::map<char,int> first;
8     std::map<char,int> second;
9
10    first['x']=8;
11    first['y']=16;
12    first['z']=32;
13
14    second=first;           // second now contains 3 ints
15    first=std::map<char,int>(); // and first is now empty
16
17    std::cout << "Size of first: " << first.size() << '\n';
18    std::cout << "Size of second: " << second.size() << '\n';
19    return 0;
20 }

```

Output:

```

Size of first: 0
Size of second: 3

```

## Complexity

For the *copy assignment* (1): Linear in sizes (destructions, copies).

For the *move assignment* (2): Linear in current container *size* (destructions).\*

For the *initializer list assignment* (3): Up to logarithmic in sizes (destructions, move-assignments) -- linear if *il* is already sorted.

\* Additional complexity for assignments if allocators do not *propagate*.

## Iterator validity

All iterators, references and pointers related to this container are invalidated.

In the *move assignment*, iterators, pointers and references referring to elements in *x* are also invalidated.

## Data races

All copied elements are accessed.

The *move assignment* (2) modifies *x*.

The container and all its elements are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.

If *allocator\_traits::construct* is not supported with the appropriate arguments for the element constructions, or if *value\_type* is not *copy assignable* (or *move assignable* for (2)), it causes *undefined behavior*.

## See also

<a href="#">map::insert</a>	Insert elements (public member function )
<a href="#">map::operator[]</a>	Access element (public member function )
<a href="#">map::map</a>	Construct map (public member function )

## /map/map/operator[]

public member function

### std::map::operator[]

<map>

```

mapped_type& operator[] (const key_type& k);
mapped_type& operator[] (const key_type& k);
mapped_type& operator[] (key_type&& k);

```

#### Access element

If *k* matches the key of an element in the container, the function returns a reference to its mapped value.

If *k* does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the *container size* by one, even if no mapped value is assigned to the element (the element is constructed using its default constructor).

A similar member function, [map::at](#), has the same behavior when an element with the key exists, but throws an exception when it does not.

A call to this function is equivalent to:

```

(*((this->insert(make_pair(k,mapped_type()))).first)).second

```

## Parameters

*k*

Key value of the element whose mapped value is accessed.

Member type *key\_type* is the type of the keys for the elements stored in the container, defined in [map](#) as an alias of its first template parameter (*key*). If an rvalue (second version), the key is moved instead of copied when a new element is inserted.

## Return value

A reference to the mapped value of the element with a key value equivalent to *k*.

Member type *mapped\_type* is the type of the mapped values in the container, defined in [map](#) as an alias of its second template parameter (*T*).

## Example

```
1 // accessing mapped values
2 #include <iostream>
3 #include <map>
4 #include <string>
5
6 int main ()
7 {
8     std::map<char, std::string> mymap;
9
10    mymap['a']="an element";
11    mymap['b']="another element";
12    mymap['c']=mymap['b'];
13
14    std::cout << "mymap['a'] is " << mymap['a'] << '\n';
15    std::cout << "mymap['b'] is " << mymap['b'] << '\n';
16    std::cout << "mymap['c'] is " << mymap['c'] << '\n';
17    std::cout << "mymap['d'] is " << mymap['d'] << '\n';
18
19    std::cout << "mymap now contains " << mymap.size() << " elements.\n";
20
21    return 0;
22 }
```

Notice how the last access (to element 'd') inserts a new element in the `map` with that key and initialized to its default value (an empty string) even though it is accessed only to retrieve its value. Member function `map::find` does not produce this effect.

Output:

```
mymap['a'] is an element
mymap['b'] is another element
mymap['c'] is another element
mymap['d'] is
mymap now contains 4 elements.
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed, and potentially modified.

The function accesses an element and returns a reference that can be used to modify its mapped value. Concurrently accessing other elements is safe. If the function inserts a new element, concurrently iterating ranges in the container is not safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If a new element is inserted and `allocator_traits::construct` cannot construct an element with `k` and a default-constructed `mapped_type` (or if `mapped_type` is not `default constructible`), it causes *undefined behavior*.

## See also

<code>map::find</code>	Get iterator to element (public member function )
<code>map::insert</code>	Insert elements (public member function )
<code>map::operator=</code>	Copy container content (public member function )

# /map/map/operators

function

## std::relational operators (map)

<map>

```
(1) template <class Key, class T, class Compare, class Alloc>
(1)   bool operator== ( const map<Key,T,Compare,Alloc>& lhs,
(1)           const map<Key,T,Compare,Alloc>& rhs );
(2) template <class Key, class T, class Compare, class Alloc>
(2)   bool operator!= ( const map<Key,T,Compare,Alloc>& lhs,
(2)           const map<Key,T,Compare,Alloc>& rhs );
(3) template <class Key, class T, class Compare, class Alloc>
(3)   bool operator< ( const map<Key,T,Compare,Alloc>& lhs,
(3)           const map<Key,T,Compare,Alloc>& rhs );
(4) template <class Key, class T, class Compare, class Alloc>
(4)   bool operator<= ( const map<Key,T,Compare,Alloc>& lhs,
(4)           const map<Key,T,Compare,Alloc>& rhs );
(5) template <class Key, class T, class Compare, class Alloc>
(5)   bool operator> ( const map<Key,T,Compare,Alloc>& lhs,
(5)           const map<Key,T,Compare,Alloc>& rhs );
(6) template <class Key, class T, class Compare, class Alloc>
(6)   bool operator>= ( const map<Key,T,Compare,Alloc>& lhs,
(6)           const map<Key,T,Compare,Alloc>& rhs );
```

### Relational operators for map

Performs the appropriate comparison operation between the `map` containers `lhs` and `rhs`.

The *equality comparison* (`operator==`) is performed by first comparing `sizes`, and if they match, the elements are compared sequentially using `operator==`, stopping at the first mismatch (as if using algorithm `equal`).

The *less-than comparison* (`operator<`) behaves as if using algorithm `lexicographical_compare`, which compares the elements sequentially using `operator<` in a reciprocal manner (i.e., checking both  $a < b$  and  $b < a$ ) and stopping at the first occurrence.

The other operations also use the operators `==` and `<` internally to compare the elements, behaving as if the following equivalent operations were performed:

operation	equivalent operation
<code>a != b</code>	<code>!(a == b)</code>
<code>a &gt; b</code>	<code>b &lt; a</code>
<code>a &lt;= b</code>	<code>!(b &lt; a)</code>
<code>a &gt;= b</code>	<code>!(a &lt; b)</code>

Notice that none of these operations take into consideration the `internal comparison object` of either container, but compare the elements (of type `value_type`) directly.

`value_type` is a `pair` type, and as such, by default, two elements will compare equal only if both their `key` and `mapped value` compare equal, and one compare lower than the other only if the first `key` is lower, or if the `keys` are equivalent and the `mapped value` is lower.

These operators are overloaded in header `<map>`.

## Parameters

`lhs, rhs`  
map containers (to the left- and right-hand side of the operator, respectively), having both the same template parameters (`key`, `T`, `Compare` and `Alloc`).

## Example

```
1 // map comparisons
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> foo,bar;
8     foo['a']=100;
9     foo['b']=200;
10    bar['a']=10;
11    bar['z']=1000;
12
13    // foo ({a,100},{b,200}) vs bar ({a,10},{z,1000}):
14    if (foo==bar) std::cout << "foo and bar are equal\n";
15    if (foo!=bar) std::cout << "foo and bar are not equal\n";
16    if (foo< bar) std::cout << "foo is less than bar\n";
17    if (foo> bar) std::cout << "foo is greater than bar\n";
18    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
19    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
20
21    return 0;
22 }
```

Output:

```
foo and bar are not equal
foo is greater than bar
foo is greater than or equal to bar
```

## Return Value

true if the condition holds, and false otherwise.

## Complexity

Up to linear in the `size` of `lhs` and `rhs`.

For (1) and (2), constant if the `sizes` of `lhs` and `rhs` differ, and up to linear in that `size` (equality comparisons) otherwise.  
For the others, up to linear in the smaller `size` (each representing two comparisons with `operator<`).

## Iterator validity

No changes.

## Data races

Both containers, `lhs` and `rhs`, are accessed.

Up to all of their contained elements may be accessed.

## Exception safety

If the type of the elements supports the appropriate operation with no-throw guarantee, the function never throws exceptions (no-throw guarantee). In any case, the function cannot modify its arguments.

## See also

<code>map::key_comp</code>	Return key comparison object (public member function )
<code>map::value_comp</code>	Return value comparison object (public member function )
<code>map::operator=</code>	Copy container content (public member function )
<code>map::swap</code>	Swap content (public member function )

# /map/map/rbegin

public member function

## std::map::rbegin

<map>

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
    reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
```

### Return reverse iterator to reverse beginning

Returns a *reverse iterator* pointing to the last element in the container (i.e., its *reverse beginning*).

*Reverse iterators* iterate backwards: increasing them moves them towards the beginning of the container.

rbegin points to the element preceding the one that would be pointed to by member `end`.

### Parameters

none

### Return Value

A *reverse iterator* to the *reverse beginning* of the sequence container.

If the `map` object is `const`-qualified, the function returns a `const_reverse_iterator`. Otherwise, it returns a `reverse_iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are *reverse bidirectional iterator* types pointing to elements. See `map` member types.

### Example

```
1 // map::rbegin/rend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['x'] = 100;
10    mymap['y'] = 200;
11    mymap['z'] = 300;
12
13    // show content:
14    std::map<char,int>::reverse_iterator rit;
15    for (rit=mymap.rbegin(); rit!=mymap.rend(); ++rit)
16        std::cout << rit->first << " => " << rit->second << '\n';
17
18    return 0;
19 }
```

Output:

```
z => 300
y => 200
x => 100
```

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">map::rend</a>	Return reverse iterator to reverse end (public member function )
<a href="#">map::begin</a>	Return iterator to beginning (public member function )
<a href="#">map::end</a>	Return iterator to end (public member function )

# /map/map/rend

public member function

## std::map::rend

<map>

```
reverse_iterator rend();
const_reverse_iterator rend() const;
    reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

### Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element right before the first element in the `map` container (which is considered its *reverse end*).

The range between `map::rbegin` and `map::rend` contains all the elements of the container (in reverse order).

### Parameters

none

### Return Value

A reverse iterator to the *reverse end* of the sequence container.

If the `map` object is `const`-qualified, the function returns a `const_reverse_iterator`. Otherwise, it returns a `reverse_iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are reverse `bidirectional iterator` types pointing to elements. See `map` member types.

### Example

```
1 // map::rbegin/rend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['x'] = 100;
10    mymap['y'] = 200;
11    mymap['z'] = 300;
12
13    // show content:
14    std::map<char,int>::reverse_iterator rit;
15    for (rit=mymap.rbegin(); rit!=mymap.rend(); ++rit)
16        std::cout << rit->first << " => " << rit->second << '\n';
17
18    return 0;
19 }
```

Output:

```
z => 300
y => 200
x => 100
```

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">map::rbegin</a>	Return reverse iterator to reverse beginning (public member function )
<a href="#">map::begin</a>	Return iterator to beginning (public member function )
<a href="#">map::end</a>	Return iterator to end (public member function )

## /map/map/size

public member function

## std::map::size

<map>

```
size_type size() const;
size_type size() const noexcept;
```

### Return container size

Returns the number of elements in the [map](#) container.

## Parameters

none

## Return Value

The number of elements in the container.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // map::size
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     mymap['a']=101;
9     mymap['b']=202;
10    mymap['c']=302;
11
12    std::cout << "mymap.size() is " << mymap.size() << '\n';
13
14    return 0;
15 }
```

Output:

```
mymap.size() is 3
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">map::max_size</a>	Return maximum size ( <a href="#">public member function</a> )
<a href="#">map::empty</a>	Test whether container is empty ( <a href="#">public member function</a> )

# /map/map/swap

public member function

## std::map::swap

<map>

```
void swap (map& x);
```

### Swap content

Exchanges the content of the container by the content of *x*, which is another [map](#) of the same type. Sizes may differ.

After the call to this member function, the elements in this container are those which were in *x* before the call, and the elements of *x* are those which were in this. All iterators, references and pointers remain valid for the swapped objects.

Notice that a non-member function exists with the same name, [swap](#), overloading that algorithm with an optimization that behaves like this member function.

Whether the internal container allocators and [comparison objects](#) are swapped is undefined.

Whether the internal container allocators are swapped is not defined, unless in the case the appropriate allocator trait indicates explicitly that they shall propagate.

The internal [comparison objects](#) are always exchanged, using [swap](#).

## Parameters

x

Another [map](#) container of the same type as this (i.e., with the same template parameters, `Key`, `T`, `Compare` and `Alloc`) whose content is swapped with that of this container.

## Return value

none

## Example

```
1 // swap maps
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> foo,bar;
8
9     foo[ 'x' ]=100;
10    foo[ 'y' ]=200;
11
12    bar[ 'a' ]=11;
13    bar[ 'b' ]=22;
14    bar[ 'c' ]=33;
15
16    foo.swap(bar);
17
18    std::cout << "foo contains:\n";
19    for (std::map<char,int>::iterator it=foo.begin(); it!=foo.end(); ++it)
20        std::cout << it->first << " => " << it->second << '\n';
21
22    std::cout << "bar contains:\n";
23    for (std::map<char,int>::iterator it=bar.begin(); it!=bar.end(); ++it)
24        std::cout << it->first << " => " << it->second << '\n';
25
26    return 0;
27 }
```

Output:

```
foo contains:
a => 11
b => 22
c => 33
bar contains:
x => 100
y => 200
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.

Note that the *end iterators* do not refer to elements and may be invalidated.

## Data races

Both the container and *x* are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both containers compare equal, or if their *allocator traits* indicate that the allocators shall *propagate*, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">swap (map)</a>	Exchanges the contents of two maps ( <a href="#">function template</a> )
<a href="#">swap_ranges</a>	Exchange values of two ranges ( <a href="#">function template</a> )

# /map/map/swap-free

function template

## std::swap (map)

<map>

```
template <class Key, class T, class Compare, class Alloc>
void swap (map<Key,T,Compare,Alloc>& x, map<Key,T,Compare,Alloc>& y);
```

### Exchanges the contents of two maps

The contents of container *x* are exchanged with those of *y*. Both container objects must be of the same type (same template parameters), although sizes may differ.

After the call to this member function, the elements in *x* are those which were in *y* before the call, and the elements of *y* are those which were in *x*. All iterators, references and pointers remain valid for the swapped objects.

This is an overload of the generic algorithm [swap](#) that improves its performance by mutually transferring ownership over their assets to the other container (i.e., the containers exchange references to their data, without actually performing any element copy or movement): It behaves as if *x.swap(y)* was called.

## Parameters

*x,y*

`map` containers of the same type (i.e., having both the same template parameters: `key`, `T`, `Compare` and `Alloc`).

## Return value

none

## Example

```
1 // swap maps
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> foo,bar;
8
9     foo['x']=100;
10    foo['y']=200;
11
12    bar['a']=11;
13    bar['b']=22;
14    bar['c']=33;
15
16    swap(foo,bar);
17
18    std::cout << "foo contains:\n";
19    for (std::map<char,int>::iterator it=foo.begin(); it!=foo.end(); ++it)
20        std::cout << it->first << " => " << it->second << '\n';
21
22    std::cout << "bar contains:\n";
23    for (std::map<char,int>::iterator it=bar.begin(); it!=bar.end(); ++it)
24        std::cout << it->first << " => " << it->second << '\n';
25
26    return 0;
27 }
```

Output:

```
foo contains:
a => 11
b => 22
c => 33
bar contains:
x => 100
y => 200
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.

Note that the `end` iterators do not refer to elements and may be invalidated.

## Data races

Both containers, `x` and `y`, are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both `maps` compare equal, or if their allocator traits indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">map::swap</a>	Swap content (public member function )
<a href="#">swap</a>	Exchange values of two objects (function template )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )

## /map/map/upper\_bound

public member function

### std::map::upper\_bound

<map>

```
iterator upper_bound (const key_type& k);
const_iterator upper_bound (const key_type& k) const;
```

#### Return iterator to upper bound

Returns an iterator pointing to the first element in the container whose key is considered to go after `k`.

The function uses its internal comparison object (`key_comp`) to determine this, returning an iterator to the first element for which `key_comp(k,element_key)` would return `true`.

If the `map` class is instantiated with the default comparison type (`less`), the function returns an iterator to the first element whose key is greater than `k`.

A similar member function, `lower_bound`, has the same behavior as `upper_bound`, except in the case that the `map` contains an element with a key equivalent to `k`: In this case `lower_bound` returns an iterator pointing to that element, whereas `upper_bound` returns an iterator pointing to the next element.

## Parameters

`k`

Key to search for.

Member type `key_type` is the type of the elements in the container, defined in `map` as an alias of its first template parameter (`key`).

## Return value

An iterator to the the first element in the container whose key is considered to go after `k`, or `map::end` if no keys are considered to go after `k`.

If the `map` object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are `bidirectional_iterator` types pointing to elements.

## Example

```
1 // map::lower_bound/upper_bound
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     std::map<char,int>::iterator itlow,itup;
9
10    mymap[ 'a' ]=20;
11    mymap[ 'b' ]=40;
12    mymap[ 'c' ]=60;
13    mymap[ 'd' ]=80;
14    mymap[ 'e' ]=100;
15
16    itlow=mymap.lower_bound ('b'); // itlow points to b
17    itup=mymap.upper_bound ('d'); // itup points to e (not d!)
18
19    mymap.erase(itlow,itup);      // erases [itlow,itup)
20
21    // print content:
22    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
23        std::cout << it->first << " => " << it->second << '\n';
24
25    return 0;
26 }
```

```
a => 20
e => 100
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>map::lower_bound</code>	Return iterator to lower bound ( <a href="#">public member function</a> )
<code>map::equal_range</code>	Get range of equal elements ( <a href="#">public member function</a> )
<code>map::find</code>	Get iterator to element ( <a href="#">public member function</a> )
<code>map::count</code>	Count elements with a specific key ( <a href="#">public member function</a> )

## /map/map/value\_comp

public member function

### `std::map::value_comp`

`<map>`

`value_compare value_comp() const;`

#### **Return value comparison object**

Returns a comparison object that can be used to compare two elements to get whether the key of the first one goes before the second.

The arguments taken by this function object are of member type `value_type` (defined in `map` as an alias of `pair<const key_type,mapped_type>`), but the

`mapped_type` part of the value is not taken into consideration in this comparison.

The comparison object returned is an object of the member type `map::value_compare`, which is a nested class that uses the internal `comparison` object to generate the appropriate comparison functional class. It is defined with the same behavior as:

```
1 template <class Key, class T, class Compare, class Alloc>
2 class map<Key,T,Compare,Alloc>::value_compare
3 { // in C++98, it is required to inherit binary_function<value_type,value_type,bool>
4     friend class map;
5 protected:
6     Compare comp;
7     value_compare (Compare c) : comp(c) {} // constructed with map's comparison object
8 public:
9     typedef bool result_type;
10    typedef value_type first_argument_type;
11    typedef value_type second_argument_type;
12    bool operator() (const value_type& x, const value_type& y) const
13    {
14        return comp(x.first, y.first);
15    }
16 }
```

The public member of this comparison class returns `true` if the key of the first argument is considered to go before that of the second (according to the *strict weak ordering* specified by the container's `comparison` object, `key_comp`), and `false` otherwise.

Notice that `value_compare` has no public constructor, therefore no objects can be directly created from this nested class outside `map` members.

## Parameters

none

## Return value

The comparison object for element values.

Member type `value_compare` is a nested class type (described above).

## Example

```
1 // map::value_comp
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['x']=1001;
10    mymap['y']=2002;
11    mymap['z']=3003;
12
13    std::cout << "mymap contains:\n";
14
15    std::pair<char,int> highest = *mymap.rbegin();           // last element
16
17    std::map<char,int>::iterator it = mymap.begin();
18    do {
19        std::cout << it->first << " => " << it->second << '\n';
20    } while ( mymap.value_comp()(*it++, highest) );
21
22    return 0;
23 }
```

Output:

```
mymap contains:
x => 1001
y => 2002
z => 3003
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<a href="#">map::key_comp</a>	Return key comparison object (public member function )
<a href="#">map::find</a>	Get iterator to element (public member function )
<a href="#">map::count</a>	Count elements with a specific key (public member function )

<code>map::lower_bound</code>	Return iterator to lower bound (public member function )
<code>map::upper_bound</code>	Return iterator to upper bound (public member function )

## /map/multimap

class template

### std::multimap

<map>

```
template < class Key, // multimap::key_type
           class T, // multimap::mapped_type
           class Compare = less<Key>, // multimap::key_compare
           class Alloc = allocator<pair<const Key, T> > // multimap::allocator_type
         > class multimap;
```

#### Multiple-key map

Multimaps are associative containers that store elements formed by a combination of a *key value* and a *mapped value*, following a specific order, and where multiple elements can have equivalent keys.

In a *multimap*, the *key values* are generally used to sort and uniquely identify the elements, while the *mapped values* store the content associated to this *key*. The types of *key* and *mapped value* may differ, and are grouped together in member type *value\_type*, which is a *pair* type combining both:

```
typedef pair<const Key, T> value_type;
```

Internally, the elements in a *multimap* are always sorted by its *key* following a specific *strict weak ordering* criterion indicated by its internal *comparison object* (of type *Compare*).

*multimap* containers are generally slower than *unordered\_multimap* containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Multimaps are typically implemented as *binary search trees*.

### Container properties

#### Associative

Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

#### Ordered

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

#### Map

Each element associates a *key* to a *mapped value*: Keys are meant to identify the elements whose main content is the *mapped value*.

#### Multiple equivalent keys

Multiple elements in the container can have equivalent *keys*.

#### Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

### Template parameters

#### Key

Type of the *keys*. Each element in a *map* is identified by its *key value*.

Aliased as member type *multimap::key\_type*.

#### T

Type of the mapped value. Each element in a *multimap* stores some data as its *mapped value*.

Aliased as member type *multimap::mapped\_type*.

#### Compare

A binary predicate that takes two element keys as arguments and returns a *bool*. The expression *comp(a,b)*, where *comp* is an object of this type and *a* and *b* are element keys, shall return *true* if *a* is considered to go before *b* in the *strict weak ordering* the function defines.

The *multimap* object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent (by comparing them reflexively: they are equivalent if *!comp(a,b) && !comp(b,a)*).

This can be a function pointer or a function object (see *constructor* for an example). This defaults to *less<T>*, which returns the same as applying the *less-than operator* (*a<b*).

Aliased as member type *multimap::key\_compare*.

#### Alloc

Type of the allocator object used to define the storage allocation model. By default, the *allocator* class template is used, which defines the simplest memory allocation model and is value-independent.

Aliased as member type *multimap::allocator\_type*.

### Member types

member type	definition	notes
<code>key_type</code>	The first template parameter ( <code>key</code> )	
<code>mapped_type</code>	The second template parameter ( <code>T</code> )	
<code>value_type</code>	<code>pair&lt;const key_type,mapped_type&gt;</code>	
<code>key_compare</code>	The third template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;key_type&gt;</code>
<code>value_compare</code>	<i>Nested function class to compare elements</i>	see <code>value_comp</code>
<code>allocator_type</code>	The fourth template parameter ( <code>Alloc</code> )	defaults to: <code>allocator&lt;value_type&gt;</code>
<code>reference</code>	<code>allocator_type::reference</code>	for the default allocator: <code>value_type&amp;</code>
<code>const_reference</code>	<code>allocator_type::const_reference</code>	for the default allocator: <code>const value_type&amp;</code>
<code>pointer</code>	<code>allocator_type::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_type::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a bidirectional <code>iterator</code> to <code>value_type</code>	convertible to <code>const_iterator</code>

<code>const_iterator</code>	a bidirectional iterator to <code>const value_type</code>	
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>
member type	definition	notes
<code>key_type</code>	The first template parameter ( <code>key</code> )	
<code>mapped_type</code>	The second template parameter ( <code>T</code> )	
<code>value_type</code>	<code>pair&lt;const key_type, mapped_type&gt;</code>	
<code>key_compare</code>	The third template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;key_type&gt;</code>
<code>value_compare</code>	<i>Nested function class to compare elements</i>	see <code>value_comp</code>
<code>allocator_type</code>	The fourth template parameter ( <code>Alloc</code> )	defaults to: <code>allocator&lt;value_type&gt;</code>
<code>reference</code>	<code>value_type&amp;</code>	
<code>const_reference</code>	<code>const value_type&amp;</code>	
<code>pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a bidirectional iterator to <code>value_type</code>	convertible to <code>const_iterator</code>
<code>const_iterator</code>	a bidirectional iterator to <code>const value_type</code>	
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>

## Member functions

<b>(constructor)</b>	Construct multimap <a href="#">(public member function )</a>
<b>(destructor)</b>	Multimap destructor <a href="#">(public member function )</a>
<b>operator=</b>	Copy container content <a href="#">(public member function )</a>

### Iterators:

<code>begin</code>	Return iterator to beginning <a href="#">(public member function )</a>
<code>end</code>	Return iterator to end <a href="#">(public member function )</a>
<code>rbegin</code>	Return reverse iterator to reverse beginning <a href="#">(public member function )</a>
<code>rend</code>	Return reverse iterator to reverse end <a href="#">(public member function )</a>
<code>cbegin</code>	Return <code>const_iterator</code> to beginning <a href="#">(public member function )</a>
<code>cend</code>	Return <code>const_iterator</code> to end <a href="#">(public member function )</a>
<code>crbegin</code>	Return <code>const_reverse_iterator</code> to reverse beginning <a href="#">(public member function )</a>
<code>crend</code>	Return <code>const_reverse_iterator</code> to reverse end <a href="#">(public member function )</a>

### Capacity:

<code>empty</code>	Test whether container is empty <a href="#">(public member function )</a>
<code>size</code>	Return container size <a href="#">(public member function )</a>
<code>max_size</code>	Return maximum size <a href="#">(public member function )</a>

### Modifiers:

<code>insert</code>	Insert element <a href="#">(public member function )</a>
<code>erase</code>	Erase elements <a href="#">(public member function )</a>
<code>swap</code>	Swap content <a href="#">(public member function )</a>
<code>clear</code>	Clear content <a href="#">(public member function )</a>
<code>emplace</code>	Construct and insert element <a href="#">(public member function )</a>
<code>emplace_hint</code>	Construct and insert element with hint <a href="#">(public member function )</a>

### Observers:

<code>key_comp</code>	Return key comparison object <a href="#">(public member function )</a>
<code>value_comp</code>	Return value comparison object <a href="#">(public member function )</a>

### Operations:

<code>find</code>	Get iterator to element <a href="#">(public member function )</a>
<code>count</code>	Count elements with a specific key <a href="#">(public member function )</a>
<code>lower_bound</code>	Return iterator to lower bound <a href="#">(public member function )</a>
<code>upper_bound</code>	Return iterator to upper bound <a href="#">(public member function )</a>
<code>equal_range</code>	Get range of equal elements <a href="#">(public member function )</a>

### Allocator:

<code>get_allocator</code>	Get allocator <a href="#">(public member function )</a>
----------------------------	---

## /map/multimap/begin

public member function

## std::multimap::begin

&lt;map&gt;

```
iterator begin();
const_iterator begin() const;
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

### Return iterator to beginning

Returns an iterator referring to the first element in the `multimap` container.

Because `multimap` containers keep their elements ordered at all times, `begin` points to the element that goes first following the container's sorting criterion.

If the container is `empty`, the returned iterator value shall not be dereferenced.

### Parameters

none

### Return Value

An iterator to the first element in the container.

If the `multimap` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an iterator.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements (of type `value_type`). Notice that `value_type` in `multimap` containers is an alias of `pair<const key_type, mapped_type>`.

### Example

```
1 // multimap::begin/end
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     mymultimap.insert (std::pair<char,int>('a',10));
10    mymultimap.insert (std::pair<char,int>('b',20));
11    mymultimap.insert (std::pair<char,int>('b',150));
12
13    // show content:
14    for (std::multimap<char,int>::iterator it=mymultimap.begin(); it!=mymultimap.end(); ++it)
15        std::cout << (*it).first << " => " << (*it).second << '\n';
16
17    return 0;
18 }
```

Output:

```
a => 10
b => 20
b => 150
```

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">multimap::end</a>	Return iterator to end ( <a href="#">public member function</a> )
<a href="#">multimap::rbegin</a>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<a href="#">multimap::rend</a>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )

## /map/multimap/cbegin

public member function

## std::multimap::cbegin

&lt;map&gt;

```
const_iterator cbegin() const noexcept;
```

**Return `const_iterator` to beginning**

Returns a `const_iterator` pointing to the first element in the container.

A `const_iterator` is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also `const`), just like the iterator returned by `multimap::begin`, but it cannot be used to modify the contents it points to, even if the `multimap` object is not itself `const`.

If the container is `empty`, the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

A `const_iterator` to the beginning of the sequence.

Member type `const_iterator` is a [bidirectional iterator](#) type that points to const elements (of type `const value_type`). Notice that `value_type` in `multimap` containers is an alias of `pair<const key_type, mapped_type>`.

## Example

```
1 // multimap::cbegin/cend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap = { {'x',100}, {'y',200}, {'x',300} };
8
9     // print content:
10    std::cout << "mymultimap contains:";
11    for (auto it = mymultimap.cbegin(); it != mymultimap.cend(); ++it)
12        std::cout << "[" << it->first << ':' << it->second << "]";
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
mymultimap contains: [x:100] [x:300] [y:200]
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multimap::cend</a>	Return <code>const_iterator</code> to end ( <a href="#">public member function</a> )
<a href="#">multimap::begin</a>	Return iterator to beginning ( <a href="#">public member function</a> )
<a href="#">multimap::crbegin</a>	Return <code>const_reverse_iterator</code> to reverse beginning ( <a href="#">public member function</a> )

## /map/multimap/cend

public member function

### std::multimap::cend

`<map>`

```
const_iterator cend() const noexcept;
```

#### Return `const_iterator` to end

Returns a `const_iterator` pointing to the *past-the-end* element in the container.

A `const_iterator` is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also `const`), just like the iterator returned by `multimap::end`, but it cannot be used to modify the contents it points to, even if the `multimap` object is not itself `const`.

If the container is `empty`, this function returns the same as `multimap::cbegin`.

The value returned shall not be dereferenced.

## Parameters

none

## Return Value

A const\_iterator to the element past the end of the sequence.

Member type const\_iterator is a [bidirectional iterator](#) type that points to const elements.

## Example

```
1 // multimap::cbegin/cend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap = { {'x',100}, {'y',200}, {'x',300} };
8
9     // print content:
10    std::cout << "mymultimap contains:" ;
11    for (auto it = mymultimap.cbegin(); it != mymultimap.cend(); ++it)
12        std::cout << "[" << it->first << ':' << it->second << "]";
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
mymultimap contains: [x:100] [x:300] [y:200]
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multimap::end</a>	Return iterator to end ( <a href="#">public member function</a> )
<a href="#">multimap::cbegin</a>	Return const_iterator to beginning ( <a href="#">public member function</a> )

# /map/multimap/clear

public member function

## std::multimap::clear

<map>

```
void clear();
void clear() noexcept;
```

### Clear content

Removes all elements from the multimap container (which are destroyed), leaving the container with a [size](#) of 0.

## Parameters

none

## Return value

none

## Example

```
1 // multimap::clear
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8     std::multimap<char,int>::iterator it;
9
10    mymultimap.insert(std::pair<char,int>('b',80));
11    mymultimap.insert(std::pair<char,int>('b',120));
12    mymultimap.insert(std::pair<char,int>('q',360));
13
14    std::cout << "mymultimap contains:\n";
```

```

15 for (it=mymultimap.begin(); it!=mymultimap.end(); ++it)
16     std::cout << (*it).first << " => " << (*it).second << '\n';
17
18 mymultimap.clear();
19
20 mymultimap.insert(std::pair<char,int>('a',11));
21 mymultimap.insert(std::pair<char,int>('x',22));
22
23 std::cout << "mymultimap contains:\n";
24 for (it=mymultimap.begin(); it != mymultimap.end(); ++it)
25     std::cout << (*it).first << " => " << (*it).second << '\n';
26
27 return 0;
28 }

```

Output:

```

mymultimap contains:
b => 80
b => 120
q => 360
mymap contains:
a => 11
x => 22

```

## Complexity

Linear in `size` (destructions).

## Iterator validity

All iterators, pointers and references related to this container are invalidated.

## Data races

The container is modified.

All contained elements are modified.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>multimap::erase</code>	Erase elements ( <a href="#">public member function</a> )
<code>multimap::size</code>	Return container size ( <a href="#">public member function</a> )
<code>multimap::empty</code>	Test whether container is empty ( <a href="#">public member function</a> )

# /map/multimap/count

public member function

## std::multimap::count

<map>

`size_type count (const key_type& k) const;`

### Count elements with a specific key

Searches the container for elements with a key equivalent to *k* and returns the number of matches.

Two keys are considered equivalent if the container's `comparison object` returns `false` reflexively (i.e., no matter the order in which the keys are passed as arguments).

## Parameters

*k*

Key to search for.

Member type `key_type` is the type of the element keys in the container, defined in `map` as an alias of its first template parameter (`key`).

## Return value

The number of elements in the container contains that have a key equivalent to *k*.

Member type `size_type` is an unsigned integral type.

## Example

```

1 // multimap::count
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymm;
8
9     mymm.insert(std::make_pair('x',50));
10    mymm.insert(std::make_pair('y',100));
11    mymm.insert(std::make_pair('y',150));
12    mymm.insert(std::make_pair('y',200));

```

```

13 mymm.insert(std::make_pair('z',250));
14 mymm.insert(std::make_pair('z',300));
15
16 for (char c='x'; c<='z'; c++)
17 {
18     std::cout << "There are " << mymm.count(c) << " elements with key " << c << ":";
19     std::multimap<char,int>::iterator it;
20     for (it=mymm.equal_range(c).first; it!=mymm.equal_range(c).second; ++it)
21         std::cout << ' ' << (*it).second;
22     std::cout << '\n';
23 }
24
25 return 0;
26 }
```

Output:

```

There are 1 elements with key x: 50
There are 3 elements with key y: 100 150 200
There are 2 elements with key z: 250 300
```

## Complexity

Logarithmic in `size`, plus linear in the number of matches.

## Iterator validity

No changes.

## Data races

The container is accessed.

No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>multimap::find</code>	Get iterator to element ( <a href="#">public member function</a> )
<code>multimap::equal_range</code>	Get range of equal elements ( <a href="#">public member function</a> )
<code>multimap::size</code>	Return container size ( <a href="#">public member function</a> )
<code>multimap::lower_bound</code>	Return iterator to lower bound ( <a href="#">public member function</a> )
<code>multimap::upper_bound</code>	Return iterator to upper bound ( <a href="#">public member function</a> )

## /map/multimap/crbegin

public member function

### std::multimap::crbegin

<map>

`const_reverse_iterator crbegin() const noexcept;`

**Return `const_reverse_iterator` to reverse beginning**

Returns a `const_reverse_iterator` pointing to the last element in the container (i.e., its *reverse beginning*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse beginning* of the sequence.

Member type `const_reverse_iterator` is a reverse `bidirectional iterator` type that points to a `const` element (see [multimap member types](#)).

## Example

```

1 // multimap::crbegin/crend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap = { {'x',100}, {'y',200}, {'x',300} };
8
9     // print content:
10    std::cout << "mymultimap backwards:" ;
11    for (auto rit = mymultimap.crbegin(); rit != mymultimap.crend(); ++rit)
12        std::cout << " [" << rit->first << ':' << rit->second << ']';
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
mymultimap backwards: [y:200] [x:300] [x:100]
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multimap::crend</a>	Return const_reverse_iterator to reverse end (public member function )
<a href="#">multimap::begin</a>	Return iterator to beginning (public member function )
<a href="#">multimap::rbegin</a>	Return reverse iterator to reverse beginning (public member function )

# /map/multimap/crend

public member function

## std::multimap::crend

<map>

```
const_reverse_iterator crend() const noexcept;
```

### Return const\_reverse\_iterator to reverse end

Returns a const\_reverse\_iterator pointing to the theoretical element preceding the first element in the container (which is considered its *reverse end*).

## Parameters

none

## Return Value

A const\_reverse\_iterator to the *reverse end* of the sequence.

Member type const\_reverse\_iterator is a reverse [bidirectional iterator](#) type that points to a const element (see [multimap member types](#)).

## Example

```
1 // multimap::crbegin/crend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap = { {'x',100}, {'y',200}, {'x',300} };
8
9     // print content:
10    std::cout << "mymultimap backwards:" ;
11    for (auto rit = mymultimap.crbegin(); rit != mymultimap.crend(); ++rit)
12        std::cout << "[" << rit->first << ':' << rit->second << ']';
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
mymultimap backwards: [y:200] [x:300] [x:100]
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multimap::end</a>	Return iterator to end (public member function )
<a href="#">multimap::crbegin</a>	Return const_reverse_iterator to reverse beginning (public member function )
<a href="#">multimap::rend</a>	Return reverse iterator to reverse end (public member function )

# /map/multimap/emplace

public member function

## std::multimap::emplace

<map>

```
template <class... Args>
iterator emplace (Args&&... args);
```

### Construct and insert element

Inserts a new element in the [multimap](#). This new element is constructed in place using *args* as the arguments for the construction of a *value\_type* (which is an object of a [pair](#) type).

This effectively increases the container [size](#) by one.

Internally, [multimap](#) containers keep all their elements sorted by key following the criterion specified by its [comparison object](#). The element is always inserted in its respective position following this ordering.

The element is constructed in-place by calling [allocator\\_traits::construct](#) with *args* forwarded.

A similar member function exists, [insert](#), which either copies or moves existing objects into the container.

The relative ordering of elements with equivalent keys is preserved, and newly inserted elements follow those with equivalent keys already in the container.

## Parameters

args

Arguments used to construct a new object of the *mapped type* for the inserted element.

Arguments forwarded to construct the new element (of type [pair<const key\\_type, mapped\\_type>](#)).

This can be one of:

- Two arguments: one for the *key*, the other for the *mapped value*.
- A single argument of a [pair](#) type with a value for the *key* as first member, and a value for the *mapped value* as second.
- [piecewise\\_construct](#) as first argument, and two additional arguments with [tuples](#) to be forwarded as arguments for the *key value* and for the *mapped value* respectively.

See [pair::pair](#) for more info.

## Return value

An iterator to the newly inserted element.

Member type [iterator](#) is a [bidirectional iterator](#) type that points to an element.

## Example

```
1 // multimap::emplace
2 #include <iostream>
3 #include <string>
4 #include <map>
5
6 int main ()
7 {
8     std::multimap<std::string, float> mymultimap;
9
10    mymultimap.emplace("apple",1.50);
11    mymultimap.emplace("coffee",2.10);
12    mymultimap.emplace("apple",1.40);
13
14    std::cout << "mymultimap contains:";
15    for (auto& x: mymultimap)
16        std::cout << " [" << x.first << ':' << x.second << ']';
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
mymultimap contains: [apple:1.5] [apple:1.4] [coffee:2.1]
```

## Complexity

Logarithmic in the container [size](#).

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

`multimap::emplace_hint` Construct and insert element with hint (public member function)

`multimap::insert` Insert element (public member function)

`multimap::erase` Erase elements (public member function)

# /map/multimap/emplace\_hint

public member function

## std::multimap::emplace\_hint

<map>

```
template <class... Args>
iterator emplace_hint (const_iterator position, Args&&... args);
```

### Construct and insert element with hint

Inserts a new element in the `multimap`, with a hint on the insertion *position*. This new element is constructed in place using *args* as the arguments for the construction of a `value_type` (which is an object of a `pair` type).

This effectively increases the container `size` by one.

The value in *position* is used as a hint on the insertion point. The element will nevertheless be inserted at its corresponding position following the order described by its internal `comparison` object, but this hint is used by the function to begin its search for the insertion point, speeding up the process considerably when the actual insertion point is either *position* or close to it.

The element is constructed in-place by calling `allocator_traits::construct` with *args* forwarded.

The relative ordering of equivalent elements is preserved, and newly inserted elements follow their equivalents already in the container.

## Parameters

### position

Hint for the position where the element can be inserted.

The function optimizes its insertion time if *position* points to the element that will follow the inserted element (or to the `end`, if it would be the last).

Notice that this does not force the new element to be in that position within the `multimap` container (the elements in a `multimap` always follow a specific order).

`const_iterator` is a member type, defined as a `bidirectional iterator` type that points to elements.

### args

Arguments used to construct a new object of the *mapped type* for the inserted element.

Arguments forwarded to construct the new element (of type `pair<const key_type, mapped_type>`).

This can be one of:

- Two arguments: one for the *key*, the other for the *mapped value*.
- A single argument of a `pair` type with a value for the *key* as first member, and a value for the *mapped value* as second.
- `piecewise_construct` as first argument, and two additional arguments with `tuples` to be forwarded as arguments for the *key value* and for the *mapped value* respectively.

See `pair::pair` for more info.

## Return value

An iterator to the newly inserted element.

Member type `iterator` is a `bidirectional iterator` type that points to an element.

## Example

```
1 // multimap::emplace_hint
2 #include <iostream>
3 #include <string>
4 #include <map>
5
6 int main ()
7 {
8     std::multimap<std::string,int> mymultimap;
9     auto it = mymultimap.end();
10
11    it = mymultimap.emplace_hint(it,"foo",10);
12    mymultimap.emplace_hint(it,"bar",20);
13    mymultimap.emplace_hint(mymultimap.end(),"foo",30);
14
15    std::cout << "mymultimap contains:" ;
16    for (auto& x: mymultimap)
17        std::cout << " [" << x.first << ':' << x.second << ']';
18    std::cout << '\n';
19
20    return 0;
21 }
```

Output:

```
| mymultimap contains: [bar:20] [foo:10] [foo:30]
```

## Complexity

Generally, logarithmic in the container [size](#).  
Amortized constant if the insertion point for the element is [position](#).

## Iterator validity

No changes.

## Data races

The container is modified.  
Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
If [allocator\\_traits::construct](#) is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<a href="#">multimap::emplace</a>	Construct and insert element (public member function )
<a href="#">multimap::insert</a>	Insert element (public member function )
<a href="#">multimap::erase</a>	Erase elements (public member function )

# /map/multimap/empty

public member function

## std::multimap::empty

<map>

```
bool empty() const;
bool empty() const noexcept;
```

### Test whether container is empty

Returns whether the [multimap](#) container is empty (i.e. whether its [size](#) is 0).

This function does not modify the container in any way. To clear the content of a [multimap](#) container, see [multimap::clear](#).

## Parameters

none

## Return Value

true if the container [size](#) is 0, false otherwise.

## Example

```
1 // multimap::empty
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     mymultimap.insert (std::pair<char,int>('b',101));
10    mymultimap.insert (std::pair<char,int>('b',202));
11    mymultimap.insert (std::pair<char,int>('q',505));
12
13    while (!mymultimap.empty())
14    {
15        std::cout << mymultimap.begin()->first << " => ";
16        std::cout << mymultimap.begin()->second << '\n';
17        mymultimap.erase(mymultimap.begin());
18    }
19
20    return 0;
21 }
```

Output:

```
b => 101
b => 202
q => 505
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">multimap::clear</a>	Clear content (public member function )
<a href="#">multimap::erase</a>	Erase elements (public member function )
<a href="#">multimap::size</a>	Return container size (public member function )

# /map/multimap/end

public member function

## std::multimap::end

<map>

```
iterator end();
const_iterator end() const;
iterator end() noexcept;
const_iterator end() const noexcept;
```

### Return iterator to end

Returns an iterator referring to the *past-the-end* element in the [multimap](#) container.

The *past-the-end* element is the theoretical element that would follow the last element in the [multimap](#) container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with [multimap::begin](#) to specify a range including all the elements in the container.

If the container is [empty](#), this function returns the same as [multimap::begin](#).

## Parameters

none

## Return Value

An iterator to the *past-the-end* element in the container.

If the [multimap](#) object is [const](#)-qualified, the function returns a [const\\_iterator](#). Otherwise, it returns an [iterator](#).

Member types [iterator](#) and [const\\_iterator](#) are [bidirectional iterator](#) types pointing to elements.

## Example

```
1 // multimap::begin/end
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     mymultimap.insert (std::pair<char,int>('a',10));
10    mymultimap.insert (std::pair<char,int>('b',20));
11    mymultimap.insert (std::pair<char,int>('b',150));
12
13 // show content:
14 for (std::multimap<char,int>::iterator it=mymultimap.begin(); it!=mymultimap.end(); ++it)
15     std::cout << (*it).first << " => " << (*it).second << '\n';
16
17 return 0;
18 }
```

Output:

```
a => 10
b => 20
b => 150
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the [const](#) nor the non-[const](#) versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<b>multimap::begin</b>	Return iterator to beginning (public member function )
<b>multimap::rbegin</b>	Return reverse iterator to reverse beginning (public member function )
<b>multimap::rend</b>	Return reverse iterator to reverse end (public member function )

## /map/multimap/equal\_range

public member function

### std::multimap::equal\_range

<map>

```
pair<const_iterator,const_iterator> equal_range (const key_type& k) const;
pair<iterator,iterator> equal_range (const key_type& k);
```

#### Get range of equal elements

Returns the bounds of a range that includes all the elements in the container which have a key equivalent to *k*.

If no matches are found, the range returned has a length of zero, with both iterators pointing to the first element that has a key considered to go after *k* according to the container's internal comparison object (*key\_comp*).

Two keys are considered equivalent if the container's comparison object returns false reflexively (i.e., no matter the order in which the keys are passed as arguments).

## Parameters

*k*

Key to search for.

Member type *key\_type* is the type of the elements in the container, defined in *multimap* as an alias of its first template parameter (*key*).

## Return value

The function returns a *pair*, whose member *pair::first* is the lower bound of the range (the same as *lower\_bound*), and *pair::second* is the upper bound (the same as *upper\_bound*).

If the *multimap* object is const-qualified, the function returns a *pair* of *const\_iterator*. Otherwise, it returns a *pair* of *iterator*.

Member types *iterator* and *const\_iterator* are *bidirectional iterator* types pointing to elements (of type *value\_type*).

Notice that *value\_type* in *multimap* containers is itself also a *pair* type: *pair<const key\_type, mapped\_type>*.

## Example

```
1 // multimap::equal_range
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymm;
8
9     mymm.insert(std::pair<char,int>('a',10));
10    mymm.insert(std::pair<char,int>('b',20));
11    mymm.insert(std::pair<char,int>('b',30));
12    mymm.insert(std::pair<char,int>('b',40));
13    mymm.insert(std::pair<char,int>('c',50));
14    mymm.insert(std::pair<char,int>('c',60));
15    mymm.insert(std::pair<char,int>('d',60));
16
17    std::cout << "mymm contains:\n";
18    for (char ch='a'; ch<='d'; ch++)
19    {
20        std::pair <std::multimap<char,int>::iterator, std::multimap<char,int>::iterator> ret;
21        ret = mymm.equal_range(ch);
22        std::cout << ch << " =>";
23        for (std::multimap<char,int>::iterator it=ret.first; it!=ret.second; ++it)
24            std::cout << ' ' << it->second;
25        std::cout << '\n';
26    }
27
28    return 0;
29 }
```

```
mymm contains:
a => 10
b => 20 30 40
c => 50 60
d => 60
```

## Complexity

Logarithmic in *size*.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<b>multimap::count</b>	Count elements with a specific key ( <a href="#">public member function</a> )
<b>multimap::lower_bound</b>	Return iterator to lower bound ( <a href="#">public member function</a> )
<b>multimap::upper_bound</b>	Return iterator to upper bound ( <a href="#">public member function</a> )
<b>multimap::find</b>	Get iterator to element ( <a href="#">public member function</a> )

# /map/multimap/erase

public member function

## std::multimap::erase

<map>

(1)	void erase (iterator position);
(2)	size_type erase (const key_type& k);
(3)	void erase (iterator first, iterator last);
(1)	iterator erase (const_iterator position);
(2)	size_type erase (const key_type& k);
(3)	iterator erase (const_iterator first, const_iterator last);

### Erase elements

Removes elements from the [multimap](#) container.

This effectively reduces the container [size](#) by the number of elements removed, which are destroyed.

The parameters determine the elements removed:

### Parameters

#### position

Iterator pointing to a single element to be removed from the [multimap](#).

Member types [iterator](#) and [const\\_iterator](#) are [bidirectional iterator](#) types that point to elements.

#### k

Key to be removed from the [multimap](#). All elements with a key equivalent to this are removed from the container.

Member type [key\\_type](#) is the type of the elements in the container, defined in [multimap](#) as an alias of its first template parameter ([key](#)).

#### first, last

Iterators specifying a range within the [multimap](#) container to be removed: `[first, last]`. i.e., the range includes all the elements between `first` and `last`, including the element pointed by `first` but not the one pointed by `last`.

Member types [iterator](#) and [const\\_iterator](#) are [bidirectional iterator](#) types that point to elements.

### Return value

For the key-based version (2), the function returns the number of elements erased.

Member type [size\\_type](#) is an unsigned integral type.

The other versions return no value.

The other versions return an iterator to the element that follows the last element removed (or [multimap::end](#), if the last element was removed).

Member type [iterator](#) is a [bidirectional iterator](#) type that points to an element.

### Example

```
1 // erasing from multimap
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     // insert some values:
10    mymultimap.insert(std::pair<char,int>('a',10));
11    mymultimap.insert(std::pair<char,int>('b',20));
12    mymultimap.insert(std::pair<char,int>('b',30));
13    mymultimap.insert(std::pair<char,int>('c',40));
14    mymultimap.insert(std::pair<char,int>('d',50));
15    mymultimap.insert(std::pair<char,int>('d',60));
16    mymultimap.insert(std::pair<char,int>('e',70));
17    mymultimap.insert(std::pair<char,int>('f',80));
18
19    std::multimap<char,int>::iterator it = mymultimap.find('b');
```

```

20     mymultimap.erase (it);                                // erasing by iterator (1 element)
21
22     mymultimap.erase ('d');                               // erasing by key (2 elements)
23
24     it=mymultimap.find ('e');
25     mymultimap.erase ( it, mymultimap.end() ); // erasing by range
26
27     // show content:
28     for (it=mymultimap.begin(); it!=mymultimap.end(); ++it)
29         std::cout << (*it).first <> " => " << (*it).second << '\n';
30
31     return 0;
32 }

```

Output:

```

a => 10
b => 30
c => 40

```

## Complexity

For the first version (`erase(position)`), amortized constant.

For the second version (`erase(val)`), logarithmic in container `size`, plus linear in the number of elements removed.

For the last version (`erase(first,last)`), linear in the distance between `first` and `last`.

## Iterator validity

Iterators, pointers and references referring to elements removed by the function are invalidated.

All other iterators, pointers and references keep their validity.

## Data races

The container is modified.

The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the container is not.

## Exception safety

Unless the container's `comparison object` throws, this function never throws exceptions (no-throw guarantee).

Otherwise, if a single element is to be removed, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If an invalid `position` or range is specified, it causes *undefined behavior*.

## See also

<a href="#">multimap::clear</a>	Clear content (public member function )
<a href="#">multimap::insert</a>	Insert element (public member function )
<a href="#">multimap::find</a>	Get iterator to element (public member function )

## /map/multimap/find

public member function

### std::multimap::find

<map>

```

iterator find (const key_type& k);
const_iterator find (const key_type& k) const;

```

#### Get iterator to element

Searches the container for an element with a `key` equivalent to `k` and returns an iterator to it if found, otherwise it returns an iterator to `multimap::end`.

Notice that this function returns an iterator to a single element (of the possibly multiple elements with equivalent keys). To obtain the entire range of equivalent elements, see `multimap::equal_range`.

Two keys are considered equivalent if the container's `comparison object` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

## Parameters

`k`

Key to be searched for.

Member type `key_type` is the type of the keys for the elements in the container, defined in `multimap` as an alias of its first template parameter (`key`).

## Return value

An iterator to the element, if an element with specified key is found, or `multimap::end` otherwise.

If the `multimap` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are `bidirectional iterator` types pointing to elements (of type `value_type`).

Notice that `value_type` in `multimap` containers is an alias of `pair<const key_type, mapped_type>`.

## Example

```

1 // multimap::find
2 #include <iostream>
3 #include <map>
4

```

```

5 int main ()
6 {
7     std::multimap<char,int> mymm;
8
9     mymm.insert (std::make_pair('x',10));
10    mymm.insert (std::make_pair('y',20));
11    mymm.insert (std::make_pair('z',30));
12    mymm.insert (std::make_pair('z',40));
13
14    std::multimap<char,int>::iterator it = mymm.find('x');
15    mymm.erase (it);
16    mymm.erase (mymm.find('z'));
17
18    // print content:
19    std::cout << "elements in mymm:" << '\n';
20    std::cout << "y => " << mymm.find('y')->second << '\n';
21    std::cout << "z => " << mymm.find('z')->second << '\n';
22
23    return 0;
24 }

```

Output:

```

elements in mymm:
y => 20
z => 40

```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<a href="#">multimap::equal_range</a>	Get range of equal elements (public member function )
<a href="#">multimap::count</a>	Count elements with a specific key (public member function )
<a href="#">multimap::lower_bound</a>	Return iterator to lower bound (public member function )
<a href="#">multimap::upper_bound</a>	Return iterator to upper bound (public member function )

## /map/multimap/get\_allocator

public member function

### std::multimap::get\_allocator

<map>

```

allocator_type get_allocator() const;
allocator_type get_allocator() const noexcept;

```

#### Get allocator

Returns a copy of the allocator object associated with the `multimap`.

## Parameters

none

## Return Value

The allocator.

Member type `allocator_type` is the type of the allocator used by the container, defined in `multimap` as an alias of its fourth template parameter (`Alloc`).

## Example

```

1 // map::get_allocator
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     int psize;
8     std::multimap<char,int> mymm;
9     std::pair<const char,int>* p;
10
11    // allocate an array of 5 elements using mymm's allocator:
12    p=mymm.get_allocator().allocate(5);
13
14    // assign some values to array

```

```

15     psizes = sizeof(std::multimap<char,int>::value_type)*5;
16     std::cout << "The allocated array has a size of " << psizes << " bytes.\n";
17     mymm.get_allocator().deallocate(p,5);
18
19     return 0;
20 }
21 }
```

The example shows an elaborate way to allocate memory for an array of pairs using the same allocator used by the container.  
A possible output is:

The allocated array has a size of 40 bytes.

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

Copying any instantiation of the [default allocator](#) is also guaranteed to never throw.

## See also

<a href="#">allocator</a>	Default allocator (class template )
---------------------------	-------------------------------------

# /map/multimap/insert

public member function

## std::multimap::insert

<map>

single element (1)	iterator insert (const value_type& val);
with hint (2)	iterator insert (iterator position, const value_type& val);
range (3)	template <class InputIterator> void insert (InputIterator first, InputIterator last);
single element (1)	iterator insert (const value_type& val); template <class P> iterator insert (P&& val);
with hint (2)	iterator insert (const_iterator position, const value_type& val); template <class P> iterator insert (const_iterator position, P&& val);
range (3)	template <class InputIterator> void insert (InputIterator first, InputIterator last);
initializer list (4)	void insert (initializer_list<value_type> il);

### Insert element

Extends the container by inserting new elements, effectively increasing the container [size](#) by the number of elements inserted.

Internally, [multimap](#) containers keep all their elements sorted by key following the criterion specified by its [comparison object](#). The elements are always inserted in its respective position following this ordering.

There are no guarantees on the relative order of equivalent elements.

The relative ordering of elements with equivalent keys is preserved, and newly inserted elements follow those with equivalent keys already in the container.

The parameters determine how many elements are inserted and to which values they are initialized:

## Parameters

### val

Value to be copied to (or moved as) the inserted element.

Member type [value\\_type](#) is the type of the elements in the container, defined in [multimap](#) as [pair<const key\\_type,mapped\\_type>](#) (see [multimap member types](#)).

The template parameter [P](#) shall be a type convertible to [value\\_type](#).

The signatures taking an argument of type [P&&](#) are only called if [std::is\\_constructible<value\\_type,P&&>](#) is true.

If [P](#) is instantiated as a reference type, the argument is copied.

### position

Hint for the position where the element can be inserted.

The function optimizes its insertion time if [position](#) points to the element that will [precede](#) the inserted element.

The function optimizes its insertion time if [position](#) points to the element that will [follow](#) the inserted element (or to the [end](#), if it would be the last).

Notice that this is just a hint and does not force the new element to be inserted at that position within the [multimap](#) container (the elements in a [multimap](#) always follow a specific order depending on their key).

Member types [iterator](#) and [const\\_iterator](#) are defined in [multimap](#) as [bidirectional iterator](#) types that point to elements.

### first, last

Iterators specifying a range of elements. Copies of the elements in the range [\[first,last\)](#) are inserted in the container.

Notice that the range includes all the elements between [first](#) and [last](#), including the element pointed by [first](#) but not the one pointed by [last](#).

The function template argument `InputIterator` shall be an input iterator type that points to elements of a type from which `value_type` objects can be constructed.

i1

An `initializer_list` object. Copies of these elements are inserted.  
These objects are automatically constructed from `initializer_list` declarators.  
Member type `value_type` is the type of the elements contained in the container, defined in `multimap` as `pair<const key_type, mapped_type>` (see `multimap` member types).

## Return value

In the versions returning a value, this is an iterator pointing to the newly inserted element in the `multiset`.

Member type `iterator` is a `bidirectional iterator` type that points to elements.

## Example

```
1 // multimap::insert (C++98)
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8     std::multimap<char,int>::iterator it;
9
10    // first insert function version (single parameter):
11    mymultimap.insert ( std::pair<char,int>('a',100) );
12    mymultimap.insert ( std::pair<char,int>('z',150) );
13    it=mymultimap.insert ( std::pair<char,int>('b',75) );
14
15    // second insert function version (with hint position):
16    mymultimap.insert ( it, std::pair<char,int>('c',300)); // max efficiency inserting
17    mymultimap.insert ( it, std::pair<char,int>('z',400)); // no max efficiency inserting
18
19    // third insert function version (range insertion):
20    std::multimap<char,int> anothermultimap;
21    anothermultimap.insert(mymultimap.begin(),mymultimap.find('c'));
22
23    // showing contents:
24    std::cout << "mymultimap contains:\n";
25    for (it=mymultimap.begin(); it!=mymultimap.end(); ++it)
26        std::cout << (*it).first << " => " << (*it).second << '\n';
27
28    std::cout << "anothermultimap contains:\n";
29    for (it=anothermultimap.begin(); it!=anothermultimap.end(); ++it)
30        std::cout << (*it).first << " => " << (*it).second << '\n';
31
32    return 0;
33 }
```

Output:

```
mymultimap contains:
a => 100
b => 75
c => 300
z => 400
z => 150
anothermultimap contains:
a => 100
b => 75
```

## Complexity

If a single element is inserted, logarithmic in `size` in general, but amortized constant if a hint is given and the `position` given is the optimal.

If N elements are inserted,  $N \log(\text{size}+N)$  in general, but linear in `size+N` if the elements are already sorted according to the same ordering criterion used by the container.

If N elements are inserted,  $N \log(\text{size}+N)$ .

Implementations may optimize if the range is already sorted.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

If a single element is to be inserted, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if an invalid `position` is specified, it causes undefined behavior.

## See also

<code>multimap::find</code>	Get iterator to element (public member function )
<code>multimap::erase</code>	Erase elements (public member function )

# /map/multimap/key\_comp

public member function

## std::multimap::key\_comp

<map>

key\_compare key\_comp() const;

### Return key comparison object

Returns a copy of the *comparison object* used by the container to compare keys.

By default, this is a `less` object, which returns the same as `operator<`.

This object determines the order of the elements in the container: it is a function pointer or a function object that takes two arguments of the same type as the element keys, and returns `true` if the first argument is considered to go before the second in the *strict weak ordering* it defines, and `false` otherwise.

Two keys are considered equivalent if `key_comp` returns `false` reflexively (i.e., no matter the order in which the keys are passed as arguments).

### Parameters

none

### Return value

The comparison object.

Member type `key_compare` is the type of the *comparison object* associated to the container, defined in `multimap` as an alias of its third template parameter (`Compare`).

### Example

```
1 // multimap::key_comp
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     std::multimap<char,int>::key_compare mycomp = mymultimap.key_comp();
10
11    mymultimap.insert (std::make_pair('a',100));
12    mymultimap.insert (std::make_pair('b',200));
13    mymultimap.insert (std::make_pair('b',211));
14    mymultimap.insert (std::make_pair('c',300));
15
16    std::cout << "mymultimap contains:\n";
17
18    char highest = mymultimap.rbegin()->first;      // key value of last element
19
20    std::multimap<char,int>::iterator it = mymultimap.begin();
21    do {
22        std::cout << (*it).first << " => " << (*it).second << '\n';
23    } while ( mycomp((*it++).first, highest) );
24
25    std::cout << '\n';
26
27    return 0;
28 }
```

Output:

```
mymultimap contains:
a => 100
b => 200
b => 211
c => 300
```

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

### See also

<a href="#">multimap::value_comp</a>	Return value comparison object (public member function )
<a href="#">multimap::find</a>	Get iterator to element (public member function )
<a href="#">multimap::count</a>	Count elements with a specific key (public member function )

**multimap::lower\_bound** Return iterator to lower bound ([public member function](#))

**multimap::upper\_bound** Return iterator to upper bound ([public member function](#))

## /map/multimap/lower\_bound

public member function

### std::multimap::lower\_bound

<map>

```
iterator lower_bound (const key_type& k);
const_iterator lower_bound (const key_type& k) const;
```

#### Return iterator to lower bound

Returns an iterator pointing to the first element in the container whose key is not considered to go before *k* (i.e., either it is equivalent or goes after).

The function uses its internal comparison object ([key\\_comp](#)) to determine this, returning an iterator to the first element for which [key\\_comp\(element\\_key, k\)](#) would return `false`.

If the [multimap](#) class is instantiated with the default comparison type ([less](#)), the function returns an iterator to the first element whose key is not less than *k*.

A similar member function, [upper\\_bound](#), has the same behavior as [lower\\_bound](#), except in the case that the [multimap](#) contains elements with keys equivalent to *k*: In this case, [lower\\_bound](#) returns an iterator pointing to the first of such elements, whereas [upper\\_bound](#) returns an iterator pointing to the element following the last.

#### Parameters

*k*

Key to search for.

Member type [key\\_type](#) is the type of the elements in the container, defined in [map](#) as an alias of its first template parameter ([key](#)).

#### Return value

An iterator to the the first element in the container whose key is not considered to go before *k*, or [multimap::end](#) if all keys are considered to go before *k*.

If the [multimap](#) object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an iterator.

Member types [iterator](#) and [const\\_iterator](#) are [bidirectional iterator](#) types pointing to elements (of type [value\\_type](#)). Notice that [value\\_type](#) in [multimap](#) containers is itself also a pair type: `pair<const key_type, mapped_type>`.

#### Example

```
1 // multimap::lower_bound/upper_bound
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8     std::multimap<char,int>::iterator it,itlow,itup;
9
10    mymultimap.insert(std::make_pair('a',10));
11    mymultimap.insert(std::make_pair('b',121));
12    mymultimap.insert(std::make_pair('c',1001));
13    mymultimap.insert(std::make_pair('c',2002));
14    mymultimap.insert(std::make_pair('d',11011));
15    mymultimap.insert(std::make_pair('e',44));
16
17    itlow = mymultimap.lower_bound ('b'); // itlow points to b
18    itup = mymultimap.upper_bound ('d'); // itup points to e (not d)
19
20    // print range [itlow,itup):
21    for (it=itlow; it!=itup; ++it)
22        std::cout << (*it).first << " => " << (*it).second << '\n';
23
24    return 0;
25 }
```

```
b => 121
c => 1001
c => 2002
d => 11011
```

#### Complexity

Logarithmic in [size](#).

#### Iterator validity

No changes.

#### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container). No mapped values are accessed: concurrently accessing or modifying elements is safe.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<b>multimap::upper_bound</b>	Return iterator to upper bound (public member function )
<b>multimap::equal_range</b>	Get range of equal elements (public member function )
<b>multimap::find</b>	Get iterator to element (public member function )
<b>multimap::count</b>	Count elements with a specific key (public member function )

## /map/multimap/max\_size

public member function

### std::multimap::max\_size

<map>

```
size_type max_size() const;
size_type max_size() const noexcept;
```

#### Return maximum size

Returns the maximum number of elements that the **multimap** container can hold.

This is the maximum potential **size** the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size: it can still fail to allocate storage at any point before that size is reached.

#### Parameters

none

#### Return Value

The maximum number of elements a **multimap** container can hold as content.

Member type **size\_type** is an unsigned integral type.

#### Example

```
1 // multimap::max_size
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<int,int> mymultimap;
8
9     if (mymultimap.max_size() >= 1000u)
10    {
11        for (int i=0; i<1000; i++)
12            mymultimap.insert(std::make_pair(i,0));
13        std::cout << "The multimap contains 1000 elements.\n";
14    }
15    else std::cout << "The multimap could not hold 1000 elements.\n";
16
17    return 0;
18 }
```

Here, member **max\_size** is used to check beforehand whether the multimap will allow for 1000 elements to be inserted.

#### Complexity

Constant.

#### Iterator validity

No changes.

#### Data races

The container is accessed.

No elements are accessed: concurrently accessing or modifying them is safe.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

#### See also

<b>multimap::size</b>	Return container size (public member function )
-----------------------	---

## /map/multimap/multimap

public member function

### std::multimap::multimap

<map>

```

empty (1) explicit multimap (const key_compare& comp = key_compare(),
                             const allocator_type& alloc = allocator_type());
template <class InputIterator>
range (2)     multimap (InputIterator first, InputIterator last,
                       const key_compare& comp = key_compare(),
                       const allocator_type& alloc = allocator_type());
copy (3)     multimap (const multimap& x);

empty (1)     explicit multimap (const key_compare& comp = key_compare(),
                                 const allocator_type& alloc = allocator_type());
explicit multimap (const allocator_type& alloc);
template <class InputIterator>
range (2)     multimap (InputIterator first, InputIterator last,
                       const key_compare& comp = key_compare(),
                       const allocator_type& alloc = allocator_type());
copy (3)     multimap (const multimap& x);
multimap (const multimap& x, const allocator_type& alloc);
move (4)     multimap (multimap& x, const allocator_type& alloc);
multimap (initializer_list<value_type> il,
          const key_compare& comp = key_compare(),
          const allocator_type& alloc = allocator_type());

multimap();
empty (1)     explicit multimap (const key_compare& comp,
                                 const allocator_type& alloc = allocator_type());
explicit multimap (const allocator_type& alloc);
template <class InputIterator>
range (2)     multimap (InputIterator first, InputIterator last,
                       const key_compare& comp = key_compare(),
                       const allocator_type& alloc = allocator_type());
template <class InputIterator>
range (2)     multimap (InputIterator first, InputIterator last,
                       const allocator_type& alloc = allocator_type());
copy (3)     multimap (const multimap& x);
multimap (const multimap& x, const allocator_type& alloc);
move (4)     multimap (multimap& x, const allocator_type& alloc);
multimap (multimap& x, const allocator_type& alloc);
multimap (initializer_list<value_type> il,
          const key_compare& comp = key_compare(),
          const allocator_type& alloc = allocator_type());
initializer list (5) multimap (initializer_list<value_type> il,
                               const allocator_type& alloc = allocator_type());

```

## Construct multimap

Constructs a `multimap` container object, initializing its contents depending on the constructor version used:

### (1) empty container constructor (default constructor)

Constructs an `empty` container, with no elements.

### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element constructed from its corresponding element in that range.

### (3) copy constructor

Constructs a container with a copy of each of the elements in `x`.

The container keeps an internal copy of `alloc` and `comp`, which are used to allocate storage and to sort the elements throughout its lifetime.  
The copy constructor (3) creates a container that keeps and uses copies of `x`'s `allocator` and `comparison object`.

The storage for the elements is allocated using this `internal allocator`.

The elements are sorted according to the `comparison object`.

### (1) empty container constructors (default constructor)

Constructs an `empty` container, with no elements.

### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element `emplace-constructed` from its corresponding element in that range.

### (3) copy constructor (and copying with allocator)

Constructs a container with a copy of each of the elements in `x`.

### (4) move constructor (and moving with allocator)

Constructs a container that acquires the elements of `x`.

If `alloc` is specified and is different from `x`'s allocator, the elements are moved. Otherwise, no elements are constructed (their ownership is directly transferred).

`x` is left in an unspecified but valid state.

### (5) initializer list constructor

Constructs a container with a copy of each of the elements in `il`.

The container keeps an internal copy of `alloc`, which is used to allocate and deallocate storage for its elements, and to construct and destroy them (as specified by its `allocator_traits`). If no `alloc` argument is passed to the constructor, a default-constructed allocator is used, except in the following cases:

- The copy constructor (3, *first signature*) creates a container that keeps and uses a copy of the allocator returned by calling the appropriate `selected_on_container_copy_construction` trait on `x`'s allocator.

- The move constructor (4, *first signature*) acquires `x`'s allocator.

The container also keeps an internal copy of `comp` (or `x`'s `comparison object`), which is used to establish the order of the elements in the container and to check for elements with equivalent keys.

All elements are `copied`, `moved` or otherwise `constructed` by calling `allocator_traits::construct` with the appropriate arguments.

The elements are sorted according to the `comparison object`. If elements with equivalent keys are passed to the constructor, their relative order is preserved.

## Parameters

**comp**  
 Binary predicate that, taking two *element keys* as argument, returns `true` if the first argument goes before the second argument in the *strict weak ordering* it defines, and `false` otherwise.  
 This shall be a function pointer or a function object.  
 Member type `key_compare` is the internal comparison object type used by the container, defined in `multimap` as an alias of its third template parameter (`Compare`).  
 If `key_compare` uses the default `less` (which has no state), this parameter is not relevant.

**alloc**  
 Allocator object.  
 The container keeps and uses an internal copy of this allocator.  
 Member type `allocator_type` is the internal allocator type used by the container, defined in `multimap` as an alias of its fourth template parameter (`Alloc`).  
 If `allocator_type` is an instantiation of the default `allocator` (which has no state), this parameter is not relevant.

**first, last**  
 Input iterators to the initial and final positions in a range. The range used is `[first, last)`, which includes all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
 The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type from which `value_type` objects can be constructed (in `multimap`, `value_type` is an alias of `pair<const key_type, mapped_type>`)

**x**  
 Another `multimap` object of the same type (with the same class template arguments `Key`, `T`, `Compare` and `Alloc`), whose contents are either copied or acquired.

**il**  
 An `initializer_list` object.  
 These objects are automatically constructed from `initializer_list` declarators.  
 Member type `value_type` is the type of the elements in the container, defined in `multimap` as an alias of `pair<const key_type, mapped_type>` (see `multimap types`).

## Example

```

1 // constructing multimaps
2 #include <iostream>
3 #include <map>
4
5 bool fncomp (char lhs, char rhs) {return lhs<rhs;}
6
7 struct classcomp {
8     bool operator() (const char& lhs, const char& rhs) const
9     {return lhs<rhs;}
10 };
11
12 int main ()
13 {
14     std::multimap<char,int> first;
15
16     first.insert(std::pair<char,int>('a',10));
17     first.insert(std::pair<char,int>('b',15));
18     first.insert(std::pair<char,int>('b',20));
19     first.insert(std::pair<char,int>('c',25));
20
21     std::multimap<char,int> second (first.begin(),first.end());
22
23     std::multimap<char,int> third (second);
24
25     std::multimap<char,int,classcomp> fourth;           // class as Compare
26
27     bool(*fn_pt)(char,char) = fncomp;
28     std::multimap<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as comp
29
30     return 0;
31 }
```

The code does not produce any output, but demonstrates some ways in which a `multimap` container can be constructed.

## Complexity

Constant for the *empty constructors* (1), and for the *move constructors* (4) (unless `alloc` is different from `x`'s allocator).  
 For all other cases, linear in the distance between the iterators (copy constructions) if the elements are already sorted according to the same criterion. For unsorted sequences, linearithmic ( $N \log N$ ) in that distance (sorting,copy constructions).

## Iterator validity

The *move constructors* (4), invalidate all iterators, pointers and references related to `x` if the elements are moved.

## Data races

All copied elements are accessed.  
 The *move constructors* (4) modify `x`.

## Exception safety

**Strong guarantee:** no effects in case an exception is thrown.  
 If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

## See also

<code>multimap::operator=</code>	Copy container content (public member function )
<code>multimap::insert</code>	Insert element (public member function )

## /map/multimap/~multimap

public member function

### std::multimap::~multimap

<map>

`~multimap();`

#### Multimap destructor

Destroys the container object.

This destroys all container elements, and deallocates all the storage capacity allocated by the `multimap` container using its `allocator`.

This calls `allocator_traits::destroy` on each of the contained elements, and deallocates all the storage capacity allocated by the `multimap` container using its `allocator`.

#### Complexity

Linear in `multimap::size` (destructors).

#### Iterator validity

All iterators, pointers and references are invalidated.

#### Data races

The container and all its elements are modified.

#### Exception safety

**No-throw guarantee:** never throws exceptions.

## /map/multimap/operator=

public member function

### std::multimap::operator=

<map>

```
copy (1) multimap& operator= (const multimap& x);
copy (1) multimap& operator= (const multimap& x);
move (2) multimap& operator= (multimap& x);
initializer list (3) multimap& operator= (initializer_list<value_type> il);
```

#### Copy container content

Assigns new contents to the container, replacing its current content.

Copies all the elements from `x` into the container, changing its `size` accordingly.

The container preserves its `current allocator`, which is used to allocate additional storage if needed.

The `copy assignment` (1) copies all the elements from `x` into the container (with `x` preserving its contents).

The `move assignment` (2) moves the elements of `x` into the container (`x` is left in an unspecified but valid state).

The `initializer list assignment` (3) copies the elements of `il` into the container.

The new container `size` is the same as the `size` of `x` (or `il`) before the call.

The container preserves its `current allocator`, except if the `allocator traits` indicate `x`'s allocator should `propagate`. This `allocator` is used (through its `traits`) to `allocate` or `deallocate` if there are changes in storage requirements, and to `construct` or `destroy` elements, if needed.

The elements stored in the container before the call are either assigned to or destroyed.

#### Parameters

`x`

A `multimap` object of the same type (i.e., with the same template parameters, `key`, `T`, `Compare` and `Alloc`).

`il`

An `initializer_list` object. The compiler will automatically construct such objects from `initializer_list` declarators.

Member type `value_type` is the type of the elements in the container, defined in `multimap` as an alias of `pair<const key_type, mapped_type>` (see `multimap` member types).

#### Return value

`*this`

#### Example

```
1 // assignment operator with multimaps
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> foo,bar;
```

```

9  foo.insert(std::make_pair('x',32));
10 foo.insert(std::make_pair('y',64));
11 foo.insert(std::make_pair('y',96));
12 foo.insert(std::make_pair('z',128));
13
14 bar = foo;           // bar now contains 4 ints
15 foo.clear();         // and first is now empty
16
17 std::cout << "Size of foo: " << foo.size() << '\n';
18 std::cout << "Size of bar: " << bar.size() << '\n';
19 return 0;
20 }
```

Output:

```
Size of foo: 0
Size of bar: 4
```

## Complexity

For the *copy assignment* (1): Linear in sizes (destructions, copies).

For the *move assignment* (2): Linear in current container `size` (destructions).\*

For the *initializer list assignment* (3): Up to logarithmic in sizes (destructions, move-assignments) -- linear if *il* is already sorted.

\* Additional complexity for assignments if allocators do not *propagate*.

## Iterator validity

All iterators, references and pointers related to this container are invalidated.

In the *move assignment*, iterators, pointers and references referring to elements in *x* are also invalidated.

## Data races

All copied elements are accessed.

The *move assignment* (2) modifies *x*.

The container and all its elements are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if `value_type` is not `copy assignable` (or `move assignable` for (2)), it causes *undefined behavior*.

## See also

<code>multimap::insert</code>	Insert element (public member function )
<code>multimap::find</code>	Get iterator to element (public member function )
<code>multimap::multimap</code>	Construct multimap (public member function )

# /map/multimap/operators

function

## std::relational operators (multimap)

<map>

```

template <class Key, class T, class Compare, class Alloc>
(1)  bool operator== ( const multimap<Key,T,Compare,Alloc>& lhs,
                      const multimap<Key,T,Compare,Alloc>& rhs );
template <class Key, class T, class Compare, class Alloc>
(2)  bool operator!= ( const multimap<Key,T,Compare,Alloc>& lhs,
                      const multimap<Key,T,Compare,Alloc>& rhs );
template <class Key, class T, class Compare, class Alloc>
(3)  bool operator< ( const multimap<Key,T,Compare,Alloc>& lhs,
                      const multimap<Key,T,Compare,Alloc>& rhs );
template <class Key, class T, class Compare, class Alloc>
(4)  bool operator<= ( const multimap<Key,T,Compare,Alloc>& lhs,
                      const multimap<Key,T,Compare,Alloc>& rhs );
template <class Key, class T, class Compare, class Alloc>
(5)  bool operator> ( const multimap<Key,T,Compare,Alloc>& lhs,
                      const multimap<Key,T,Compare,Alloc>& rhs );
template <class Key, class T, class Compare, class Alloc>
(6)  bool operator>= ( const multimap<Key,T,Compare,Alloc>& lhs,
                      const multimap<Key,T,Compare,Alloc>& rhs );
```

### Relational operators for multimap

Performs the appropriate comparison operation between the `multimap` containers *lhs* and *rhs*.

The *equality comparison* (`operator==`) is performed by first comparing `sizes`, and if they match, the elements are compared sequentially using `operator==`, stopping at the first mismatch (as if using algorithm `equal`).

The *less-than comparison* (`operator<`) behaves as if using algorithm `lexicographical_compare`, which compares the elements sequentially using `operator<` in a reciprocal manner (i.e., checking both  $a < b$  and  $b < a$ ) and stopping at the first occurrence.

The other operations also use the operators == and < internally to compare the elements, behaving as if the following equivalent operations were performed:

operation	equivalent operation
<code>a != b</code>	<code>!(a == b)</code>
<code>a &gt; b</code>	<code>b &lt; a</code>
<code>a &lt;= b</code>	<code>!(b &lt; a)</code>
<code>a &gt;= b</code>	<code>!(a &lt; b)</code>

Notice that none of these operations take into consideration the [internal comparison object](#) of neither container, but compare the elements (of type `value_type`) directly.

`value_type` is a [pair](#) type, and as such, by default, two elements will compare equal only if both their `key` and `mapped value` compare equal, and one compare lower than the other only if the first `key` is lower, or if the `keys` are equivalent and the `mapped value` is lower.

These operators are overloaded in header .

## Parameters

`lhs, rhs`  
multimap containers (to the left- and right-hand side of the operator, respectively), having both the same template parameters (`key`, `T`, `Compare` and `Alloc`).

## Example

```
1 // multimap comparisons
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> foo,bar;
8
9     foo.insert (std::make_pair('a',100));
10    foo.insert (std::make_pair('z',900));
11    bar.insert (std::make_pair('b',250));
12    bar.insert (std::make_pair('b',350));
13
14    // foo ({{a,100},{z,900}}) vs bar ({{b,250},{b,350}}):
15    if (foo==bar) std::cout << "foo and bar are equal\n";
16    if (foo!=bar) std::cout << "foo and bar are not equal\n";
17    if (foo< bar) std::cout << "foo is less than bar\n";
18    if (foo> bar) std::cout << "foo is greater than bar\n";
19    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
20    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
21
22    return 0;
23 }
```

Output:

```
foo and bar are not equal
foo is less than bar
foo is less than or equal to bar
```

## Return Value

true if the condition holds, and false otherwise.

## Complexity

Up to linear in the `size` of `lhs` and `rhs`.

For (1) and (2), constant if the `sizes` of `lhs` and `rhs` differ, and up to linear in that `size` (equality comparisons) otherwise.  
For the others, up to linear in the smaller `size` (each representing two comparisons with `operator<`).

## Iterator validity

No changes.

## Data races

Both containers, `lhs` and `rhs`, are accessed.

Up to all of their contained elements may be accessed.

## Exception safety

If the type of the elements supports the appropriate operation with no-throw guarantee, the function never throws exceptions (no-throw guarantee).  
In any case, the function cannot modify its arguments.

## See also

<a href="#">multimap::key_comp</a>	Return key comparison object ( <a href="#">public member function</a> )
<a href="#">multimap::value_comp</a>	Return value comparison object ( <a href="#">public member function</a> )
<a href="#">multimap::operator=</a>	Copy container content ( <a href="#">public member function</a> )
<a href="#">multimap::swap</a>	Swap content ( <a href="#">public member function</a> )

# /map/multimap/rbegin

public member function

## std::multimap::rbegin

<map>

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

```
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
```

### Return reverse iterator to reverse beginning

Returns a *reverse iterator* pointing to the last element in the container (i.e., its *reverse beginning*).

*Reverse iterators* iterate backwards: increasing them moves them towards the beginning of the container.

`rbegin` points to the element preceding the one that would be pointed to by member `end`.

### Parameters

none

### Return Value

A *reverse iterator* to the *reverse beginning* of the sequence container.

If the `multimap` object is `const`-qualified, the function returns a `const_reverse_iterator`. Otherwise, it returns a `reverse_iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are *reverse bidirectional iterator* types pointing to elements. See `multimap` member types.

### Example

```
1 // multimap::rbegin/rend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     mymultimap.insert (std::make_pair('x',10));
10    mymultimap.insert (std::make_pair('y',20));
11    mymultimap.insert (std::make_pair('y',150));
12    mymultimap.insert (std::make_pair('z',9));
13
14    // show content:
15    std::multimap<char,int>::reverse_iterator rit;
16    for (rit=mymultimap.rbegin(); rit!=mymultimap.rend(); ++rit)
17        std::cout << rit->first << " => " << rit->second << '\n';
18
19    return 0;
20 }
```

Output:

```
z => 9
y => 150
y => 20
x => 10
```

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">multimap::rend</a>	Return reverse iterator to reverse end (public member function )
<a href="#">multimap::begin</a>	Return iterator to beginning (public member function )
<a href="#">multimap::end</a>	Return iterator to end (public member function )

## /map/multimap/rend

public member function

### std::multimap::rend

<map>

```
reverse_iterator rend();
const_reverse_iterator rend() const;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

**Return reverse iterator to reverse end**

Returns a *reverse iterator* pointing to the theoretical element right before the first element in the `multimap` container (which is considered its *reverse end*).

The range between `multimap::rbegin` and `multimap::rend` contains all the elements of the container (in reverse order).

## Parameters

none

## Return Value

A reverse iterator to the *reverse end* of the sequence container.

If the `multimap` object is `const`-qualified, the function returns a `const_reverse_iterator`. Otherwise, it returns a `reverse_iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are reverse `bidirectional iterator` types pointing to elements. See `multimap` member types.

## Example

```
1 // multimap::rbegin/rend
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     mymultimap.insert (std::make_pair('x',10));
10    mymultimap.insert (std::make_pair('y',20));
11    mymultimap.insert (std::make_pair('y',150));
12    mymultimap.insert (std::make_pair('z',9));
13
14    // show content:
15    std::multimap<char,int>::reverse_iterator rit;
16    for (rit=mymultimap.rbegin(); rit!=mymultimap.rend(); ++rit)
17        std::cout << rit->first << " => " << rit->second << '\n';
18
19    return 0;
20 }
```

Output:

```
z => 9
y => 150
y => 20
x => 10
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container).

No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multimap::rbegin</a>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<a href="#">multimap::begin</a>	Return iterator to beginning ( <a href="#">public member function</a> )
<a href="#">multimap::end</a>	Return iterator to end ( <a href="#">public member function</a> )

# /map/multimap/size

public member function

## std::multimap::size

<map>

```
size_type size() const;
size_type size() const noexcept;
```

### Return container size

Returns the number of elements in the `multimap` container.

## Parameters

none

## Return Value

The number of elements in the container.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // multimap::size
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     mymultimap.insert(std::make_pair('x',100));
10    mymultimap.insert(std::make_pair('y',200));
11    mymultimap.insert(std::make_pair('y',350));
12    mymultimap.insert(std::make_pair('z',500));
13
14    std::cout << "mymultimap.size() is " << mymultimap.size() << '\n';
15
16    return 0;
17 }
```

Output:

```
mymultimap.size() is 4
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">multimap::max_size</a>	Return maximum size ( <a href="#">public member function</a> )
------------------------------------	--

<a href="#">multimap::empty</a>	Test whether container is empty ( <a href="#">public member function</a> )
---------------------------------	--

# /map/multimap/swap

public member function

## std::multimap::swap

<map>

`void swap (multimap& x);`

### Swap content

Exchanges the content of the container by the content of `x`, which is another `multimap` of the same type. Sizes may differ.

After the call to this member function, the elements in this container are those which were in `x` before the call, and the elements of `x` are those which were in this. All iterators, references and pointers remain valid for the swapped objects.

Notice that a non-member function exists with the same name, `swap`, overloading that algorithm with an optimization that behaves like this member function.

Whether the internal container <code>allocators</code> and <code>comparison objects</code> are swapped is undefined.
--

Whether the internal container <code>allocators</code> are swapped is not defined, unless in the case the appropriate allocator trait indicates explicitly that they shall propagate.
---

The internal <code>comparison objects</code> are always exchanged, using <code>swap</code> .
--

## Parameters

x

Another `multimap` container of the same type as this (i.e., with the same template parameters, `Key`, `T`, `Compare` and `Alloc`) whose content is swapped with that of this container.

## Return value

none

## Example

```

1 // swap multimaps
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> foo,bar;
8
9     foo.insert(std::make_pair('x',100));
10    foo.insert(std::make_pair('y',200));
11
12    bar.insert(std::make_pair('a',11));
13    bar.insert(std::make_pair('b',22));
14    bar.insert(std::make_pair('a',55));
15
16    foo.swap(bar);
17
18    std::cout << "foo contains:\n";
19    for (std::multimap<char,int>::iterator it=foo.begin(); it!=foo.end(); ++it)
20        std::cout << (*it).first << " => " << (*it).second << '\n';
21
22    std::cout << "bar contains:\n";
23    for (std::multimap<char,int>::iterator it=bar.begin(); it!=bar.end(); ++it)
24        std::cout << (*it).first << " => " << (*it).second << '\n';
25
26    return 0;
27 }

```

Output:

```

foo contains:
a => 11
a => 55
b => 22
bar contains:
x => 100
y => 200

```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.

Note that the *end iterators* do not refer to elements and may be invalidated.

## Data races

Both the container and *x* are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both containers compare equal, or if their *allocator traits* indicate that the allocators shall *propagate*, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">swap (multimap)</a>	Exchanges the contents of two multimaps ( <a href="#">function template</a> )
<a href="#">swap_ranges</a>	Exchange values of two ranges ( <a href="#">function template</a> )

## /map/multimap/swap-free

function template

### std::swap (multimap)

<map>

```

template <class Key, class T, class Compare, class Alloc>
void swap (multimap<Key,T,Compare,Alloc>& x, multimap<Key,T,Compare,Alloc>& y);

```

#### Exchanges the contents of two multimaps

The contents of container *x* are exchanged with those of *y*. Both container objects must be of the same type (same template parameters), although sizes may differ.

After the call to this member function, the elements in *x* are those which were in *y* before the call, and the elements of *y* are those which were in *x*. All iterators, references and pointers remain valid for the swapped objects.

This is an overload of the generic algorithm [swap](#) that improves its performance by mutually transferring ownership over their assets to the other container (i.e., the containers exchange references to their data, without actually performing any element copy or movement): It behaves as if *x.swap(y)* was called.

## Parameters

*x,y*

multimap containers of the same type (i.e., having both the same template parameters: *Key*, *T*, *Compare* and *Alloc*).

## Return value

none

## Example

```
1 // swap multimaps
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> foo,bar;
8
9     foo.insert(std::make_pair('x',100));
10    foo.insert(std::make_pair('y',200));
11
12    bar.insert(std::make_pair('a',11));
13    bar.insert(std::make_pair('b',22));
14    bar.insert(std::make_pair('a',55));
15
16    swap(foo,bar);
17
18    std::cout << "foo contains:\n";
19    for (std::multimap<char,int>::iterator it=foo.begin(); it!=foo.end(); ++it)
20        std::cout << (*it).first << " => " << (*it).second << '\n';
21
22    std::cout << "bar contains:\n";
23    for (std::multimap<char,int>::iterator it=bar.begin(); it!=bar.end(); ++it)
24        std::cout << (*it).first << " => " << (*it).second << '\n';
25
26    return 0;
27 }
```

Output:

```
foo contains:
a => 11
a => 55
b => 22
bar contains:
x => 100
y => 200
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.

Note that the *end iterators* do not refer to elements and may be invalidated.

## Data races

Both containers, *x* and *y*, are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both *multimaps* compare equal, or if their *allocator traits* indicate that the allocators shall *propagate*, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">multimap::swap</a>	Swap content (public member function )
<a href="#">swap</a>	Exchange values of two objects (function template )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )

## /map/multimap/upper\_bound

public member function

### std::multimap::upper\_bound

<map>

```
iterator upper_bound (const key_type& k);
const_iterator upper_bound (const key_type& k) const;
```

#### Return iterator to upper bound

Returns an iterator pointing to the first element in the container whose key is considered to go after *k*.

The function uses its internal *comparison object* (*key\_comp*) to determine this, returning an iterator to the first element for which *key\_comp(k,element\_key)* would return *true*.

If the *multimap* class is instantiated with the default comparison type (*less*), the function returns an iterator to the first element whose key is greater than *k*.

A similar member function, *lower\_bound*, has the same behavior as *upper\_bound*, except in the case that the *multimap* contains elements with keys equivalent to *k*: In this case *lower\_bound* returns an iterator pointing to the first of such elements, whereas *upper\_bound* returns an iterator pointing to the element following the last.

## Parameters

---

k

Key to search for.

Member type `key_type` is the type of the elements in the container, defined in `multimap` as an alias of its first template parameter (`key`).

## Return value

---

An iterator to the the first element in the container whose key is considered to go after `k`, or `multimap::end` if no keys are considered to go after `k`.

If the `multimap` object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are `bidirectional iterator` types pointing to elements.

## Example

---

```
1 // multimap::lower_bound/upper_bound
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8     std::multimap<char,int>::iterator it,itlow,itup;
9
10    mymultimap.insert(std::make_pair('a',10));
11    mymultimap.insert(std::make_pair('b',121));
12    mymultimap.insert(std::make_pair('c',1001));
13    mymultimap.insert(std::make_pair('c',2002));
14    mymultimap.insert(std::make_pair('d',11011));
15    mymultimap.insert(std::make_pair('e',44));
16
17    itlow = mymultimap.lower_bound ('b'); // itlow points to b
18    itup = mymultimap.upper_bound ('d'); // itup points to e (not d)
19
20    // print range [itlow,itup]:
21    for (it=itlow; it!=itup; ++it)
22        std::cout << (*it).first << " => " << (*it).second << '\n';
23
24    return 0;
25 }
```

```
b => 121
c => 1001
c => 2002
d => 11011
```

## Complexity

---

Logarithmic in `size`.

## Iterator validity

---

No changes.

## Data races

---

The container is accessed (neither the const nor the non-const versions modify the container).  
No mapped values are accessed: concurrently accessing or modifying elements is safe.

## Exception safety

---

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

---

<code>multimap::lower_bound</code>	Return iterator to lower bound (public member function )
<code>multimap::equal_range</code>	Get range of equal elements (public member function )
<code>multimap::find</code>	Get iterator to element (public member function )
<code>multimap::count</code>	Count elements with a specific key (public member function )

## /map/multimap/value\_comp

public member function

**std::multimap::value\_comp**

`<map>`

`value_compare value_comp() const;`

### Return value comparison object

Returns a comparison object that can be used to compare two elements to get whether the `key` of the first one goes before the second.

The arguments taken by this function object are of member type `value_type` (defined in `multimap` as an alias of `pair<const key_type,mapped_type>`), but the `mapped_type` part of the value is not taken into consideration in this comparison.

The comparison object returned is an object of the member type `multimap::value_compare`, which is a nested class that uses the internal `comparison object` to

generate the appropriate comparison functional class. It is defined with the same behavior as:

```
1 template <class Key, class T, class Compare, class Alloc>
2 class multimap<Key,T,Compare,Alloc>::value_compare
3 { // in C++98, it is required to inherit binary_function<value_type,value_type,bool>
4     friend class multimap;
5 protected:
6     Compare comp;
7     value_compare (Compare c) : comp(c) {} // constructed with multimap's comparison object
8 public:
9     typedef bool result_type;
10    typedef value_type first_argument_type;
11    typedef value_type second_argument_type;
12    bool operator() (const value_type& x, const value_type& y) const
13    {
14        return comp(x.first, y.first);
15    }
16 }
```

The public member of this comparison class returns `true` if the key of the first argument is considered to go before that of the second (according to the *strict weak ordering* specified by the container's `comparison object`, `key_comp`), and `false` otherwise.

Notice that `value_compare` has no public constructor, therefore no objects can be directly created from this nested class outside `multimap` members.

## Parameters

none

## Return value

The comparison object for element values.

Member type `value_compare` is a nested class type (described above).

## Example

```
1 // multimap::value_comp
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::multimap<char,int> mymultimap;
8
9     mymultimap.insert(std::make_pair('x',101));
10    mymultimap.insert(std::make_pair('y',202));
11    mymultimap.insert(std::make_pair('y',252));
12    mymultimap.insert(std::make_pair('z',303));
13
14    std::cout << "mymultimap contains:\n";
15
16    std::pair<char,int> highest = *mymultimap.rbegin();           // last element
17
18    std::multimap<char,int>::iterator it = mymultimap.begin();
19    do {
20        std::cout << (*it).first << " => " << (*it).second << '\n';
21    } while ( mymultimap.value_comp()(*it++, highest) );
22
23    return 0;
24 }
```

Output:

```
mymultimap contains:
x => 101
y => 202
y => 252
z => 303
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

No contained elements are accessed: concurrently accessing or modifying them is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<a href="#">multimap::key_comp</a>	Return key comparison object ( <a href="#">public member function</a> )
<a href="#">multimap::find</a>	Get iterator to element ( <a href="#">public member function</a> )
<a href="#">multimap::count</a>	Count elements with a specific key ( <a href="#">public member function</a> )
<a href="#">multimap::lower_bound</a>	Return iterator to lower bound ( <a href="#">public member function</a> )

## /queue

header

### <queue>

#### Queue header

Header that defines the `queue` and `priority_queue` container adaptor classes:

#### Classes

<code>queue</code>	FIFO queue (class template )
<code>priority_queue</code>	Priority queue (class template )

## /queue/priority\_queue

class template

### `std::priority_queue`

<queue>

```
template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type> > class priority_queue;
```

#### Priority queue

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some *strict weak ordering* criterion.

This context is similar to a *heap*, where elements can be inserted at any moment, and only the *max heap* element can be retrieved (the one at the top in the *priority queue*).

Priority queues are implemented as *container adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *popped* from the "back" of the specific container, which is known as the *top* of the priority queue.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall be accessible through *random access iterators* and support the following operations:

- `empty()`
- `size()`
- `front()`
- `push_back()`
- `pop_back()`

The standard container classes `vector` and `deque` fulfill these requirements. By default, if no container class is specified for a particular `priority_queue` class instantiation, the standard container `vector` is used.

Support of *random access iterators* is required to keep a heap structure internally at all times. This is done automatically by the container adaptor by automatically calling the algorithm functions `make_heap`, `push_heap` and `pop_heap` when needed.

#### Template parameters

T

Type of the elements.

Aliased as member type `priority_queue::value_type`.

Container

Type of the internal *underlying container* object where the elements are stored.

Its `value_type` shall be T.

Aliased as member type `priority_queue::container_type`.

Compare

A binary predicate that takes two elements (of type T) as arguments and returns a `bool`.

The expression `comp(a,b)`, where `comp` is an object of this type and `a` and `b` are elements in the container, shall return `true` if `a` is considered to go before `b` in the *strict weak ordering* the function defines.

The `priority_queue` uses this function to maintain the elements sorted in a way that preserves *heap properties* (i.e., that the element popped is the last according to this *strict weak ordering*).

This can be a function pointer or a function object, and defaults to `less<T>`, which returns the same as applying the *less-than operator* (`a < b`).

#### Member types

member type	definition	notes
<code>value_type</code>	The first template parameter (T)	Type of the elements
<code>container_type</code>	The second template parameter (Container)	Type of the <i>underlying container</i>
<code>size_type</code>	an unsigned integral type	usually the same as <code>size_t</code>

member type	definition	notes
<code>value_type</code>	The first template parameter (T)	Type of the elements
<code>container_type</code>	The second template parameter (Container)	Type of the <i>underlying container</i>
<code>reference</code>	<code>container_type::reference</code>	usually, <code>value_type&amp;</code>
<code>const_reference</code>	<code>container_type::const_reference</code>	usually, <code>const value_type&amp;</code>
<code>size_type</code>	an unsigned integral type	usually, the same as <code>size_t</code>

## Member functions

(constructor)	Construct priority queue (public member function )
<b>empty</b>	Test whether container is empty (public member function )
<b>size</b>	Return size (public member function )
<b>top</b>	Access top element (public member function )
<b>push</b>	Insert element (public member function )
<b>emplace</b>	Construct and insert element (public member function )
<b>pop</b>	Remove top element (public member function )
<b>swap</b>	Swap contents (public member function )

## Non-member function overloads

<b>swap (queue)</b>	Exchange contents of priority queues (public member function )
---------------------	--

## Non-member class specializations

<b>uses_allocator&lt;queue&gt;</b>	Uses allocator for priority queue (class template )
------------------------------------	---

# /queue/priority\_queue/emplace

public member function

## std::priority\_queue::emplace

<queue>

template <class... Args> void emplace (Args&... args);

### Construct and insert element

Adds a new element to the `priority_queue`. This new element is constructed in place passing `args` as the arguments for its constructor.

This member function effectively calls the member function `emplace_back` of the *underlying container*, *forwarding args*, and then reorders it to its location in the *heap* by calling the `push_heap` algorithm on the range that includes all the elements of the container.

## Parameters

`args`  
Arguments forwarded to construct the new element.

## Return value

none

## Example

```
1 // priority_queue::emplace
2 #include <iostream>           // std::cout
3 #include <queue>              // std::priority_queue
4 #include <string>              // std::string
5
6 int main ()
7 {
8     std::priority_queue<std::string> mypq;
9
10    mypq.emplace("orange");
11    mypq.emplace("strawberry");
12    mypq.emplace("apple");
13    mypq.emplace("pear");
14
15    std::cout << "mypq contains:" ;
16    while ( !mypq.empty() )
17    {
18        std::cout << ' ' << mypq.top();
19        mypq.pop();
20    }
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
mypq contains: strawberry pear orange apple
```

## Complexity

One call to `emplace_back` on the *underlying container* and one call to `push_heap` on the range that includes all the elements of the *underlying container*.

## Data races

The container and up to all its contained elements are modified.

## Exception safety

Provides the same level of guarantees as the operations performed on the *underlying container* object.

## See also

<a href="#">priority_queue::push</a>	Insert element (public member function )
<a href="#">priority_queue::pop</a>	Remove top element (public member function )

# /queue/priority\_queue/empty

public member function

## std::priority\_queue::empty

<queue>

`bool empty() const;`

### Test whether container is empty

Returns whether the [priority\\_queue](#) is empty: i.e. whether its [size](#) is zero.

This member function effectively calls member [empty](#) of the *underlying container* object.

## Parameters

none

## Return Value

true if the *underlying container*'s size is 0, false otherwise.

## Example

```
1 // priority_queue::empty
2 #include <iostream>           // std::cout
3 #include <queue>             // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> mypq;
8     int sum (0);
9
10    for (int i=1;i<=10;i++) mypq.push(i);
11
12    while (!mypoq.empty())
13    {
14        sum += mypq.top();
15        mypq.pop();
16    }
17
18    std::cout << "total: " << sum << '\n';
19
20    return 0;
21 }
```

The example initializes the content of the priority queue to a sequence of numbers (form 1 to 10). It then pops the elements one by one until it is empty and calculates their sum.

Output:

total: 55

## Complexity

Constant (calling [empty](#) on the *underlying container*).

## Data races

The container is accessed.

## Exception safety

Provides the same level of guarantees as the operation performed on the container (no-throw guarantee for standard container types).

## See also

<a href="#">priority_queue::size</a>	Return size (public member function )
--------------------------------------	---------------------------------------

# /queue/priority\_queue/pop

public member function

## std::priority\_queue::pop

<queue>

`void pop();`

### Remove top element

Removes the element on top of the [priority\\_queue](#), effectively reducing its [size](#) by one. The element removed is the one with the highest value.

The value of this element can be retrieved before being popped by calling member [priority\\_queue::top](#).

This member function effectively calls the [pop\\_heap](#) algorithm to keep the *heap property* of priority\_queues and then calls the member function [pop\\_back](#) of the

underlying container object to remove the element.

This calls the removed element's destructor.

## Parameters

none

## Return value

none

## Example

```
1 // priority_queue::push/pop
2 #include <iostream>           // std::cout
3 #include <queue>             // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> mypq;
8
9     mypq.push(30);
10    mypq.push(100);
11    mypq.push(25);
12    mypq.push(40);
13
14    std::cout << "Popping out elements..." ;
15    while ( !mypoq.empty() )
16    {
17        std::cout << ' ' << mypq.top();
18        mypq.pop();
19    }
20    std::cout << '\n';
21
22    return 0;
23 }
```

Output:

```
Popping out elements... 100 40 30 25
```

## Complexity

One call to `pop_heap` and one call to `pop_back` on the *underlying container*.

## Data races

The container and up to all its contained elements are modified.

## Exception safety

Provides the same level of guarantees as the operations performed on the *underlying container* object.

## See also

`priority_queue::push` Insert element ([public member function](#))

`priority_queue::empty` Test whether container is empty ([public member function](#))

# /queue/priority\_queue/priority\_queue

public member function

## std::priority\_queue::priority\_queue

<queue>

```
initialize (1) explicit priority_queue (const Compare& comp = Compare(),
                                         const Container& ctnr = Container());
template <class InputIterator>
range (2) priority_queue (InputIterator first, InputIterator last,
                         const Compare& comp = Compare(),
                         const Container& ctnr = Container());

initialize (1) priority_queue (const Compare& comp, const Container& ctnr);
template <class InputIterator>
range (2) priority_queue (InputIterator first, InputIterator last,
                         const Compare& comp, const Container& ctnr);

move-initialize (3) explicit priority_queue (const Compare& comp = Compare(),
                                             Container&& ctnr = Container());
template <class InputIterator>
move-range (4) priority_queue (InputIterator first, InputIterator last,
                               const Compare& comp, Container&& ctnr = Container());
template <class Alloc> explicit priority_queue (const Alloc& alloc);
template <class Alloc> priority_queue (const Compare& comp, const Alloc& alloc);
template <class Alloc> priority_queue (const Compare& comp, const Container& ctnr,
                                         const Alloc& alloc);

allocator versions (5) template <class Alloc> priority_queue (const Compare& comp, Container&& ctnr,
                                         const Alloc& alloc);
template <class Alloc> priority_queue (const priority_queue& x, const Alloc& alloc);
template <class Alloc> priority_queue (priority_queue& x, const Alloc& alloc);
```

## Construct priority queue

Constructs a `priority_queue` container adaptor object.

A `priority_queue` keeps internally a comparing function and a container object as data, which are copies of `comp` and `ctnr` respectively.

The `range version` (2), on top that, inserts the elements between `first` and `last` (before the container is converted into a `heap`).

A `priority_queue` keeps internally a comparing function and a container object as data. The comparing function is a copy of `comp` and the *underlying container* depends on the constructor used:

### (1) initialization constructor

The *underlying container* is a copy of `ctnr`, sorted by the `make_heap` algorithm.

### (2) range initialization constructor

The *underlying container* is a copy of `ctnr`, with the insertion of the elements in the range `[first, last)`, and then sorted by `make_heap`.

### (3) move-initialization constructor

The *underlying container* acquires the value of `ctnr` by *move-constructing* it. The elements are sorted by `make_heap`.

### (4) move-range initialization constructor

The *underlying container* acquires the value of `ctnr` by *move-conconstructing*. It also inserts the elements in the range `[first, last)` and then sorts them with `make_heap`.

### (5) allocator constructors

Constructs a `priority_queue` using a specific *allocator* value.

The constructor effectively initializes the comparison and container objects and then calls algorithm `make_heap` on the range that includes all its elements before returning.

## Parameters

`comp`  
Comparison object to be used to order the `heap`.  
This may be a function pointer or function object able to perform a *strict weak ordering* by comparing its two arguments.  
Compare is the third class template parameter ( by default: `less<T>`).

`ctnr`  
Container object.  
Container is the second class template parameter (the type of the *underlying container* for the `priority_queue`; by default: `vector<T>`).

`first, last`  
`Input iterators` to the initial and final positions in a sequence. The elements in this sequence are inserted into the *underlying container* before sorting it.  
The range used is `[first, last)`, which includes all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

## Example

```
1 // constructing priority queues
2 #include <iostream>           // std::cout
3 #include <queue>              // std::priority_queue
4 #include <vector>              // std::vector
5 #include <functional>          // std::greater
6
7 class mycomparison
8 {
9     bool reverse;
10 public:
11     mycomparison(const bool& revparam=false)
12     {reverse=revparam;}
13     bool operator() (const int& lhs, const int&rhs) const
14     {
15         if (reverse) return (lhs>rhs);
16         else return (lhs<rhs);
17     }
18 };
19
20 int main ()
21 {
22     int myints[] = {10,60,50,20};
23
24     std::priority_queue<int> first;
25     std::priority_queue<int> second (myints,myints+4);
26     std::priority_queue<int, std::vector<int>, std::greater<int> >
27         third (myints,myints+4);
28     // using mycomparison:
29     typedef std::priority_queue<int, std::vector<int>, mycomparison> mypq_type;
30
31     mypq_type fourth;           // less-than comparison
32     mypq_type fifth (mycomparison(true)); // greater-than comparison
33
34     return 0;
35 }
```

The example does not produce any output, but it constructs different `priority_queue` objects:

- First is empty.
- Second contains the four ints defined for `myints`, with 60 (the highest) at its top.
- Third has the same four ints, but because it uses `greater` instead of the default (which is `less`), it has 10 as its top element.
- Fourth and fifth are very similar to first: they are both empty, except that these use `mycomparison` for comparisons, which is a special stateful comparison function that behaves differently depending on a flag set on construction.

## Complexity

One container construction, and one call to `make_heap`, plus linear in the number of elements in the range `[first, last)` (if specified).

## Iterator validity

Moving constructors may invalidate all iterators, pointers and references related to their moved argument.

## Data races

All copied elements are accessed.

The *moving constructors* may modify their rvalue reference argument.

## Exception safety

Provides the same level of guarantees as the operation performed on the container.

## See also

[priority\\_queue::push](#) Insert element (public member function )

# /queue/priority\_queue/push

public member function

## std::priority\_queue::push

<queue>

```
void push (const value_type& val);
void push (const value_type& val);
void push (value_type&& val);
```

### Insert element

Inserts a new element in the `priority_queue`. The content of this new element is initialized to `val`.

This member function effectively calls the member function `push_back` of the *underlying container* object, and then reorders it to its location in the *heap* by calling the `push_heap` algorithm on the range that includes all the elements of the container.

## Parameters

`val`

Value to which the inserted element is initialized.

Member type `value_type` is the type of the elements in the container (defined as an alias of the first class template parameter, `T`).

## Return value

none

## Example

```
1 // priority_queue::push/pop
2 #include <iostream>           // std::cout
3 #include <queue>              // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> mypq;
8
9     mypq.push(30);
10    mypq.push(100);
11    mypq.push(25);
12    mypq.push(40);
13
14    std::cout << "Popping out elements...";
15    while (!mypoq.empty())
16    {
17        std::cout << ' ' << mypq.top();
18        mypq.pop();
19    }
20    std::cout << '\n';
21
22    return 0;
23 }
```

Output:

```
Popping out elements... 100 40 30 25
```

## Complexity

One call to `push_back` on the *underlying container* and one call to `push_heap` on the range that includes all the elements of the *underlying container*.

## Data races

The container and up to all its contained elements are modified.

## Exception safety

Provides the same level of guarantees as the operations performed on the *underlying container* object.

## See also

[priority\\_queue::pop](#) Remove top element (public member function )

## /queue/priority\_queue/size

public member function

### **std::priority\_queue::size**

<queue>

`size_type size() const;`

#### **Return size**

Returns the number of elements in the `priority_queue`.

This member function effectively calls member `size` of the underlying container object.

#### **Parameters**

none

#### **Return Value**

The number of elements in the *underlying container*.

Member type `size_type` is an unsigned integral type.

#### **Example**

```

1 // priority_queue::size
2 #include <iostream>           // std::cout
3 #include <queue>              // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> myints;
8     std::cout << "0. size: " << myints.size() << '\n';
9
10    for (int i=0; i<5; i++) myints.push(i);
11    std::cout << "1. size: " << myints.size() << '\n';
12
13    myints.pop();
14    std::cout << "2. size: " << myints.size() << '\n';
15
16    return 0;
17 }
```

Output:

```
0. size: 0
1. size: 5
2. size: 4
```

#### **Complexity**

Constant (calling `size` on the *underlying container*).

#### **See also**

## /queue/priority\_queue/swap

public member function

### **std::priority\_queue::swap**

<queue>

`void swap (priority_queue& x) noexcept /*see below*/;`

#### **Swap contents**

Exchanges the contents of the container adaptor by those of *x*, swapping both the *underlying container* value and their *comparison function* using the corresponding `swap` non-member functions (unqualified).

This member function has a `noexcept` specifier that matches the combined `noexcept` of the `swap` operations on the *underlying container* and the *comparison functions*.

#### **Parameters**

x

Another `priority_queue` container adaptor of the same type (i.e., instantiated with the same template parameters, `T`, `Container` and `Compare`). Sizes may differ.

#### **Return value**

none

#### **Example**

```

1 // priority_queue::swap
2 #include <iostream>           // std::cout
3 #include <queue>             // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> foo,bar;
8     foo.push(15); foo.push(30); foo.push(10);
9     bar.push(101); bar.push(202);
10
11    foo.swap(bar);
12
13    std::cout << "size of foo: " << foo.size() << '\n';
14    std::cout << "size of bar: " << bar.size() << '\n';
15
16    return 0;
17 }

```

Output:

```

size of foo: 2
size of bar: 3

```

## Complexity

Constant.

## Data races

Both `*this` and `x` are modified.

## Exception safety

Provides the same level of guarantees as the operation performed on the *underlying container* objects.

## See also

<a href="#">swap (priority_queue)</a>	Exchange contents of priority queues (public member function )
<a href="#">swap</a>	Exchange values of two objects (function template )

# /queue/priority\_queue/swap-free

public member function

## std::swap (priority\_queue)

<queue>

```

template <class T, class Container, class Compare>
void swap (queue<T,Container,Compare>& x, queue <T,Container,Compare>& y) noexcept(noexcept(x.swap(y)));

```

### Exchange contents of priority queues

Exchanges the contents of `x` and `y`.

## Parameters

`x,y`  
`priority_queue` containers of the same type. Size may differ.

## Return value

none

## Example

```

1 // swap priority_queues
2 #include <iostream>           // std::cout
3 #include <queue>             // std::priority_queue, std::swap(priority_queue)
4
5 int main ()
6 {
7     std::priority_queue<int> foo,bar;
8     foo.push(15); foo.push(30); foo.push(10);
9     bar.push(101); bar.push(202);
10
11    swap(foo,bar);
12
13    std::cout << "size of foo: " << foo.size() << '\n';
14    std::cout << "size of bar: " << bar.size() << '\n';
15
16    return 0;
17 }

```

Output:

```

size of foo: 2
size of bar: 3

```

## Complexity

Constant.

## Data races

Both containers, *x* and *y*, are modified.

## Exception safety

Provides the same level of guarantees as the operation performed on the *underlying container* objects.

## See also

<a href="#">priority_queue::swap</a>	Swap contents (public member function )
<a href="#">swap</a>	Exchange values of two objects (function template )

## /queue/priority\_queue/top

public member function

### std::priority\_queue::top

<queue>

```
const value_type& top() const;
const_reference top() const;
```

#### Access top element

Returns a constant reference to the *top element* in the *priority\_queue*.

The *top element* is the element that compares higher in the *priority\_queue*, and the next that is removed from the container when *priority\_queue::pop* is called.

This member function effectively calls member *front* of the underlying container object.

## Parameters

none

## Return value

A reference to the top element in the *priority\_queue*.

Member type *value\_type* is the type of the elements in the container (defined as an alias of the first class template parameter, *T*).

Member type *const\_reference* is an alias of the *underlying container's* type with the same name.

## Example

```
1 // priority_queue::top
2 #include <iostream>           // std::cout
3 #include <queue>              // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> mypq;
8
9     mypq.push(10);
10    mypq.push(20);
11    mypq.push(15);
12
13    std::cout << "mypq.top() is now " << mypq.top() << '\n';
14
15    return 0;
16 }
```

Output:

```
mypq.top() is now 20
```

## Complexity

Constant (calling *front* on the *underlying container*).

## Data races

The container is accessed.

The constant reference returned can be used to directly access the next element.

## Exception safety

Provides the same level of guarantees as the operation performed on the container (no-throw guarantee for standard non-empty containers).

## See also

<a href="#">priority_queue::pop</a>	Remove top element (public member function )
<a href="#">priority_queue::push</a>	Insert element (public member function )

## /queue/priority\_queue/uses\_allocator

class template

### std::uses\_allocator<priority\_queue>

<queue>

```
template <class T, class Container, class Compare, class Alloc>
struct uses_allocator<priority_queue<T,Container,Compare>,Alloc>;
```

#### Uses allocator for priority queue

This trait specialization of `uses_allocator` informs whether the `priority_queue` accepts an allocator convertible from `Alloc`, by inheriting either from `true_type` or from `false_type`/`samp`.

It is equivalent to the `uses_allocator` instantiation for its *underlying container*.

It is defined in `<queue>` as:

```
1 template <class T, class Container, class Compare, class Alloc>
2   struct uses_allocator<priority_queue<T,Container,Compare>,Alloc>
3     : uses_allocator<Container,Alloc>::type {}
```

#### See also

`uses_allocator` Uses allocator (class template )

`priority_queue::priority_queue` Construct priority queue (public member function )

## /queue/queue

class template

### std::queue

<queue>

```
template <class T, class Container = deque<T> > class queue;
```

#### FIFO queue

`queues` are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.

`queues` are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed* into the "back" of the specific container and *popped* from its "front".

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

- `empty`
- `size`
- `front`
- `back`
- `push_back`
- `pop_front`

The standard container classes `deque` and `list` fulfill these requirements. By default, if no container class is specified for a particular `queue` class instantiation, the standard container `deque` is used.

#### Template parameters

T

Type of the elements.

Aliased as member type `queue::value_type`.

Container

Type of the internal *underlying container* object where the elements are stored.

Its `value_type` shall be T.

Aliased as member type `queue::container_type`.

#### Member types

member type	definition	notes
<code>value_type</code>	The first template parameter (T)	Type of the elements
<code>container_type</code>	The second template parameter (Container)	Type of the <i>underlying container</i>
<code>size_type</code>	an unsigned integral type	usually the same as <code>size_t</code>
member type	definition	notes
<code>value_type</code>	The first template parameter (T)	Type of the elements
<code>container_type</code>	The second template parameter (Container)	Type of the <i>underlying container</i>
<code>reference</code>	<code>container_type::reference</code>	usually, <code>value_type&amp;</code>
<code>const_reference</code>	<code>container_type::const_reference</code>	usually, <code>const value_type&amp;</code>
<code>size_type</code>	an unsigned integral type	usually, the same as <code>size_t</code>

#### Member functions

<code>(constructor)</code>	Construct queue (public member function )
<code>empty</code>	Test whether container is empty (public member function )
<code>size</code>	Return size (public member function )

<b>front</b>	Access next element (public member function )
<b>back</b>	Access last element (public member function )
<b>push</b>	Insert element (public member function )
<b>emplace</b>	Construct and insert element (public member function )
<b>pop</b>	Remove next element (public member function )
<b>swap</b>	Swap contents (public member function )

## Non-member function overloads

<b>relational operators</b>	Relational operators for queue (function )
<b>swap (queue)</b>	Exchange contents of queues (public member function )

## Non-member class specializations

<b>uses_allocator&lt;queue&gt;</b>	Uses allocator for queue (class template )
------------------------------------	--

# /queue/queue/back

public member function

## std::queue::back

<queue>

<b>value_type&amp; back();</b>
<b>const value_type&amp; back() const;</b>
<b>reference&amp; back();</b>
<b>const_reference&amp; back() const;</b>

### Access last element

Returns a reference to the last element in the `queue`. This is the "newest" element in the queue (i.e. the last element pushed into the `queue`).

This member function effectively calls member `back` of the underlying container object.

### Parameters

none

### Return value

A reference to the last element in the `queue`.

Member type `value_type` is the type of the elements in the container (defined as an alias of the first class template parameter, `T`).

Member types `reference` and `const_reference` are aliases of the *underlying containers*'s types with the same name.

### Example

```
1 // queue::back
2 #include <iostream>           // std::cout
3 #include <queue>              // std::queue
4
5 int main ()
6 {
7     std::queue<int> myqueue;
8
9     myqueue.push(12);
10    myqueue.push(75); // this is now the back
11
12    myqueue.back() -= myqueue.front();
13
14    std::cout << "myqueue.back() is now " << myqueue.back() << '\n';
15
16    return 0;
17 }
```

Output:

myqueue.back() is now 63

### Complexity

Constant (calling `back` on the *underlying container*).

### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container).  
The reference returned can be used to access or modify the last element.

### Exception safety

Provides the same level of guarantees as the operation performed on the container (no-throw guarantee for standard non-empty containers).

### See also

<b>queue::front</b>	Access next element (public member function )
---------------------	---

## /queue/queue/emplace

public member function

### std::queue::emplace

&lt;queue&gt;

```
template <class... Args> void emplace (Args&... args);
```

#### Construct and insert element

Adds a new element at the end of the `queue`, after its current last element. This new element is constructed in place passing `args` as the arguments for its constructor.

This member function effectively calls the member function `emplace_back` of the *underlying container*, *forwarding args*.

#### Parameters

`args`

Arguments forwarded to construct the new element.

#### Return value

none

#### Example

```
1 // queue::emplace
2 #include <iostream>           // std::cin, std::cout
3 #include <queue>              // std::queue
4 #include <string>             // std::string, std::getline(string)
5
6 int main ()
7 {
8     std::queue<std::string> myqueue;
9
10    myqueue.emplace ("First sentence");
11    myqueue.emplace ("Second sentence");
12
13    std::cout << "myqueue contains:\n";
14    while (!myqueue.empty())
15    {
16        std::cout << myqueue.front() << '\n';
17        myqueue.pop();
18    }
19
20    return 0;
21 }
```

Output:

```
myqueue contains:
First sentence
Second sentence
```

#### Complexity

One call to `emplace_back` on the *underlying container*.

#### Data races

The container and up to all its contained elements are modified.

#### Exception safety

Provides the same level of guarantees as the operation performed on the *underlying container* object.

#### See also

## /queue/queue/empty

public member function

### std::queue::empty

&lt;queue&gt;

```
bool empty() const;
```

#### Test whether container is empty

Returns whether the `queue` is empty: i.e. whether its `size` is zero.

This member function effectively calls member `empty` of the *underlying container* object.

## Parameters

none

## Return Value

true if the *underlying container's* size is 0, false otherwise.

## Example

```
1 // queue::empty
2 #include <iostream>           // std::cout
3 #include <queue>             // std::queue
4
5 int main ()
6 {
7     std::queue<int> myqueue;
8     int sum (0);
9
10    for (int i=1;i<=10;i++) myqueue.push(i);
11
12    while (!myqueue.empty())
13    {
14        sum += myqueue.front();
15        myqueue.pop();
16    }
17
18    std::cout << "total: " << sum << '\n';
19
20    return 0;
21 }
```

The example initializes the content of the queue to a sequence of numbers (from 1 to 10). It then pops the elements one by one until it is empty and calculates their sum.

Output:

```
total: 55
```

## Complexity

Constant (calling `empty` on the *underlying container*).

## Data races

The container is accessed.

## Exception safety

Provides the same level of guarantees as the operation performed on the container (no-throw guarantee for standard container types).

## See also

<code>queue::size</code>	Return size (public member function )
--------------------------	---------------------------------------

# /queue/queue/front

public member function

## std::queue::front

<queue>

```
    value_type& front();
const value_type& front() const;
    reference& front();
const_reference& front() const;
```

### Access next element

Returns a reference to the *next element* in the `queue`.

The *next element* is the "oldest" element in the `queue` and the same element that is popped out from the `queue` when `queue::pop` is called.

This member function effectively calls member `front` of the *underlying container* object.

## Parameters

none

## Return value

A reference to the *next element* in the `queue`.

Member type <code>value_type</code> is the type of the elements in the container (defined as an alias of the first class template parameter, <code>T</code> ).
--

Member types <code>reference</code> and <code>const_reference</code> are aliases of the <i>underlying container's</i> types with the same name.
---

## Example

```
1 // queue::front
2 #include <iostream>           // std::cout
```

```

3 #include <queue>           // std::queue
4
5 int main ()
6 {
7     std::queue<int> myqueue;
8
9     myqueue.push(77);
10    myqueue.push(16);
11
12    myqueue.front() -= myqueue.back();    // 77-16=61
13
14    std::cout << "myqueue.front() is now " << myqueue.front() << '\n';
15
16    return 0;
17 }

```

Output:

```
myqueue.front() is now 61
```

## Complexity

Constant (calling `front` on the *underlying container*).

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
The reference returned can be used to access or modify the next element.

## Exception safety

Provides the same level of guarantees as the operation performed on the container (no-throw guarantee for standard non-empty containers).

## See also

<code>queue::pop</code>	Remove next element ( <a href="#">public member function</a> )
<code>queue::back</code>	Access last element ( <a href="#">public member function</a> )

# /queue/queue/operators

function

## std::relational operators (queue)

<queue>

```

(1) template <class T, class Container>
    bool operator==(const queue<T,Container>& lhs, const queue<T,Container>& rhs);
(2) template <class T, class Container>
    bool operator!=(const queue<T,Container>& lhs, const queue<T,Container>& rhs);
(3) template <class T, class Container>
    bool operator< (const queue<T,Container>& lhs, const queue<T,Container>& rhs);
(4) template <class T, class Container>
    bool operator<= (const queue<T,Container>& lhs, const queue<T,Container>& rhs);
(5) template <class T, class Container>
    bool operator> (const queue<T,Container>& lhs, const queue<T,Container>& rhs);
(6) template <class T, class Container>
    bool operator>= (const queue<T,Container>& lhs, const queue<T,Container>& rhs);

```

### Relational operators for queue

Performs the appropriate comparison operation between *lhs* and *rhs*.

Each of these operator overloads calls the same operator on the *underlying container* objects.

## Parameters

*lhs*, *rhs*  
`queue` objects (to the left- and right-hand side of the operator, respectively).

## Return Value

true if the condition holds, and false otherwise.

## Complexity

Constant (a single call to the comparison operator on the *underlying containers*). Although notice that this operation on the *underlying containers* is itself up to linear in the `size` of the smaller object for standard containers.

## Data races

Both containers, *lhs* and *rhs*, are accessed.  
Up to all of their contained elements may be accessed.

## Exception safety

Provides the same level of guarantees as the operation performed on the container.

## See also

<code>queue::queue</code>	Construct queue (public member function )
<code>queue::swap</code>	Swap contents (public member function )

## /queue/queue/pop

public member function

### `std::queue::pop`

<queue>

`void pop();`

#### Remove next element

Removes the next element in the `queue`, effectively reducing its `size` by one.

The element removed is the "oldest" element in the `queue` whose value can be retrieved by calling member `queue::front`.

This calls the removed element's destructor.

This member function effectively calls the member function `pop_front` of the *underlying container* object.

#### Parameters

none

#### Return value

none

#### Example

```

1 // queue::push/pop           // std::cin, std::cout
2 #include <iostream>          // std::queue
3 #include <queue>             // std::queue
4
5 int main ()
6 {
7     std::queue<int> myqueue;
8     int myint;
9
10    std::cout << "Please enter some integers (enter 0 to end):\n";
11
12    do {
13        std::cin >> myint;
14        myqueue.push (myint);
15    } while (myint);
16
17    std::cout << "myqueue contains: ";
18    while (!myqueue.empty())
19    {
20        std::cout << ' ' << myqueue.front();
21        myqueue.pop();
22    }
23    std::cout << '\n';
24
25    return 0;
26 }
```

The example uses `push` to add new elements to the queue, which are then popped out in the same order.

#### Complexity

Constant (calling `pop_front` on the *underlying container*).

#### Data races

The container and up to all its contained elements are modified.

#### Exception safety

Provides the same level of guarantees as the operation performed on the *underlying container* object.

#### See also

<code>queue::push</code>	Insert element (public member function )
<code>queue::empty</code>	Test whether container is empty (public member function )

## /queue/queue/push

public member function

### `std::queue::push`

<queue>

```

void push (const value_type& val);
void push (const value_type& val);
void push (value_type&& val);
```

#### Insert element

Inserts a new element at the end of the `queue`, after its current last element. The content of this new element is initialized to `val`.

This member function effectively calls the member function `push_back` of the *underlying container* object.

## Parameters

`val`

Value to which the inserted element is initialized.

Member type `value_type` is the type of the elements in the container (defined as an alias of the first class template parameter, `T`).

## Return value

`none`

## Example

```
1 // queue::push/pop
2 #include <iostream>           // std::cin, std::cout
3 #include <queue>             // std::queue
4
5 int main ()
6 {
7     std::queue<int> myqueue;
8     int myint;
9
10    std::cout << "Please enter some integers (enter 0 to end):\n";
11
12    do {
13        std::cin >> myint;
14        myqueue.push (myint);
15    } while (myint);
16
17    std::cout << "myqueue contains: ";
18    while (!myqueue.empty())
19    {
20        std::cout << ' ' << myqueue.front();
21        myqueue.pop();
22    }
23    std::cout << '\n';
24
25    return 0;
26 }
```

The example uses `push` to add new elements to the queue, which are then popped out in the same order.

## Complexity

One call to `push_back` on the *underlying container*.

## Data races

The container and up to all its contained elements are modified.

## Exception safety

Provides the same level of guarantees as the operation performed on the *underlying container* object.

## See also

<code>queue::pop</code>	Remove next element (public member function )
<code>queue::size</code>	Return size (public member function )

# /queue/queue/queue

public member function

## `std::queue::queue`

`<queue>`

```
explicit queue (const container_type& ctrnr = container_type());
    initialize (1) explicit queue (const container_type& ctrnr);
    move-initialize (2) explicit queue (container_type&& ctrnr = container_type());
    allocator (3) template <class Alloc> explicit queue (const Alloc& alloc);
    init + allocator (4) template <class Alloc> queue (const container_type& ctrnr, const Alloc& alloc);
    move-init + allocator (5) template <class Alloc> queue (container_type&& ctrnr, const Alloc& alloc);
    copy + allocator (6) template <class Alloc> queue (const queue& x, const Alloc& alloc);
    move + allocator (7) template <class Alloc> queue (queue&& x, const Alloc& alloc);
```

## Construct queue

Constructs a `queue` container adaptor object.

A container adaptor keeps internally a container object as data. This container object is a copy of the `ctrnr` argument passed to the constructor, if any, otherwise it is an empty container.

A container adaptor keeps internally a container object as data, which is initialized by this constructor:

### (1) initialization constructor

Constructs a container adaptor whose internal container is initialized to a copy of `ctrnr`.

<b>(2) move-initialization constructor</b>	Constructs a container adaptor whose internal container acquires the value of <i>ctnr</i> by <i>move-constructing</i> it.
<b>(3) allocator constructor</b>	Constructs a container adaptor whose internal container is constructed with <i>alloc</i> as argument.
<b>(4) initialization with allocator constructor</b>	Constructs a container adaptor whose internal container is constructed with <i>cntr</i> and <i>alloc</i> as arguments.
<b>(5) move-initialization with allocator constructor</b>	Constructs a container adaptor whose internal container is constructed with <code>std::move(cntr)</code> and <i>alloc</i> as arguments.
<b>(6) copy with allocator constructor</b>	Constructs a container adaptor whose internal container is constructed with <i>x</i> 's internal container as first argument and <i>alloc</i> as second.
<b>(7) move with allocator constructor</b>	Constructs a container adaptor whose internal container is constructed by <i>moving</i> <i>x</i> 's internal container as first argument and passing <i>alloc</i> as second.

## Parameters

<code>ctnr</code>	Container object. <code>container_type</code> is the type of the <i>underlying container type</i> (defined as an alias of the second class template parameter, <code>Container</code> ; see <a href="#">member types</a> ).
<code>alloc</code>	Allocator object. <code>Alloc</code> shall be a type for which <code>uses_allocator::value</code> is true (for other types, the constructor does not even participate in overload resolution).
<code>x</code>	A <code>queue</code> of the same type (i.e., with the same template arguments, <code>T</code> and <code>Container</code> ).

## Example

```

1 // constructing queues
2 #include <iostream>           // std::cout
3 #include <deque>              // std::deque
4 #include <list>               // std::list
5 #include <queue>              // std::queue
6
7 int main ()
8 {
9     std::deque<int> mydeck (3,100);      // deque with 3 elements
10    std::list<int> mylist (2,200);        // list with 2 elements
11
12    std::queue<int> first;                // empty queue
13    std::queue<int> second (mydeck);       // queue initialized to copy of deque
14
15    std::queue<int, std::list<int> > third; // empty queue with list as underlying container
16    std::queue<int, std::list<int> > fourth (mylist);
17
18    std::cout << "size of first: " << first.size() << '\n';
19    std::cout << "size of second: " << second.size() << '\n';
20    std::cout << "size of third: " << third.size() << '\n';
21    std::cout << "size of fourth: " << fourth.size() << '\n';
22
23    return 0;
24 }
```

Output:

```

size of first: 0
size of second: 3
size of third: 0
size of fourth: 2

```

## Complexity

One container construction (which for standard containers is up to linear in its size).

## Iterator validity

*Moving constructors* (2) and (7), may invalidate all iterators, pointers and references related to their moved argument.

## Data races

All copied elements are accessed.

The *moving constructors* (2) and (7) may modify their (first) argument.

## Exception safety

Provides the same level of guarantees as the operation performed on the container.

## See also

<code>queue::push</code>	Insert element (public member function )
--------------------------	--

## /queue/queue/size

public member function

### `std::queue::size`

```
size_type size() const;
```

`<queue>`

## Return size

Returns the number of elements in the `queue`.

This member function effectively calls member `size` of the underlying container object.

### Parameters

none

### Return Value

The number of elements in the *underlying container*.

Member type `size_type` is an unsigned integral type.

### Example

```
1 // queue::size
2 #include <iostream>           // std::cout
3 #include <queue>             // std::queue
4
5 int main ()
6 {
7     std::queue<int> myints;
8     std::cout << "0. size: " << myints.size() << '\n';
9
10    for (int i=0; i<5; i++) myints.push(i);
11    std::cout << "1. size: " << myints.size() << '\n';
12
13    myints.pop();
14    std::cout << "2. size: " << myints.size() << '\n';
15
16    return 0;
17 }
```

Output:

```
0. size: 0
1. size: 5
2. size: 4
```

### Complexity

Constant (calling `size` on the *underlying container*).

### Data races

The container is accessed.

### Exception safety

Provides the same level of guarantees as the operation performed on the container (no-throw guarantee for standard container types).

### See also

`queue::empty` Test whether container is empty ([public member function](#))

## /queue/queue/swap

public member function

### std::queue::swap

`<queue>`

```
void swap (queue& x) noexcept /*see below*/;
```

#### Swap contents

Exchanges the contents of the container adaptor (`*this`) by those of `x`.

This member function calls the non-member function `swap` (unqualified) to swap the *underlying containers*.

The `noexcept` specifier matches the `swap` operation on the *underlying container*.

### Parameters

x

Another `queue` container adaptor of the same type (i.e., instantiated with the same template parameters, `T` and `Container`). Sizes may differ.

### Return value

none

### Example

```
1 // queue::swap
2 #include <iostream>           // std::cout
3 #include <queue>             // std::queue
4
```

```

5 int main ()
6 {
7     std::queue<int> foo,bar;
8     foo.push(10); foo.push(20); foo.push(30);
9     bar.push(111); bar.push(222);
10
11    foo.swap(bar);
12
13    std::cout << "size of foo: " << foo.size() << '\n';
14    std::cout << "size of bar: " << bar.size() << '\n';
15
16    return 0;
17 }

```

Output:

```

size of foo: 2
size of bar: 3

```

## Complexity

Constant.

## Data races

Both `*this` and `x` are modified.

## Exception safety

Provides the same level of guarantees as the operation performed on the *underlying container* objects.

## See also

<a href="#">swap (queue)</a>	Exchange contents of queues (public member function )
<a href="#">swap</a>	Exchange values of two objects (function template )

## /queue/queue/swap-free

public member function

### std::swap (queue)

<queue>

```

template <class T, class Container>
void swap (queue<T,Container>& x, queue<T,Container>& y) noexcept(noexcept(x.swap(y)));

```

#### Exchange contents of queues

Exchanges the contents of `x` and `y`.

## Parameters

`x,y`

queue containers of the same type. Size may differ.

## Return value

none

## Example

```

1 // swap queues
2 #include <iostream>           // std::cout
3 #include <queue>              // std::queue, std::swap(queue)
4
5 int main ()
6 {
7     std::queue<int> foo,bar;
8     foo.push(10); foo.push(20); foo.push(30);
9     bar.push(111); bar.push(222);
10
11    swap(foo,bar);
12
13    std::cout << "size of foo: " << foo.size() << '\n';
14    std::cout << "size of bar: " << bar.size() << '\n';
15
16    return 0;
17 }

```

Output:

```

size of foo: 2
size of bar: 3

```

## Complexity

Constant.

## Data races

Both container adaptors, `x` and `y`, are modified.

## Exception safety

Provides the same level of guarantees as the operation performed on the *underlying container* objects.

## See also

<code>queue::swap</code>	Swap contents (public member function )
<code>swap</code>	Exchange values of two objects (function template )

## /queue/queue/uses\_allocator

class template

### `std::uses_allocator<queue>`

<queue>

```
template <class T, class Container, class Alloc>
    struct uses_allocator<queue<T,Container>,Alloc>;
```

#### Uses allocator for queue

This trait specialization of `uses_allocator` informs whether the `queue` accepts an allocator convertible from `Alloc`, by inheriting either from `true_type` or from `false_type`/samp>.

It is equivalent to the `uses_allocator` instantiation for its *underlying container*.

It is defined in <queue> as:

```
1 template <class T, class Container, class Alloc>
2     struct uses_allocator<queue<T,Container>,Alloc>
3         : uses_allocator<Container,Alloc>::type {}
```

## See also

<code>uses_allocator</code>	Uses allocator (class template )
<code>queue::queue</code>	Construct queue (public member function )

## /set

header

### `<set>`

#### Set header

Header that defines the `set` and `multiset` container classes:

## Classes

<code>set</code>	Set (class template )
<code>multiset</code>	Multiple-key set (class template )

## Functions

<code>begin</code>	Iterator to beginning (function template )
<code>end</code>	Iterator to end (function template )

## /set/multiset

class template

### `std::multiset`

<set>

```
template < class T, // multiset::key_type/value_type
            class Compare = less<T>, // multiset::key_compare/value_compare
            class Alloc = allocator<T> > // multiset::allocator_type
        > class multiset;
```

#### Multiple-key set

Multisets are containers that store elements following a specific order, and where multiple elements can have equivalent values.

In a `multiset`, the value of an element also identifies it (the value is itself the *key*, of type `T`). The value of the elements in a `multiset` cannot be modified once in the container (the elements are always `const`), but they can be inserted or removed from the container.

Internally, the elements in a `multiset` are always sorted following a specific *strict weak ordering* criterion indicated by its internal `comparison object` (of type `Compare`).

`multiset` containers are generally slower than `unordered_multiset` containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Multisets are typically implemented as *binary search trees*.

## Container properties

### Associative

Elements in associative containers are referenced by their key and not by their absolute position in the container.

### Ordered

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

### Set

The value of an element is also the key used to identify it.

### Multiple equivalent keys

Multiple elements in the container can have equivalent keys.

### Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

## Template parameters

T

Type of the elements. Each element in a `multiset` container is also identified by this value (each value is itself also the element's key). Aliased as member types `multiset::key_type` and `multiset::value_type`.

Compare

A binary predicate that takes two arguments of the same type as the elements and returns a `bool`. The expression `comp(a,b)`, where `comp` is an object of this type and `a` and `b` are key values, shall return `true` if `a` is considered to go before `b` in the *strict weak ordering* the function defines.

The `multiset` object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent (by comparing them reflexively: they are equivalent if `!comp(a,b) && !comp(b,a)`).

This can be a function pointer or a function object (see `constructor` for an example). This defaults to `less<T>`, which returns the same as applying the *less-than operator* (`a<b`).

Aliased as member types `multiset::key_compare` and `multiset::value_compare`.

Alloc

Type of the allocator object used to define the storage allocation model. By default, the `allocator` class template is used, which defines the simplest memory allocation model and is value-independent.

Aliased as member type `multiset::allocator_type`.

## Member types

member type	definition	notes
<code>key_type</code>	The first template parameter ( <code>T</code> )	
<code>value_type</code>	The first template parameter ( <code>T</code> )	
<code>key_compare</code>	The second template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;key_type&gt;</code>
<code>value_compare</code>	The second template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;value_type&gt;</code>
<code>allocator_type</code>	The third template parameter ( <code>Alloc</code> )	defaults to: <code>allocator&lt;value_type&gt;</code>
<code>reference</code>	<code>allocator_type::reference</code>	for the default allocator: <code>value_type&amp;</code>
<code>const_reference</code>	<code>allocator_type::const_reference</code>	for the default allocator: <code>const value_type&amp;</code>
<code>pointer</code>	<code>allocator_type::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_type::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a bidirectional <code>iterator</code> to <code>value_type</code>	convertible to <code>const_iterator</code>
<code>const_iterator</code>	a bidirectional <code>iterator</code> to <code>const value_type</code>	
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>

member type	definition	notes
<code>key_type</code>	The first template parameter ( <code>T</code> )	
<code>value_type</code>	The first template parameter ( <code>T</code> )	
<code>key_compare</code>	The second template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;key_type&gt;</code>
<code>value_compare</code>	The second template parameter ( <code>Compare</code> )	defaults to: <code>less&lt;value_type&gt;</code>
<code>allocator_type</code>	The third template parameter ( <code>Alloc</code> )	defaults to: <code>allocator&lt;value_type&gt;</code>
<code>reference</code>	<code>value_type&amp;</code>	
<code>const_reference</code>	<code>const value_type&amp;</code>	
<code>pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::pointer</code>	for the default allocator: <code>value_type*</code>
<code>const_pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::const_pointer</code>	for the default allocator: <code>const value_type*</code>
<code>iterator</code>	a bidirectional <code>iterator</code> to <code>const value_type</code>	* convertible to <code>const_iterator</code>
<code>const_iterator</code>	a bidirectional <code>iterator</code> to <code>const value_type</code>	*
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	*
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	*
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>

\*Note: All iterators in a `multiset` point to const elements. Whether the `const_` member type is the same type as its non-`const_` counterpart depends on the particular library implementation, but programs should not rely on them being different to overload functions: `const_iterator` is more generic, since `iterator` is always convertible to it.

## Member functions

<code>(constructor)</code>	Construct multiset (public member function )
<code>(destructor)</code>	Multiset destructor (public member function )
<code>operator=</code>	Copy container content (public member function )

**Iterators:**

<b>begin</b>	Return iterator to beginning (public member function )
<b>end</b>	Return iterator to end (public member function )
<b>rbegin</b>	Return reverse iterator to reverse beginning (public member function )
<b>rend</b>	Return reverse iterator to reverse end (public member function )
<b>cbegin</b>	Return const_iterator to beginning (public member function )
<b>cend</b>	Return const_iterator to end (public member function )
<b>crbegin</b>	Return const_reverse_iterator to reverse beginning (public member function )
<b>crend</b>	Return const_reverse_iterator to reverse end (public member function )

**Capacity:**

<b>empty</b>	Test whether container is empty (public member function )
<b>size</b>	Return container size (public member function )
<b>max_size</b>	Return maximum size (public member function )

**Modifiers:**

<b>insert</b>	Insert element (public member function )
<b>erase</b>	Erase elements (public member function )
<b>swap</b>	Swap content (public member function )
<b>clear</b>	Clear content (public member function )
<b>emplace</b>	Construct and insert element (public member function )
<b>emplace_hint</b>	Construct and insert element with hint (public member function )

**Observers:**

<b>key_comp</b>	Return comparison object (public member function )
<b>value_comp</b>	Return comparison object (public member function )

**Operations:**

<b>find</b>	Get iterator to element (public member function )
<b>count</b>	Count elements with a specific key (public member function )
<b>lower_bound</b>	Return iterator to lower bound (public member function )
<b>upper_bound</b>	Return iterator to upper bound (public member function )
<b>equal_range</b>	Get range of equal elements (public member function )

**Allocator:**

<b>get_allocator</b>	Get allocator (public member function )
----------------------	---

## /set/multiset/begin

public member function

**std::multiset::begin**

&lt;set&gt;

```

    iterator begin();
const_iterator begin() const;
    iterator begin() noexcept;
const_iterator begin() const noexcept;

```

**Return iterator to beginning**Returns an iterator referring to the first element in the **multiset** container.Because **multiset** containers keep their elements ordered at all times, **begin** points to the element that goes first following the container's **sorting** criterion.If the container is **empty**, the returned iterator value shall not be dereferenced.**Parameters**

none

**Return Value**

An iterator to the first element in the container.

If the **multiset** object is const-qualified, the function returns a **const\_iterator**. Otherwise, it returns an **iterator**.Member types **iterator** and **const\_iterator** are **bidirectional iterator** types pointing to elements.**Example**

```

1 // multiset::begin/end
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {42,71,71,71,12};
8     std::multiset<int> mymultiset (myints,myints+5);
9

```

```

10 std::multiset<int>::iterator it;
11 std::cout << "mymultiset contains:";
12 for (std::multiset<int>::iterator it=mymultiset.begin(); it!=mymultiset.end(); ++it)
13     std::cout << ' ' << *it;
14
15 std::cout << '\n';
16
17 return 0;
18 }

```

Output:

```
mymultiset contains: 12 42 71 71 71
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>multiset::end</code>	Return iterator to end (public member function )
<code>multiset::rbegin</code>	Return reverse iterator to reverse beginning (public member function )
<code>multiset::rend</code>	Return reverse iterator to reverse end (public member function )

## /set/multiset/cbegin

public member function

### std::multiset::cbegin

<set>

`const_iterator cbegin() const noexcept;`

**Return const\_iterator to beginning**

Returns a `const_iterator` pointing to the first element in the container.

All iterators in `multiset` containers are *constant iterators* (including both `const_iterator` and `iterator` member types). These cannot be used to modify the contents they point to, but can be increased and decreased normally (unless they are themselves also `const`).

If the container is `empty`, the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

A `const_iterator` to the beginning of the sequence.

Member type `const_iterator` is a **bidirectional iterator** type that points to `const` elements.

## Example

```

1 // multiset::cbegin/cend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset = {10,20,30,20,10};
8
9     std::cout << "mymultiset contains:";
10    for (auto it=mymultiset.cbegin(); it != mymultiset.cend(); ++it)
11        std::cout << ' ' << *it;
12
13    std::cout << '\n';
14
15    return 0;
16 }

```

Output:

```
mymultiset contains: 10 10 20 20 30
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.  
Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>multiset::begin</code>	Return iterator to beginning (public member function )
<code>multiset::cend</code>	Return const_iterator to end (public member function )
<code>multiset::crbegin</code>	Return const_reverse_iterator to reverse beginning (public member function )

# /set/multiset/cend

public member function

## std::multiset::cend

<set>

`const_iterator cend() const noexcept;`

### Return const\_iterator to end

Returns a `const_iterator` pointing to the *past-the-end* element in the container.

All iterators in `multiset` containers are *constant iterators* (including both `const_iterator` and `iterator` member types). These cannot be used to modify the contents they point to, but can be increased and decreased normally (unless they are themselves also `const`).

## Parameters

none

## Return Value

A `const_iterator` to the element past the end of the sequence.

Member type `const_iterator` is a **bidirectional iterator** type that points to `const` elements.

## Example

```
1 // multiset::cbegin/cend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset = {10,20,30,20,10};
8
9     std::cout << "mymultiset contains:";
10    for (auto it=mymultiset.cbegin(); it != mymultiset.cend(); ++it)
11        std::cout << ' ' << *it;
12
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
mymultiset contains: 10 10 20 20 30
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.  
Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multiset::end</a>	Return iterator to end (public member function )
<a href="#">multiset::cbegin</a>	Return const_iterator to beginning (public member function )

## /set/multiset/clear

public member function

### std::multiset::clear

<set>

```
void clear();  
void clear() noexcept;
```

#### Clear content

Removes all elements from the multiset container (which are destroyed), leaving the container with a [size](#) of 0.

#### Parameters

none

#### Return value

none

#### Example

```
1 // multiset::clear  
2 #include <iostream>  
3 #include <set>  
4  
5 int main ()  
6 {  
7     std::multiset<int> mymultiset;  
8  
9     mymultiset.insert (11);  
10    mymultiset.insert (42);  
11    mymultiset.insert (11);  
12  
13    std::cout << "mymultiset contains:";  
14    for (std::multiset<int>::iterator it=mymultiset.begin(); it!=mymultiset.end(); ++it)  
15        std::cout << ' ' << *it;  
16    std::cout << '\n';  
17  
18    mymultiset.clear();  
19    mymultiset.insert (200);  
20    mymultiset.insert (100);  
21  
22    std::cout << "mymultiset contains:";  
23    for (std::multiset<int>::iterator it=mymultiset.begin(); it!=mymultiset.end(); ++it)  
24        std::cout << ' ' << *it;  
25  
26    std::cout << '\n';  
27  
28    return 0;  
29 }
```

Output:

```
mymultiset contains: 11 11 42  
myset contains: 100 200
```

#### Complexity

Linear in [size](#) (destructions).

#### Iterator validity

All iterators, pointers and references related to this container are invalidated.

#### Data races

The container is modified.

All contained elements are modified.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">multiset::erase</a>	Erase elements (public member function )
<a href="#">multiset::size</a>	Return container size (public member function )
<a href="#">multiset::empty</a>	Test whether container is empty (public member function )

## /set/multiset/count

public member function

### std::multiset::count

<set>

```
size_type count (const value_type& val) const;
```

#### Count elements with a specific key

Searches the container for elements equivalent to *val* and returns the number of matches.

Two elements of a *multiset* are considered equivalent if the container's *comparison object* returns *false* reflexively (i.e., no matter the order in which the elements are passed as arguments).

#### Parameters

*val*

Value to search for.

Member type *value\_type* is the type of the elements in the container, defined in *multiset* as an alias of its first template parameter (*T*).

#### Return value

The number of elements in the container that are equivalent to *val*.

Member type *size\_type* is an unsigned integral type.

#### Example

```
1 // multiset::count
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {10, 73, 12, 22, 73, 73, 12};
8     std::multiset<int> mymultiset (myints, myints+7);
9
10    std::cout << "73 appears " << mymultiset.count(73) << " times in mymultiset.\n";
11
12    return 0;
13 }
```

Output:

```
73 appears 3 times in mymultiset.
```

#### Complexity

Logarithmic in *size* and linear in the number of matches.

#### Iterator validity

No changes.

#### Data races

The container is accessed.

Concurrently accessing the elements of a *multiset* is safe.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

#### See also

## /set/multiset/crbegin

public member function

### std::multiset::crbegin

<set>

```
const_reverse_iterator crbegin() const noexcept;
```

#### Return const\_reverse\_iterator to reverse beginning

Returns a *const\_reverse\_iterator* pointing to the last element in the container (i.e., its *reverse beginning*).

#### Parameters

none

#### Return Value

A *const\_reverse\_iterator* to the *reverse beginning* of the sequence.

Member type *const\_reverse\_iterator* is a *bidirectional iterator* type that points to *const* elements.

## Example

```
1 // multiset::crbegin/crend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset = {10,20,30,20,10};
8
9     std::cout << "mymultiset backwards:" ;
10    for (auto rit=mymultiset.crbegin(); rit != mymultiset.crend(); ++rit)
11        std::cout << ' ' << *rit;
12
13    std::cout << '\n';
14
15    return 0;
16 }
```

## Output:

```
mymultiset backwards: 30 20 20 10 10
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>multiset::begin</code>	Return iterator to beginning ( <a href="#">public member function</a> )
<code>multiset::crend</code>	Return <code>const_reverse_iterator</code> to reverse end ( <a href="#">public member function</a> )
<code>multiset::rbegin</code>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )

# /set/multiset/crend

public member function

## std::multiset::crend

`<set>`

```
const_reverse_iterator crend() const noexcept;
```

### Return `const_reverse_iterator` to reverse end

Returns a `const_reverse_iterator` pointing to the element that would theoretically precede the first element in the container (which is considered its *reverse end*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse end* of the sequence.

Member type `const_reverse_iterator` is a [bidirectional iterator](#) type that points to `const` elements.

## Example

```
1 // multiset::crbegin/crend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset = {10,20,30,20,10};
8
9     std::cout << "mymultiset backwards:" ;
10    for (auto rit=mymultiset.crbegin(); rit != mymultiset.crend(); ++rit)
11        std::cout << ' ' << *rit;
12
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
mymultiset backwards: 30 20 20 10 10
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.  
Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>multiset::end</code>	Return iterator to end (public member function )
<code>multiset::crbegin</code>	Return const_reverse_iterator to reverse beginning (public member function )
<code>multiset::rend</code>	Return reverse iterator to reverse end (public member function )

# /set/multiset/emplace

public member function

## std::multiset::emplace

<set>

```
template <class... Args>
iterator emplace (Args&&... args);
```

### Construct and insert element

Inserts a new element in the `multiset`. This new element is constructed in place using `args` as the arguments for its construction.

This effectively increases the container `size` by one.

Internally, `multiset` containers keep all their elements sorted following the criterion specified by its `comparison` object. The element is always inserted in its respective position following this ordering.

The element is constructed in-place by calling `allocator_traits::construct` with `args` forwarded.

A similar member function exists, `insert`, which either copies or moves existing objects into the container.

There are no guarantees on the relative order of equivalent elements.

The relative ordering of equivalent elements is preserved, and newly inserted elements follow their equivalents already in the container.

## Parameters

`args`  
Arguments forwarded to construct the new element.

## Return value

An iterator to the newly inserted element.

Member type `iterator` is a `bidirectional iterator` type that points to elements.

## Example

```
1 // multiset::emplace
2 #include <iostream>
3 #include <set>
4 #include <string>
5
6 int main ()
7 {
8     std::multiset<std::string> mymultiset;
9
10    mymultiset.emplace("foo");
11    mymultiset.emplace("bar");
12    mymultiset.emplace("foo");
13
14    std::cout << "mymultiset contains:";
15    for (const std::string& x: mymultiset)
16        std::cout << ' ' << x;
17    std::cout << '\n';
18
19    return 0;
20 }
```

Output:

```
mymultiset contains: bar foo foo
```

## Complexity

Logarithmic in the container size.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<code>multiset::emplace_hint</code>	Construct and insert element with hint (public member function )
<code>multiset::insert</code>	Insert element (public member function )
<code>multiset::erase</code>	Erase elements (public member function )

# /set/multiset/emplace\_hint

public member function

## std::multiset::emplace\_hint

<set>

```
template <class... Args>
iterator emplace_hint (const_iterator position, Args&&... args);
```

### Construct and insert element with hint

Inserts a new element in the `multiset`, with a hint on the insertion *position*. This new element is constructed in place using *args* as the arguments for its construction.

This effectively increases the container `size` by one.

The value in *position* is used as a hint on the insertion point. The element will nevertheless be inserted at its corresponding position following the order described by its internal `comparison object`, but this hint is used by the function to begin its search for the insertion point, speeding up the process considerably when the actual insertion point is either *position* or close to it.

The element is constructed in-place by calling `allocator_traits::construct` with *args* forwarded.

There are no guarantees on the relative order of equivalent elements.

The relative ordering of equivalent elements is preserved, and newly inserted elements follow their equivalents already in the container.

## Parameters

`position`

Hint for the position where the element can be inserted.

The function optimizes its insertion time if *position* points to the element that will follow the inserted element (or to the `end`, if it would be the last).

Notice that this does not force the new element to be in that position within the `multiset` container (the elements in a `multiset` always follow a specific order).

`const_iterator` is a member type, defined as a `bidirectional iterator` type that points to elements.

`args`

Arguments forwarded to construct the new element.

## Return value

An iterator to the newly inserted element.

Member type `iterator` is a `bidirectional iterator` type that points to elements.

## Example

```
1 // multiset::emplace_hint
2 #include <iostream>
3 #include <set>
4 #include <string>
5
6 int main ()
7 {
8     std::multiset<std::string> mymultiset;
9     auto it = mymultiset.cbegin();
10
11    mymultiset.emplace_hint (it,"apple");
12    it = mymultiset.emplace_hint (mymultiset.cend(),"orange");
13    it = mymultiset.emplace_hint (it,"melon");
14    mymultiset.emplace_hint (it,"melon");
15}
```

```

16 std::cout << "mymultiset contains:";
17 for (const std::string& x: mymultiset)
18     std::cout << ' ' << x;
19     std::cout << '\n';
20
21     return 0;
22 }
23

```

Output:

```
mymultiset contains: apple melon melon orange
```

## Complexity

Generally, logarithmic in the container `size`.

Amortized constant if the insertion point for the element is `position`.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<code>multiset::emplace</code>	Construct and insert element ( <a href="#">public member function</a> )
<code>multiset::insert</code>	Insert element ( <a href="#">public member function</a> )
<code>multiset::erase</code>	Erase elements ( <a href="#">public member function</a> )

# /set/multiset/empty

public member function

## std::multiset::empty

<set>

```
bool empty() const;
```

```
bool empty() const noexcept;
```

**Test whether container is empty**

Returns whether the `multiset` container is empty (i.e. whether its `size` is 0).

This function does not modify the container in any way. To clear the content of a `multiset` container, see `multiset::clear`.

## Parameters

none

## Return Value

true if the container `size` is 0, false otherwise.

## Example

```

1 // multiset::empty
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8
9     mymultiset.insert(10);
10    mymultiset.insert(20);
11    mymultiset.insert(10);
12
13    std::cout << "mymultiset contains:";
14    while (!mymultiset.empty())
15    {
16        std::cout << ' ' << *mymultiset.begin();
17        mymultiset.erase(mymultiset.begin());
18    }
19    std::cout << '\n';
20
21    return 0;
22 }

```

Output:

```
mymultiset contains: 10 10 20
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.  
Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>multiset::clear</code>	Clear content (public member function )
<code>multiset::erase</code>	Erase elements (public member function )
<code>multiset::size</code>	Return container size (public member function )

# /set/multiset/end

public member function

## std::multiset::end

<set>

```
iterator end();
const_iterator end() const;
iterator end() noexcept;
const_iterator end() const noexcept;
```

### Return iterator to end

Returns an iterator referring to the *past-the-end* element in the `multiset` container.

The *past-the-end* element is the theoretical element that would follow the last element in the `multiset` container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with `multiset::begin` to specify a range including all the elements in the container.

If the container is `empty`, this function returns the same as `multiset::begin`.

## Parameters

none

## Return Value

An iterator to the *past-the-end* element in the container.

If the `multiset` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are `bidirectional iterator` types pointing to elements.

## Example

```
1 // multiset::begin/end
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {15,98,77,77,39};
8     std::multiset<int> mymultiset (myints,myints+5);
9
10    std::cout << "mymultiset contains:";
11    for (std::multiset<int>::iterator it=mymultiset.begin(); it!=mymultiset.end(); ++it )
12        std::cout << ' ' << *it;
13
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

```
mymultiset contains: 15 39 77 77 98
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a [multiset](#) is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multiset::begin</a>	Return iterator to beginning (public member function )
<a href="#">multiset::rbegin</a>	Return reverse iterator to reverse beginning (public member function )
<a href="#">multiset::rend</a>	Return reverse iterator to reverse end (public member function )

## /set/multiset/equal\_range

public member function

### std::multiset::equal\_range

<set>

```
pair<iterator,iterator> equal_range (const value_type& val) const;
pair<const_iterator,const_iterator> equal_range (const value_type& val) const;
pair<iterator,iterator> equal_range (const value_type& val);
```

#### Get range of equal elements

Returns the bounds of a range that includes all the elements in the container that are equivalent to *val*.

If no matches are found, the range returned has a length of zero, with both iterators pointing to the first element that is considered to go after *val* according to the container's [internal comparison object](#) ([key\\_comp](#)).

Two elements of a [multiset](#) are considered equivalent if the container's [comparison object](#) returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

## Parameters

*val*

Value to search for.

Member type `value_type` is the type of the elements in the container, defined in [multiset](#) as an alias of its first template parameter (*T*).

## Return value

The function returns a `pair`, whose member `pair::first` is the lower bound of the range (the same as [lower\\_bound](#)), and `pair::second` is the upper bound (the same as [upper\\_bound](#)).

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements.

## Example

```
1 // multiset::equal_elements
2 #include <iostream>
3 #include <set>
4
5 typedef std::multiset<int>::iterator It; // aliasing the iterator type used
6
7 int main ()
8 {
9     int myints[] = {77,30,16,2,30,30};
10    std::multiset<int> mymultiset (myints, myints+6); // 2 16 30 30 30 77
11
12    std::pair<It,It> ret = mymultiset.equal_range(30); //      ^      ^
13
14    mymultiset.erase(ret.first,ret.second);
15
16    std::cout << "mymultiset contains:" ;
17    for (It it=mymultiset.begin(); it!=mymultiset.end(); ++it)
18        std::cout << ' ' << *it;
19    std::cout << '\n';
20
21    return 0;
22 }
```

```
mymultiset contains: 2 16 77
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>multiset::count</code>	Count elements with a specific key (public member function )
<code>multiset::lower_bound</code>	Return iterator to lower bound (public member function )
<code>multiset::upper_bound</code>	Return iterator to upper bound (public member function )
<code>multiset::find</code>	Get iterator to element (public member function )

## /set/multiset/erase

public member function

### std::multiset::erase

<set>

```
(1)     void erase (iterator position);
(2) size_type erase (const value_type& val);
(3)     void erase (iterator first, iterator last);

(1) iterator erase (const_iterator position);
(2) size_type erase (const value_type& val);
(3) iterator erase (const_iterator first, const_iterator last);
```

#### Erase elements

Removes elements from the `multiset` container.

This effectively reduces the container `size` by the number of elements removed, which are destroyed.

The parameters determine the elements removed:

#### Parameters

##### position

Iterator pointing to a single element to be removed from the `multiset`. Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types that point to elements.

##### val

Value to be removed from the `multiset`. All elements with a value equivalent to this are removed from the container. Member type `value_type` is the type of the elements in the container, defined in `multiset` as an alias of its first template parameter (`T`).

##### first, last

Iterators specifying a range within the `multiset` container to be removed: `[first, last]`. i.e., the range includes all the elements between `first` and `last`, including the element pointed by `first` but not the one pointed by `last`. Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types that point to elements.

#### Return value

For the value-based version (2), the function returns the number of elements erased.

Member type `size_type` is an unsigned integral type.

The other versions return no value.

The other versions return an iterator to the element that follows the last element removed (or `multiset::end`, if the last element was removed).

Member type `iterator` is a [bidirectional iterator](#) type that points to elements.

#### Example

```
1 // erasing from multiset
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8     std::multiset<int>::iterator it;
9
10    // insert some values:
11    mymultiset.insert (40);                                // 40
12    for (int i=1; i<7; i++) mymultiset.insert(i*10);      // 10 20 30 40 40 50 60
13
14    it=mymultiset.begin();                                // ^
15    it++;                                                 //
16
17    mymultiset.erase (it);                                // 10 30 40 40 50 60
18
19    mymultiset.erase (40);                                // 10 30 50 60
20
21    it=mymultiset.find (50);
22    mymultiset.erase ( it, mymultiset.end() );            // 10 30
23
24    std::cout << "mymultiset contains:";
25    for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)
```

```

27     std::cout << ' ' << *it;
28     std::cout << '\n';
29
30     return 0;
}

```

Output:

```
mymultiset contains: 10 30
```

## Complexity

For the first version (`erase(position)`), amortized constant.

For the second version (`erase(val)`), logarithmic in container `size`, plus linear in the number of elements removed.

For the last version (`erase(first,last)`), linear in the distance between `first` and `last`.

## Iterator validity

Iterators, pointers and references referring to elements removed by the function are invalidated.

All other iterators, pointers and references keep their validity.

## Data races

The container is modified.

The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the container is not.

## Exception safety

Unless the container's `comparison object` throws, this function never throws exceptions (no-throw guarantee).

Otherwise, if a single element is to be removed, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If an invalid `position` or range is specified, it causes *undefined behavior*.

## See also

<code>multiset::clear</code>	Clear content (public member function )
<code>multiset::insert</code>	Insert element (public member function )
<code>multiset::find</code>	Get iterator to element (public member function )

# /set/multiset/find

public member function

## std::multiset::find

<set>

```

iterator find (const value_type& val) const;
const_iterator find (const value_type& val) const;
iterator      find (const value_type& val);

```

### Get iterator to element

Searches the container for an element equivalent to `val` and returns an iterator to it if found, otherwise it returns an iterator to `multiset::end`.

Notice that this function returns an iterator to a single element (of the possibly multiple equivalent elements). To obtain the entire range of equivalent elements, see `multiset::equal_range`.

Two elements of a `multiset` are considered equivalent if the container's `comparison object` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

## Parameters

`val`

Value to be searched for.

Member type `value_type` is the type of the elements in the container, defined in `multiset` as an alias of its first template parameter (`T`).

## Return value

An iterator to the element, if `val` is found, or `multiset::end` otherwise.

Member types `iterator` and `const_iterator` are `bidirectional iterator` types pointing to elements.

## Example

```

1 // multiset::find
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8     std::multiset<int>::iterator it;
9
10    // set some initial values:
11    for (int i=1; i<=5; i++) mymultiset.insert(i*10);    // 10 20 30 40 50
12
13    it=mymultiset.find(20);
14    mymultiset.erase (it);
15    mymultiset.erase (mymultiset.find(40));
16

```

```

17 std::cout << "mymultiset contains:";
18 for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)
19   std::cout << ' ' << *it;
20 std::cout << '\n';
21
22 return 0;
23 }

```

Output:

```
mymultiset contains: 10 30 50
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container). Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>multiset::equal_range</code>	Get range of equal elements ( <a href="#">public member function</a> )
<code>multiset::count</code>	Count elements with a specific key ( <a href="#">public member function</a> )
<code>multiset::lower_bound</code>	Return iterator to lower bound ( <a href="#">public member function</a> )
<code>multiset::upper_bound</code>	Return iterator to upper bound ( <a href="#">public member function</a> )

## /set/multiset/get\_allocator

public member function

### std::multiset::get\_allocator

<set>

```
allocator_type get_allocator() const;
allocator_type get_allocator() const noexcept;
```

#### Get allocator

Returns a copy of the allocator object associated with the `multiset`.

## Parameters

none

## Return Value

The allocator.

Member type `allocator_type` is the type of the allocator used by the container, defined in `multiset` as an alias of its third template parameter (`Alloc`).

## Example

```

1 // multiset::get_allocator
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7   std::multiset<int> mymultiset;
8   int * p;
9   unsigned int i;
10
11  // allocate an array of 5 elements using myset's allocator:
12  p=mymultiset.get_allocator().allocate(5);
13
14  // assign some values to array
15  for (i=0; i<5; i++) p[i]=(i+1)*10;
16
17  std::cout << "The allocated array contains:";
18  for (i=0; i<5; i++) std::cout << ' ' << p[i];
19  std::cout << '\n';
20
21  mymultiset.get_allocator().deallocate(p,5);
22
23  return 0;
24 }

```

The example shows an elaborate way to allocate memory for an array of `ints` using the same allocator used by the container.  
Output:

```
The allocated array contains: 10 20 30 40 50
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

Copying any instantiation of the `default allocator` is also guaranteed to never throw.

## See also

<code>allocator</code>	Default allocator (class template )
------------------------	-------------------------------------

# /set/multiset/insert

public member function

## `std::multiset::insert`

<set>

```
single element (1) iterator insert (const value_type& val);
with hint (2) iterator insert (iterator position, const value_type& val);
range (3) template <class InputIterator>
           void insert (InputIterator first, InputIterator last);

single element (1) iterator insert (const value_type& val);
iterator insert (value_type& val);
with hint (2) iterator insert (const_iterator position, const value_type& val);
iterator insert (const_iterator position, value_type& val);
range (3) template <class InputIterator>
           void insert (InputIterator first, InputIterator last);
initializer list (4) void insert (initializer_list<value_type> il);
```

### Insert element

Extends the container by inserting new elements, effectively increasing the container `size` by the number of elements inserted.

Internally, `multiset` containers keep all their elements sorted following the criterion specified by its `comparison` object. The elements are always inserted in its respective position following this ordering.

There are no guarantees on the relative order of equivalent elements.

The relative ordering of equivalent elements is preserved, and newly inserted elements follow their equivalents already in the container.

The parameters determine how many elements are inserted and to which values they are initialized:

## Parameters

### val

Value to be copied (or moved) to the inserted elements.

Member type `value_type` is the type of the elements in the container, defined in `multiset` as an alias of its first template parameter (`T`).

### position

Hint for the position where the element can be inserted.

The function optimizes its insertion time if `position` points to the element that will **precede** the inserted element.

The function optimizes its insertion time if `position` points to the element that will **follow** the inserted element (or to the `end`, if it would be the last).

Notice that this is just a hint and does not force the new element to be inserted at that position within the `multiset` container (the elements in a `multiset` always follow a specific order).

Member types `iterator` and `const_iterator` are defined in `map` as a **bidirectional iterator** type that point to elements.

### first, last

Iterators specifying a range of elements. Copies of the elements in the range `[first, last)` are inserted in the container.

Notice that the range includes all the elements between `first` and `last`, including the element pointed by `first` but not the one pointed by `last`.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type from which `value_type` objects can be constructed.

### il

An `initializer_list` object. Copies of these elements are inserted.

These objects are automatically constructed from `initializer_list` declarators.

Member type `value_type` is the type of the elements in the container, defined in `multiset` as an alias of its first template parameter (`T`).

## Return value

In the versions returning a value, this is an iterator pointing to the newly inserted element in the `multiset`.

Member type `iterator` is a **bidirectional iterator** type that points to elements.

## Example

```

1 // multiset::insert (C++98)
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8     std::multiset<int>::iterator it;
9
10    // set some initial values:
11    for (int i=1; i<=5; i++) mymultiset.insert(i*10); // 10 20 30 40 50
12
13    it=mymultiset.insert(25);
14
15    it=mymultiset.insert (it,27); // max efficiency inserting
16    it=mymultiset.insert (it,29); // max efficiency inserting
17    it=mymultiset.insert (it,24); // no max efficiency inserting (24<29)
18
19    int myints[] = {5,10,15};
20    mymultiset.insert (myints,myints+3);
21
22    std::cout << "mymultiset contains:";
23    for (it=mymultiset.begin(); it!=mymultiset.end(); ++it)
24        std::cout << ' ' << *it;
25    std::cout << '\n';
26
27    return 0;
28 }

```

Output:

```
myset contains: 5 10 10 15 20 24 25 27 29 30 40 50
```

## Complexity

For the first version (`insert(x)`), logarithmic.

For the second version (`insert(position,x)`), logarithmic in general, but amortized constant if `x` is inserted right after the element pointed by `position`.

For the third version (`insert (first,last)`),  $N \log(\text{size}+N)$  in general (where `N` is the distance between `first` and `last`, and `size` the `size` of the container before the insertion), but linear if the elements between `first` and `last` are already sorted according to the same ordering criterion used by the container.

## Complexity

If a single element is inserted, logarithmic in `size` in general, but amortized constant if a hint is given and the `position` given is the optimal.

If `N` elements are inserted,  $N \log(\text{size}+N)$  in general, but linear in `size+N` if the elements are already sorted according to the same ordering criterion used by the container.

If `N` elements are inserted,  $N \log(\text{size}+N)$ .

Implementations may optimize if the range is already sorted.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

If a single element is to be inserted, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if an invalid `position` is specified, it causes undefined behavior.

## See also

<code>multiset::erase</code>	Erase elements (public member function )
<code>multiset::find</code>	Get iterator to element (public member function )

## /set/multiset/key\_comp

public member function

### `std::multiset::key_comp`

<set>

`key_compare key_comp() const;`

#### Return comparison object

Returns a copy of the `comparison` object used by the container.

By default, this is a `less` object, which returns the same as `operator<`.

This object determines the order of the elements in the container: it is a function pointer or a function object that takes two arguments of the same type as the container elements, and returns `true` if the first argument is considered to go before the second in the *strict weak ordering* it defines, and `false` otherwise.

Two elements of a `multiset` are considered equivalent if `key_comp` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

In `multiset` containers, the keys to sort the elements are the values themselves, therefore `key_comp` and its sibling member function `value_comp` are equivalent.

## Parameters

none

## Return value

The comparison object.

Member type `key_compare` is the type of the *comparison object* associated to the container, defined in `multiset` as an alias of its second template parameter (`Compare`).

## Example

```
1 // multiset::key_comp
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8
9     for (int i=0; i<5; i++) mymultiset.insert(i);
10
11    std::multiset<int>::key_compare mycomp = mymultiset.key_comp();
12
13    std::cout << "mymultiset contains:";
14
15    int highest = *mymultiset.rbegin();
16
17    std::multiset<int>::iterator it = mymultiset.begin();
18    do {
19        std::cout << ' ' << *it;
20    } while ( mycomp(*it++,highest) );
21
22    std::cout << '\n';
23
24    return 0;
25 }
```

Output:

```
mymultiset contains: 0 1 2 3 4
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>multiset::value_comp</code>	Return comparison object (public member function )
<code>multiset::find</code>	Get iterator to element (public member function )
<code>multiset::count</code>	Count elements with a specific key (public member function )
<code>multiset::lower_bound</code>	Return iterator to lower bound (public member function )
<code>multiset::upper_bound</code>	Return iterator to upper bound (public member function )

## /set/multiset/lower\_bound

public member function

### std::multiset::lower\_bound

<set>

```
iterator lower_bound (const value_type& val) const;
const_iterator lower_bound (const value_type& val) const;
    iterator lower_bound (const value_type& val);
```

#### Return iterator to lower bound

Returns an iterator pointing to the first element in the container which is not considered to go before `val` (i.e., either it is equivalent or goes after).

The function uses its internal `comparison object` (`key_comp`) to determine this, returning an iterator to the first element for which `key_comp(element, val)` would return `false`.

If the `multiset` class is instantiated with the default comparison type (`less`), the function returns an iterator to the first element that is not less than `val`.

A similar member function, `upper_bound`, has the same behavior as `lower_bound`, except in the case that the `multiset` contains elements equivalent to `val`: In this case `lower_bound` returns an iterator pointing to the first of such elements, whereas `upper_bound` returns an iterator pointing to the element following the last.

## Parameters

val

Value to compare.  
Member type `value_type` is the type of the elements in the container, defined in `multiset` as an alias of its first template parameter (`T`).

## Return value

An iterator to the the first element in the container which is not considered to go before `val`, or `multiset::end` if all elements are considered to go before `val`.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements.

## Example

```
1 // multiset::lower_bound/upper_bound
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8     std::multiset<int>::iterator itlow,itup;
9
10    for (int i=1; i<8; i++) mymultiset.insert(i*10); // 10 20 30 40 50 60 70
11
12    itlow = mymultiset.lower_bound (30);           //      ^
13    itup = mymultiset.upper_bound (40);           //      ^
14
15    mymultiset.erase(itlow,itup);                 // 10 20 50 60 70
16
17    std::cout << "mymultiset contains:";
18    for (std::multiset<int>::iterator it=mymultiset.begin(); it!=mymultiset.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Notice that `lower_bound(30)` returns an iterator to 30, whereas `upper_bound(40)` returns an iterator to 50.

```
mymultiset contains: 10 20 50 60 70
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>multiset::upper_bound</code>	Return iterator to upper bound ( <a href="#">public member function</a> )
<code>multiset::equal_range</code>	Get range of equal elements ( <a href="#">public member function</a> )
<code>multiset::find</code>	Get iterator to element ( <a href="#">public member function</a> )
<code>multiset::count</code>	Count elements with a specific key ( <a href="#">public member function</a> )

## /set/multiset/max\_size

public member function

**std::multiset::max\_size**

`<set>`

```
size_type max_size() const;
size_type max_size() const noexcept;
```

### Return maximum size

Returns the maximum number of elements that the `multiset` container can hold.

This is the maximum potential `size` the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size: it can still fail to allocate storage at any point before that size is reached.

## Parameters

none

## Return Value

The maximum number of elements a `multiset` container can hold as content.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // multiset::max_size
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8
9     if (mymultiset.max_size()>1000)
10    {
11        for (int i=0; i<1000; i++) mymultiset.insert(i);
12        std::cout << "The multiset contains 1000 elements.\n";
13    }
14    else std::cout << "The multiset could not hold 1000 elements.\n";
15
16    return 0;
17 }
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a `multiset` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

`multiset::size`      Return container size (public member function )

# /set/multiset/multiset

public member function

## std::multiset::multiset

<set>

```
empty (1) explicit multiset (const key_compare& comp = key_compare(),
                             const allocator_type& alloc = allocator_type());
template <class InputIterator>
multiset (InputIterator first, InputIterator last,
          const key_compare& comp = key_compare(),
          const allocator_type& alloc = allocator_type());
range (2) multiset (InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type());
copy (3) multiset (const multiset& x);

empty (1) explicit multiset (const key_compare& comp = key_compare(),
                           const allocator_type& alloc = allocator_type());
explicit multiset (const allocator_type& alloc);
template <class InputIterator>
range (2) multiset (InputIterator first, InputIterator last,
                     const key_compare& comp = key_compare(),
                     const allocator_type& alloc = allocator_type());
copy (3) multiset (const multiset& x);
multiset (const multiset& x, const allocator_type& alloc);
move (4) multiset (multiset&& x, const allocator_type& alloc);
multiset (initializer_list<value_type> il,
          const key_compare& comp = key_compare(),
          const allocator_type& alloc = allocator_type());

multiset();
empty (1) explicit multiset (const key_compare& comp,
                           const allocator_type& alloc = allocator_type());
explicit multiset (const allocator_type& alloc);
template <class InputIterator>
multiset (InputIterator first, InputIterator last,
          const key_compare& comp = key_compare(),
          const allocator_type& alloc = allocator_type());
range (2) multiset (InputIterator first, InputIterator last,
                     const allocator_type& alloc = allocator_type());
template <class InputIterator>
multiset (InputIterator first, InputIterator last,
          const allocator_type& alloc = allocator_type());
copy (3) multiset (const multiset& x);
multiset (const multiset& x, const allocator_type& alloc);
move (4) multiset (multiset&& x, const allocator_type& alloc);
```

```

multiset (initializer_list<value_type> il,
           const key_compare& comp = key_compare(),
           const allocator_type& alloc = allocator_type());
multiset (initializer_list<value_type> il,
           const allocator_type& alloc = allocator_type());

```

## Construct multiset

Constructs a `multiset` container object, initializing its contents depending on the constructor version used:

### (1) empty container constructor (default constructor)

Constructs an `empty` container, with no elements.

### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element constructed from its corresponding element in that range.

### (3) copy constructor

Constructs a container with a copy of each of the elements in `x`.

The container keeps an internal copy of `alloc` and `comp`, which are used to allocate storage and to sort the elements throughout its lifetime. The copy constructor (3) creates a container that keeps and uses copies of `x`'s `allocator` and `comparison object`.

The storage for the elements is allocated using this `internal allocator`.

The elements are sorted according to the `comparison object`.

### (1) empty container constructors (default constructor)

Constructs an `empty` container, with no elements.

### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element `emplace-constructed` from its corresponding element in that range.

### (3) copy constructor (and copying with allocator)

Constructs a container with a copy of each of the elements in `x`.

### (4) move constructor (and moving with allocator)

Constructs a container that acquires the elements of `x`.

If `alloc` is specified and is different from `x`'s allocator, the elements are moved. Otherwise, no elements are constructed (their ownership is directly transferred).

`x` is left in an unspecified but valid state.

### (5) initializer list constructor

Constructs a container with a copy of each of the elements in `il`.

The container keeps an internal copy of `alloc`, which is used to allocate and deallocate storage for its elements, and to construct and destroy them (as specified by its `allocator_traits`). If no `alloc` argument is passed to the constructor, a default-constructed allocator is used, except in the following cases:

- The copy constructor (3, first signature) creates a container that keeps and uses a copy of the allocator returned by calling the appropriate `selected_on_container_copy_construction` trait on `x`'s allocator.

- The move constructor (4, first signature) acquires `x`'s allocator.

The container also keeps an internal copy of `comp` (or `x`'s `comparison object`), which is used to establish the order of the elements in the container and to check for equivalent elements.

All elements are `copied`, `moved` or otherwise `constructed` by calling `allocator_traits::construct` with the appropriate arguments.

The elements are sorted according to the `comparison object`. If elements that are equivalent are passed to the constructor, their relative order is preserved.

## Parameters

### comp

Binary predicate that, taking two values of the same type of those contained in the `multiset`, returns `true` if the first argument goes before the second argument in the *strict weak ordering* it defines, and `false` otherwise.

This shall be a function pointer or a function object.

Member type `key_compare` is the internal comparison object type used by the container, defined in `multiset` as an alias of its second template parameter (`Compare`).

If `key_compare` uses the default `less` (which has no state), this parameter is not relevant.

### alloc

Allocator object.

The container keeps and uses an internal copy of this allocator.

Member type `allocator_type` is the internal allocator type used by the container, defined in `multiset` as an alias of its third template parameter (`Alloc`). If `allocator_type` is an instantiation of the default `allocator` (which has no state), this parameter is not relevant.

### first, last

`Input iterators` to the initial and final positions in a range. The range used is `[first, last)`, which includes all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type from which `value_type` objects can be constructed.

### x

Another `multiset` object of the same type (with the same class template arguments `T`, `Compare` and `Alloc`), whose contents are either copied or acquired.

### il

An `initializer_list` object.

These objects are automatically constructed from `initializer_list` declarators.

Member type `value_type` is the type of the elements in the container, defined in `multiset` as an alias of its first template parameter (`T`).

## Example

```

1 // constructing multisets
2 #include <iostream>
3 #include <set>
4
5 bool fncomp (int lhs, int rhs) {return lhs<rhs;}
6
7 struct classcomp {

```

```

8  bool operator() (const int& lhs, const int& rhs) const
9  {return lhs<rhs;}
10 };
11
12 int main ()
13 {
14     std::multiset<int> first;                                // empty multiset of ints
15
16     int myints[] = {10,20,30,20,20};                         std::multiset<int> second (myints,myints+5);           // pointers used as iterators
17
18     std::multiset<int> third (second);                      // a copy of second
19
20     std::multiset<int> fourth (second.begin(), second.end()); // iterator ctor.
21
22     std::multiset<int, classcomp> fifth;                     // class as Compare
23
24     bool(*fn_pt)(int,int) = fncomp;
25     std::multiset<int, bool(*)(int,int)> sixth (fn_pt); // function pointer as Compare
26
27     return 0;
28 }

```

The code does not produce any output, but demonstrates some ways in which a `multiset` container can be constructed.

### **Complexity**

Constant for the *empty constructors* (1), and for the *move constructors* (4) (unless `alloc` is different from `x`'s allocator). For all other cases, linear in the distance between the iterators (copy constructions) if the elements are already sorted according to the same criterion. For unsorted sequences, linearithmic ( $N \log N$ ) in that distance (sorting,copy constructions).

### **Iterator validity**

The *move constructors* (4), invalidate all iterators, pointers and references related to `x` if the elements are moved.

### **Data races**

All copied elements are accessed.  
The *move constructors* (4) modify `x`.

### **Exception safety**

**Strong guarantee:** no effects in case an exception is thrown.  
If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

### **See also**

<code>multiset::operator=</code>	Copy container content (public member function )
<code>multiset::insert</code>	Insert element (public member function )

## /set/multiset/~multiset

public member function

### **std::multiset::~multiset**

<set>

#### `~multiset();`

#### **Multiset destructor**

Destroys the container object.

This destroys all container elements, and deallocates all the storage capacity allocated by the `multiset` container using its `allocator`.

This calls `allocator_traits::destroy` on each of the contained elements, and deallocates all the storage capacity allocated by the `multiset` container using its `allocator`.

### **Complexity**

Linear in `multiset::size` (destructors).

### **Iterator validity**

All iterators, pointers and references are invalidated.

### **Data races**

The container and all its elements are modified.

### **Exception safety**

**No-throw guarantee:** never throws exceptions.

## /set/multiset/operator=

public member function

## std::multiset::operator=

<set>

```
copy (1) multiset& operator= (const multiset& x);
copy (1) multiset& operator= (const multiset& x);
move (2) multiset& operator= (multiset&& x);
initializer list (3) multiset& operator= (initializer_list<value_type> il);
```

### Copy container content

Assigns new contents to the container, replacing its current content.

Copies all the elements from *x* into the container, changing its `size` accordingly.

The container preserves its `current_allocator`, which is used to allocate additional storage if needed.

The `copy assignment` (1) copies all the elements from *x* into the container (with *x* preserving its contents).

The `move assignment` (2) moves the elements of *x* into the container (*x* is left in an unspecified but valid state).

The `initializer list assignment` (3) copies the elements of *il* into the container.

The new container `size` is the same as the `size` of *x* (or *il*) before the call.

The container preserves its `current_allocator`, except if the `allocator_traits` indicate *x*'s allocator should `propagate`.

This `allocator` is used (through its `traits`) to `allocate` or `deallocate` if there are changes in storage requirements, and to `construct` or `destroy` elements, if needed.

The elements stored in the container before the call are either assigned to or destroyed.

### Parameters

*x*  
A `multiset` object of the same type (i.e., with the same template parameters, *T*, `Compare` and `Alloc`).

*il*  
An `initializer_list` object. The compiler will automatically construct such objects from `initializer_list` declarators.  
Member type `value_type` is the type of the elements in the container, defined in `multiset` as an alias of its first template parameter (*T*).

### Return value

`*this`

### Example

```
1 // assignment operator with multisets
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = { 19, 81, 36, 36, 19 };
8     std::multiset<int> first (myints, myints + 5);    // multiset with 5 ints
9     std::multiset<int> second;                          // empty multiset
10
11    second = first;                                    // now second contains the 5 ints
12    first = std::multiset<int>();                      // and first is empty
13
14    std::cout << "Size of first: " << first.size() << '\n';
15    std::cout << "Size of second: " << second.size() << '\n';
16    return 0;
17 }
```

Output:

```
Size of first: 0
Size of second: 5
```

### Complexity

For the `copy assignment` (1): Linear in sizes (destructions, copies).

For the `move assignment` (2): Linear in current container `size` (destructions).\*

For the `initializer list assignment` (3): Up to logarithmic in sizes (destructions, move-assignments) -- linear if *il* is already sorted.

\* Additional complexity for assignments if allocators do not `propagate`.

### Iterator validity

All iterators, references and pointers related to this container are invalidated.

In the `move assignment`, iterators, pointers and references referring to elements in *x* are also invalidated.

### Data races

All copied elements are accessed.

The `move assignment` (2) modifies *x*.

The container and all its elements are modified.

### Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if `value_type` is not `copy assignable` (or `move assignable` for (2)), it causes `undefined behavior`.

## See also

<a href="#">multiset::insert</a>	Insert element (public member function )
<a href="#">multiset::multiset</a>	Construct multiset (public member function )

# /set/multiset/operators

function

## std::relational operators (multiset)

<set>

```
template <class T, class Compare, class Allocator>
(1)  bool operator== ( const multiset<T,Compare,Allocator>& lhs,
                     const multiset<T,Compare,Allocator>& rhs );
template <class T, class Compare, class Allocator>
(2)  bool operator!= ( const multiset<T,Compare,Allocator>& lhs,
                     const multiset<T,Compare,Allocator>& rhs );
template <class T, class Compare, class Allocator>
(3)  bool operator< ( const multiset<T,Compare,Allocator>& lhs,
                     const multiset<T,Compare,Allocator>& rhs );
template <class T, class Compare, class Allocator>
(4)  bool operator<= ( const multiset<T,Compare,Allocator>& lhs,
                     const multiset<T,Compare,Allocator>& rhs );
template <class T, class Compare, class Allocator>
(5)  bool operator> ( const multiset<T,Compare,Allocator>& lhs,
                     const multiset<T,Compare,Allocator>& rhs );
template <class T, class Compare, class Allocator>
(6)  bool operator>= ( const multiset<T,Compare,Allocator>& lhs,
                     const multiset<T,Compare,Allocator>& rhs );
```

### Relational operators for multiset

Performs the appropriate comparison operation between the [multiset](#) containers *lhs* and *rhs*.

The *equality comparison* (`operator==`) is performed by first comparing [sizes](#), and if they match, the elements are compared sequentially using `operator==`, stopping at the first mismatch (as if using algorithm `equal`).

The *less-than comparison* (`operator<`) behaves as if using algorithm `lexicographical_compare`, which compares the elements sequentially using `operator<` in a reciprocal manner (i.e., checking both `a<b` and `b<a`) and stopping at the first occurrence.

The other operations also use the operators `==` and `<` internally to compare the elements, behaving as if the following equivalent operations were performed:

operation	equivalent operation
<code>a!=b</code>	<code>!(a==b)</code>
<code>a&gt;b</code>	<code>b&lt;a</code>
<code>a&lt;=b</code>	<code>!(b&lt;a)</code>
<code>a&gt;=b</code>	<code>!(a&lt;b)</code>

Notice that none of these operations take into consideration the [internal comparison object](#) of neither container.

These operators are overloaded in header `<set>`.

## Parameters

*lhs*, *rhs*

[multiset](#) containers (to the left- and right-hand side of the operator, respectively), having both the same template parameters (*T*, *Compare* and *Alloc*).

## Example

```
1 // multiset comparisons
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> foo,bar;
8     foo.insert(10);
9     bar.insert(20);
10    bar.insert(20);
11    foo.insert(30);
12
13    // foo ({10,30}) vs bar ({20,20}):
14    if (foo==bar) std::cout << "foo and bar are equal\n";
15    if (foo!=bar) std::cout << "foo and bar are not equal\n";
16    if (foo< bar) std::cout << "foo is less than bar\n";
17    if (foo> bar) std::cout << "foo is greater than bar\n";
18    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
19    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
20
21    return 0;
22 }
```

Output:

```
foo and bar are not equal
foo is less than bar
foo is less than or equal to bar
```

## Return Value

true if the condition holds, and false otherwise.

## Complexity

Up to linear in the [size](#) of *lhs* and *rhs*.

For (1) and (2), constant if the [sizes](#) of *lhs* and *rhs* differ, and up to linear in that [size](#) (equality comparisons) otherwise.  
For the others, up to linear in the smaller [size](#) (each representing two comparisons with [operator<](#)).

## Iterator validity

No changes.

## Data races

Both containers, *lhs* and *rhs*, are accessed.

Concurrently accessing the elements of unmodified [set](#) objects is always safe (their elements are *immutable*).

## Exception safety

If the type of the elements supports the appropriate operation with no-throw guarantee, the function never throws exceptions (no-throw guarantee).  
In any case, the function cannot modify its arguments.

## See also

<a href="#">multiset::value_comp</a>	Return comparison object (public member function )
<a href="#">multiset::operator=</a>	Copy container content (public member function )
<a href="#">multiset::swap</a>	Swap content (public member function )

# /set/multiset/rbegin

public member function

## std::multiset::rbegin

<set>

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
```

### Return reverse iterator to reverse beginning

Returns a [reverse iterator](#) pointing to the last element in the container (i.e., its *reverse beginning*).

Reverse [iterators](#) iterate backwards: increasing them moves them towards the beginning of the container.

*rbegin* points to the element preceding the one that would be pointed to by member [end](#).

## Parameters

none

## Return Value

A reverse iterator to the *reverse beginning* of the sequence container.

If the [multiset](#) object is const-qualified, the function returns a [const\\_reverse\\_iterator](#). Otherwise, it returns a [reverse\\_iterator](#).

Member types [reverse\\_iterator](#) and [const\\_reverse\\_iterator](#) are reverse [bidirectional iterator](#) types pointing to elements. See [multiset member types](#).

## Example

```
1 // multiset::rbegin/rend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {77,16,2,30,30};
8     std::multiset<int> mymultiset (myints,myints+5);
9
10    std::cout << "mymultiset contains:";
11    for (std::multiset<int>::reverse_iterator rit=mymultiset.rbegin(); rit!=mymultiset.rend(); ++rit)
12        std::cout << ' ' << *rit;
13
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

```
mymultiset contains: 77 30 30 16 2
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a [multiset](#) is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multiset::rend</a>	Return reverse iterator to reverse end (public member function )
<a href="#">multiset::begin</a>	Return iterator to beginning (public member function )
<a href="#">multiset::end</a>	Return iterator to end (public member function )

## /set/multiset/rend

public member function

### std::multiset::rend

<set>

```
    reverse_iterator rend();
const_reverse_iterator rend() const;
    reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

#### Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element right before the first element in the [multiset](#) container (which is considered its *reverse end*).

The range between [multiset::rbegin](#) and [multiset::rend](#) contains all the elements of the container, in reverse order.

## Parameters

none

## Return Value

A *reverse iterator* to the *reverse end* of the sequence container.

If the [multiset](#) object is const-qualified, the function returns a *const\_reverse\_iterator*. Otherwise, it returns a *reverse\_iterator*.

Member types *reverse\_iterator* and *const\_reverse\_iterator* are *reverse bidirectional iterator* types pointing to elements. See [multiset member types](#).

## Example

```
1 // multiset::rbegin/rend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {77,16,2,30,30};
8     std::multiset<int> mymultiset (myints,myints+5);
9
10    std::cout << "mymultiset contains:" ;
11    for (std::multiset<int>::reverse_iterator rit=mymultiset.rbegin() ; rit!=mymultiset.rend(); ++rit)
12        std::cout << ' ' << *rit;
13
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

```
mymultiset contains: 77 30 30 16 2
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a [set](#) is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">multiset::rbegin</a>	Return reverse iterator to reverse beginning (public member function )
<a href="#">multiset::begin</a>	Return iterator to beginning (public member function )
<a href="#">multiset::end</a>	Return iterator to end (public member function )

## /set/multiset/size

public member function

### std::multiset::size

<set>

```
size_type size() const;
size_type size() const noexcept;
```

#### Return container size

Returns the number of elements in the [multiset](#) container.

#### Parameters

none

#### Return Value

The number of elements in the container.

Member type `size_type` is an unsigned integral type.

#### Example

```
1 // multiset::size
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> myints;
8     std::cout << "0. size: " << myints.size() << '\n';
9
10    for (int i=0; i<10; i++) myints.insert(i);
11    std::cout << "1. size: " << myints.size() << '\n';
12
13    myints.insert (5);
14    std::cout << "2. size: " << myints.size() << '\n';
15
16    myints.erase (5);
17    std::cout << "3. size: " << myints.size() << '\n';
18
19    return 0;
20 }
```

Output:

```
0. size: 0
1. size: 10
2. size: 11
3. size: 9
```

#### Complexity

Constant.

#### Iterator validity

No changes.

#### Data races

The container is accessed.

Concurrently accessing the elements of a [multiset](#) is safe.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">multiset::max_size</a>	Return maximum size (public member function )
<a href="#">multiset::empty</a>	Test whether container is empty (public member function )

## /set/multiset/swap

public member function

### std::multiset::swap

<set>

```
void swap (multiset& x);
```

## Swap content

Exchanges the content of the container by the content of *x*, which is another [multiset](#) of the same type. Sizes may differ.

After the call to this member function, the elements in this container are those which were in *x* before the call, and the elements of *x* are those which were in this. All iterators, references and pointers remain valid for the swapped objects.

Notice that a non-member function exists with the same name, [swap](#), overloading that algorithm with an optimization that behaves like this member function.

Whether the internal container [allocators](#) and [comparison objects](#) are swapped is undefined.

Whether the internal container [allocators](#) are swapped is not defined, unless in the case the appropriate [allocator trait](#) indicates explicitly that they shall propagate.

The internal [comparison objects](#) are always exchanged, using [swap](#).

## Parameters

*x*

Another [multiset](#) container of the same type as this (i.e., with the same template parameters, *T*, *Compare* and *Alloc*) whose content is swapped with that of this container.

## Return value

none

## Example

```
1 // swap multisets
2 #include <iostream>
3 #include <set>
4
5 main ()
6 {
7     int myints[] = {19, 72, 4, 36, 20, 20};
8     std::multiset<int> first (myints, myints+3);      // 4, 19, 72
9     std::multiset<int> second (myints+3, myints+6);    // 20, 20, 36
10
11    first.swap(second);
12
13    std::cout << "first contains:";
14    for (std::multiset<int>::iterator it=first.begin(); it!=first.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    std::cout << "second contains:";
19    for (std::multiset<int>::iterator it=second.begin(); it!=second.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
first contains: 20 20 36
second contains: 4 19 72
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.

Note that the *end iterators* do not refer to elements and may be invalidated.

## Data races

Both the container and *x* are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both containers compare equal, or if their [allocator traits](#) indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">swap (multiset)</a>	Exchanges the contents of two multisets ( <a href="#">function template</a> )
<a href="#">swap_ranges</a>	Exchange values of two ranges ( <a href="#">function template</a> )

# /set/multiset/swap-free

function template

## std::swap (multiset)

&lt;set&gt;

```
template <class T, class Compare, class Alloc>
void swap (multiset<T,Compare,Alloc>& x, multiset<T,Compare,Alloc>& y);
```

### Exchanges the contents of two multisets

The contents of container *x* are exchanged with those of *y*. Both container objects must be of the same type (same template parameters), although sizes may differ.

After the call to this member function, the elements in *x* are those which were in *y* before the call, and the elements of *y* are those which were in *x*. All iterators, references and pointers remain valid for the swapped objects.

This is an overload of the generic algorithm [swap](#) that improves its performance by mutually transferring ownership over their assets to the other container (i.e., the containers exchange references to their data, without actually performing any element copy or movement): It behaves as if *x.swap(y)* was called.

### Parameters

*x,y* multiset containers of the same type (i.e., having both the same template parameters, *T*, *Compare* and *Alloc*).

### Return value

none

### Example

```
1 // swap (multiset overload)
2 #include <iostream>
3 #include <set>
4
5 main ()
6 {
7     int myints[] = {12, 75, 12, 35, 20, 35};
8     std::multiset<int> first (myints, myints+3);      // 12, 12, 75
9     std::multiset<int> second (myints+3, myints+6);   // 20, 35, 35
10
11    swap(first, second);
12
13    std::cout << "first contains:" ;
14    for (std::multiset<int>::iterator it=first.begin(); it!=first.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    std::cout << "second contains:" ;
19    for (std::multiset<int>::iterator it=second.begin(); it!=second.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
first contains: 20 35 35
second contains: 12 12 75
```

### Complexity

Constant.

### Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.

Note that the *end iterators* do not refer to elements and may be invalidated.

### Data races

Both containers, *x* and *y*, are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

### Exception safety

If the allocators in both *multisets* compare equal, or if their *allocator traits* indicate that the allocators shall *propagate*, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

### See also

<a href="#">vector::swap</a>	Swap content (public member function )
<a href="#">swap</a>	Exchange values of two objects (function template )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )

## /set/multiset/upper\_bound

public member function

&lt;set&gt;

## std::multiset::upper\_bound

```
iterator upper_bound (const value_type& val) const;
const_iterator upper_bound (const value_type& val) const;
    iterator upper_bound (const value_type& val);
```

### Return iterator to upper bound

Returns an iterator pointing to the first element in the container which is considered to go after *val*.

The function uses its internal [comparison object \(key\\_comp\)](#) to determine this, returning an iterator to the first element for which `key_comp(val,element)` would return `true`.

If the [multiset](#) class is instantiated with the default comparison type ([less](#)), the function returns an iterator to the first element that is greater than *val*.

A similar member function, [lower\\_bound](#), has the same behavior as [upper\\_bound](#), except in the case that the [multiset](#) contains elements equivalent to *val*: In this case [lower\\_bound](#) returns an iterator pointing to the first of such elements, whereas [upper\\_bound](#) returns an iterator pointing to the element following the last.

### Parameters

*val*

Value to compare.

Member type `value_type` is the type of the elements in the container, defined in [multiset](#) as an alias of its first template parameter (*T*).

### Return value

An iterator to the the first element in the container which is considered to go after *val*, or [multiset::end](#) if no elements are considered to go after *val*.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements.

### Example

```
1 // multiset::lower_bound/upper_bound
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8     std::multiset<int>::iterator itlow,itup;
9
10    for (int i=1; i<8; i++) mymultiset.insert(i*10); // 10 20 30 40 50 60 70
11
12    itlow = mymultiset.lower_bound (30);           //      ^
13    itup = mymultiset.upper_bound (40);           //      ^
14
15    mymultiset.erase(itlow,itup);                 // 10 20 50 60 70
16
17    std::cout << "mymultiset contains:";
18    for (std::multiset<int>::iterator it=mymultiset.begin(); it!=mymultiset.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Notice that `lower_bound(30)` returns an iterator to 30, whereas `upper_bound(60)` returns an iterator to 70.

```
mymultiset contains: 10 20 50 60 70
```

### Complexity

Logarithmic in `size`.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container). Concurrently accessing the elements of a [multiset](#) is safe.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

### See also

<a href="#">multiset::lower_bound</a>	Return iterator to lower bound ( <a href="#">public member function</a> )
<a href="#">multiset::equal_range</a>	Get range of equal elements ( <a href="#">public member function</a> )
<a href="#">multiset::find</a>	Get iterator to element ( <a href="#">public member function</a> )
<a href="#">multiset::count</a>	Count elements with a specific key ( <a href="#">public member function</a> )

public member function

## std::multiset::value\_comp

<set>

value\_compare value\_comp() const;

### Return comparison object

Returns a copy of the *comparison object* used by the container.

By default, this is a `less` object, which returns the same as `operator<`.

This object determines the order of the elements in the container: it is a function pointer or a function object that takes two arguments of the same type as the container elements, and returns `true` if the first argument is considered to go before the second in the *strict weak ordering* it defines, and `false` otherwise.

Two elements of a `multiset` are considered equivalent if `value_comp` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

In `multiset` containers, the keys to sort the elements are the values themselves, therefore `value_comp` and its sibling member function `key_comp` are equivalent.

### Parameters

none

### Return value

The comparison object.

Member type `value_compare` is the type of the *comparison object* associated to the container, defined in `multiset` as an alias of its second template parameter (`Compare`).

### Example

```
1 // multiset::value_comp
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::multiset<int> mymultiset;
8
9     std::multiset<int>::value_compare mycomp = mymultiset.value_comp();
10
11    for (int i=0; i<7; i++) mymultiset.insert(i);
12
13    std::cout << "mymultiset contains:";
14
15    int highest = *mymultiset.rbegin();
16    std::multiset<int>::iterator it = mymultiset.begin();
17    do {
18        std::cout << ' ' << *it;
19    } while ( mycomp(*it++,highest) );
20
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

mymultiset contains: 0 1 2 3 4 5 6

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.

Concurrently accessing the elements of a `multiset` is safe.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

### See also

<a href="#">multiset::key_comp</a>	Return comparison object (public member function )
<a href="#">multiset::find</a>	Get iterator to element (public member function )
<a href="#">multiset::count</a>	Count elements with a specific key (public member function )
<a href="#">multiset::lower_bound</a>	Return iterator to lower bound (public member function )
<a href="#">multiset::upper_bound</a>	Return iterator to upper bound (public member function )

## /set/set

class template

```
template < class T,           // set::key_type/value_type
          class Compare = less<T>, // set::key_compare/value_compare
          class Alloc = allocator<T> // set::allocator_type
        > class set;
```

**Set**

Sets are containers that store unique elements following a specific order.

In a set, the value of an element also identifies it (the value is itself the *key*, of type *T*), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always *const*), but they can be inserted or removed from the container.

Internally, the elements in a set are always sorted following a specific *strict weak ordering* criterion indicated by its internal *comparison object* (of type *Compare*).

set containers are generally slower than *unordered\_set* containers to access individual elements by their *key*, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as *binary search trees*.

**Container properties****Associative**

Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

**Ordered**

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

**Set**

The value of an element is also the *key* used to identify it.

**Unique keys**

No two elements in the container can have equivalent *keys*.

**Allocator-aware**

The container uses an allocator object to dynamically handle its storage needs.

**Template parameters****T**

Type of the elements. Each element in a set container is also uniquely identified by this value (each value is itself also the element's key). Aliased as member types *set::key\_type* and *set::value\_type*.

**Compare**

A binary predicate that takes two arguments of the same type as the elements and returns a *bool*. The expression *comp(a,b)*, where *comp* is an object of this type and *a* and *b* are key values, shall return *true* if *a* is considered to go before *b* in the *strict weak ordering* the function defines. The set object uses this expression to determine both the order the elements follow in the container and whether two element keys are equivalent (by comparing them reflexively: they are equivalent if *!comp(a,b) && !comp(b,a)*). No two elements in a set container can be equivalent. This can be a function pointer or a function object (see *constructor* for an example). This defaults to *less<T>*, which returns the same as applying the *less-than operator* (*a < b*). Aliased as member types *set::key\_compare* and *set::value\_compare*.

**Alloc**

Type of the allocator object used to define the storage allocation model. By default, the *allocator* class template is used, which defines the simplest memory allocation model and is value-independent.

Aliased as member type *set::allocator\_type*.

**Member types**

member type	definition	notes
<i>key_type</i>	The first template parameter ( <i>T</i> )	
<i>value_type</i>	The first template parameter ( <i>T</i> )	
<i>key_compare</i>	The second template parameter ( <i>Compare</i> )	defaults to: <i>less&lt;key_type&gt;</i>
<i>value_compare</i>	The second template parameter ( <i>Compare</i> )	defaults to: <i>less&lt;value_type&gt;</i>
<i>allocator_type</i>	The third template parameter ( <i>Alloc</i> )	defaults to: <i>allocator&lt;value_type&gt;</i>
<i>reference</i>	<i>allocator_type::reference</i>	for the default allocator: <i>value_type&amp;</i>
<i>const_reference</i>	<i>allocator_type::const_reference</i>	for the default allocator: <i>const value_type&amp;</i>
<i>pointer</i>	<i>allocator_type::pointer</i>	for the default allocator: <i>value_type*</i>
<i>const_pointer</i>	<i>allocator_type::const_pointer</i>	for the default allocator: <i>const value_type*</i>
<i>iterator</i>	a bidirectional <i>iterator</i> to <i>value_type</i>	convertible to <i>const_iterator</i>
<i>const_iterator</i>	a bidirectional <i>iterator</i> to <i>const value_type</i>	
<i>reverse_iterator</i>	<i>reverse_iterator&lt;iterator&gt;</i>	
<i>const_reverse_iterator</i>	<i>reverse_iterator&lt;const_iterator&gt;</i>	
<i>difference_type</i>	a signed integral type, identical to: <i>iterator_traits&lt;iterator&gt;::difference_type</i>	usually the same as <i>ptrdiff_t</i>
<i>size_type</i>	an unsigned integral type that can represent any non-negative value of <i>difference_type</i>	usually the same as <i>size_t</i>
member type	definition	notes
<i>key_type</i>	The first template parameter ( <i>T</i> )	
<i>value_type</i>	The first template parameter ( <i>T</i> )	
<i>key_compare</i>	The second template parameter ( <i>Compare</i> )	defaults to: <i>less&lt;key_type&gt;</i>
<i>value_compare</i>	The second template parameter ( <i>Compare</i> )	defaults to: <i>less&lt;value_type&gt;</i>
<i>allocator_type</i>	The third template parameter ( <i>Alloc</i> )	defaults to: <i>allocator&lt;value_type&gt;</i>
<i>reference</i>	<i>value_type&amp;</i>	
<i>const_reference</i>	<i>const value_type&amp;</i>	
<i>pointer</i>	<i>allocator_traits&lt;allocator_type&gt;::pointer</i>	for the default allocator: <i>value_type*</i>
		for the default allocator: <i>const</i>

<code>const_pointer</code>	<code>allocator_traits&lt;allocator_type&gt;::const_pointer</code>	<code>value_type*</code>
<code>iterator</code>	a bidirectional iterator to const <code>value_type</code>	* convertible to <code>const_iterator</code>
<code>const_iterator</code>	a bidirectional iterator to const <code>value_type</code>	*
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>	*
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>	*
<code>difference_type</code>	a signed integral type, identical to: <code>iterator_traits&lt;iterator&gt;::difference_type</code>	usually the same as <code>ptrdiff_t</code>
<code>size_type</code>	an unsigned integral type that can represent any non-negative value of <code>difference_type</code>	usually the same as <code>size_t</code>

\*Note: All iterators in a `set` point to const elements. Whether the `const_` member type is the same type as its non-`const_` counterpart depends on the particular library implementation, but programs should not rely on them being different to overload functions: `const_iterator` is more generic, since `iterator` is always convertible to it.

## Member functions

<b>(constructor)</b>	Construct set (public member function )
<b>(destructor)</b>	Set destructor (public member function )
<b>operator=</b>	Copy container content (public member function )

### Iterators:

<code>begin</code>	Return iterator to beginning (public member function )
<code>end</code>	Return iterator to end (public member function )
<code>rbegin</code>	Return reverse iterator to reverse beginning (public member function )
<code>rend</code>	Return reverse iterator to reverse end (public member function )
<code>cbegin</code>	Return <code>const_iterator</code> to beginning (public member function )
<code>cend</code>	Return <code>const_iterator</code> to end (public member function )
<code>crbegin</code>	Return <code>const_reverse_iterator</code> to reverse beginning (public member function )
<code>crend</code>	Return <code>const_reverse_iterator</code> to reverse end (public member function )

### Capacity:

<code>empty</code>	Test whether container is empty (public member function )
<code>size</code>	Return container size (public member function )
<code>max_size</code>	Return maximum size (public member function )

### Modifiers:

<code>insert</code>	Insert element (public member function )
<code>erase</code>	Erase elements (public member function )
<code>swap</code>	Swap content (public member function )
<code>clear</code>	Clear content (public member function )
<code>emplace</code>	Construct and insert element (public member function )
<code>emplace_hint</code>	Construct and insert element with hint (public member function )

### Observers:

<code>key_comp</code>	Return comparison object (public member function )
<code>value_comp</code>	Return comparison object (public member function )

### Operations:

<code>find</code>	Get iterator to element (public member function )
<code>count</code>	Count elements with a specific value (public member function )
<code>lower_bound</code>	Return iterator to lower bound (public member function )
<code>upper_bound</code>	Return iterator to upper bound (public member function )
<code>equal_range</code>	Get range of equal elements (public member function )

### Allocator:

<code>get_allocator</code>	Get allocator (public member function )
----------------------------	---

## /set/set/begin

public member function

**std::set::begin**

<set>

```
iterator begin();
const_iterator begin() const;
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

### Return iterator to beginning

Returns an iterator referring to the first element in the `set` container.

Because `set` containers keep their elements ordered at all times, `begin` points to the element that goes first following the container's sorting criterion.

If the container is `empty`, the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

An iterator to the first element in the container.

If the `set` object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements.

## Example

```
1 // set::begin/end
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {75,23,65,42,13};
8     std::set<int> myset (myints,myints+5);
9
10    std::cout << "myset contains:" ;
11    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
12        std::cout << ' ' << *it;
13
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

```
myset contains: 13 23 42 65 75
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

# /set/set/cbegin

public member function

## std::set::cbegin

<set>

```
const_iterator cbegin() const noexcept;
```

### Return const\_iterator to beginning

Returns a `const_iterator` pointing to the first element in the container.

All iterators in `set` containers are *constant iterators* (including both `const_iterator` and `iterator` member types). These cannot be used to modify the contents they point to, but can be increased and decreased normally (unless they are themselves also `const`).

If the container is `empty`, the returned iterator value shall not be dereferenced.

## Parameters

none

## Return Value

A `const_iterator` to the beginning of the sequence.

Member type `const_iterator` is a [bidirectional iterator](#) type that points to `const` elements.

## Example

```
1 // set::cbegin/cend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
```

```

7 std::set<int> myset = {50,20,60,10,25};
8 std::cout << "myset contains:";
9 for (auto it=myset.cbegin(); it != myset.cend(); ++it)
10    std::cout << ' ' << *it;
11
12 std::cout << '\n';
13
14 return 0;
15 }
16 }
```

Output:

```
myset contains: 10 20 25 50 60
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>set::begin</code>	Return iterator to beginning (public member function )
<code>set::cend</code>	Return const_iterator to end (public member function )
<code>set::crbegin</code>	Return const_reverse_iterator to reverse beginning (public member function )

## /set/set/cend

public member function

### std::set::cend

<set>

`const_iterator cend() const noexcept;`

#### Return const\_iterator to end

Returns a `const_iterator` pointing to the *past-the-end* element in the container.

All iterators in `set` containers are *constant iterators* (including both `const_iterator` and `iterator` member types). These cannot be used to modify the contents they point to, but can be increased and decreased normally (unless they are themselves also `const`).

## Parameters

none

## Return Value

A `const_iterator` to the element past the end of the sequence.

Member type `const_iterator` is a **bidirectional iterator** type that points to a `const` element.

## Example

```

1 // set::cbegin/cend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset = {50,20,60,10,25};
8
9     std::cout << "myset contains:";
10    for (auto it=myset.cbegin(); it != myset.cend(); ++it)
11        std::cout << ' ' << *it;
12
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
myset contains: 10 20 25 50 60
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a [set](#) is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">set::end</a>	Return iterator to end ( <a href="#">public member function</a> )
<a href="#">set::cbegin</a>	Return const_iterator to beginning ( <a href="#">public member function</a> )

# /set/set/clear

public member function

## std::set::clear

<set>

```
void clear();
void clear() noexcept;
```

### Clear content

Removes all elements from the [set](#) container (which are destroyed), leaving the container with a [size](#) of 0.

## Parameters

none

## Return value

none

## Example

```
1 // set::clear
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8
9     myset.insert (100);
10    myset.insert (200);
11    myset.insert (300);
12
13    std::cout << "myset contains:" ;
14    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    myset.clear();
19    myset.insert (1101);
20    myset.insert (2202);
21
22    std::cout << "myset contains:" ;
23    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
24        std::cout << ' ' << *it;
25    std::cout << '\n';
26
27    return 0;
28 }
```

Output:

```
myset contains: 100 200 300
myset contains: 1101 2202
```

## Complexity

Linear in [size](#) (destructions).

## Iterator validity

All iterators, pointers and references related to this container are invalidated.

## Data races

The container is modified.  
All contained elements are modified.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

# /set/set/count

public member function

## std::set::count

<set>

`size_type count (const value_type& val) const;`

### Count elements with a specific value

Searches the container for elements equivalent to `val` and returns the number of matches.

Because all elements in a `set` container are unique, the function can only return 1 (if the element is found) or zero (otherwise).

Two elements of a `set` are considered equivalent if the container's comparison object returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

## Parameters

`val`

Value to search for.

Member type `value_type` is the type of the elements in the container, defined in `set` as an alias of its first template parameter (`T`).

## Return value

1 if the container contains an element equivalent to `val`, or zero otherwise.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // set::count
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8
9     // set some initial values:
10    for (int i=1; i<5; ++i) myset.insert(i*3);      // set: 3 6 9 12
11
12    for (int i=0; i<10; ++i)
13    {
14        std::cout << i;
15        if (myset.count(i)!=0)
16            std::cout << " is an element of myset.\n";
17        else
18            std::cout << " is not an element of myset.\n";
19    }
20
21    return 0;
22 }
```

Output:

```
0 is not an element of myset.
1 is not an element of myset.
2 is not an element of myset.
3 is an element of myset.
4 is not an element of myset.
5 is not an element of myset.
6 is an element of myset.
7 is not an element of myset.
8 is not an element of myset.
9 is an element of myset.
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed.  
Concurrently accessing the elements of a `set` is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<a href="#">set::find</a>	Get iterator to element (public member function )
<a href="#">set::size</a>	Return container size (public member function )
<a href="#">set::equal_range</a>	Get range of equal elements (public member function )

## /set/set/crbegin

public member function

### std::set::crbegin

<set>

`const_reverse_iterator crbegin() const noexcept;`

**Return const\_reverse\_iterator to reverse beginning**

Returns a `const_reverse_iterator` pointing to the last element in the container (i.e., its *reverse beginning*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse beginning* of the sequence.

Member type `const_reverse_iterator` is a bidirectional iterator type that points to a `const` element.

## Example

```
1 // set::crbegin/crend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset = {50,20,60,10,25};
8
9     std::cout << "myset backwards:";
10    for (auto rit=myset.crbegin(); rit != myset.crend(); ++rit)
11        std::cout << ' ' << *rit;
12
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
myset backwards: 60 50 25 20 10
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">set::begin</a>	Return iterator to beginning (public member function )
<a href="#">set::crend</a>	Return const_reverse_iterator to reverse end (public member function )
<a href="#">set::rbegin</a>	Return reverse iterator to reverse beginning (public member function )

## /set/set/crend

public member function

### std::set::crend

<set>

`const_reverse_iterator crend() const noexcept;`

## **Return const\_reverse\_iterator to reverse end**

Returns a `const_reverse_iterator` pointing to the element that would theoretically precede the first element in the container (which is considered its *reverse end*).

### **Parameters**

none

### **Return Value**

A `const_reverse_iterator` to the *reverse end* of the sequence.

Member type `const_reverse_iterator` is a [bidirectional iterator](#) type that points to a `const` element.

### **Example**

```

1 // set::crbegin/crend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset = {50,20,60,10,25};
8
9     std::cout << "myset backwards:";
10    for (auto rit=myset.crbegin(); rit != myset.crend(); ++rit)
11        std::cout << ' ' << *rit;
12
13    std::cout << '\n';
14
15    return 0;
16 }
```

Output:

```
myset backwards: 60 50 25 20 10
```

### **Complexity**

Constant.

### **Iterator validity**

No changes.

### **Data races**

The container is accessed.

Concurrently accessing the elements of a `set` is safe.

### **Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### **See also**

<code>set::end</code>	Return iterator to end ( <a href="#">public member function</a> )
<code>set::crbegin</code>	Return <code>const_reverse_iterator</code> to reverse beginning ( <a href="#">public member function</a> )
<code>set::rend</code>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )

## /set/set/emplace

public member function

### **std::set::emplace**

`<set>`

```
template <class... Args>
pair<iterator,bool> emplace (Args&&... args);
```

#### **Construct and insert element**

Inserts a new element in the `set`, if unique. This new element is constructed in place using `args` as the arguments for its construction.

The insertion only takes place if no other element in the container is equivalent to the one being emplaced (elements in a `set` container are unique).

If inserted, this effectively increases the container `size` by one.

Internally, `set` containers keep all their elements sorted following the criterion specified by its `comparison object`. The element is always inserted in its respective position following this ordering.

The element is constructed in-place by calling `allocator_traits::construct` with `args` forwarded.

A similar member function exists, `insert`, which either copies or moves existing objects into the container.

### **Parameters**

`args`

Arguments forwarded to construct the new element.

## Return value

If the function successfully inserts the element (because no equivalent element existed already in the `set`), the function returns a pair of an iterator to the newly inserted element and a value of `true`.

Otherwise, it returns an iterator to the equivalent element within the container and a value of `false`.

Member type `iterator` is a [bidirectional iterator](#) type that points to an element.  
`pair` is a class template declared in `<utility>` (see `pair`).

## Example

```
1 // set::emplace
2 #include <iostream>
3 #include <set>
4 #include <string>
5
6 int main ()
7 {
8     std::set<std::string> myset;
9
10    myset.emplace("foo");
11    myset.emplace("bar");
12    auto ret = myset.emplace("foo");
13
14    if (!ret.second) std::cout << "foo already exists in myset\n";
15
16    return 0;
17 }
```

Output:

```
foo already exists in myset
```

## Complexity

Logarithmic in the container size.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with the appropriate arguments, it causes *undefined behavior*.

## See also

<code>set::emplace_hint</code>	Construct and insert element with hint (public member function )
<code>set::insert</code>	Insert element (public member function )
<code>set::erase</code>	Erase elements (public member function )

# /set/set/emplace\_hint

public member function

## `std::set::emplace_hint`

`<set>`

```
template <class... Args>
iterator emplace_hint (const_iterator position, Args&&... args);
```

### Construct and insert element with hint

Inserts a new element in the `set`, if unique, with a hint on the insertion *position*. This new element is constructed in place using *args* as the arguments for its construction.

The insertion only takes place if no other element in the container is equivalent to the one being emplaced (elements in a `set` container are unique).

If inserted, this effectively increases the container `size` by one.

The value in *position* is used as a hint on the insertion point. The element will nevertheless be inserted at its corresponding position following the order described by its internal `comparison object`, but this hint is used by the function to begin its search for the insertion point, speeding up the process considerably when the actual insertion point is either *position* or close to it.

The element is constructed in-place by calling `allocator_traits::construct` with *args* forwarded.

## Parameters

### `position`

Hint for the position where the element can be inserted.

The function optimizes its insertion time if *position* points to the element that will follow the inserted element (or to the `end`, if it would be the last).

Notice that this does not force the new element to be inserted at that position within the `set` container (the elements in a `set` always follow a specific order).

`const_iterator` is a member type, defined as a [bidirectional iterator](#) type that points to elements.

`args`

Arguments forwarded to construct the new element.

## Return value

If the function successfully inserts the element (because no equivalent element existed already in the `set`), the function returns an iterator to the newly inserted element.

Otherwise, it returns an iterator to the equivalent element within the container.

Member type `iterator` is a [bidirectional iterator](#) type that points to an element.

## Example

```
1 // set::emplace_hint
2 #include <iostream>
3 #include <set>
4 #include <string>
5
6 int main ()
7 {
8     std::set<std::string> myset;
9     auto it = myset.cbegin();
10
11    myset.emplace_hint (it, "alpha");
12    it = myset.emplace_hint (myset.cend(), "omega");
13    it = myset.emplace_hint (it, "epsilon");
14    it = myset.emplace_hint (it, "beta");
15
16    std::cout << "myset contains:";
17    for (const std::string& x: myset)
18        std::cout << ' ' << x;
19    std::cout << '\n';
20
21    return 0;
22 }
```

Output:

```
myset contains: alpha beta epsilon omega
```

## Complexity

Generally, logarithmic in the container `size`.

Amortized constant if the insertion point for the element is `position`.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if an invalid `position` is specified, it causes *undefined behavior*.

## See also

<code>set::emplace</code>	Construct and insert element ( <a href="#">public member function</a> )
<code>set::insert</code>	Insert element ( <a href="#">public member function</a> )
<code>set::erase</code>	Erase elements ( <a href="#">public member function</a> )

# /set/set/empty

public member function

## `std::set::empty`

`<set>`

```
bool empty() const;
bool empty() const noexcept;
```

### Test whether container is empty

Returns whether the `set` container is empty (i.e. whether its `size` is 0).

This function does not modify the container in any way. To clear the content of a `set` container, see `set::clear`.

## Parameters

none

## Return Value

true if the container size is 0, false otherwise.

## Example

```
1 // set::empty
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8
9     myset.insert(20);
10    myset.insert(30);
11    myset.insert(10);
12
13    std::cout << "myset contains:";
14    while (!myset.empty())
15    {
16        std::cout << ' ' << *myset.begin();
17        myset.erase(myset.begin());
18    }
19    std::cout << '\n';
20
21    return 0;
22 }
```

Output:

```
myset contains: 10 20 30
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>set::clear</code>	Clear content (public member function )
<code>set::erase</code>	Erase elements (public member function )
<code>set::size</code>	Return container size (public member function )

# /set/set/end

public member function

## `std::set::end`

<set>

<code>iterator end();</code>
<code>const_iterator end() const;</code>
<code>iterator end() noexcept;</code>
<code>const_iterator end() const noexcept;</code>

### Return iterator to end

Returns an iterator referring to the *past-the-end* element in the `set` container.

The *past-the-end* element is the theoretical element that would follow the last element in the `set` container. It does not point to any element, and thus shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with `set::begin` to specify a range including all the elements in the container.

If the container is `empty`, this function returns the same as `set::begin`.

## Parameters

none

## Return Value

An iterator to the *past-the-end* element in the container.

If the `set` object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are `bidirectional iterator` types pointing to elements.

## Example

```
1 // set::begin/end
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {75,23,65,42,13};
8     std::set<int> myset (myints,myints+5);
9
10    std::cout << "myset contains:" ;
11    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
12        std::cout << ' ' << *it;
13
14    std::cout << '\n';
15
16    return 0;
17 }
```

Output:

```
myset contains: 13 23 42 65 75
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>set::begin</code>	Return iterator to beginning ( <a href="#">public member function</a> )
<code>set::rbegin</code>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<code>set::rend</code>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )

## /set/set/equal\_range

public member function

### `std::set::equal_range`

`<set>`

```
pair<iterator,iterator> equal_range (const value_type& val) const;
pair<const_iterator,const_iterator> equal_range (const value_type& val) const;
pair<iterator,iterator> equal_range (const value_type& val);
```

#### Get range of equal elements

Returns the bounds of a range that includes all the elements in the container that are equivalent to `val`.

Because all elements in a `set` container are unique, the range returned will contain a single element at most.

If no matches are found, the range returned has a length of zero, with both iterators pointing to the first element that is considered to go after `val` according to the container's [internal comparison object \(key\\_comp\)](#).

Two elements of a `set` are considered equivalent if the container's [comparison object](#) returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

## Parameters

`val`

Value to search for.

Member type `value_type` is the type of the elements in the container, defined in `set` as an alias of its first template parameter (`T`).

## Return value

The function returns a `pair`, whose member `pair::first` is the lower bound of the range (the same as `lower_bound`), and `pair::second` is the upper bound (the same as `upper_bound`).

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements.

## Example

```

1 // set::equal_elements
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8
9     for (int i=1; i<=5; i++) myset.insert(i*10);    // myset: 10 20 30 40 50
10
11 std::pair<std::set<int>::const_iterator, std::set<int>::const_iterator> ret;
12 ret = myset.equal_range(30);
13
14 std::cout << "the lower bound points to: " << *ret.first << '\n';
15 std::cout << "the upper bound points to: " << *ret.second << '\n';
16
17 return 0;
18 }

```

```

the lower bound points to: 30
the upper bound points to: 40

```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a `set` is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>set::count</code>	Count elements with a specific value ( <a href="#">public member function</a> )
<code>set::lower_bound</code>	Return iterator to lower bound ( <a href="#">public member function</a> )
<code>set::upper_bound</code>	Return iterator to upper bound ( <a href="#">public member function</a> )
<code>set::find</code>	Get iterator to element ( <a href="#">public member function</a> )

## /set/set/erase

public member function

### `std::set::erase`

<`set`>

```

(1)      void erase (iterator position);
(2) size_type erase (const value_type& val);
(3)      void erase (iterator first, iterator last);

(1) iterator erase (const_iterator position);
(2) size_type erase (const value_type& val);
(3) iterator erase (const_iterator first, const_iterator last);

```

#### Erase elements

Removes from the `set` container either a single element or a range of elements (`[first, last]`).

This effectively reduces the container `size` by the number of elements removed, which are destroyed.

## Parameters

### position

Iterator pointing to a single element to be removed from the `set`.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types that point to elements.

### val

Value to be removed from the `set`.

Member type `value_type` is the type of the elements in the container, defined in `set` as an alias of its first template parameter (`T`).

### first, last

Iterators specifying a range within the `set` container to be removed: `[first, last]`. i.e., the range includes all the elements between `first` and `last`, including the element pointed by `first` but not the one pointed by `last`.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types that point to elements.

## Return value

For the value-based version (2), the function returns the number of elements erased, which in `set` containers is at most 1.

Member type `size_type` is an unsigned integral type.

The other versions return no value.

The other versions return an iterator to the element that follows the last element removed (or `set::end`, if the last element was removed).

Member type `iterator` is a [bidirectional iterator](#) type that points to elements.

## Example

```
1 // erasing from set
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     std::set<int>::iterator it;
9
10    // insert some values:
11    for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90
12
13    it = myset.begin();
14    ++it;                                // "it" points now to 20
15
16    myset.erase (it);
17
18    myset.erase (40);
19
20    it = myset.find (60);
21    myset.erase (it, myset.end());
22
23    std::cout << "myset contains:";
24    for (it=myset.begin(); it!=myset.end(); ++it)
25        std::cout << ' ' << *it;
26    std::cout << '\n';
27
28    return 0;
29 }
```

Output:

```
myset contains: 10 30 50
```

## Complexity

For the first version (`erase(position)`), amortized constant.

For the second version (`erase(val)`), logarithmic in container `size`.

For the last version (`erase(first,last)`), linear in the distance between `first` and `last`.

## Iterator validity

Iterators, pointers and references referring to elements removed by the function are invalidated.

All other iterators, pointers and references keep their validity.

## Data races

The container is modified.

The elements removed are modified. Concurrently accessing other elements is safe, although iterating ranges in the container is not.

## Exception safety

Unless the container's `comparison object` throws, this function never throws exceptions (no-throw guarantee).

Otherwise, if a single element is to be removed, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If an invalid `position` or range is specified, it causes *undefined behavior*.

## See also

<code>set::clear</code>	Clear content ( <a href="#">public member function</a> )
<code>set::insert</code>	Insert element ( <a href="#">public member function</a> )
<code>set::find</code>	Get iterator to element ( <a href="#">public member function</a> )

# /set/set/find

public member function

## `std::set::find`

`<set>`

```
iterator find (const value_type& val) const;
const_iterator find (const value_type& val) const;
iterator      find (const value_type& val);
```

### Get iterator to element

Searches the container for an element equivalent to `val` and returns an iterator to it if found, otherwise it returns an iterator to `set::end`.

Two elements of a `set` are considered equivalent if the container's `comparison object` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

## Parameters

`val`

Value to be searched for.

Member type `value_type` is the type of the elements in the container, defined in `set` as an alias of its first template parameter (`T`).

## Return value

An iterator to the element, if `val` is found, or `set::end` otherwise.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements.

## Example

```
1 // set::find
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     std::set<int>::iterator it;
9
10    // set some initial values:
11    for (int i=1; i<=5; i++) myset.insert(i*10);    // set: 10 20 30 40 50
12
13    it=myset.find(20);
14    myset.erase (it);
15    myset.erase (myset.find(40));
16
17    std::cout << "myset contains:";
18    for (it=myset.begin(); it!=myset.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Output:

```
myset contains: 10 30 50
```

## Complexity

Logarithmic in `size`.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container). Concurrently accessing the elements of a `set` is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<code>set::count</code>	Count elements with a specific value ( <a href="#">public member function</a> )
<code>set::lower_bound</code>	Return iterator to lower bound ( <a href="#">public member function</a> )
<code>set::upper_bound</code>	Return iterator to upper bound ( <a href="#">public member function</a> )

## /set/set/get\_allocator

public member function

### std::set::get\_allocator

`<set>`

```
allocator_type get_allocator() const;
allocator_type get_allocator() const noexcept;
```

#### Get allocator

Returns a copy of the allocator object associated with the `set`.

## Parameters

none

## Return Value

The allocator.

Member type `allocator_type` is the type of the allocator used by the container, defined in `set` as an alias of its third template parameter (`Alloc`).

## Example

```

1 // set::get_allocator
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     int * p;
9     unsigned int i;
10
11    // allocate an array of 5 elements using myset's allocator:
12    p=myset.get_allocator().allocate(5);
13
14    // assign some values to array
15    for (i=0; i<5; i++) p[i]=(i+1)*10;
16
17    std::cout << "The allocated array contains:";
18    for (i=0; i<5; i++) std::cout << ' ' << p[i];
19    std::cout << '\n';
20
21    myset.get_allocator().deallocate(p,5);
22
23    return 0;
24 }

```

The example shows an elaborate way to allocate memory for an array of ints using the same allocator used by the container.  
Output:

The allocated array contains: 10 20 30 40 50

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a [set](#) is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

Copying any instantiation of the [default allocator](#) is also guaranteed to never throw.

## See also

<a href="#">allocator</a>	Default allocator (class template )
---------------------------	-------------------------------------

## /set/set/insert

public member function

### std::set::insert

<set>

single element (1)	pair<iterator,bool> insert (const value_type& val);
with hint (2)	iterator insert (iterator position, const value_type& val);
range (3)	template <class InputIterator> void insert (InputIterator first, InputIterator last);
single element (1)	pair<iterator,bool> insert (const value_type& val); pair<iterator,bool> insert (value_type&& val);
with hint (2)	iterator insert (const_iterator position, const value_type& val); iterator insert (const_iterator position, value_type&& val);
range (3)	template <class InputIterator> void insert (InputIterator first, InputIterator last);
initializer list (4)	void insert (initializer_list<value_type> il);

### Insert element

Extends the container by inserting new elements, effectively increasing the container [size](#) by the number of elements inserted.

Because elements in a [set](#) are unique, the insertion operation checks whether each inserted element is equivalent to an element already in the container, and if so, the element is not inserted, returning an iterator to this existing element (if the function returns a value).

For a similar container allowing for duplicate elements, see [multiset](#).

Internally, [set](#) containers keep all their elements sorted following the criterion specified by its [comparison object](#). The elements are always inserted in its respective position following this ordering.

The parameters determine how many elements are inserted and to which values they are initialized:

## Parameters

val

Value to be copied (or moved) to the inserted elements.

Member type [value\\_type](#) is the type of the elements in the container, defined in [set](#) as an alias of its first template parameter ( $\tau$ ).

position

Hint for the position where the element can be inserted.

The function optimizes its insertion time if *position* points to the element that will **precede** the inserted element.

The function optimizes its insertion time if *position* points to the element that will **follow** the inserted element (or to the *end*, if it would be the last).

Notice that this is just a hint and does not force the new element to be inserted at that position within the *set* container (the elements in a *set* always follow a specific order).

Member types *iterator* and *const\_iterator* are defined in *map* as a **bidirectional iterator** type that point to elements.

*first*, *last*

Iterators specifying a range of elements. Copies of the elements in the range [*first*,*last*) are inserted in the container.

Notice that the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.

The function template argument *InputIterator* shall be an **input iterator** type that points to elements of a type from which *value\_type* objects can be constructed.

*il*

An **initializer\_list** object. Copies of these elements are inserted.

These objects are automatically constructed from *initializer\_list* declarators.

Member type *value\_type* is the type of the elements in the container, defined in *set* as an alias of its first template parameter (*T*).

## Return value

The single element versions (1) return a **pair**, with its member *pair*::*first* set to an iterator pointing to either the newly inserted element or to the equivalent element already in the *set*. The *pair*::*second* element in the **pair** is set to **true** if a new element was inserted or **false** if an equivalent element already existed.

The versions with a hint (2) return an iterator pointing to either the newly inserted element or to the element that already had its same value in the *set*.

Member type *iterator* is a **bidirectional iterator** type that points to elements.

*pair* is a class template declared in **<utility>** (see **pair**).

## Example

```
1 // set::insert (C++98)
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     std::set<int>::iterator it;
9     std::pair<std::set<int>::iterator,bool> ret;
10
11    // set some initial values:
12    for (int i=1; i<=5; ++i) myset.insert(i*10);    // set: 10 20 30 40 50
13
14    ret = myset.insert(20);                // no new element inserted
15
16    if (ret.second==false) it=ret.first; // "it" now points to element 20
17
18    myset.insert (it,25);              // max efficiency inserting
19    myset.insert (it,24);              // max efficiency inserting
20    myset.insert (it,26);              // no max efficiency inserting
21
22    int myints[] = {5,10,15};        // 10 already in set, not inserted
23    myset.insert (myints,myints+3);
24
25    std::cout << "myset contains:";
26    for (it=myset.begin(); it!=myset.end(); ++it)
27        std::cout << ' ' << *it;
28    std::cout << '\n';
29
30    return 0;
31 }
```

Output:

```
myset contains: 5 10 15 20 24 25 26 30 40 50
```

## Complexity

If a single element is inserted, logarithmic in *size* in general, but amortized constant if a hint is given and the *position* given is the optimal.

If *N* elements are inserted,  $N \log(\text{size} + N)$  in general, but linear in  $\text{size} + N$  if the elements are already sorted according to the same ordering criterion used by the container.

If *N* elements are inserted,  $N \log(\text{size} + N)$ .

Implementations may optimize if the range is already sorted.

## Iterator validity

No changes.

## Data races

The container is modified.

Concurrently accessing existing elements is safe, although iterating ranges in the container is not.

## Exception safety

If a single element is to be inserted, there are no changes in the container in case of exception (strong guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

If *allocator\_traits*::*construct* is not supported with the appropriate arguments for the element constructions, or if an invalid *position* is specified, it causes *undefined behavior*.

## See also

<b>set::erase</b>	Erase elements (public member function )
<b>set::find</b>	Get iterator to element (public member function )

## /set/set/key\_comp

public member function

### std::set::key\_comp

<set>

`key_compare key_comp() const;`

#### Return comparison object

Returns a copy of the *comparison object* used by the container.

By default, this is a `less` object, which returns the same as `operator<`.

This object determines the order of the elements in the container: it is a function pointer or a function object that takes two arguments of the same type as the container elements, and returns `true` if the first argument is considered to go before the second in the *strict weak ordering* it defines, and `false` otherwise.

Two elements of a `set` are considered equivalent if `key_comp` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

In `set` containers, the keys to sort the elements are the values themselves, therefore `key_comp` and its sibling member function `value_comp` are equivalent.

#### Parameters

none

#### Return value

The comparison object.

Member type `key_compare` is the type of the *comparison object* associated to the container, defined in `set` as an alias of its second template parameter (`Compare`).

#### Example

```

1 // set::key_comp
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     int highest;
9
10    std::set<int>::key_compare mycomp = myset.key_comp();
11
12    for (int i=0; i<=5; i++) myset.insert(i);
13
14    std::cout << "myset contains:" ;
15
16    highest=*myset.rbegin();
17    std::set<int>::iterator it=myset.begin();
18    do {
19        std::cout << ' ' << *it;
20    } while ( mycomp(*(++it),highest) );
21
22    std::cout << '\n';
23
24    return 0;
25 }
```

Output:

`myset contains: 0 1 2 3 4`

#### Complexity

Constant.

#### Iterator validity

No changes.

#### Data races

The container is accessed.

Concurrently accessing the elements of a `set` is safe.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

#### See also

<b>set::value_comp</b>	Return comparison object (public member function )
<b>set::find</b>	Get iterator to element (public member function )
<b>set::count</b>	Count elements with a specific value (public member function )

<b>set::lower_bound</b>	Return iterator to lower bound ( <a href="#">public member function</a> )
<b>set::upper_bound</b>	Return iterator to upper bound ( <a href="#">public member function</a> )

## /set/set/lower\_bound

public member function

### std::set::lower\_bound

<set>

```
iterator lower_bound (const value_type& val) const;
    iterator lower_bound (const value_type& val);
const_iterator lower_bound (const value_type& val) const;
```

#### Return iterator to lower bound

Returns an iterator pointing to the first element in the container which is not considered to go before *val* (i.e., either it is equivalent or goes after).

The function uses its internal [comparison object](#) (`key_comp`) to determine this, returning an iterator to the first element for which `key_comp(element, val)` would return `false`.

If the `set` class is instantiated with the default comparison type ([less](#)), the function returns an iterator to the first element that is not less than *val*.

A similar member function, [upper\\_bound](#), has the same behavior as `lower_bound`, except in the case that the `set` contains an element equivalent to *val*: In this case `lower_bound` returns an iterator pointing to that element, whereas `upper_bound` returns an iterator pointing to the next element.

#### Parameters

*val*

Value to compare.

Member type `value_type` is the type of the elements in the container, defined in `set` as an alias of its first template parameter (`T`).

#### Return value

An iterator to the the first element in the container which is not considered to go before *val*, or `set::end` if all elements are considered to go before *val*.

Member types `iterator` and `const_iterator` are [bidirectional iterator](#) types pointing to elements.

#### Example

```
1 // set::lower_bound/upper_bound
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     std::set<int>::iterator itlow,itup;
9
10    for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90
11
12    itlow=myset.lower_bound (30);           //      ^
13    itup=myset.upper_bound (60);          //      ^
14
15    myset.erase(itlow,itup);             // 10 20 70 80 90
16
17    std::cout << "myset contains:" ;
18    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Notice that `lower_bound(30)` returns an iterator to 30, whereas `upper_bound(60)` returns an iterator to 70.

`myset contains: 10 20 70 80 90`

#### Complexity

Logarithmic in `size`.

#### Iterator validity

No changes.

#### Data races

The container is accessed (neither the `const` nor the non-`const` versions modify the container). Concurrently accessing the elements of a `set` is safe.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

#### See also

<b>set::upper_bound</b>	Return iterator to upper bound ( <a href="#">public member function</a> )
<b>set::equal_range</b>	Get range of equal elements ( <a href="#">public member function</a> )

<b>set::find</b>	Get iterator to element (public member function )
<b>set::count</b>	Count elements with a specific value (public member function )

## /set/set/max\_size

public member function

### std::set::max\_size

<set>

```
size_type max_size() const;
size_type max_size() const noexcept;
```

#### Return maximum size

Returns the maximum number of elements that the `set` container can hold.

This is the maximum potential `size` the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size: it can still fail to allocate storage at any point before that size is reached.

#### Parameters

none

#### Return Value

The maximum number of elements a `set` container can hold as content.

Member type `size_type` is an unsigned integral type.

#### Example

```
1 // set::max_size
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int i;
8     std::set<int> myset;
9
10    if (myset.max_size()>1000)
11    {
12        for (i=0; i<1000; i++) myset.insert(i);
13        std::cout << "The set contains 1000 elements.\n";
14    }
15    else std::cout << "The set could not hold 1000 elements.\n";
16
17    return 0;
18 }
```

Here, member `max_size` is used to check beforehand whether the `set` will allow for 1000 elements to be inserted.

#### Complexity

Constant.

#### Iterator validity

No changes.

#### Data races

The container is accessed.

Concurrently accessing the elements of a `set` is safe.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

#### See also

<b>set::size</b>	Return container size (public member function )
------------------	---

## /set/set/operator=

public member function

### std::set::operator=

<set>

```
copy (1)  set& operator= (const set& x);
copy (1)  set& operator= (const set& x);
move (2)  set& operator= (set&& x);
initializer list (3) set& operator= (initializer_list<value_type> il);
```

#### Copy container content

Assigns new contents to the container, replacing its current content.

Copies all the elements from *x* into the container, changing its [size](#) accordingly.

The container preserves its [current allocator](#), which is used to allocate additional storage if needed.

The [copy assignment](#) (1) copies all the elements from *x* into the container (with *x* preserving its contents).

The [move assignment](#) (2) moves the elements of *x* into the container (*x* is left in an unspecified but valid state).

The [initializer list assignment](#) (3) copies the elements of *il* into the container.

The new container [size](#) is the same as the [size](#) of *x* (or *il*) before the call.

The container preserves its [current allocator](#), except if the [allocator traits](#) indicate *x*'s allocator should [propagate](#). This allocator is used (through its [traits](#)) to [allocate](#) or [deallocate](#) if there are changes in storage requirements, and to [construct](#) or [destroy](#) elements, if needed.

The elements stored in the container before the call are either assigned to or destroyed.

## Parameters

*x*  
A [set](#) object of the same type (i.e., with the same template parameters, *T*, [Compare](#) and [Alloc](#)).

*il*  
An [initializer\\_list](#) object. The compiler will automatically construct such objects from [initializer list](#) declarators.  
Member type [value\\_type](#) is the type of the elements in the container, defined in [set](#) as an alias of its first template parameter (*T*).

## Return value

[\\*this](#)

## Example

```
1 // assignment operator with sets
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = { 12, 82, 37, 64, 15 };
8     std::set<int> first (myints, myints + 5);    // set with 5 ints
9     std::set<int> second;                         // empty set
10
11    second = first;                             // now second contains the 5 ints
12    first = std::set<int>();                   // and first is empty
13
14    std::cout << "Size of first: " << int (first.size()) << '\n';
15    std::cout << "Size of second: " << int (second.size()) << '\n';
16    return 0;
17 }
```

Output:

```
Size of first: 0
Size of second: 5
```

## Complexity

For the [copy assignment](#) (1): Linear in sizes (destructions, copies).

For the [move assignment](#) (2): Linear in current container [size](#) (destructions).\*

For the [initializer list assignment](#) (3): Up to logarithmic in sizes (destructions, move-assignments) -- linear if *il* is already sorted.

\* Additional complexity for assignments if allocators do not [propagate](#).

## Iterator validity

All iterators, references and pointers related to this container are invalidated.

In the [move assignment](#), iterators, pointers and references referring to elements in *x* are also invalidated.

## Data races

All copied elements are accessed.

The [move assignment](#) (2) modifies *x*.

The container and all its elements are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.

If [allocator\\_traits::construct](#) is not supported with the appropriate arguments for the element constructions, or if [value\\_type](#) is not [copy assignable](#) (or [move assignable](#) for (2)), it causes [undefined behavior](#).

## See also

<a href="#">set::insert</a>	Insert element (public member function )
<a href="#">set::set</a>	Construct set (public member function )

function  
**std::relational operators (set)** <set>

---

```
template <class T, class Compare, class Alloc>
(1)  bool operator==( const set<T,Compare,Alloc>& lhs,
        const set<T,Compare,Alloc>& rhs );
template <class T, class Compare, class Alloc>
(2)  bool operator!=( const set<T,Compare,Alloc>& lhs,
        const set<T,Compare,Alloc>& rhs );
template <class T, class Compare, class Alloc>
(3)  bool operator<( const set<T,Compare,Alloc>& lhs,
        const set<T,Compare,Alloc>& rhs );
template <class T, class Compare, class Alloc>
(4)  bool operator<=( const set<T,Compare,Alloc>& lhs,
        const set<T,Compare,Alloc>& rhs );
template <class T, class Compare, class Alloc>
(5)  bool operator>( const set<T,Compare,Alloc>& lhs,
        const set<T,Compare,Alloc>& rhs );
template <class T, class Compare, class Alloc>
(6)  bool operator>=( const set<T,Compare,Alloc>& lhs,
        const set<T,Compare,Alloc>& rhs );
```

#### Relational operators for set

Performs the appropriate comparison operation between the `set` containers *lhs* and *rhs*.

The *equality comparison* (`operator==`) is performed by first comparing *sizes*, and if they match, the elements are compared sequentially using `operator==`, stopping at the first mismatch (as if using algorithm `equal`).

The *less-than comparison* (`operator<`) behaves as if using algorithm `lexicographical_compare`, which compares the elements sequentially using `operator<` in a reciprocal manner (i.e., checking both  $a < b$  and  $b < a$ ) and stopping at the first occurrence.

The other operations also use the operators `==` and `<` internally to compare the elements, behaving as if the following equivalent operations were performed:

operator	equivalent operation
<code>a != b</code>	<code>!(a == b)</code>
<code>a &gt; b</code>	<code>b &lt; a</code>
<code>a &lt;= b</code>	<code>!(b &lt; a)</code>
<code>a &gt;= b</code>	<code>!(a &lt; b)</code>

Notice that none of these operations take into consideration the `internal comparison object` of neither container.

These operators are overloaded in header `<set>`.

#### Parameters

`lhs, rhs`

`set` containers (to the left- and right-hand side of the operator, respectively), having both the same template parameters (*T*, `Compare` and `Alloc`).

#### Example

```
1 // set comparisons
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> foo,bar;
8     foo.insert(10);
9     bar.insert(20);
10    bar.insert(30);
11    foo.insert(40);
12
13    // foo ({10,40}) vs bar ({20,30}):
14    if (foo==bar) std::cout << "foo and bar are equal\n";
15    if (foo!=bar) std::cout << "foo and bar are not equal\n";
16    if (foo< bar) std::cout << "foo is less than bar\n";
17    if (foo> bar) std::cout << "foo is greater than bar\n";
18    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
19    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
20
21    return 0;
22 }
```

Output:

```
foo and bar are not equal
foo is less than bar
foo is less than or equal to bar
```

#### Return Value

true if the condition holds, and false otherwise.

#### Complexity

Up to linear in the `size` of *lhs* and *rhs*.

For (1) and (2), constant if the `sizes` of *lhs* and *rhs* differ, and up to linear in that `size` (equality comparisons) otherwise.  
For the others, up to linear in the smaller `size` (each representing two comparisons with `operator<`).

#### Iterator validity

No changes.

## Data races

Both containers, *lhs* and *rhs*, are accessed.  
Concurrently accessing the elements of unmodified `set` objects is always safe (their elements are *immutable*).

## Exception safety

If the type of the elements supports the appropriate operation with no-throw guarantee, the function never throws exceptions (no-throw guarantee).  
In any case, the function cannot modify its arguments.

## See also

<code>set::value_comp</code>	Return comparison object (public member function )
<code>set::operator=</code>	Copy container content (public member function )
<code>set::swap</code>	Swap content (public member function )

## /set/set/rbegin

public member function

### `std::set::rbegin`

<set>

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
```

#### Return reverse iterator to reverse beginning

Returns a *reverse iterator* pointing to the last element in the container (i.e., its *reverse beginning*).

*Reverse iterators* iterate backwards: increasing them moves them towards the beginning of the container.

`rbegin` points to the element preceding the one that would be pointed to by member `end`.

## Parameters

none

## Return Value

A *reverse iterator* to the *reverse beginning* of the sequence container.

If the `set` object is *const-qualified*, the function returns a `const_reverse_iterator`. Otherwise, it returns a `reverse_iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are *reverse bidirectional iterator* types pointing to elements. See `set` member types.

## Example

```
1 // set::rbegin/rend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {21,64,17,78,49};
8     std::set<int> myset (myints,myints+5);
9
10    std::set<int>::reverse_iterator rit;
11
12    std::cout << "myset contains:";
13    for (rit=myset.rbegin(); rit != myset.rend(); ++rit)
14        std::cout << ' ' << *rit;
15
16    std::cout << '\n';
17
18    return 0;
19 }
```

Output:

```
myset contains: 78 64 49 21 17
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the *const* nor the non-*const* versions modify the container).  
Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">set::rend</a>	Return reverse iterator to reverse end (public member function )
<a href="#">set::begin</a>	Return iterator to beginning (public member function )
<a href="#">set::end</a>	Return iterator to end (public member function )

## /set/set/rend

public member function

### std::set::rend

<set>

```
reverse_iterator rend();
const_reverse_iterator rend() const;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

#### Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element right before the first element in the `set` container (which is considered its *reverse end*).

The range between `set::rbegin` and `set::rend` contains all the elements of the container (in reverse order).

## Parameters

none

## Return Value

A reverse iterator to the *reverse end* of the sequence container.

If the `set` object is const-qualified, the function returns a `const_reverse_iterator`. Otherwise, it returns a `reverse_iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are reverse `bidirectional iterator` types pointing to elements. See `set` member types.

## Example

```
1 // set::rbegin/rend
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {21,64,17,78,49};
8     std::set<int> myset (myints,myints+5);
9
10    std::set<int>::reverse_iterator rit;
11
12    std::cout << "myset contains:";
13    for (rit=myset.rbegin(); rit != myset.rend(); ++rit)
14        std::cout << ' ' << *rit;
15
16    std::cout << '\n';
17
18    return 0;
19 }
```

Output:

```
myset contains: 78 64 49 21 17
```

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container). Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">set::rbegin</a>	Return reverse iterator to reverse beginning (public member function )
<a href="#">set::begin</a>	Return iterator to beginning (public member function )

## /set/set/set

public member function

**std::set::set**

<set>

```

empty (1) explicit set (const key_compare& comp = key_compare(),
                        const allocator_type& alloc = allocator_type());
template <class InputIterator>
range (2)   set (InputIterator first, InputIterator last,
                  const key_compare& comp = key_compare(),
                  const allocator_type& alloc = allocator_type());
copy (3)   set (const set& x);

empty (1)   explicit set (const key_compare& comp = key_compare(),
                           const allocator_type& alloc = allocator_type());
explicit set (const allocator_type& alloc);
template <class InputIterator>
range (2)   set (InputIterator first, InputIterator last,
                  const key_compare& comp = key_compare(),
                  const allocator_type& alloc = allocator_type());
copy (3)   set (const set& x);
set (const set& x, const allocator_type& alloc);
move (4)   set (set&& x);
set (set&& x, const allocator_type& alloc);
set (initializer_list<value_type> il,
      const key_compare& comp = key_compare(),
      const allocator_type& alloc = allocator_type());

empty (1)   set();
explicit set (const key_compare& comp,
              const allocator_type& alloc = allocator_type());
explicit set (const allocator_type& alloc);
template <class InputIterator>
range (2)   set (InputIterator first, InputIterator last,
                  const key_compare& comp = key_compare(),
                  const allocator_type& alloc = allocator_type());
template <class InputIterator>
set (InputIterator first, InputIterator last,
      const allocator_type& alloc = allocator_type());
copy (3)   set (const set& x);
set (const set& x, const allocator_type& alloc);
move (4)   set (set&& x);
set (set&& x, const allocator_type& alloc);
set (initializer_list<value_type> il,
      const key_compare& comp = key_compare(),
      const allocator_type& alloc = allocator_type());
initializer list (5) set (initializer_list<value_type> il,
                           const allocator_type& alloc = allocator_type());

```

### Construct set

Constructs a `set` container object, initializing its contents depending on the constructor version used:

#### (1) empty container constructor (default constructor)

Constructs an `empty` container, with no elements.

#### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element constructed from its corresponding element in that range.

#### (3) copy constructor

Constructs a container with a copy of each of the elements in `x`.

The container keeps an internal copy of `alloc` and `comp`, which are used to allocate storage and to sort the elements throughout its lifetime.  
The copy constructor (3) creates a container that keeps and uses copies of `x`'s allocator and comparison object.

The storage for the elements is allocated using this [internal allocator](#).

#### (1) empty container constructors (default constructor)

Constructs an `empty` container, with no elements.

#### (2) range constructor

Constructs a container with as many elements as the range `[first, last)`, with each element *emplace-constructed* from its corresponding element in that range.

#### (3) copy constructor (and copying with allocator)

Constructs a container with a copy of each of the elements in `x`.

#### (4) move constructor (and moving with allocator)

Constructs a container that acquires the elements of `x`.

If `alloc` is specified and is different from `x`'s allocator, the elements are moved. Otherwise, no elements are constructed (their ownership is directly transferred).

`x` is left in an unspecified but valid state.

#### (5) initializer list constructor

Constructs a container with a copy of each of the elements in `il`.

The container keeps an internal copy of `alloc`, which is used to allocate and deallocate storage for its elements, and to construct and destroy them (as specified by its [allocator\\_traits](#)). If no `alloc` argument is passed to the constructor, a default-constructed allocator is used, except in the following cases:

- The copy constructor (3, *first signature*) creates a container that keeps and uses a copy of the allocator returned by calling the appropriate [selected\\_on\\_container\\_copy\\_construction](#) trait on `x`'s allocator.

- The move constructor (4, *first signature*) acquires `x`'s allocator.

The container also keeps an internal copy of `comp` (or `x`'s comparison object), which is used to establish the order of the elements in the container and to check

for equivalent elements.

All elements are *copied*, *moved* or otherwise *constructed* by calling `allocator_traits::construct` with the appropriate arguments.

The elements are sorted according to the *comparison object*. If more than one equivalent element is passed to the constructor, only the first one is preserved.

## Parameters

`comp`

Binary predicate that, taking two values of the same type of those contained in the `set`, returns `true` if the first argument goes before the second argument in the *strict weak ordering* it defines, and `false` otherwise.  
This shall be a function pointer or a function object.  
Member type `key_compare` is the internal comparison object type used by the container, defined in `set` as an alias of its second template parameter (`Compare`).  
If `key_compare` uses the default `less` (which has no state), this parameter is not relevant.

`alloc`

Allocator object.  
The container keeps and uses an internal copy of this allocator.  
Member type `allocator_type` is the internal allocator type used by the container, defined in `set` as an alias of its third template parameter (`Alloc`).  
If `allocator_type` is an instantiation of the default `allocator` (which has no state), this parameter is not relevant.

`first, last`

*Input iterators* to the initial and final positions in a range. The range used is `[first, last)`, which includes all the elements between `first` and `last`, including the element pointed by `first` but not the element pointed by `last`.  
The function template argument `InputIterator` shall be an *input iterator* type that points to elements of a type from which `value_type` objects can be constructed.

`x`

Another `set` object of the same type (with the same class template arguments `T`, `Compare` and `Alloc`), whose contents are either copied or acquired.

`il`

An *initializer\_list* object.  
These objects are automatically constructed from *initializer list* declarators.  
Member type `value_type` is the type of the elements in the container, defined in `set` as an alias of its first template parameter (`T`).

## Example

```
1 // constructing sets
2 #include <iostream>
3 #include <set>
4
5 bool fncomp (int lhs, int rhs) {return lhs<rhs;}
6
7 struct classcomp {
8     bool operator() (const int& lhs, const int& rhs) const
9     {return lhs<rhs;}
10 };
11
12 int main ()
13 {
14     std::set<int> first;                                // empty set of ints
15
16     int myints[] = {10,20,30,40,50};                    // range
17     std::set<int> second (myints,myints+5);           // a copy of second
18
19     std::set<int> third (second);                     // iterator ctor.
20
21     std::set<int> fourth (second.begin(), second.end()); // class as Compare
22
23     std::set<int,classcomp> fifth;                   // function pointer as Compare
24
25     bool(*fn_pt)(int,int) = fncomp;
26     std::set<int,bool(*)(int,int)> sixth (fn_pt); // move constructors
27
28     return 0;
29 }
```

The code does not produce any output, but demonstrates some ways in which a `set` container can be constructed.

## Complexity

Constant for the *empty constructors* (1), and for the *move constructors* (4) (unless `alloc` is different from `x`'s allocator).

For all other cases, linear in the distance between the iterators (copy constructions) if the elements are already sorted according to the same criterion. For unsorted sequences, linearithmic ( $N \log N$ ) in that distance (sorting,copy constructions).

## Iterator validity

The *move constructors* (4), invalidate all iterators, pointers and references related to `x` if the elements are moved.

## Data races

All copied elements are accessed.

The *move constructors* (4) modify `x`.

## Exception safety

**Strong guarantee:** no effects in case an exception is thrown.

If `allocator_traits::construct` is not supported with the appropriate arguments for the element constructions, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

## See also

<b>set::operator=</b>	Copy container content (public member function )
<b>set::clear</b>	Clear content (public member function )
<b>set::insert</b>	Insert element (public member function )

## /set/set/~set

public member function

### std::set::~set

<set>

`~set();`

#### Set destructor

Destroys the container object.

This destroys all container elements, and deallocates all the storage capacity allocated by the `set` container using its allocator.

This calls `allocator_traits::destroy` on each of the contained elements, and deallocates all the storage capacity allocated by the `set` container using its allocator.

#### Complexity

Linear in `set::size` (destructors).

#### Iterator validity

All iterators, pointers and references are invalidated.

#### Data races

The container and all its elements are modified.

#### Exception safety

**No-throw guarantee:** never throws exceptions.

## /set/set/size

public member function

### std::set::size

<set>

`size_type size() const;`  
`size_type size() const noexcept;`

#### Return container size

Returns the number of elements in the `set` container.

#### Parameters

none

#### Return Value

The number of elements in the container.

Member type `size_type` is an unsigned integral type.

#### Example

```
1 // set::size
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myints;
8     std::cout << "0. size: " << myints.size() << '\n';
9
10    for (int i=0; i<10; ++i) myints.insert(i);
11    std::cout << "1. size: " << myints.size() << '\n';
12
13    myints.insert (100);
14    std::cout << "2. size: " << myints.size() << '\n';
15
16    myints.erase(5);
17    std::cout << "3. size: " << myints.size() << '\n';
18
19    return 0;
20 }
```

Output:

0. size: 0  
1. size: 10  
2. size: 11  
3. size: 10

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.  
Concurrently accessing the elements of a `set` is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>set::max_size</code>	Return maximum size (public member function )
<code>set::empty</code>	Test whether container is empty (public member function )

# /set/set/swap

public member function

## std::set::swap

<set>

`void swap (set& x);`

### Swap content

Exchanges the content of the container by the content of `x`, which is another `set` of the same type. Sizes may differ.

After the call to this member function, the elements in this container are those which were in `x` before the call, and the elements of `x` are those which were in this. All iterators, references and pointers remain valid for the swapped objects.

Notice that a non-member function exists with the same name, `swap`, overloading that algorithm with an optimization that behaves like this member function.

Whether the internal container allocators and comparison objects are swapped is undefined.

Whether the internal container allocators are swapped is not defined, unless in the case the appropriate allocator trait indicates explicitly that they shall propagate.

The internal comparison objects are always exchanged, using `swap`.

## Parameters

`x`

Another `set` container of the same type as this (i.e., with the same template parameters, `T`, `Compare` and `Alloc`) whose content is swapped with that of this container.

## Return value

none

## Example

```
1 // swap sets
2 #include <iostream>
3 #include <set>
4
5 main ()
6 {
7     int myints[]={12,75,10,32,20,25};
8     std::set<int> first (myints,myints+3);      // 10,12,75
9     std::set<int> second (myints+3,myints+6);    // 20,25,32
10
11    first.swap(second);
12
13    std::cout << "first contains:";
14    for (std::set<int>::iterator it=first.begin(); it!=first.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    std::cout << "second contains:";
19    for (std::set<int>::iterator it=second.begin(); it!=second.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
first contains: 20 25 32
second contains: 10 12 75
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, but now are referring to elements in the other container, and iterate in it.

Note that the *end iterators* do not refer to elements and may be invalidated.

## Data races

Both the container and *x* are modified.

No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both containers compare equal, or if their *allocator traits* indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<b>swap (set)</b>	Exchanges the contents of two sets (function template )
<b>swap_ranges</b>	Exchange values of two ranges (function template )

# /set/set/swap-free

function template

## std::swap (set)

<set>

```
template <class T, class Compare, class Alloc>
void swap (set<T,Compare,Alloc>& x, set<T,Compare,Alloc>& y);
```

### Exchanges the contents of two sets

The contents of container *x* are exchanged with those of *y*. Both container objects must be of the same type (same template parameters), although sizes may differ.

After the call to this member function, the elements in *x* are those which were in *y* before the call, and the elements of *y* are those which were in *x*. All iterators, references and pointers remain valid for the swapped objects.

This is an overload of the generic algorithm *swap* that improves its performance by mutually transferring ownership over their assets to the other container (i.e., the containers exchange references to their data, without actually performing any element copy or movement): It behaves as if *x.swap(y)* was called.

## Parameters

*x,y*  
set containers of the same type (i.e., having both the same template parameters, *T*, *Compare* and *Alloc*).

## Return value

none

## Example

```
1 // swap (set overload)
2 #include <iostream>
3 #include <set>
4
5 main ()
6 {
7     int myints[]={12,75,10,32,20,25};
8     std::set<int> first (myints,myints+3);      // 10,12,75
9     std::set<int> second (myints+3,myints+6);   // 20,25,32
10
11    swap(first,second);
12
13    std::cout << "first contains:" ;
14    for (std::set<int>::iterator it=first.begin(); it!=first.end(); ++it)
15        std::cout << ' ' << *it;
16    std::cout << '\n';
17
18    std::cout << "second contains:" ;
19    for (std::set<int>::iterator it=second.begin(); it!=second.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
first contains: 20 25 32
second contains: 10 12 75
```

## Complexity

Constant.

## Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.  
Note that the *end iterators* do not refer to elements and may be invalidated.

## Data races

Both containers, *x* and *y*, are modified.  
No contained elements are accessed by the call (although see *iterator validity* above).

## Exception safety

If the allocators in both *sets* compare equal, or if their allocator traits indicate that the allocators shall *propagate*, the function never throws exceptions (no-throw guarantee).  
Otherwise, it causes *undefined behavior*.

## See also

<a href="#">set::swap</a>	Swap content (public member function )
<a href="#">swap</a>	Exchange values of two objects (function template )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )

# /set/set/upper\_bound

public member function

## std::set::upper\_bound

<set>

```
iterator upper_bound (const value_type& val) const;
    iterator upper_bound (const value_type& val);
const_iterator upper_bound (const value_type& val) const;
```

### Return iterator to upper bound

Returns an iterator pointing to the first element in the container which is considered to go after *val*.

The function uses its internal comparison object (*key\_comp*) to determine this, returning an iterator to the first element for which *key\_comp(val,element)* would return *true*.

If the *set* class is instantiated with the default comparison type (*less*), the function returns an iterator to the first element that is greater than *val*.

A similar member function, *lower\_bound*, has the same behavior as *upper\_bound*, except in the case that the *set* contains an element equivalent to *val*: In this case *lower\_bound* returns an iterator pointing to that element, whereas *upper\_bound* returns an iterator pointing to the next element.

## Parameters

*val*

Value to compare.

Member type *value\_type* is the type of the elements in the container, defined in *set* as an alias of its first template parameter (*T*).

## Return value

An iterator to the the first element in the container which is considered to go after *val*, or *set::end* if no elements are considered to go after *val*.

Member types *iterator* and *const\_iterator* are *bidirectional iterator* types pointing to elements.

## Example

```
1 // set::lower_bound/upper_bound
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     std::set<int>::iterator itlow,itup;
9
10    for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90
11
12    itlow=myset.lower_bound (30);           //      ^
13    itup=myset.upper_bound (60);           //      ^
14
15    myset.erase(itlow,itup);              // 10 20 70 80 90
16
17    std::cout << "myset contains:";
18    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
```

Notice that *lower\_bound(30)* returns an iterator to 30, whereas *upper\_bound(60)* returns an iterator to 70.

```
myset contains: 10 20 70 80 90
```

## Complexity

Logarithmic in `size`.

## Iterator validity

---

No changes.

## Data races

---

The container is accessed (neither the `const` nor the non-`const` versions modify the container). Concurrently accessing the elements of a `set` is safe.

## Exception safety

---

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

---

<code>set::lower_bound</code>	Return iterator to lower bound ( <a href="#">public member function</a> )
<code>set::equal_range</code>	Get range of equal elements ( <a href="#">public member function</a> )
<code>set::find</code>	Get iterator to element ( <a href="#">public member function</a> )
<code>set::count</code>	Count elements with a specific value ( <a href="#">public member function</a> )

# /set/set/value\_comp

public member function

## std::set::value\_comp

---

<set>

`value_compare value_comp() const;`

### Return comparison object

Returns a copy of the *comparison object* used by the container.

By default, this is a `less` object, which returns the same as `operator<`.

This object determines the order of the elements in the container: it is a function pointer or a function object that takes two arguments of the same type as the container elements, and returns `true` if the first argument is considered to go before the second in the *strict weak ordering* it defines, and `false` otherwise.

Two elements of a `set` are considered equivalent if `value_comp` returns `false` reflexively (i.e., no matter the order in which the elements are passed as arguments).

In `set` containers, the keys to sort the elements are the values themselves, therefore `value_comp` and its sibling member function `key_comp` are equivalent.

## Parameters

---

none

## Return value

---

The comparison object.

Member type `value_compare` is the type of the *comparison object* associated to the container, defined in `set` as an alias of its second template parameter (`Compare`).

## Example

---

```
1 // set::value_comp
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8
9     std::set<int>::value_compare mycomp = myset.value_comp();
10
11    for (int i=0; i<=5; i++) myset.insert(i);
12
13    std::cout << "myset contains:";
14
15    int highest=*myset.rbegin();
16    std::set<int>::iterator it=myset.begin();
17    do {
18        std::cout << ' ' << *it;
19    } while ( mycomp(*(++it),highest) );
20
21    std::cout << '\n';
22
23    return 0;
24 }
```

Output:

```
myset contains: 0 1 2 3 4
```

## Complexity

---

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed.

Concurrently accessing the elements of a [set](#) is safe.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.

## See also

<a href="#">set::key_comp</a>	Return comparison object (public member function )
<a href="#">set::find</a>	Get iterator to element (public member function )
<a href="#">set::count</a>	Count elements with a specific value (public member function )
<a href="#">set::lower_bound</a>	Return iterator to lower bound (public member function )
<a href="#">set::upper_bound</a>	Return iterator to upper bound (public member function )

# /string

header

## <string>

### Strings

This header introduces string types, character traits and a set of converting functions:

### Class templates

<a href="#">basic_string</a>	Generic string class (class template )
<a href="#">char_traits</a>	Character traits (class template )

### Class instantiations

<a href="#">string</a>	String class (class )
<a href="#">u16string</a>	String of 16-bit characters (class )
<a href="#">u32string</a>	String of 32-bit characters (class )
<a href="#">wstring</a>	Wide string (class )

### Functions

#### Convert from strings

<a href="#">stoi</a>	Convert string to integer (function template )
<a href="#">stol</a>	Convert string to long int (function template )
<a href="#">stoul</a>	Convert string to unsigned integer (function template )
<a href="#">stoll</a>	Convert string to long long (function template )
<a href="#">stoull</a>	Convert string to unsigned long long (function template )
<a href="#">stof</a>	Convert string to float (function template )
<a href="#">stod</a>	Convert string to double (function template )
<a href="#">stold</a>	Convert string to long double (function template )

#### Convert to strings

<a href="#">to_string</a>	Convert numerical value to string (function )
<a href="#">to_wstring</a>	Convert numerical value to wide string (function )

#### Range access

<a href="#">begin</a>	Iterator to beginning (function template )
<a href="#">end</a>	Iterator to end (function template )

# /string/basic\_string

class template

## std::basic\_string

<string>

```
template < class charT,
          class traits = char_traits<charT>,      // basic_string::traits_type
          class Alloc = allocator<charT>           // basic_string::allocator_type
        > class basic_string;
```

### Generic string class

The [basic\\_string](#) is the generalization of class [string](#) for any character type (see [string](#) for a description).

## Template parameters

**charT**  
Character type.  
The string is formed by a sequence of characters of this type.  
This shall be a non-array POD type.

**traits**  
**Character traits** class that defines essential properties of the characters used by **basic\_string** objects (see **char\_traits**).  
**traits::char\_type** shall be the same as **charT**.  
Aliased as member type **basic\_string::traits\_type**.

**Alloc**  
Type of the allocator object used to define the storage allocation model. By default, the **allocator** class template is used, which defines the simplest memory allocation model and is value-independent.  
Aliased as member type **basic\_string::allocator\_type**.

Note: Because the first template parameter is not aliased as any member type, **charT** is used throughout this reference to refer to this type.

## Template instantiations

<b>string</b>	String class (class )
<b>wstring</b>	Wide string (class )
<b>u16string</b>	String of 16-bit characters (class )
<b>u32string</b>	String of 32-bit characters (class )

## Member types

member type	definition	notes
<b>traits_type</b>	The second template parameter ( <b>traits</b> )	defaults to: <b>char_traits&lt;charT&gt;</b>
<b>allocator_type</b>	The third template parameter ( <b>Alloc</b> )	defaults to: <b>allocator&lt;charT&gt;</b>
<b>value_type</b>	<b>traits_type::char_type</b>	shall be the same as <b>charT</b>
<b>reference</b>	<b>allocator_type::reference</b>	for the default <b>allocator::charT&amp;</b>
<b>const_reference</b>	<b>allocator_type::const_reference</b>	for the default <b>allocator::const charT&amp;</b>
<b>pointer</b>	<b>allocator_type::pointer</b>	for the default <b>allocator::charT*</b>
<b>const_pointer</b>	<b>allocator_type::const_pointer</b>	for the default <b>allocator::const charT*</b>
<b>iterator</b>	a random access iterator to <b>charT</b>	convertible to <b>const_iterator</b>
<b>const_iterator</b>	a random access iterator to <b>const charT</b>	
<b>reverse_iterator</b>	<b>reverse_iterator&lt;iterator&gt;</b>	
<b>const_reverse_iterator</b>	<b>reverse_iterator&lt;const_iterator&gt;</b>	
<b>difference_type</b>	<b>allocator_type::difference_type</b>	usually the same as <b>ptrdiff_t</b>
<b>size_type</b>	<b>allocator_type::size_type</b>	usually the same as <b>size_t</b>

member type	definition	notes
<b>traits_type</b>	The second template parameter ( <b>traits</b> )	defaults to: <b>char_traits&lt;charT&gt;</b>
<b>allocator_type</b>	The third template parameter ( <b>Alloc</b> )	defaults to: <b>allocator&lt;charT&gt;</b>
<b>value_type</b>	<b>traits_type::char_type</b>	shall be the same as <b>charT</b>
<b>reference</b>	<b>value_type&amp;</b>	
<b>const_reference</b>	<b>const value_type&amp;</b>	
<b>pointer</b>	<b>allocator_traits&lt;allocator_type&gt;::pointer</b>	for the default <b>allocator::charT*</b>
<b>const_pointer</b>	<b>allocator_traits&lt;allocator_type&gt;::const_pointer</b>	for the default <b>allocator::const charT*</b>
<b>iterator</b>	a random access iterator to <b>charT</b>	convertible to <b>const_iterator</b>
<b>const_iterator</b>	a random access iterator to <b>const charT</b>	
<b>reverse_iterator</b>	<b>reverse_iterator&lt;iterator&gt;</b>	
<b>const_reverse_iterator</b>	<b>reverse_iterator&lt;const_iterator&gt;</b>	
<b>difference_type</b>	<b>allocator_traits&lt;allocator_type&gt;::difference_type</b>	usually the same as <b>ptrdiff_t</b>
<b>size_type</b>	<b>allocator_traits&lt;allocator_type&gt;::size_type</b>	usually the same as <b>size_t</b>

## Member functions

<b>(constructor)</b>	Construct <b>basic_string</b> object (public member function )
<b>(destructor)</b>	String destructor (public member function )
<b>operator=</b>	String assignment (public member function )

### Iterators:

<b>begin</b>	Return iterator to beginning (public member function )
<b>end</b>	Return iterator to end (public member function )
<b>rbegin</b>	Return reverse iterator to reverse beginning (public member function )
<b>rend</b>	Return reverse iterator to reverse end (public member function )
<b>cbegin</b>	Return <b>const_iterator</b> to beginning (public member function )
<b>cend</b>	Return <b>const_iterator</b> to end (public member function )
<b>crbegin</b>	Return <b>const_reverse_iterator</b> to reverse beginning (public member function )
<b>crend</b>	Return <b>const_reverse_iterator</b> to reverse end (public member function )

### Capacity:

<b>size</b>	Return size (public member function )
<b>length</b>	Return length of string (public member function )
<b>max_size</b>	Return maximum size (public member function )
<b>resize</b>	Resize string (public member function )

<b>capacity</b>	Return size of allocated storage (public member function )
<b>reserve</b>	Request a change in capacity (public member function )
<b>clear</b>	Clear string (public member function )
<b>empty</b>	Test whether string is empty (public member function )
<b>shrink_to_fit</b>	Shrink to fit (public member function )

#### Element access:

<b>operator[]</b>	Get character of string (public member function )
<b>at</b>	Get character of string (public member function )
<b>back</b>	Access last character (public member function )
<b>front</b>	Access first character (public member function )

#### Modifiers:

<b>operator+=</b>	Append to string (public member function )
<b>append</b>	Append to string (public member function )
<b>push_back</b>	Append character to string (public member function )
<b>assign</b>	Assign content to string (public member function )
<b>insert</b>	Insert into string (public member function )
<b>erase</b>	Erase characters from string (public member function )
<b>replace</b>	Replace portion of string (public member function )
<b>swap</b>	Swap string values (public member function )
<b>pop_back</b>	Delete last character (public member function )

#### String operations:

<b>c_str</b>	Get C-string equivalent
<b>data</b>	Get string data (public member function )
<b>get_allocator</b>	Get allocator (public member function )
<b>copy</b>	Copy sequence of characters from string (public member function )
<b>find</b>	Find first occurrence in string (public member function )
<b>rfind</b>	Find last occurrence in string (public member function )
<b>find_first_of</b>	Find character in string (public member function )
<b>find_last_of</b>	Find character in string from the end (public member function )
<b>find_first_not_of</b>	Find non-matching character in string (public member function )
<b>find_last_not_of</b>	Find non-matching character in string from the end (public member function )
<b>substr</b>	Generate substring (public member function )
<b>compare</b>	Compare strings (public member function )

#### Non-member function overloads

<b>operator+</b>	Concatenate strings (function template )
<b>relational operators</b>	Relational operators for basic_string (function template )
<b>swap</b>	Exchanges the values of two strings (function template )
<b>operator&gt;&gt;</b>	Extract string from stream (function template )
<b>operator&lt;&lt;</b>	Insert string into stream (function template )
<b>getline</b>	Get line from stream into string (function template )

#### Member constants

<b>npos</b>	Maximum value of size_type (public static member constant )
-------------	---

## /string/basic\_string/append

public member function

### std::basic\_string::append

<string>

```

string(1) basic_string& append (const basic_string& str);
substring(2) basic_string& append (const basic_string& str, size_type subpos, size_type sublen);
c-string(3) basic_string& append (const charT* s);
buffer(4) basic_string& append (const charT* s, size_type n);
fill(5) basic_string& append (size_type n, charT c);
range(6) template <class InputIterator>
          basic_string& append (InputIterator first, InputIterator last);

string(1) basic_string& append (const basic_string& str);
substring(2) basic_string& append (const basic_string& str, size_type subpos, size_type sublen);
c-string(3) basic_string& append (const charT* s);
buffer(4) basic_string& append (const charT* s, size_type n);
fill(5) basic_string& append (size_type n, charT c);
range(6) template <class InputIterator>
          basic_string& append (InputIterator first, InputIterator last);
initializer_list(7) basic_string& append (initializer_list<charT> il);

```

```

    string(1) basic_string& append (const basic_string& str);
    substring(2) basic_string& append (const basic_string& str, size_type subpos, size_type sublen = npos);
    c-string(3) basic_string& append (const charT* s);
    buffer(4) basic_string& append (const charT* s, size_type n);
    fill(5) basic_string& append (size_type n, charT c);
    template <class InputIterator>
    range(6) basic_string& append (InputIterator first, InputIterator last);
    initializer_list(7) basic_string& append (initializer_list<charT> il);

```

## Append to string

Extends the [basic\\_string](#) by appending additional characters at the end of its current value:

### (1) string

Appends a copy of *str*.

### (2) substring

Appends a copy of a substring of *str*. The substring is the portion of *str* that begins at the character position *subpos* and spans *sublen* characters (or until the end of *str*, if either *str* is too short or if *sublen* is [basic\\_string::npos](#)).

### (3) c-string

Appends a copy of the string formed by the null-terminated character sequence (C-string) pointed by *s*.  
The length of this character sequence is determined by calling [traits\\_type::length\(s\)](#).

### (4) buffer

Appends a copy of the first *n* characters in the array of characters pointed by *s*.

### (5) fill

Appends *n* consecutive copies of character *c*.

### (6) range

Appends a copy of the sequence of characters in the range [*first*,*last*), in the same order.

### (7) initializer list

Appends a copy of each of the characters in *il*, in the same order.

## Parameters

*str* Another [basic\\_string](#) object of the same type (with the same class template arguments *charT*, *traits* and *Alloc*), whose value is appended.

*subpos* Position of the first character in *str* that is copied to the object as a substring.  
If this is greater than *str*'s [length](#), it throws [out\\_of\\_range](#).  
Note: The first character in *str* is denoted by a value of 0 (not 1).

*sublen* Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).  
A value of [basic\\_string::npos](#) indicates all characters until the end of *str*.

*s* Pointer to an array of characters (such as a *c-string*).

*n* Number of characters to copy.

*c* Character value, repeated *n* times.

*first*, *last* [Input iterators](#) to the initial and final positions in a range. The range used is [*first*,*last*), which includes all the characters between *first* and *last*, including the character pointed by *first* but not the character pointed by *last*.  
The function template argument *InputIterator* shall be an [input iterator](#) type that points to elements of a type convertible to *charT*.  
If *InputIterator* is an integral type, the arguments are casted to the proper types so that signature (5) is used instead.

*il* An [initializer\\_list](#) object.  
These objects are automatically constructed from *initializer list* declarators.

*charT* is [basic\\_string](#)'s character type (i.e., its first template parameter).

Member type *size\_type* is an unsigned integral type.

## Return Value

\*this

## Example

```

1 // appending to string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str;
8     std::string str2="Writing ";
9     std::string str3="print 10 and then 5 more";
10
11    // used in the same order as described above:
12    str.append(str2);           // "Writing "
13    str.append(str3,6,3);       // "10 "
14    str.append("dots are cool",5); // "dots "
15    str.append("here: ");       // "here: "
16    str.append(10u,'.');// "....."
17    str.append(str3.begin()+8,str3.end()); // " and then 5 more"
18    str.append<int>(5,0x2E); // "...."
19
20    std::cout << str << '\n';

```

```
21 |     return 0;
22 }
```

#### Output:

```
Writing 10 dots here: ..... and then 5 more.....
```

### Complexity

Unspecified, but generally up to linear in the new `string` length.

### Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

### Data races

The object is modified.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If `s` does not point to an array long enough, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

If `subpos` is greater than `str`'s `length`, an `out_of_range` exception is thrown.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default_allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

### See also

<code>basic_string::operator+=</code>	Append to string (public member function )
<code>basic_string::assign</code>	Assign content to string (public member function )
<code>basic_string::insert</code>	Insert into string (public member function )
<code>basic_string::replace</code>	Replace portion of string (public member function )

## /string/basic\_string/assign

public member function

### std::basic\_string::assign

`<string>`

```
string (1) basic_string& assign (const basic_string& str);
substring (2) basic_string& assign (const basic_string& str, size_type subpos, size_type sublen);
c-string (3) basic_string& assign (const charT* s);
buffer (4) basic_string& assign (const charT* s, size_type n);
fill (5) basic_string& assign (size_type n, charT c);
range (6) template <class InputIterator>
           basic_string& assign (InputIterator first, InputIterator last);

string (1) basic_string& assign (const basic_string& str);
substring (2) basic_string& assign (const basic_string& str, size_type subpos, size_type sublen);
c-string (3) basic_string& assign (const charT* s);
buffer (4) basic_string& assign (const charT* s, size_type n);
fill (5) basic_string& assign (size_type n, charT c);
range (6) template <class InputIterator>
           basic_string& assign (InputIterator first, InputIterator last);
initializer list(7) basic_string& assign (initializer_list<charT> il);
move (8) basic_string& assign (basic_string&& str) noexcept;

string (1) basic_string& assign (const basic_string& str);
substring (2) basic_string& assign (const basic_string& str, size_type subpos, size_type sublen = npos);
c-string (3) basic_string& assign (const charT* s);
buffer (4) basic_string& assign (const charT* s, size_type n);
fill (5) basic_string& assign (size_type n, charT c);
range (6) template <class InputIterator>
           basic_string& assign (InputIterator first, InputIterator last);
initializer list(7) basic_string& assign (initializer_list<charT> il);
move (8) basic_string& assign (basic_string&& str) noexcept;
```

### Assign content to string

Assigns a new value to the string, replacing its current contents.

#### (1) `string`

Copies `str`.

#### (2) `substring`

Copies the portion of `str` that begins at the character position `subpos` and spans `sublen` characters (or until the end of `str`, if either `str` is too short or if `sublen` is `basic_string::npos`).

#### (3) `c-string`

Copies the null-terminated character sequence (C-string) pointed by `s`.  
The `length` is determined by calling `traits_type::length(s)`.

#### (4) `buffer`

Copies the first `n` characters from the array of characters pointed by `s`.

**(5) fill**

Replaces the current value by  $n$  consecutive copies of character  $c$ .

**(6) range**

Copies the sequence of characters in the range  $[first, last)$ , in the same order.

**(7) initializer list**

Copies each of the characters in  $il$ , in the same order.

**(8) move**

Acquires the contents of  $str$ .

$str$  is left in an unspecified but valid state.

---

## Parameters

<code>str</code>	Another <code>basic_string</code> object of the same type (with the same class template arguments <code>charT</code> , <code>traits</code> and <code>Alloc</code> ), whose value is either copied or moved.
<code>subpos</code>	Position of the first character in $str$ that is copied to the object as a substring. If this is greater than $str$ 's <code>length</code> , it throws <code>out_of_range</code> . Note: The first character in $str$ is denoted by a value of 0 (not 1).
<code>sublen</code>	Length of the substring to be copied (if the string is shorter, as many characters as possible are copied). A value of <code>basic_string::npos</code> indicates all characters until the end of $str$ .
<code>s</code>	Pointer to an array of characters (such as a <i>c-string</i> ).
<code>n</code>	Number of characters to copy.
<code>c</code>	Character value, repeated $n$ times.
<code>first, last</code>	<code>Input iterators</code> to the initial and final positions in a range. The range used is $[first, last)$ , which includes all the characters between $first$ and $last$ , including the character pointed by $first$ but not the character pointed by $last$ . The function template argument <code>InputIterator</code> shall be an <code>input iterator</code> type that points to elements of a type convertible to <code>charT</code> . If <code>InputIterator</code> is an integral type, the arguments are casted to the proper types so that signature (5) is used instead.
<code>il</code>	An <code>initializer_list</code> object. These objects are automatically constructed from <i>initializer list</i> declarators.

`charT` is `basic_string`'s character type (i.e., its first template parameter).

Member type `size_type` is an unsigned integral type.

---

## Return Value

`*this`

---

## Example

```

1 // string::assign
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str;
8     std::string base="The quick brown fox jumps over a lazy dog.";
9
10    // used in the same order as described above:
11
12    str.assign(base);
13    std::cout << str << '\n';
14
15    str.assign(base,10,9);           // "brown fox"
16    std::cout << str << '\n';      // "pangram"
17
18    str.assign("pangrams are cool",7);
19    std::cout << str << '\n';      // "c-string"
20
21    str.assign("c-string");
22    std::cout << str << '\n';      // "*****"
23
24    str.assign(10,'*');
25    std::cout << str << '\n';      // "*****"
26
27    str.assign<int>(10,0x2D);
28    std::cout << str << '\n';      // "-----"
29
30    str.assign(base.begin()+16,base.end()-12);
31    std::cout << str << '\n';      // "fox jumps over"
32
33    return 0;
34 }
```

Output:

```
The quick brown fox jumps over a lazy dog.
brown fox
pangram
c-string
*****
```

-----  
fox jumps over

## Complexity

Unspecified.

Unspecified, but generally linear in the new string length (and constant for the move version).

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

The move assign form (8), modifies str.

## Exception safety

For the move assign (8), the function does not throw exceptions (no-throw guarantee).

In all other cases, there are no effects in case an exception is thrown (strong guarantee).

If s does not point to an array long enough, or if the range specified by [first, last) is not valid, it causes *undefined behavior*.

If subpos is greater than str's length, an `out_of_range` exception is thrown.

If the resulting string length would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default_allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

`basic_string::operator=` String assignment (public member function )

`basic_string::append` Append to string (public member function )

`basic_string::insert` Insert into string (public member function )

`basic_string::replace` Replace portion of string (public member function )

# /string/basic\_string/at

public member function

## std::basic\_string::at

<string>

reference at (size\_type pos);  
const\_reference at (size\_type pos) const;

### Get character of string

Returns a reference to the character at position pos in the `basic_string`.

The function automatically checks whether pos is the valid position of a character in the string (i.e., whether pos is less than the string length), throwing an `out_of_range` exception if it is not.

## Parameters

pos

Value with the position of a character within the string.

Note: The first character in a `basic_string` is denoted by a value of 0 (not 1).

If it is not the position of a character, an `out_of_range` exception is thrown.

Member type `size_type` is an unsigned integral type.

## Return value

The character at the specified position in the string.

If the `basic_string` object is const-qualified, the function returns a `const_reference`. Otherwise, it returns a `reference`.

Member types `reference` and `const_reference` are the reference types to the characters in the container; They shall be aliases of `charT&` and `const charT&` respectively.

## Example

```
1 // string::at
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for (unsigned i=0; i<str.length(); ++i)
9     {
10         std::cout << str.at(i);
11     }
12     return 0;
13 }
```

This code prints out the content of a string character by character using the `at` member function:

Test string

## **Complexity**

Unspecified.

## **Iterator validity**

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

## **Data races**

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The reference returned can be used to access or modify characters.

## **Complexity**

Constant.

## **Iterator validity**

No changes.

## **Data races**

The object is accessed (neither the const nor the non-const versions modify it).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## **Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the [basic\\_string](#).

If *pos* is not less than the [string length](#), an [out\\_of\\_range](#) exception is thrown.

## **See also**

[basic\\_string::operator\[\]](#) Get character of string ([public member function](#))

[basic\\_string::substr](#) Generate substring ([public member function](#))

[basic\\_string::find](#) Find first occurrence in string ([public member function](#))

[basic\\_string::replace](#) Replace portion of string ([public member function](#))

# /string/basic\_string/back

public member function

## **std::basic\_string::back**

<string>

charT& back();  
const charT& back() const;

### **Access last character**

Returns a reference to the last character of the [basic\\_string](#).

This function shall not be called on [empty strings](#).

## **Parameters**

none

## **Return value**

A reference to the last character in the [basic\\_string](#).

If the [basic\\_string](#) object is const-qualified, the function returns a `const charT&`. Otherwise, it returns a `charT&`.

`charT` is [basic\\_string](#)'s character type (i.e., its first template parameter).

## **Example**

```

1 // string::back
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("hello world.");
8     str.back() = '!';
9     std::cout << str << '\n';
10    return 0;
11 }
```

Output:

hello world!

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The container is accessed (neither the const nor the non-const versions modify the container).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## Exception safety

If the `basic_string` is not `empty`, the function never throws exceptions (no-throw guarantee). Otherwise, it causes *undefined behavior*.

## See also

`basic_string::front` Access first character ([public member function](#))

`basic_string::push_back` Append character to string ([public member function](#))

`basic_string::pop_back` Delete last character ([public member function](#))

`basic_string::at` Get character of string ([public member function](#))

`basic_string::operator[]` Get character of string ([public member function](#))

# /string/basic\_string/~basic\_string

public member function

## std::basic\_string::~basic\_string

<string>

`~basic_string()`

### String destructor

Destroys the `basic_string` object.

This deallocates all the storage `capacity` allocated by the `basic_string` using its `allocator`.

## Complexity

Unspecified, but generally constant.

## Iterator validity

All iterators, pointers and references are invalidated.

## Data races

The object is modified.

## Exception safety

**No-throw guarantee:** never throws exceptions.

# /string/basic\_string/basic\_string

public member function

## std::basic\_string::basic\_string

<string>

`default (1) explicit basic_string (const allocator_type& alloc = allocator_type());`

`copy (2) basic_string (const basic_string& str);`

`substring (3) basic_string (const basic_string& str, size_type pos, size_type len = npos,`

`const allocator_type& alloc = allocator_type());`

`from c-string (4) basic_string (const charT* s, const allocator_type& alloc = allocator_type());`

`from sequence (5) basic_string (const charT* s, size_type n,`

`const allocator_type& alloc = allocator_type());`

`fill (6) basic_string (size_type n, charT c,`

`const allocator_type& alloc = allocator_type());`

`template <class InputIterator>`

`range (7) basic_string (Inputiterator first, Inputiterator last,`

`const allocator_type& alloc = allocator_type());`

`default (1) explicit basic_string (const allocator_type& alloc = allocator_type());`

`copy (2) basic_string (const basic_string& str);`

`basic_string (const basic_string& str, const allocator_type& alloc);`

`basic_string (const basic_string& str, size_type pos, size_type len = npos,`

`const allocator_type& alloc = allocator_type());`

`from c-string (4) basic_string (const charT* s, const allocator_type& alloc = allocator_type());`

`from buffer (5) basic_string (const charT* s, size_type n,`

`const allocator_type& alloc = allocator_type());`

```

fill (6) basic_string (size_type n, chart c,
                      const allocator_type& alloc = allocator_type());
template <class InputIterator>
range (7) basic_string (InputIterator first, InputIterator last,
                       const allocator_type& alloc = allocator_type());
initializer list (8) basic_string (initializer_list<chart> il,
                                    const allocator_type& alloc = allocator_type());
move (9) basic_string (basic_string&& str) noexcept;
           basic_string (basic_string&& str, const allocator_type& alloc);

default (1) basic_string();
copy (2) basic_string (const basic_string& str);
substring (3) basic_string (const basic_string& str, size_type pos, size_type len =npos,
                           const allocator_type& alloc = allocator_type());
from c-string (4) basic_string (const chart* s, const allocator_type& alloc = allocator_type());
from buffer (5) basic_string (const chart* s, size_type n,
                           const allocator_type& alloc = allocator_type());
fill (6) basic_string (size_type n, chart c,
                      const allocator_type& alloc = allocator_type());
template <class InputIterator>
range (7) basic_string (InputIterator first, InputIterator last,
                       const allocator_type& alloc = allocator_type());
initializer list (8) basic_string (initializer_list<chart> il,
                                    const allocator_type& alloc = allocator_type());
move (9) basic_string (basic_string&& str) noexcept;
           basic_string (basic_string&& str, const allocator_type& alloc);

```

## Construct basic\_string object

Constructs a `basic_string` object, initializing its value depending on the constructor version used:

### (1) empty string constructor (default constructor)

Constructs an `empty` string, with a `length` of zero characters.

### (2) copy constructors

Constructs a copy of `str`.

### (3) substring constructor

Copies the portion of `str` that begins at the character position `pos` and spans `len` characters (or until the end of `str`, if either `str` is too short or if `len` is `basic_string::npos`).

### (4) from c-string

Copies the null-terminated character sequence (C-string) pointed by `s`.  
The `length` is determined by calling `traits_type::length(s)`.

### (5) from buffer

Copies the first `n` characters from the array of characters pointed by `s`.

### (6) fill constructor

Fills the string with `n` consecutive copies of character `c`.

### (7) range constructor

Copies the sequence of characters in the range `[first, last]`, in the same order.

### (8) initializer list

Copies each of the characters in `il`, in the same order.

### (9) move constructors

Acquires the contents of `str`.

`str` is left in an unspecified but valid state.

The `basic_string` object keeps an internal copy of `alloc`, which is used to allocate and free storage for the characters it contains throughout its lifetime.  
The copy constructor (2) creates an object that keeps and uses a copy of `str`'s allocator.

The `basic_string` object keeps an internal copy of `alloc`, which is used to allocate and free storage for the characters it contains.

The copy constructor (2, *first signature*) creates a container that keeps and uses a copy of the allocator returned by calling the appropriate `selected_on_container_copy_construction` trait on `str`'s allocator.

The move constructor (9, *first signature*) acquires `str`'s allocator.

The `basic_string` object keeps an internal copy of `alloc`, which is used to allocate and free storage for the characters it contains.

The default constructor (1, *first signature*) uses a default-constructed allocator.

The copy constructor (2, *first signature*) creates a container that keeps and uses a copy of the allocator returned by calling the appropriate `selected_on_container_copy_construction` trait on `str`'s allocator.

The move constructor (9, *first signature*) acquires `str`'s allocator.

## Parameters

### alloc

Allocator object.

The container keeps and uses an internal copy of this allocator.

Member type `allocator_type` is the internal allocator type used by the container, defined in `basic_string` as an alias of its third template parameter (`Alloc`).

If `allocator_type` is an instantiation of the default `allocator` (which has no state), this is not relevant.

### str

Another `basic_string` object of the same type (with the same class template arguments `chart`, `traits` and `Alloc`), whose value is either copied or acquired.

### pos

Position of the first character in `str` that is copied to the object as a substring.

If this is greater than `str`'s `length`, it throws `out_of_range`.

Note: The first character in `str` is denoted by a value of 0 (not 1).

### len

Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).

A value of `basic_string::npos` indicates all characters until the end of `str`.

### s

Pointer to an array of characters (such as a *c-string*).

**n** Number of characters to copy.

**c** Character to fill the string with. Each of the *n* characters in the string will be initialized to a copy of this value.

**first, last** Input iterators to the initial and final positions in a range. The range used is [first, last), which includes all the characters between *first* and *last*, including the character pointed by *first* but not the character pointed by *last*.  
The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type convertible to `charT`. If `InputIterator` is an integral type, the arguments are casted to the proper types so that signature (5) is used instead.

**il** An `initializer_list` object.  
These objects are automatically constructed from *initializer list* declarators.

`charT` is `basic_string`'s character type (i.e., its first template parameter). Member type `size_type` is an unsigned integral type.

## Example

```
1 // string constructor
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string s0 ("Initial string");
8
9     // constructors used in the same order as described above:
10    std::string s1;
11    std::string s2 (s0);
12    std::string s3 (s0, 8, 3);
13    std::string s4 ("A character sequence", 6);
14    std::string s5 ("Another character sequence");
15    std::string s6 (10, 'x');
16    std::string s7a (10, 42);
17    std::string s7b (s0.begin(), s0.begin() + 7);
18
19    std::cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: " << s3;
20    std::cout << "\ns4: " << s4 << "\ns5: " << s5 << "\ns6: " << s6;
21    std::cout << "\ns7a: " << s7a << "\ns7b: " << s7b << '\n';
22    return 0;
23 }
```

Output:

```
s1:
s2: Initial string
s3: str
s4: A char
s5: Another character sequence
s6: xxxxxxxxxxxx
s7a: ****
s7b: Initial
```

## Complexity

Unspecified.

Unspecified, but generally linear in the resulting `string length` (and constant for *move constructors*).

## Iterator validity

The *move constructors* (9) may invalidate iterators, pointers and references related to *str*.

## Data races

The *move constructors* (9) modify *str*.

## Exception safety

The *move constructor* with no allocator argument (9, *first*) never throws exceptions (no-throw guarantee). In all other cases, there are no effects in case an exception is thrown (strong guarantee).

If *s* is a null pointer, if *n* == *npos*, or if the range specified by [first, last) is not valid, it causes *undefined behavior*.

If *pos* is greater than *str*'s `length`, an `out_of_range` exception is thrown.

If *n* is greater than the array pointed by *s*, it causes *undefined behavior*.

If the resulting `string length` would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function fails when attempting to allocate storage.

## See also

<code>basic_string::operator=</code>	String assignment (public member function )
--------------------------------------	---

<code>basic_string::assign</code>	Assign content to string (public member function )
-----------------------------------	--

<code>basic_string::resize</code>	Resize string (public member function )
-----------------------------------	---

<code>basic_string::clear</code>	Clear string (public member function )
----------------------------------	--

# /string/basic\_string/begin

public member function

## std::basic\_string::begin

<string>

```
    iterator begin();
const_iterator begin() const;
    iterator begin() noexcept;
const_iterator begin() const noexcept;
```

### Return iterator to beginning

Returns an iterator pointing to the first character of the string.

#### Parameters

none

#### Return Value

An iterator to the beginning of the string.

If the `basic_string` object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are `random access iterator` types (pointing to a character and to a const character, respectively).

#### Example

```
1 // string::begin/end
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for ( std::string::iterator it=str.begin(); it!=str.end(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
Test string
```

#### Complexity

Unspecified.

#### Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

#### Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

#### Complexity

Unspecified, but generally constant.

#### Iterator validity

No changes.

#### Data races

The object is accessed (neither the const nor the non-const versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

#### See also

<code>basic_string::end</code>	Return iterator to end ( <a href="#">public member function</a> )
--------------------------------	---

<code>basic_string::rbegin</code>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
-----------------------------------	---

## /string/basic\_string/capacity

public member function

### std::basic\_string::capacity

<string>

```
size_type capacity() const;
size_type capacity() const noexcept;
```

#### Return size of allocated storage

Returns the size of the storage space currently allocated for the `basic_string`, expressed in terms of characters.

This `capacity` is not necessarily equal to the `string length`. It can be equal or greater, with the extra space allowing the object to optimize its operations when new characters are added to the `basic_string`.

Notice that this `capacity` does not suppose a limit on the `length` of the `basic_string`. When this `capacity` is exhausted and more is needed, it is automatically expanded by the object (reallocating its storage space). The theoretical limit on the `length` of a `basic_string` is given by member `max_size`.

The `capacity` of a `basic_string` can be altered any time the object is modified, even if this modification implies a reduction in size or if the capacity has not been exhausted (this is in contrast with the guarantees given to `capacity` in `vector` containers).

The `capacity` of a `basic_string` can be explicitly altered by calling member `reserve`.

#### Parameters

none

#### Return Value

The size of the storage capacity currently allocated for the `basic_string`.

Member type `size_type` is an unsigned integral type.

#### Example

```
1 // comparing size, length, capacity and max_size
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "size: " << str.size() << "\n";
9     std::cout << "length: " << str.length() << "\n";
10    std::cout << "capacity: " << str.capacity() << "\n";
11    std::cout << "max_size: " << str.max_size() << "\n";
12    return 0;
13 }
```

A possible output for this program could be:

```
size: 11
length: 11
capacity: 15
max_size: 429496729
```

#### Complexity

Unspecified, but generally constant.

#### Iterator validity

No changes.

#### Data races

The object is accessed.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

#### See also

<code>basic_string::reserve</code>	Request a change in capacity ( <a href="#">public member function</a> )
<code>basic_string::length</code>	Return length of string ( <a href="#">public member function</a> )
<code>basic_string::size</code>	Return size ( <a href="#">public member function</a> )
<code>basic_string::max_size</code>	Return maximum size ( <a href="#">public member function</a> )

## /string/basic\_string/cbegin

public member function

### std::basic\_string::cbegin

<string>

```
const_iterator cbegin() const noexcept;
```

#### Return const\_iterator to beginning

Returns a const\_iterator pointing to the first character of the string.

A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by `basic_string::begin`, but it cannot be used to modify the contents it points to, even if the `basic_string` object is not itself const.

#### Parameters

none

#### Return Value

A const\_iterator to the beginning of the string.

Member type const\_iterator is a random access iterator type that points to a const character.

#### Example

```
1 // string::cbegin/cend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Lorem ipsum");
8     for (auto it=str.cbegin(); it!=str.cend(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
 Lorem ipsum
```

#### Complexity

Unspecified, but generally constant.

#### Iterator validity

No changes.

#### Data races

The object is accessed.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

#### See also

<code>basic_string::begin</code>	Return iterator to beginning (public member function )
<code>basic_string::cend</code>	Return const_iterator to end (public member function )
<code>basic_string::crbegin</code>	Return const_reverse_iterator to reverse beginning (public member function )

## /string/basic\_string/cend

public member function

#### std::basic\_string::cend

`<string>`

```
const_iterator cend() const noexcept;
```

#### Return const\_iterator to end

Returns a const\_iterator pointing to the *past-the-end* character of the string.

A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by `basic_string::end`, but it cannot be used to modify the contents it points to, even if the `basic_string` object is not itself const.

If the object is an `empty string`, this function returns the same as `basic_string::cbegin`.

The value returned shall not be dereferenced.

#### Parameters

none

#### Return Value

A const\_iterator to the past-the-end of the string.

Member type const\_iterator is a random access iterator type that points to a const character.

## Example

```
1 // string::cbegin/cend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Lorem ipsum");
8     for (auto it=str.cbegin(); it!=str.cend(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
LoREM ipsum
```

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">basic_string::end</a>	Return iterator to end (public member function )
<a href="#">basic_string::cbegin</a>	Return const_iterator to beginning (public member function )
<a href="#">basic_string::crend</a>	Return const_reverse_iterator to reverse end (public member function )

# /string/basic\_string/clear

public member function

## std::basic\_string::clear

<string>

```
void clear();
```

```
void clear() noexcept;
```

### Clear string

Erases the contents of the [basic\\_string](#), which becomes an [empty string](#) (with a [length](#) of 0 characters).

## Parameters

none

## Return value

none

## Example

```
1 // string::clear
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     char c;
8     std::string str;
9     std::cout << "Please type some lines of text. Enter a dot (.) to finish:\n";
10    do {
11        c = std::cin.get();
12        str += c;
13        if (c=='\n')
14        {
15            std::cout << str;
16            str.clear();
17        }
18    } while (c!='.');
19    return 0;
20 }
```

This program repeats every line introduced by the user until a the line contains a dot ('.'). Every newline character ('\n') triggers the repetition of the line and the clearing of the current string content.

## Complexity

Unspecified, but generally constant.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">basic_string::erase</a>	Erase characters from string (public member function )
<a href="#">basic_string::resize</a>	Resize string (public member function )
<a href="#">basic_string::empty</a>	Test whether string is empty (public member function )

# /string/basic\_string/compare

public member function

## std::basic\_string::compare

<string>

string (1)	int compare (const basic_string& str) const;
substrings (2)	int compare (size_type pos, size_type len, const basic_string& str) const;
	int compare (size_type pos, size_type len, const basic_string& str, size_type subpos, size_type sublen) const;
c-string (3)	int compare (const charT* s) const;
	int compare (size_type pos, size_type len, const charT* s) const;
buffer (4)	int compare (size_type pos, size_type len, const charT* s, size_type n) const;
string (1)	int compare (const basic_string& str) const noexcept;
substrings (2)	int compare (size_type pos, size_type len, const basic_string& str) const;
	int compare (size_type pos, size_type len, const basic_string& str, size_type subpos, size_type sublen) const;
c-string (3)	int compare (const charT* s) const;
	int compare (size_type pos, size_type len, const charT* s) const;
buffer (4)	int compare (size_type pos, size_type len, const charT* s, size_type n) const;
string (1)	int compare (const basic_string& str) const noexcept;
substrings (2)	int compare (size_type pos, size_type len, const basic_string& str) const;
	int compare (size_type pos, size_type len, const basic_string& str, size_type subpos, size_type sublen = npos) const;
c-string (3)	int compare (const charT* s) const;
	int compare (size_type pos, size_type len, const charT* s) const;
buffer (4)	int compare (size_type pos, size_type len, const charT* s, size_type n) const;

### Compare strings

Compares the value of the `basic_string` object (or a substring) to the sequence of characters specified by its arguments.

The *compared string* is the value of the `basic_string` object or -if the signature used has a *pos* and a *len* parameters- the substring that begins at its character in position *pos* and spans *len* characters.

This string is compared to a *comparing string*, which is determined by the other arguments passed to the function.

The sequences are compared using `traits_type::compare`.

## Parameters

str

Another `basic_string` object of the same type (with the same class template arguments `charT`, `traits` and `Alloc`), used entirely (or partially) as the *comparing string*.

pos

Position of the first character in the *compared string*.

If this is greater than the `string length`, it throws `out_of_range`.

Note: The first character is denoted by a value of 0 (not 1).

len

Length of *compared string* (if the string is shorter, as many characters as possible).

A value of `basic_string::npos` indicates all characters until the end of the string.

subpos, sublen

Same as *pos* and *len* above, but for the *comparing string*.

s

Pointer to an array of characters.

If argument *n* is specified (4), the first *n* characters in the array are used as the *comparing string*.

Otherwise (3), a null-terminated sequence is expected: the length of the sequence with the characters to use as *comparing string* is determined by the first occurrence of a null character.

n

Number of characters to compare.

`charT` is `basic_string`'s character type (i.e., its first template parameter). Member type `size_type` is an unsigned integral type.

### Return Value

Returns a signed integral indicating the relation between the strings:

value	relation between compared string and comparing string
0	They compare equal
<0	Either the value of the first character that does not match is lower in the <i>compared string</i> , or all compared characters match but the <i>compared string</i> is shorter.
>0	Either the value of the first character that does not match is greater in the <i>compared string</i> , or all compared characters match but the <i>compared string</i> is longer.

### Example

```
1 // comparing apples with apples
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str1 ("green apple");
8     std::string str2 ("red apple");
9
10    if (str1.compare(str2) != 0)
11        std::cout << str1 << " is not " << str2 << '\n';
12
13    if (str1.compare(6,5,"apple") == 0)
14        std::cout << "still, " << str1 << " is an apple\n";
15
16    if (str2.compare(str2.size()-5,5,"apple") == 0)
17        std::cout << "and " << str2 << " is also an apple\n";
18
19    if (str1.compare(6,5,str2,4,5) == 0)
20        std::cout << "therefore, both are apples\n";
21
22    return 0;
23 }
```

Output:

```
green apple is not red apple
still, green apple is an apple
and red apple is also an apple
therefore, both are apples
```

### Complexity

Unspecified, but generally up to linear in both the *compared* and *comparing string*'s lengths.

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string` (except (1), which is guaranteed to not throw).

If `s` does not point to an array long enough, it causes *undefined behavior*.

If `pos` is greater than the `string` length, or if `subpos` is greater than `str`'s `length`, an `out_of_range` exception is thrown.

### See also

<code>basic_string::find</code>	Find first occurrence in string (public member function )
<code>basic_string::replace</code>	Replace portion of string (public member function )
<code>basic_string::substr</code>	Generate substring (public member function )
<code>relational operators (basic_string)</code>	Relational operators for <code>basic_string</code> (function template )

## /string/basic\_string/copy

public member function

### `std::basic_string::copy`

`<string>`

```
size_type copy (charT* s, size_type len, size_type pos = 0) const;
```

#### Copy sequence of characters from string

Copies a substring of the current value of the `basic_string` object into the array pointed by `s`. This substring contains the `len` characters that start at position `pos`.

The function does not append a null character at the end of the copied content.

## Parameters

---

s	Pointer to an array of characters. The array shall contain enough storage for the copied characters.
len	Number of characters to copy (if the string is shorter, as many characters as possible are copied).
pos	Position of the first character to be copied. If this is greater than the <a href="#">string length</a> , it throws <a href="#">out_of_range</a> . Note: The first character in the <a href="#">basic_string</a> is denoted by a value of 0 (not 1).

## Return value

---

The number of characters copied to the array pointed by s. This may be equal to len or to [length\(\) - pos](#) (if the string value is shorter than pos+len).

Member type [size\\_type](#) is an unsigned integral type.

## Example

---

```
1 // string::copy
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     char buffer[20];
8     std::string str ("Test string...");
9     std::size_t length = str.copy(buffer,6,5);
10    buffer[length]='\0';
11    std::cout << "buffer contains: " << buffer << '\n';
12    return 0;
13 }
```

Output:

```
buffer contains: string
```

## Complexity

---

Linear in the number of characters copied.

## Iterator validity

---

No changes.

## Data races

---

The object is accessed.

## Exception safety

---

**Strong guarantee:** if an exception is thrown, there are no changes in the [basic\\_string](#).

If s does not point to an array long enough, it causes *undefined behavior*.

If pos is greater than the [string length](#), an [out\\_of\\_range](#) exception is thrown.

## See also

---

<a href="#">basic_string::substr</a>	Generate substring ( <a href="#">public member function</a> )
<a href="#">basic_string::assign</a>	Assign content to string ( <a href="#">public member function</a> )
<a href="#">basic_string::c_str</a>	Get C-string equivalent
<a href="#">basic_string::replace</a>	Replace portion of string ( <a href="#">public member function</a> )
<a href="#">basic_string::insert</a>	Insert into string ( <a href="#">public member function</a> )
<a href="#">basic_string::append</a>	Append to string ( <a href="#">public member function</a> )

## /string/basic\_string/crbegin

public member function

### std::basic\_string::crbegin

<string>

```
const_reverse_iterator crbegin() const noexcept;
```

**Return const\_reverse\_iterator to reverse beginning**

Returns a const\_reverse\_iterator pointing to the last character of the string (i.e., its *reverse beginning*).

## Parameters

---

none

## Return Value

---

A `const_reverse_iterator` to the *reverse beginning* of the string.

Member type `const_reverse_iterator` is a reverse random access iterator type that points to a const character.

## Example

```
1 // string::crbegin/crend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("lorem ipsum");
8     for (auto rit=str.crbegin(); rit!=str.crend(); ++rit)
9         std::cout << *rit;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
muspi merol
```

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>basic_string::begin</code>	Return iterator to beginning (public member function )
<code>basic_string::crend</code>	Return <code>const_reverse_iterator</code> to reverse end (public member function )
<code>basic_string::rbegin</code>	Return reverse iterator to reverse beginning (public member function )

# /string/basic\_string/crend

public member function

## std::basic\_string::crend

`<string>`

`const_reverse_iterator crend() const noexcept;`

### Return const\_reverse\_iterator to reverse end

Returns a `const_reverse_iterator` pointing to the theoretical character preceding the first character of the string (which is considered its *reverse end*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse end* of the string.

Member type `const_reverse_iterator` is a reverse random access iterator type that points to a const character.

## Example

```
1 // string::crbegin/crend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("lorem ipsum");
8     for (auto rit=str.crbegin(); rit!=str.crend(); ++rit)
9         std::cout << *rit;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
muspi merol
```

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">basic_string::begin</a>	Return iterator to beginning (public member function )
<a href="#">basic_string::crend</a>	Return const_reverse_iterator to reverse end (public member function )
<a href="#">basic_string::rbegin</a>	Return reverse iterator to reverse beginning (public member function )

# /string/basic\_string/c\_str

## basic\_string::c\_str

```
const charT* c_str() const;
const charT* c_str() const noexcept;
```

### Get C-string equivalent

Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the [basic\\_string](#) object.

This array includes the same sequence of characters that make up the value of the [basic\\_string](#) object plus an additional terminating null-character (`charT()`) at the end.

A program shall not alter any of the characters in this sequence.

The pointer returned points to the internal array currently used by the [basic\\_string](#) object to store the characters that conform its value.

Both [basic\\_string::data](#) and [basic\\_string::c\\_str](#) are synonyms and return the same value.

The pointer returned may be invalidated by further calls to other member functions that modify the object.

## Parameters

none

## Return Value

A pointer to the c-string representation of the [basic\\_string](#) object's value.

`charT` is [basic\\_string](#)'s character type (i.e., its first template parameter).

## Example

```
1 // strings and c-strings
2 #include <iostream>
3 #include <cstring>
4 #include <string>
5
6 int main ()
7 {
8     std::string str ("Please split this sentence into tokens");
9
10    char * cstr = new char [str.length()+1];
11    std::strcpy (cstr, str.c_str());
12
13    // cstr now contains a c-string copy of str
14
15    char * p = std::strtok (cstr, " ");
16    while (p!=0)
17    {
18        std::cout << p << '\n';
19        p = strtok(NULL, " ");
20    }
21
22    delete[] cstr;
23    return 0;
24 }
```

Output:

```
Please
split
this
sentence
into
tokens
```

## **Complexity, iterator, access, exceptions**

Unspecified or contradictory specifications.

### **Complexity**

Constant.

### **Iterator validity**

No changes.

### **Data races**

The object is accessed.

### **Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

## **See also**

[\*\*basic\\_string::data\*\*](#) Get string data (public member function )

[\*\*basic\\_string::copy\*\*](#) Copy sequence of characters from string (public member function )

[\*\*basic\\_string::operator\[\]\*\*](#) Get character of string (public member function )

[\*\*basic\\_string::front\*\*](#) Access first character (public member function )

## **/string/basic\_string/data**

public member function

### **std::basic\_string::data**

<string>

```
const charT* data() const;
const charT* data() const noexcept;
```

#### **Get string data**

Returns a pointer to an array that contains the same sequence of characters as the characters that make up the value of the [basic\\_string](#) object.

Accessing the value at `data() + size()` produces *undefined behavior*: There are no guarantees that a null character terminates the character sequence pointed by the value returned by this function. See [basic\\_string::c\\_str](#) for a function that provides such guarantee.

A program shall not alter any of the characters in this sequence.

Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the [basic\\_string](#) object.

This array includes the same sequence of characters that make up the value of the [basic\\_string](#) object plus an additional terminating null-character (`charT()`) at the end.

The pointer returned points to the internal array currently used by the [basic\\_string](#) object to store the characters that conform its value.

Both [basic\\_string::data](#) and [basic\\_string::c\\_str](#) are synonyms and return the same value.

The pointer returned may be invalidated by further calls to other member functions that modify the object.

### **Parameters**

none

### **Return Value**

A pointer to the c-string representation of the [basic\\_string](#) object's value.

`charT` is [basic\\_string](#)'s character type (i.e., its first template parameter).

### **Example**

```
1 // string::data
2 #include <iostream>
3 #include <string>
4 #include <cstring>
5
6 int main ()
7 {
8     int length;
9
10    std::string str = "Test string";
11    char* cstr = "Test string";
12
13    if ( str.length() == std::strlen(cstr) )
14    {
15        std::cout << "str and cstr have the same length.\n";
16
17        if ( memcmp (cstr, str.data(), str.length() ) == 0 )
18            std::cout << "str and cstr have the same content.\n";
19    }
```

```
20 |     return 0;
21 }
```

Output:

```
str and cstr have the same length.
str and cstr have the same content.
```

### Complexity, iterator, access, exceptions

Unspecified or contradictory specifications.

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

### See also

<a href="#">basic_string::c_str</a>	Get C-string equivalent
<a href="#">basic_string::copy</a>	Copy sequence of characters from string ( <a href="#">public member function</a> )
<a href="#">basic_string::operator[]</a>	Get character of string ( <a href="#">public member function</a> )
<a href="#">basic_string::front</a>	Access first character ( <a href="#">public member function</a> )

## /string/basic\_string/empty

public member function

### std::basic\_string::empty

<string>

```
bool empty() const;
bool empty() const noexcept;
```

#### Test whether string is empty

Returns whether the [basic\\_string](#) is empty (i.e. whether its [length](#) is 0).

This function does not modify the value of the string in any way. To clear the content of a [basic\\_string](#), see [basic\\_string::clear](#).

### Parameters

none

### Return Value

true if the [string length](#) is 0, false otherwise.

### Example

```
1 // string::empty
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string content;
8     std::string line;
9     std::cout << "Please introduce a text. Enter an empty line to finish:\n";
10    do {
11        getline(std::cin,line);
12        content += line + '\n';
13    } while (!line.empty());
14    std::cout << "The text you introduced was:\n" << content;
15    return 0;
16 }
```

This program reads the user input line by line and stores it into string [content](#) until an empty line is introduced.

### Complexity

Unspecified, but generally constant.

Constant.

### Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">basic_string::clear</a>	Clear string (public member function )
<a href="#">basic_string::size</a>	Return size (public member function )
<a href="#">basic_string::length</a>	Return length of string (public member function )

# /string/basic\_string/end

public member function

## std::basic\_string::end

<string>

```
iterator end();
const_iterator end() const;
iterator end() noexcept;
const_iterator end() const noexcept;
```

### Return iterator to end

Returns an iterator pointing to the *past-the-end* character of the string.

The *past-the-end* character is a theoretical character that would follow the last character in the string. It shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with [basic\\_string::begin](#) to specify a range including all the characters in the string.

If the object is an empty string, this function returns the same as [basic\\_string::begin](#).

## Parameters

none

## Return Value

An iterator to the past-the-end of the string.

If the [basic\\_string](#) object is const-qualified, the function returns a [const\\_iterator](#). Otherwise, it returns an [iterator](#).

Member types [iterator](#) and [const\\_iterator](#) are [random access iterator](#) types (pointing to a character and to a const character, respectively).

## Example

```
1 // string::begin/end
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for ( std::string::iterator it=str.begin(); it!=str.end(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
Test string
```

## Complexity

Unspecified.

## Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

## Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

## Complexity

Unspecified, but generally constant.

### Iterator validity

No changes.

### Data races

The object is accessed (neither the const nor the non-const versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">basic_string::begin</a>	Return iterator to beginning (public member function )
<a href="#">basic_string::rbegin</a>	Return reverse iterator to reverse beginning (public member function )
<a href="#">basic_string::rend</a>	Return reverse iterator to reverse end (public member function )

## /string/basic\_string/erase

public member function

### std::basic\_string::erase

<string>

```
sequence (1) basic_string& erase (size_type pos = 0, size_type len = npos);
character (2)     iterator erase (iterator p);
range (3)         iterator erase (iterator first, iterator last);

sequence (1) basic_string& erase (size_type pos = 0, size_type len = npos);
character (2)     iterator erase (const_iterator p);
range (3)         iterator erase (const_iterator first, const_iterator last);
```

#### Erase characters from string

Erases part of the `basic_string`, reducing its length:

##### (1) sequence

Erases the portion of the string value that begins at the character position `pos` and spans `len` characters (or until the *end of the string*, if either the content is too short or if `len` is `basic_string::npos`).

Notice that the default argument erases all characters in the string (like member function `clear`).

##### (2) character

Erases the character pointed by `p`.

##### (3) range

Erases the sequence of characters in the range `[first, last]`.

### Parameters

`pos`

Position of the first character to be erased.

If this is greater than the `string length`, it throws `out_of_range`.

Note: The first character in `str` is denoted by a value of 0 (not 1).

`len`

Number of characters to erase (if the string is shorter, as many characters as possible are erased).

A value of `basic_string::npos` indicates all characters until the end of the string.

`p`

Iterator to the character to be removed.

`first, last`

Iterators specifying a range within the `basic_string`] to be removed: `[first, last]`. i.e., the range includes all the characters between `first` and `last`, including the character pointed by `first` but not the one pointed by `last`.

Member type `size_type` is an unsigned integral type.

Member types `iterator` and `const_iterator` are `random access iterator` types that point to characters of the `basic_string`.

### Return value

The *sequence version (1)* returns `*this`.

The others return an iterator referring to the character that now occupies the position of the first character erased, or `basic_string::end` if no such character exists.

Member type `iterator` is a `random access iterator` type that points to characters of the `basic_string`.

### Example

```
1 // string::erase
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7 }
```

```

8 std::string str ("This is an example sentence.");
9 std::cout << str << '\n';
10 str.erase (10,8); // "This is an example sentence."
11 // ^^^^^^
12 std::cout << str << '\n';
13 str.erase (str.begin() + 9); // "This is an sentence."
14 // ^
15 std::cout << str << '\n';
16 str.erase (str.begin() + 5, str.end() - 9); // "This is a sentence."
17 // ^^^^
18 std::cout << str << '\n';
19 // "This sentence."
20 return 0;
}

```

Output:

```

This is an example sentence.
This is an sentence.
This is a sentence.
This sentence.

```

## Complexity

Unspecified, but generally up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If `pos` is greater than the `string` length, an `out_of_range` exception is thrown.

An invalid `p` in (2), or an invalid range in (3), causes *undefined behavior*.

For (1) and (3), if an exception is thrown, there are no changes in the `basic_string` (strong guarantee).  
For (2), it never throws exceptions (no-throw guarantee).

If `pos` is greater than the `string` length, an `out_of_range` exception is thrown.

An invalid range in (3), causes *undefined behavior*.

## See also

<code>basic_string::clear</code>	Clear string (public member function)
<code>basic_string::replace</code>	Replace portion of string (public member function )
<code>basic_string::insert</code>	Insert into string (public member function )
<code>basic_string::assign</code>	Assign content to string (public member function )
<code>basic_string::append</code>	Append to string (public member function )

## /string/basic\_string/find

public member function

### std::basic\_string::find

<string>

```

string (1) size_type find (const basic_string& str, size_type pos = 0) const;
c-string (2) size_type find (const charT* s, size_type pos = 0) const;
buffer (3) size_type find (const charT* s, size_type pos, size_type n) const;
character (4) size_type find (charT c, size_type pos = 0) const;

string (1) size_type find (const basic_string& str, size_type pos = 0) const noexcept;
c-string (2) size_type find (const charT* s, size_type pos = 0) const;
buffer (3) size_type find (const charT* s, size_type pos, size_type n) const;
character (4) size_type find (charT c, size_type pos = 0) const noexcept;

```

#### Find first occurrence in string

Searches the `basic_string` for the first occurrence of the sequence specified by its arguments.

When `pos` is specified, the search only includes characters at or after position `pos`, ignoring any possible occurrences that include characters before `pos`.

The function uses `traits_type::eq` to determine character equivalences.

Notice that unlike member `find_first_of`, whenever more than one character is being searched for, it is not enough that just one of these characters match, but the entire sequence must match.

## Parameters

`str`

Another `basic_string` with the subject to search for.

<b>pos</b>	Position of the first character in the string to be considered in the search. If this is greater than the <a href="#">string length</a> , the function never finds matches. Note: The first character is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.
<b>s</b>	Pointer to an array of characters. If argument <i>n</i> is specified (3), the sequence to match are the first <i>n</i> characters in the array. Otherwise (2), a null-terminated sequence is expected: the length of the sequence to match is determined by the first occurrence of a null character.
<b>n</b>	Length of sequence of characters to match.
<b>c</b>	Individual character to be searched for.

`charT` is [basic\\_string](#)'s character type (i.e., its first template parameter).  
Member type `size_type` is an unsigned integral type.

## Return Value

The position of the first character of the first match.  
If no matches were found, the function returns [basic\\_string::npos](#).

Member type `size_type` is an unsigned integral type.

## Example

```

1 // string::find
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("There are two needles in this haystack with needles.");
8     std::string str2 ("needle");
9
10    // different member versions of find in the same order as above:
11    std::string::size_type found = str.find(str2);
12    if (found!=std::string::npos)
13        std::cout << "first 'needle' found at: " << found << '\n';
14
15    found=str.find("needles are small",found+1,6);
16    if (found!=std::string::npos)
17        std::cout << "second 'needle' found at: " << found << '\n';
18
19    found=str.find("haystack");
20    if (found!=std::string::npos)
21        std::cout << "'haystack' also found at: " << found << '\n';
22
23    found=str.find('.');
24    if (found!=std::string::npos)
25        std::cout << "Period found at: " << found << '\n';
26
27    // let's replace the first needle:
28    str.replace(str.find(str2),str2.length(),"preposition");
29    std::cout << str << '\n';
30
31    return 0;
32 }
```

Notice how parameter `pos` is used to search for a second instance of the same search string. Output:

```

first 'needle' found at: 14
second 'needle' found at: 44
'haystack' also found at: 30
Period found at: 51
There are two prepositions in this haystack with needles.
```

## Complexity

Unspecified, but generally up to linear in `length()`-`pos` times the length of the sequence to match (worst case).

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If `s` does not point to an array long enough, it causes *undefined behavior*.  
Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<a href="#">basic_string::rfind</a>	Find last occurrence in string (public member function )
-------------------------------------	--

<a href="#">basic_string::find_first_of</a>	Find character in string (public member function )
---	--

<a href="#">basic_string::find_last_of</a>	Find character in string from the end (public member function )
--	---

<a href="#">basic_string::find_first_not_of</a>	Find non-matching character in string (public member function )
---	---

**basic\_string::find\_last\_not\_of** Find non-matching character in string from the end (public member function )

**basic\_string::replace** Replace portion of string (public member function )

**basic\_string::substr** Generate substring (public member function )

## /string/basic\_string/find\_first\_not\_of

public member function

### std::basic\_string::find\_first\_not\_of

<string>

```
string(1) size_type find_first_not_of (const basic_string& str, size_type pos = 0) const;
c-string(2) size_type find_first_not_of (const charT* s, size_type pos = 0) const;
buffer(3) size_type find_first_not_of (const charT* s, size_type pos, size_type n) const;
character(4) size_type find_first_not_of (charT c, size_type pos = 0) const;

string(1) size_type find_first_not_of (const basic_string& str, size_type pos = 0) const noexcept;
c-string(2) size_type find_first_not_of (const charT* s, size_type pos = 0) const;
buffer(3) size_type find_first_not_of (const charT* s, size_type pos, size_type n) const;
character(4) size_type find_first_not_of (charT c, size_type pos = 0) const noexcept;
```

#### Find non-matching character in string

Searches the [basic\\_string](#) for the first character that does not match any of the characters specified in its arguments.

When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences before that character.

The function uses [traits\\_type::eq](#) to determine character equivalences.

#### Parameters

str	Another <a href="#">basic_string</a> with the set of characters to be used in the search.
pos	Position of the first character in the string to be considered in the search. If this is greater than the <a href="#">string length</a> , the function never finds matches. Note: The first character is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.
s	Pointer to an array of characters. If argument <i>n</i> is specified (3), the first <i>n</i> characters in the array are used in the search. Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters used in the search is determined by the first occurrence of a null character.
n	Number of character values to search for.
c	Individual character to be searched for.

charT is [basic\\_string](#)'s character type (i.e., its first template parameter).

Member type [size\\_type](#) is an unsigned integral type.

#### Return Value

The position of the first character that does not match.

If no such characters are found, the function returns [basic\\_string::npos](#).

Member type [size\\_type](#) is an unsigned integral type.

#### Example

```
1 // string::find_first_not_of
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("look for non-alphabetic characters...");
8
9     std::string::size_type found = str.find_first_not_of("abcdefghijklmnopqrstuvwxyz ");
10
11    if (found!=std::string::npos)
12    {
13        std::cout << "The first non-alphabetic character is " << str[found];
14        std::cout << " at position " << found << '\n';
15    }
16
17    return 0;
18 }
```

The first non-alphabetic character is - at position 12

#### Complexity

Unspecified, but generally up to linear in [length\(\)](#)-*pos* times the number of characters to match (worst case).

#### Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If *s* does not point to an array long enough, it causes *undefined behavior*. Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<a href="#">basic_string::find</a>	Find first occurrence in string (public member function )
<a href="#">basic_string::find_first_of</a>	Find character in string (public member function )
<a href="#">basic_string::find_last_not_of</a>	Find non-matching character in string from the end (public member function )
<a href="#">basic_string::replace</a>	Replace portion of string (public member function )
<a href="#">basic_string::substr</a>	Generate substring (public member function )

## /string/basic\_string/find\_first\_of

public member function

### std::basic\_string::find\_first\_of

<string>

```
string (1) size_type find_first_of (const basic_string& str, size_type pos = 0) const;
c-string (2) size_type find_first_of (const charT* s, size_type pos = 0) const;
buffer (3) size_type find_first_of (const charT* s, size_type pos, size_type n) const;
character (4) size_type find_first_of (charT c, size_type pos = 0) const;

string (1) size_type find_first_of (const basic_string& str, size_type pos = 0) const noexcept;
c-string (2) size_type find_first_of (const charT* s, size_type pos = 0) const;
buffer (3) size_type find_first_of (const charT* s, size_type pos, size_type n) const;
character (4) size_type find_first_of (charT c, size_type pos = 0) const noexcept;
```

#### Find character in string

Searches the [basic\\_string](#) for the first character that matches **any** of the characters specified in its arguments.

When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences before *pos*.

Notice that it is enough for one single character of the sequence to match (not all of them). See [basic\\_string::find](#) for a function that matches entire sequences.

The function uses [traits\\_type::eq](#) to determine character equivalences.

## Parameters

<b>str</b>	Another <a href="#">basic_string</a> with the characters to search for.
<b>pos</b>	Position of the first character in the string to be considered in the search. If this is greater than the <a href="#">string length</a> , the function never finds matches. Note: The first character is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.
<b>s</b>	Pointer to an array of characters. If argument <i>n</i> is specified (3), the first <i>n</i> characters in the array are searched for. Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters to match is determined by the first occurrence of a null character.
<b>n</b>	Number of character values to search for.
<b>c</b>	Individual character to be searched for.

*charT* is [basic\\_string](#)'s character type (i.e., its first template parameter).

Member type *size\_type* is an unsigned integral type.

## Return Value

The position of the first character that matches.

If no matches are found, the function returns [basic\\_string::npos](#).

Member type *size\_type* is an unsigned integral type.

## Example

```
1 // string::find_first_of
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Please, replace the vowels in this sentence by asterisks.");
8     std::string::size_type found = str.find_first_of("aeiou");
9     while (found!=std::string::npos)
10    {
```

```

11     str[found]='*';
12     found=str.find_first_of("aeiou",found+1);
13 }
14
15 std::cout << str << '\n';
16
17 return 0;
18 }
```

Pl\*\*s\*, r\*pl\*c\* th\* v\*w\*ls \*n th\*s s\*nt\*nc\* by \*st\*r\*sks.

## Complexity

Unspecified, but generally up to linear in `length()`-pos times the number of characters to match (worst case).

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If `s` does not point to an array long enough, it causes *undefined behavior*. Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<code>basic_string::find</code>	Find first occurrence in string (public member function )
<code>basic_string::find_last_of</code>	Find character in string from the end (public member function )
<code>basic_string::find_first_not_of</code>	Find non-matching character in string (public member function )
<code>basic_string::replace</code>	Replace portion of string (public member function )
<code>basic_string::substr</code>	Generate substring (public member function )

## /string/basic\_string/find\_last\_not\_of

public member function

### std::basic\_string::find\_last\_not\_of

<string>

```

string(1) size_type find_last_not_of (const basic_string& str, size_type pos = npos) const;
c-string(2) size_type find_last_not_of (const charT* s, size_type pos = npos) const;
buffer(3) size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
character(4) size_type find_last_not_of (charT c, size_type pos = npos) const;

string(1) size_type find_last_not_of (const basic_string& str, size_type pos = npos) const noexcept;
c-string(2) size_type find_last_not_of (const charT* s, size_type pos = npos) const;
buffer(3) size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
character(4) size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;
```

#### Find non-matching character in string from the end

Searches the `basic_string` for the last character that does not match any of the characters specified in its arguments.

When `pos` is specified, the search only includes characters at or before position `pos`, ignoring any possible occurrences after `pos`.

The function uses `traits_type::eq` to determine character equivalences.

## Parameters

`str` Another `basic_string` with the set of characters to be used in the search.

`pos` Position of the last character in the string to be considered in the search.  
Any value greater or equal than the `string length` (including `basic_string::npos`) means that the entire string is searched.  
Note: The first character is denoted by a value of 0 (not 1).

`s` Pointer to an array of characters.  
If argument `n` is specified (3), the first `n` characters in the array are used in the search.  
Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters used in the search is determined by the first occurrence of a null character.

`n` Number of character values to search for.

`c` Individual character to be searched for.

`charT` is `basic_string`'s character type (i.e., its first template parameter).  
Member type `size_type` is an unsigned integral type.

## Return Value

The position of the first character that does not match.  
If no such characters are found, the function returns `basic_string::npos`.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // string::find_last_not_of
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Please, erase trailing white-spaces  \n");
8     std::string whitespaces (" \t\f\v\n\r");
9
10    std::string::size_type found = str.find_last_not_of(whitespaces);
11    if (found!=std::string::npos)
12        str.erase(found+1);
13    else
14        str.clear();           // str is all whitespace
15
16    std::cout << '[' << str << "]\n";
17
18    return 0;
19 }
```

[Please, erase trailing white-spaces]

## Complexity

Unspecified, but generally up to linear in the `string length` (or `pos`) times the number of characters to match (worst case).

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If `s` does not point to an array long enough, it causes *undefined behavior*.  
Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<code>basic_string::find</code>	Find first occurrence in string (public member function )
<code>basic_string::find_last_of</code>	Find character in string from the end (public member function )
<code>basic_string::find_first_not_of</code>	Find non-matching character in string (public member function )
<code>basic_string::replace</code>	Replace portion of string (public member function )
<code>basic_string::substr</code>	Generate substring (public member function )

# /string/basic\_string/find\_last\_of

public member function

## std::basic\_string::find\_last\_of

`<string>`

```
string (1) size_type find_last_of (const basic_string& str, size_type pos = npos) const;
c-string (2) size_type find_last_of (const charT* s, size_type pos = npos) const;
buffer (3) size_type find_last_of (const charT* s, size_type pos, size_type n) const;
character (4) size_type find_last_of (charT c, size_type pos = npos) const;

string (1) size_type find_last_of (const basic_string& str, size_type pos = npos) const noexcept;
c-string (2) size_type find_last_of (const charT* s, size_type pos = npos) const;
buffer (3) size_type find_last_of (const charT* s, size_type pos, size_type n) const;
character (4) size_type find_last_of (charT c, size_type pos = npos) const noexcept;
```

### Find character in string from the end

Searches the `basic_string` for the last character that matches **any** of the characters specified in its arguments.

When `pos` is specified, the search only includes characters at or before position `pos`, ignoring any possible occurrences after `pos`.

The function uses `traits_type::eq` to determine character equivalences.

## Parameters

`str`

Another `basic_string` with the characters to search for.

`pos`

Position of the last character in the string to be considered in the search.

Any value greater or equal than the `string length` (including `basic_string::npos`) means that the entire string is searched.

Note: The first character is denoted by a value of 0 (not 1).

s	Pointer to an array of characters. If argument <i>n</i> is specified (3), the first <i>n</i> characters in the array are searched for. Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters to match is determined by the first occurrence of a null character.
n	Number of character values to search for.
c	Individual character to be searched for.

`charT` is `basic_string`'s character type (i.e., its first template parameter).  
Member type `size_type` is an unsigned integral type.

### Return Value

The position of the last character that matches.  
If no matches are found, the function returns `basic_string::npos`.

Member type `size_type` is an unsigned integral type.

### Example

```

1 // string::find_last_of
2 #include <iostream>
3 #include <string>
4
5 void SplitFilename (const std::string& str)
6 {
7     std::cout << "Splitting: " << str << '\n';
8     std::string::size_type found = str.find_last_of("//");
9     std::cout << " path: " << str.substr(0,found) << '\n';
10    std::cout << " file: " << str.substr(found+1) << '\n';
11 }
12
13 int main ()
14 {
15     std::string str1 ("/usr/bin/man");
16     std::string str2 ("c:\\windows\\winhelp.exe");
17
18     SplitFilename (str1);
19     SplitFilename (str2);
20
21     return 0;
22 }
```

```

Splitting: /usr/bin/man
path: /usr/bin
file: man
Splitting: c:\\windows\\winhelp.exe
path: c:\\windows
file: winhelp.exe
```

### Complexity

Unspecified, but generally up to linear in the string length (or *pos*) times the number of characters to match (worst case).

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

If *s* does not point to an array long enough, it causes *undefined behavior*.  
Otherwise, the function never throws exceptions (no-throw guarantee).

### See also

<a href="#">basic_string::find</a>	Find first occurrence in string (public member function )
<a href="#">basic_string::rfind</a>	Find last occurrence in string (public member function )
<a href="#">basic_string::find_first_of</a>	Find character in string (public member function )
<a href="#">basic_string::find_last_not_of</a>	Find non-matching character in string from the end (public member function )
<a href="#">basic_string::replace</a>	Replace portion of string (public member function )
<a href="#">basic_string::substr</a>	Generate substring (public member function )

## /string/basic\_string/front

public member function

**std::basic\_string::front**

<string>

```
    charT& front();
const charT& front() const;
```

### Access first character

Returns a reference to the first character of the [basic\\_string](#).

Unlike member [basic\\_string::begin](#), which returns an iterator to this same character, this function returns a direct reference.

This function shall not be called on [empty strings](#).

### Parameters

none

### Return value

A reference to the first character in the [basic\\_string](#).

If the [basic\\_string](#) object is const-qualified, the function returns a `const charT&`. Otherwise, it returns a `charT&`.

`charT` is [basic\\_string](#)'s character type (i.e., its first template parameter).

### Example

```
1 // string::front
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("test string");
8     str.front() = 'T';
9     std::cout << str << '\n';
10    return 0;
11 }
```

Output:

```
Test string
```

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

### Exception safety

If the [basic\\_string](#) is not [empty](#), the function never throws exceptions (no-throw guarantee).

Otherwise, it causes [undefined behavior](#).

### See also

<a href="#">basic_string::back</a>	Access last character ( <a href="#">public member function</a> )
<a href="#">basic_string::at</a>	Get character of string ( <a href="#">public member function</a> )
<a href="#">basic_string::operator[]</a>	Get character of string ( <a href="#">public member function</a> )

## /string/basic\_string/get\_allocator

public member function

### std::basic\_string::get\_allocator

`<string>`

```
allocator_type get_allocator() const;
allocator_type get_allocator() const noexcept;
```

### Get allocator

Returns a copy of the allocator object associated with the [basic\\_string](#).

### Parameters

none

### Return Value

The allocator.

Member type `allocator_type` is the type of the allocator used by the container, defined in [basic\\_string](#) as an alias of its third template parameter (`Alloc`).

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

Copying any instantiation of the [default allocator](#) is also guaranteed to never throw.

## See also

[allocator](#) | Default allocator (class template)

# /string/basic\_string/getline

function template

## std::getline (basic\_string)

<string>

```
template <class charT, class traits, class Alloc>
(1)   basic_istream<charT,traits>& getline (basic_istream<charT,traits>& is,
                                basic_string<charT,traits,Alloc>& str, charT delim);
template <class charT, class traits, class Alloc>
(2)   basic_istream<charT,traits>& getline (basic_istream<charT,traits>& is,
                                basic_string<charT,traits,Alloc>& str);
```

```
template <class charT, class traits, class Alloc>
basic_istream<charT,traits>& getline (basic_istream<charT,traits>& is,
                                         basic_string<charT,traits,Alloc>& str, charT delim);
(1) template <class charT, class traits, class Alloc>
basic_istream<charT,traits>& getline (basic_istream<charT,traits>&& is,
                                         basic_string<charT,traits,Alloc>& str, charT delim);
template <class charT, class traits, class Alloc>
basic_istream<charT,traits>& getline (basic_istream<charT,traits>& is,
                                         basic_string<charT,traits,Alloc>& str);
(2) template <class charT, class traits, class Alloc>
basic_istream<charT,traits>& getline (basic_istream<charT,traits>&& is,
                                         basic_string<charT,traits,Alloc>& str);
```

### Get line from stream into string

Extracts characters from *is* and stores them into *str* until the delimitation character *delim* is found (or the newline character, for (2)).

The extraction also stops if the end of file is reached in *is* or if some other error occurs during the input operation.

If the delimiter is found, it is extracted and discarded (i.e. it is not stored and the next input operation will begin after it).

Note that any content in *str* before the call is replaced by the newly extracted sequence.

Each extracted character is appended to the [basic\\_string](#) as if its member [push\\_back](#) was called.

## Parameters

*is*      [basic\\_istream](#) object from which characters are extracted.

*str*      [basic\\_string](#) object where the extracted line is stored.

## Return Value

The same as parameter *is*.

A call to this function may set any of the internal state flags of *is* if:

flag	error
eofbit	The end of the source of characters is reached during its operations.
failbit	The input obtained could not be interpreted as a valid textual representation of an object of this type. In this case, <i>distr</i> preserves the parameters and internal data it had before the call. Notice that some <i>eofbit</i> cases will also set <i>failbit</i> .
badbit	An error other than the above happened.

(see [ios\\_base::iostate](#) for more info on these)

Additionally, in any of these cases, if the appropriate flag has been set with *is*'s member function [basic\\_ios::exceptions](#), an exception of type [ios\\_base::failure](#) is thrown.

## Example

```
1 // extract to string
2 #include <iostream>
3 #include <string>
4
```

```

5 main ()
6 {
7     std::string name;
8
9     std::cout << "Please, enter your full name: ";
10    std::getline (std::cin, name);
11    std::cout << "Hello, " << name << "!\n";
12
13    return 0;
14 }

```

## Complexity

Unspecified, but generally linear in the resulting *length* of *str*.

## Iterator validity

Any iterators, pointers and references related to *str* may be invalidated.

## Data races

Both objects, *is* and *str*, are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, both *is* and *str* end up in a valid state.

## See also

<b>basic_istream::getline</b>	Get line (public member function )
<b>operator&gt;&gt; (basic_string)</b>	Extract string from stream (function template )

# /string/basic\_string/insert

public member function

## std::basic\_string::insert

<string>

```

string (1) basic_string& insert (size_type pos, const basic_string& str);
substring (2) basic_string& insert (size_type pos, const basic_string& str,
                                   size_type subpos, size_type sublen);
c-string (3) basic_string& insert (size_type pos, const charT* s);
buffer (4) basic_string& insert (size_type pos, const charT* s, size_type n);
fill (5) basic_string& insert (size_type pos, size_type n, charT c);
           void insert (iterator p, size_type n, charT c);
single character (6) iterator insert (iterator p, charT c);
range (7) template <class InputIterator>
           void insert (iterator p, InputIterator first, InputIterator last);

string (1) basic_string& insert (size_type pos, const basic_string& str);
substring (2) basic_string& insert (size_type pos, const basic_string& str,
                                   size_type subpos, size_type sublen);
c-string (3) basic_string& insert (size_type pos, const charT* s);
buffer (4) basic_string& insert (size_type pos, const charT* s, size_type n);
fill (5) basic_string& insert (size_type pos, size_type n, charT c);
           iterator insert (const_iterator p, size_type n, charT c);
single character (6) iterator insert (const_iterator p, charT c);
range (7) template <class InputIterator>
           iterator insert (iterator p, InputIterator first, InputIterator last);
initializer list (8) basic_string& insert (const_iterator p, initializer_list<charT> il);

string (1) basic_string& insert (size_type pos, const basic_string& str);
substring (2) basic_string& insert (size_type pos, const basic_string& str,
                                   size_type subpos, size_type sublen =npos);
c-string (3) basic_string& insert (size_type pos, const charT* s);
buffer (4) basic_string& insert (size_type pos, const charT* s, size_type n);
fill (5) basic_string& insert (size_type pos, size_type n, charT c);
           iterator insert (const_iterator p, size_type n, charT c);
single character (6) iterator insert (const_iterator p, charT c);
range (7) template <class InputIterator>
           iterator insert (iterator p, InputIterator first, InputIterator last);
initializer list (8) basic_string& insert (const_iterator p, initializer_list<charT> il);

```

## Insert into string

Inserts additional characters into the `basic_string` right before the character indicated by *pos* (or *p*):

### (1) string

Inserts a copy of *str*.

### (2) substring

Inserts a copy of a substring of *str*. The substring is the portion of *str* that begins at the character position *subpos* and spans *sublen* characters (or until the end of *str*, if either *str* is too short or if *sublen* is *npos*).

### (3) c-string

Inserts a copy of the string formed by the null-terminated character sequence (C-string) pointed by *s*. The length of this character sequence is determined by calling `traits_type::length(s)`.

### (4) buffer

Inserts a copy of the first *n* characters in the array of characters pointed by *s*.

**(5) *fill***

Inserts *n* consecutive copies of character *c*.

**(6) *single character***

Inserts character *c*.

**(7) *range***

Inserts a copy of the sequence of characters in the range [*first*,*last*), in the same order.

**(8) *initializer list***

Inserts a copy of each of the characters in *il*, in the same order.

---

## Parameters

*pos*

Insertion point: The new contents are inserted before the character at position *pos*.

If this is greater than the object's `length`, it throws `out_of_range`.

Note: The first character is denoted by a value of 0 (not 1).

*str*

Another `basic_string` object of the same type (with the same class template arguments `charT`, `traits` and `Alloc`).

*subpos*

Position of the first character in *str* that is inserted into the object as a substring.

If this is greater than *str*'s `length`, it throws `out_of_range`.

Note: The first character in *str* is denoted by a value of 0 (not 1).

*sublen*

Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).

A value of `npos` indicates all characters until the end of *str*.

*s*

Pointer to an array of characters (such as a *c-string*).

*n*

Number of characters to insert.

*c*

Character value.

*p*

Iterator pointing to the insertion point: The new contents are inserted before the character pointed by *p*.

*iterator* is a member type, defined as a `random access iterator` type that points to characters of the `basic_string`.

*first*, *last*

*Input iterators* to the initial and final positions in a range. The range used is [*first*,*last*), which includes all the characters between *first* and *last*, including the character pointed by *first* but not the character pointed by *last*.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type convertible to `charT`.

*il*

An `initializer_list` object.

These objects are automatically constructed from `initializer list` declarators.

`charT` is `basic_string`'s character type (i.e., its first template parameter).

Member type `size_type` is an unsigned integral type.

---

## Return value

The signatures returning a reference to `basic_string`, return `*this`.

Those returning an iterator, return an iterator pointing to the first character inserted.

Member type `iterator` is a `random access iterator` type that points to characters of the `basic_string`.

---

## Example

```
1 // inserting into a string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str="to be question";
8     std::string str2="the ";
9     std::string str3="or not to be";
10    std::string::iterator it;
11
12    // used in the same order as described above:
13    str.insert(6,str2);           // to be (the )question
14    str.insert(6,str3,3,4);      // to be (not )the question
15    str.insert(10,"that is cool",8); // to be not (that is )the question
16    str.insert(10,"to be ");      // to be not (to be )that is the question
17    str.insert(15,1,':');        // to be not to be(:) that is the question
18    it = str.insert(str.begin()+5,','); // to be(,) not to be: that is the question
19    str.insert (str.end(),3,'.');// to be, not to be: that is the question(...
20    str.insert (it+2,str3.begin(),str3.begin()+3); // (or )
21
22    std::cout << str << '\n';
23    return 0;
24 }
```

Output:

```
to be, or not to be: that is the question...
```

---

## Complexity

Unspecified, but generally up to linear in the new `string` length.

## **Iterator validity**

Any iterators, pointers and references related to this object may be invalidated.

## **Data races**

The object is modified.

## **Exception safety**

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If `s` does not point to an array long enough, or if either `p` or the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

If `pos` is greater than the `string` length, or if `subpos` is greater than `str`'s `length`, an `out_of_range` exception is thrown.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## **See also**

<code>basic_string::append</code>	Append to string (public member function )
<code>basic_string::replace</code>	Replace portion of string (public member function )
<code>basic_string::substr</code>	Generate substring (public member function )
<code>basic_string::operator=</code>	String assignment (public member function )
<code>basic_string::operator+=</code>	Append to string (public member function )

# /string/basic\_string/length

public member function

## **std::basic\_string::length**

`<string>`

```
size_type length() const;
size_type length() const noexcept;
```

### **Return length of string**

Returns the length of the string, in terms of number of characters.

This is the number of actual characters that conform the contents of the `basic_string`, which is not necessarily equal to its storage capacity.

Both `basic_string::size` and `basic_string::length` are synonyms and return the same value.

## **Parameters**

none

## **Return Value**

The number of characters in the string.

Member type `size_type` is an unsigned integral type.

## **Example**

```
1 // string::length
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "The size of str is " << str.length() << " characters.\n";
9     return 0;
10 }
```

Output:

```
The size of str is 11 characters
```

## **Complexity**

Unspecified.

Constant.

## **Iterator validity**

No changes.

## **Data races**

The object is accessed.

## **Exception safety**

---

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">basic_string::size</a>	Return size (public member function )
<a href="#">basic_string::resize</a>	Resize string (public member function )
<a href="#">basic_string::max_size</a>	Return maximum size (public member function )
<a href="#">basic_string::capacity</a>	Return size of allocated storage (public member function )

## /string/basic\_string/max\_size

public member function

### std::basic\_string::max\_size

<string>

```
size_type max_size() const;
size_type max_size() const noexcept;
```

#### Return maximum size

Returns the maximum length the `basic_string` can reach.

This is the maximum potential `length` the string can reach due to known system or library implementation limitations, but the object is not guaranteed to be able to reach that length: it can still fail to allocate storage at any point before that length is reached.

## Parameters

none

## Return Value

The maximum length the `basic_string` can reach.

Member type `size_type` is an unsigned integral type.

## Example

```
1 // comparing size, length, capacity and max_size
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "size: " << str.size() << "\n";
9     std::cout << "length: " << str.length() << "\n";
10    std::cout << "capacity: " << str.capacity() << "\n";
11    std::cout << "max_size: " << str.max_size() << "\n";
12    return 0;
13 }
```

A possible output for this program could be:

```
size: 11
length: 11
capacity: 15
max_size: 4294967291
```

## Complexity

Unspecified, but generally constant.

Constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">basic_string::capacity</a>	Return size of allocated storage (public member function )
<a href="#">basic_string::size</a>	Return size (public member function )
<a href="#">basic_string::resize</a>	Resize string (public member function )

## /string/basic\_string/npos

```
public static member constant  
std::basic_string::npos <string>  
static const size_type npos = -1;
```

#### Maximum value of size\_type

npos is a static member constant value with the greatest possible value for an element of member type size\_type.

This value, when used as the value for a *len* (or *sublen*) parameter in `basic_string`'s member functions, means "until the end of the string".

As a return value, it is usually used to indicate no matches.

This constant is defined with a value of -1, which because member type size\_type is an unsigned integral type, it is the largest possible representable value for this type.

## /string/basic\_string/operator<<

```
function template  
std::operator<< (basic_string) <string>  
template <class charT, class traits, class Alloc>  
    basic_ostream<charT,traits>& operator<< (basic_ostream<charT,traits>& os,  
                                                const basic_string<charT,traits,Alloc>& str);
```

#### Insert string into stream

Inserts the sequence of characters that conforms value of *str* into *os*.

This function overloads `operator<<` to behave as described in `basic_ostream::operator<<` for c-strings, but applied to `basic_string` objects.

#### Parameters

*os*      `basic_ostream` object where characters are inserted.

*str*      `basic_string` object with the content to insert.

#### Return Value

The same as parameter *os*.

If some error happens during the output operation, the stream's *badbit* flag is set, and if the appropriate flag has been set with `basic_ios::exceptions`, an exception is thrown.

#### Example

```
1 // inserting strings into output streams  
2 #include <iostream>  
3 #include <string>  
4  
5 main ()  
6 {  
7     std::string str = "Hello world!";  
8     std::cout << str << '\n';  
9     return 0;  
10 }
```

#### Complexity

Unspecified, but generally linear in *str*'s length.

#### Iterator validity

No changes.

#### Data races

Objects *os* is modified.

#### Exception safety

**Basic guarantee:** if an exception is thrown, both *is* and *str* end up in a valid state.

#### See also

<code>ostream::operator&lt;&lt;</code>	Insert formatted output (public member function )
<code>operator&gt;&gt; (basic_string)</code>	Extract string from stream (function template )

## /string/basic\_string/operator=

public member function

**std::basic\_string::operator=** <string>

```

string (1) basic_string& operator= (const basic_string& str);
c-string (2) basic_string& operator= (const charT* s);
character (3) basic_string& operator= (charT c);

string (1) basic_string& operator= (const basic_string& str);
c-string (2) basic_string& operator= (const charT* s);
character (3) basic_string& operator= (charT c);
initializer list (4) basic_string& operator= (initializer_list<charT> il);
move (5) basic_string& operator= (basic_string&& str) noexcept;

```

## String assignment

Assigns a new value to the string, replacing its current contents.

(See member function [assign](#) for additional assignment options).

## Parameters

`str`

A `basic_string` object of the same type (with the same class template arguments `charT`, `traits` and `Alloc`), whose value is either copied (1) or moved (5) if different from `*this` (if moved, `str` is left in an unspecified but valid state).

`s`

Pointer to a null-terminated sequence of characters.  
The sequence is copied as the new value for the string.  
The `length` is determined by calling `traits_type::length(s)`.

`c`

A character.  
The string value is set to a single copy of this character (the string length becomes 1).

`il`

An `initializer_list` object.  
These objects are automatically constructed from `initializer list` declarators.  
The characters are copied, in the same order.

`charT` is `basic_string`'s character type (i.e., its first template parameter).

## Return Value

`*this`

## Example

```

1 // string assigning
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str1, str2, str3;
8     str1 = "Test string: ";    // c-string
9     str2 = 'x';                // single character
10    str3 = str1 + str2;        // string
11
12    std::cout << str3 << '\n';
13    return 0;
14 }

```

Output:

Test string: x

## Complexity

Unspecified.

Unspecified, but generally linear in the new string length (and constant for the move version).

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

The move assignment (5) modifies `str`.

## Exception safety

For the move assignment (5), the function does not throw exceptions (no-throw guarantee).  
In all other cases, there are no effects in case an exception is thrown (strong guarantee).

If the resulting string length would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

[basic\\_string::assign](#) Assign content to string (public member function )

[basic\\_string::operator+=](#) Append to string (public member function )

<b>basic_string::insert</b>	Insert into string (public member function )
<b>basic_string::replace</b>	Replace portion of string (public member function )
<b>basic_string::basic_string</b>	Construct basic_string object (public member function )
<b>basic_string::compare</b>	Compare strings (public member function )

## /string/basic\_string/operator>>

function template

### std::operator>> (basic\_string)

<string>

```
template <class charT, class traits, class Alloc>
basic_istream<charT,traits>& operator>> (basic_istream<charT,traits>& is,
                                                basic_string<charT,traits,Alloc>& str);
```

#### Extract string from stream

Extracts a string from the input stream *is*, storing the sequence in *str*, which is overwritten (the previous value of *str* is replaced).

This function overloads `operator>>` to behave as described in `basic_istream::operator>>` for c-strings, but applied to `basic_string` objects.

Each extracted character is appended to the `basic_string` as if its member `push_back` was called.

Notice that the `basic_istream` extraction operations use whitespaces as separators; Therefore, this operation will only extract what can be considered a word from the stream. To extract entire lines of text, see the `basic_string` overload of global function `getline`.

#### Parameters

- is*      `basic_istream` object from which characters are extracted.
- str*     `basic_string` object where the extracted content is stored.

#### Return Value

The same as parameter *is*.

A call to this function may set any of the internal state flags of *is* if:

flag	error
<code>eofbit</code>	The end of the source of characters is reached during its operations.
<code>failbit</code>	The input obtained could not be interpreted as a valid textual representation of an object of this type. In this case, <i>distr</i> preserves the parameters and internal data it had before the call. Notice that some <code>eofbit</code> cases will also set <code>failbit</code> .
<code>badbit</code>	An error other than the above happened. (see <code>ios_base::iostate</code> for more info on these)

Additionally, in any of these cases, if the appropriate flag has been set with *is*'s member function `basic_ios::exceptions`, an exception of type `ios_base::failure` is thrown.

#### Example

```
1 // extract to string
2 #include <iostream>
3 #include <string>
4
5 main ()
6 {
7     std::string name;
8
9     std::cout << "Please, enter your name: ";
10    std::cin >> name;
11    std::cout << "Hello, " << name << "!\n";
12
13    return 0;
14 }
```

#### Complexity

Unspecified, but generally linear in the resulting length of *str*.

#### Iterator validity

Any iterators, pointers and references related to *str* may be invalidated.

#### Data races

Both objects, *is* and *str*, are modified.

#### Exception safety

**Basic guarantee:** if an exception is thrown, both *is* and *str* end up in a valid state.

#### See also

<b>istream::operator&gt;&gt;</b>	Extract formatted input (public member function )
----------------------------------	---

**getline (basic\_string)** Get line from stream into string (function template )

**operator<< (basic\_string)** Insert string into stream (function template )

## /string/basic\_string/operator[]

public member function

### std::basic\_string::operator[]

<string>

```
reference operator[] (size_type pos);
const_reference operator[] (size_type pos) const;
```

#### Get character of string

Returns a reference to the character at position *pos* in the `basic_string`.

If *pos* is equal to the `string length` and the `string object` is `const-qualified`, the function returns a reference to a null character (`charT()`).

If *pos* is equal to the `string length`, the function returns a reference to the null character that follows the last character in the string (which should not be modified).

#### Parameters

*pos*

Value with the position of a character within the string.

Note: The first character in a `basic_string` is denoted by a value of 0 (not 1).

Member type `size_type` is an unsigned integral type.

#### Return value

The character at the specified position in the string.

If the `basic_string` object is `const-qualified`, the function returns a `const_reference`. Otherwise, it returns a `reference`.

Member types `reference` and `const_reference` are the reference types to the characters in the container; They shall be aliases of `charT&` and `const charT&` respectively.

#### Example

```
1 // string::operator[]
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for (int i=0; i<str.length(); ++i)
9     {
10         std::cout << str[i];
11     }
12     return 0;
13 }
```

This code prints out the content of a string character by character using the offset operator on *str*:

Test string

#### Complexity

Unspecified.

#### Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

#### Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The reference returned can be used to access or modify characters.

#### Exception safety

If *pos* is less than the `string length`, the function never throws exceptions (no-throw guarantee).

If *pos* is equal to the `string length`, the `const-version` never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

Note that using the reference returned to modify elements that are out of bounds (including the character at *pos*) also causes *undefined behavior*.

#### Complexity

Constant.

#### Iterator validity

No changes.

## Data races

The object is accessed (neither the const nor the non-const versions modify it).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## Exception safety

If *pos* is less or equal to the string length, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

Note that using the reference returned to modify elements that are out of bounds (including the character at *pos*) also causes *undefined behavior*.

## See also

<a href="#">basic_string::at</a>	Get character of string (public member function )
<a href="#">basic_string::substr</a>	Generate substring (public member function )
<a href="#">basic_string::find</a>	Find first occurrence in string (public member function )
<a href="#">basic_string::replace</a>	Replace portion of string (public member function )

# /string/basic\_string/operator+

function template

## std::operator+ (basic\_string)

<string>

string (1)	template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, const basic_string<charT,traits,Alloc>& rhs);
c-string (2)	template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
character (3)	template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, charT rhs); template <class charT, class traits, class Alloc> basic_string operator+ (charT lhs, const basic_string<charT,traits,Alloc>& rhs);
string (1)	template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, const basic_string<charT,traits,Alloc>& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (basic_string<charT,traits,Alloc>&& lhs, basic_string<charT,traits,Alloc>&& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (basic_string<charT,traits,Alloc>&& lhs, const basic_string<charT,traits,Alloc>& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, basic_string<charT,traits,Alloc>&& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs); template <class charT, class traits, class Alloc> basic_string operator+ (basic_string<charT,traits,Alloc>&& lhs, const charT* rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const charT* lhs, basic_string<charT,traits,Alloc>&& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, charT rhs); template <class charT, class traits, class Alloc> basic_string operator+ (basic_string<charT,traits,Alloc>&& lhs, charT rhs); template <class charT, class traits, class Alloc> basic_string operator+ (charT lhs, const basic_string<charT,traits,Alloc>& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (charT lhs, basic_string<charT,traits,Alloc>&& rhs);
c-string (2)	template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs); template <class charT, class traits, class Alloc> basic_string operator+ (basic_string<charT,traits,Alloc>&& lhs, const charT* rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const charT* lhs, basic_string<charT,traits,Alloc>&& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, charT rhs); template <class charT, class traits, class Alloc> basic_string operator+ (basic_string<charT,traits,Alloc>&& lhs, charT rhs); template <class charT, class traits, class Alloc> basic_string operator+ (charT lhs, const basic_string<charT,traits,Alloc>& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (charT lhs, basic_string<charT,traits,Alloc>&& rhs);
character (3)	template <class charT, class traits, class Alloc> basic_string operator+ (const basic_string<charT,traits,Alloc>& lhs, charT rhs); template <class charT, class traits, class Alloc> basic_string operator+ (charT lhs, const basic_string<charT,traits,Alloc>& rhs); template <class charT, class traits, class Alloc> basic_string operator+ (charT lhs, basic_string<charT,traits,Alloc>&& rhs);

## Concatenate strings

Returns a newly constructed `basic_string` object with its value being the concatenation of the characters in *lhs* followed by those of *rhs*.

In the signatures taking at least one *rvalue reference* as argument, the returned object is *move-constructed* by passing this argument, which is left in an unspecified but valid state. If both arguments are *rvalue references*, only one of them is moved (it is unspecified which), with the other one preserving its value.

## Parameters

*lhs*, *rhs*

Arguments to the left- and right-hand side of the operator, respectively.  
If of type `charT*`, it shall point to a null-terminated character sequence.  
`charT` is `basic_string`'s character type (i.e., its first template parameter).

## Example

```
1 // concatenating strings
2 #include <iostream>
3 #include <string>
4
5 main ()
6 {
7     std::string firstlevel ("com");
8     std::string secondlevel ("cplusplus");
9     std::string scheme ("http://");
```

```

10 std::string hostname;
11 std::string url;
12
13 hostname = "www." + seconlevel + '.' + firstlevel;
14 url = scheme + hostname;
15
16 std::cout << url << '\n';
17
18 return 0;
19 }

```

Output:

<http://www.cplusplus.com>

## Return Value

A `basic_string` object whose value is the concatenation of *lhs* and *rhs*.

## Complexity

Unspecified, but generally linear in the resulting `string length` (and linear in the length of the non-moved argument for signatures with *rvalue references*).

## Iterator validity

The signatures with *rvalue references* may invalidate iterators, pointers and references related to the moved `basic_string` object.

## Data races

The signatures with *rvalue references* modify the moved `basic_string` object.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in either `basic_string` objects.

If *s* is not a null-terminated character sequence, it causes *undefined behavior*.

If the resulting `string length` would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>basic_string::append</code>	Append to string (public member function )
<code>basic_string::insert</code>	Insert into string (public member function )
<code>basic_string::operator+=</code>	Append to string (public member function )

# /string/basic\_string/operator+=

public member function

## std::basic\_string::operator+=

<string>

```

string (1) basic_string& operator+=(const basic_string& str);
c-string (2) basic_string& operator+=(const charT* s);
character (3) basic_string& operator+=(charT c);

string (1) basic_string& operator+=(const basic_string& str);
c-string (2) basic_string& operator+=(const charT* s);
character (3) basic_string& operator+=(charT c);
initializer list (4) basic_string& operator+=(initializer_list<charT> il);

```

### Append to string

Extends the `basic_string` by appending additional characters at the end of its current value:

(See member function `append` for additional appending options).

## Parameters

<code>str</code>	A <code>basic_string</code> object of the same type (with the same class template arguments <code>charT</code> , <code>traits</code> and <code>Alloc</code> ), whose value is copied at the end.
<code>s</code>	Pointer to a null-terminated sequence of characters. The sequence is copied at the end of the string. The <code>length</code> is determined by calling <code>traits_type::length(s)</code> .
<code>c</code>	A character, which is appended to the current value of the string.
<code>il</code>	An <code>initializer_list</code> object. These objects are automatically constructed from <code>initializer_list</code> declarators. The characters are appended to the string, in the same order.

`charT` is `basic_string`'s character type (i.e., its first template parameter).

## Return Value

\*this

## Example

```
1 // string::operator+=
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string name ("John");
8     std::string family ("Smith");
9     name += " K. ";           // c-string
10    name += family;         // string
11    name += '\n';           // character
12
13    std::cout << name;
14
15 }
```

Output:

```
John K. Smith
```

## Complexity

Unspecified, but generally up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>basic_string::append</code>	Append to string (public member function )
<code>basic_string::assign</code>	Assign content to string (public member function )
<code>basic_string::operator=</code>	String assignment (public member function )
<code>basic_string::insert</code>	Insert into string (public member function )
<code>basic_string::replace</code>	Replace portion of string (public member function )

# /string/basic\_string/operators

function template

## std::relational operators (basic\_string)

<string>

```
template <class charT, class traits, class Alloc>
    bool operator== (const basic_string<charT,traits,Alloc>& lhs,
                      const basic_string<charT,traits,Alloc>& rhs);
(1) template <class charT, class traits, class Alloc>
    bool operator== (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator== (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator!= (const basic_string<charT,traits,Alloc>& lhs,
                     const basic_string<charT,traits,Alloc>& rhs);
(2) template <class charT, class traits, class Alloc>
    bool operator!= (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator!= (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator< (const basic_string<charT,traits,Alloc>& lhs,
                    const basic_string<charT,traits,Alloc>& rhs);
(3) template <class charT, class traits, class Alloc>
    bool operator< (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator< (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator<= (const basic_string<charT,traits,Alloc>& lhs,
                     const basic_string<charT,traits,Alloc>& rhs);
(4) template <class charT, class traits, class Alloc>
    bool operator<= (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator<= (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator> (const basic_string<charT,traits,Alloc>& lhs,
                    const basic_string<charT,traits,Alloc>& rhs);
```

```

(5) template <class charT, class traits, class Alloc>
    bool operator> (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator>= (const basic_string<charT,traits,Alloc>& lhs,
                     const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator<= (const basic_string<charT,traits,Alloc>& lhs,
                     const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator== (const basic_string<charT,traits,Alloc>& lhs,
                      const basic_string<charT,traits,Alloc>& rhs) noexcept;
(1) template <class charT, class traits, class Alloc>
    bool operator== (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator== (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator!= (const basic_string<charT,traits,Alloc>& lhs,
                     const basic_string<charT,traits,Alloc>& rhs) noexcept;
(2) template <class charT, class traits, class Alloc>
    bool operator!= (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator!= (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator< (const basic_string<charT,traits,Alloc>& lhs,
                    const basic_string<charT,traits,Alloc>& rhs) noexcept;
(3) template <class charT, class traits, class Alloc>
    bool operator< (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator< (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator<= (const basic_string<charT,traits,Alloc>& lhs,
                     const basic_string<charT,traits,Alloc>& rhs) noexcept;
(4) template <class charT, class traits, class Alloc>
    bool operator<= (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator<= (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator> (const basic_string<charT,traits,Alloc>& lhs,
                    const basic_string<charT,traits,Alloc>& rhs) noexcept;
(5) template <class charT, class traits, class Alloc>
    bool operator> (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator> (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);
template <class charT, class traits, class Alloc>
    bool operator>= (const basic_string<charT,traits,Alloc>& lhs,
                     const basic_string<charT,traits,Alloc>& rhs) noexcept;
(6) template <class charT, class traits, class Alloc>
    bool operator>= (const charT* lhs, const basic_string<charT,traits,Alloc>& rhs);
template <class charT, class traits, class Alloc>
    bool operator>= (const basic_string<charT,traits,Alloc>& lhs, const charT* rhs);

```

### Relational operators for `basic_string`

Performs the appropriate comparison operation between the `basic_string` objects *lhs* and *rhs*.

The functions use `basic_string::compare` for the comparison, which depends on the `compare` member of its `character traits`.

These operators are overloaded in header `<string>`.

### Parameters

*lhs*, *rhs*  
 Arguments to the left- and right-hand side of the operator, respectively.  
 If of type `charT*`, it shall point to a null-terminated character sequence.

### Example

```

1 // string comparisons
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::string foo = "alpha";
8     std::string bar = "beta";
9
10    if (foo==bar) std::cout << "foo and bar are equal\n";
11    if (foo!=bar) std::cout << "foo and bar are not equal\n";
12    if (foo< bar) std::cout << "foo is less than bar\n";
13    if (foo> bar) std::cout << "foo is greater than bar\n";
14    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
15    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
16
17    return 0;
18 }

```

Output:

```

foo and bar are not equal
foo is less than bar
foo is less than or equal to bar

```

## Return Value

true if the condition holds, and false otherwise.

## Complexity

Unspecified, but generally up to linear in both *lhs* and *rhs*'s lengths.

## Iterator validity

No changes.

## Data races

Both objects, *lhs* and *rhs*, are accessed.

## Exception safety

If an argument of type `char*` does not point to null-terminated character sequence, it causes *undefined behavior*. Otherwise, if an exception is thrown, there are no changes in the `basic_string` (strong guarantee).

If an argument of type `char*` does not point to null-terminated character sequence, it causes *undefined behavior*.

For operations between `basic_string` objects, exceptions are never thrown (no-throw guarantee).

For other cases, if an exception is thrown, there are no changes in the `basic_string` (strong guarantee).

## See also

<code>basic_string::compare</code>	Compare strings (public member function )
<code>basic_string::find</code>	Find first occurrence in string (public member function )
<code>basic_string::operator=</code>	String assignment (public member function )
<code>basic_string::swap</code>	Swap string values (public member function )

# /string/basic\_string/pop\_back

public member function

## std::basic\_string::pop\_back

<string>

`void pop_back();`

### Delete last character

Erases the last character of the `basic_string`, effectively reducing its length by one.

## Parameters

none

## Return value

none

## Example

```
1 // string::pop_back
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("hello world!");
8     str.pop_back();
9     std::cout << str << '\n';
10    return 0;
11 }
```

hello world

## Complexity

Unspecified, but generally constant.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

If the `basic_string` is empty, it causes *undefined behavior*.

Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<a href="#">basic_string::back</a>	Access last character (public member function )
<a href="#">basic_string::push_back</a>	Append character to string (public member function )
<a href="#">basic_string::erase</a>	Erase characters from string (public member function )

## /string/basic\_string/push\_back

public member function

### std::basic\_string::push\_back

<string>

`void push_back (charT c);`

#### Append character to string

Appends character c to the end of the `basic_string`, increasing its `length` by one.

## Parameters

c

Character added to the `basic_string`.  
`charT` is `basic_string`'s character type (i.e., its first template parameter).

## Return value

none

## Example

```
1 // string::push_back
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5
6 int main ()
7 {
8     std::string str;
9     std::ifstream file ("test.txt",std::ios::in);
10    if (file) {
11        while (!file.eof()) str.push_back(file.get());
12    }
13    std::cout << str << '\n';
14    return 0;
15 }
```

This example reads an entire file character by character, appending each character to a string object using `push_back`.

## Complexity

Unspecified; Generally amortized constant, but up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<a href="#">basic_string::back</a>	Access last character (public member function )
<a href="#">basic_string::pop_back</a>	Delete last character (public member function )
<a href="#">basic_string::append</a>	Append to string (public member function )
<a href="#">basic_string::insert</a>	Insert into string (public member function )
<a href="#">basic_string::end</a>	Return iterator to end (public member function )

## /string/basic\_string/rbegin

public member function

### std::basic\_string::rbegin

<string>

```
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
```

```
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
```

### Return reverse iterator to reverse beginning

Returns a *reverse iterator* pointing to the last character of the string (i.e., its *reverse beginning*).

*Reverse iterators* iterate backwards: increasing them moves them towards the beginning of the string.

*rbegin* points to the character right before the one that would be pointed to by member *end*.

### Parameters

none

### Return Value

A reverse iterator to the *reverse beginning* of the string.

If the `basic_string` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are reverse `random access iterator` types (pointing to a character and to a `const` character, respectively).

### Example

```
1 // string::rbegin/rend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("now step live...");
8     for (std::string::reverse_iterator rit=str.rbegin(); rit!=str.rend(); ++rit)
9         std::cout << *rit;
10    return 0;
11 }
```

This code prints out the reversed content of a string character by character using a reverse iterator that iterates between `rbegin` and `rend`. Notice how even though the reverse iterator is increased, the iteration goes backwards through the string (this is a feature of reverse iterators).

The actual output is:

```
...evil pets won
```

### Complexity

Unspecified.

### Iterator validity

Generally, no changes.

On some implementations, the non-`const` version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

### Data races

The object is accessed, and in some implementations, the non-`const` version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

### Complexity

Unspecified, but generally constant.

### Iterator validity

No changes.

### Data races

The object is accessed (neither the `const` nor the non-`const` versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<code>basic_string::rend</code>	Return reverse iterator to reverse end (public member function )
---------------------------------	--

<code>basic_string::begin</code>	Return iterator to beginning (public member function )
----------------------------------	--

<code>basic_string::end</code>	Return iterator to end (public member function )
--------------------------------	--

## /string/basic\_string/rend

public member function

## std::basic\_string::rend

<string>

```
reverse_iterator rend();
const_reverse_iterator rend() const;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

### Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element preceding the first character of the string (which is considered its *reverse end*).

The range between `basic_string::rbegin` and `basic_string::rend` contains all the characters of the `basic_string` (in reverse order).

### Parameters

none

### Return Value

A reverse iterator to the *reverse end* of the string.

If the `basic_string` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are reverse `random access iterator` types (pointing to a character and to a `const` character, respectively).

### Example

```
1 // string::rbegin/rend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("now step live...");
8     for (std::string::reverse_iterator rit=str.rbegin(); rit!=str.rend(); ++rit)
9         std::cout << *rit;
10    return 0;
11 }
```

This code prints out the reversed content of a string character by character using a reverse iterator that iterates between `rbegin` and `rend`. Notice how even though the reverse iterator is increased, the iteration goes backwards through the string (this is a feature of reverse iterators).

The actual output is:

...evil pets won

### Complexity

Unspecified.

### Iterator validity

Generally, no changes.

On some implementations, the non-`const` version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

### Data races

The object is accessed, and in some implementations, the non-`const` version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

### Complexity

Unspecified, but generally constant.

### Iterator validity

No changes.

### Data races

The object is accessed (neither the `const` nor the non-`const` versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">basic_string::rbegin</a>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<a href="#">basic_string::begin</a>	Return iterator to beginning ( <a href="#">public member function</a> )
<a href="#">basic_string::end</a>	Return iterator to end ( <a href="#">public member function</a> )

# /string/basic\_string/replace

public member function

## std::basic\_string::replace

<string>

```
string (1) basic_string& replace (size_type pos, size_type len, const basic_string& str);
basic_string& replace (iterator i1, iterator i2, const basic_string& str);

substring (2) basic_string& replace (size_type pos, size_type len, const basic_string& str,
                                         size_type subpos, size_type sublen);

c-string (3) basic_string& replace (size_type pos, size_type len, const charT* s);
basic_string& replace (iterator i1, iterator i2, const charT* s);

buffer (4) basic_string& replace (size_type pos, size_type len, const charT* s, size_type n);
basic_string& replace (iterator i1, iterator i2, const charT* s, size_type n);

fill (5) basic_string& replace (size_type pos, size_type len, size_type n, chart c);
basic_string& replace (iterator i1, iterator i2, size_type n, chart c);

template <class InputIterator>
range (6) basic_string& replace (iterator i1, iterator i2,
                                 InputIterator first, InputIterator last);

string (1) basic_string& replace (size_type pos, size_type len, const basic_string& str);
basic_string& replace (const_iterator i1, const_iterator i2, const basic_string& str);

substring (2) basic_string& replace (size_type pos, size_type len, const basic_string& str,
                                         size_type subpos, size_type sublen);

c-string (3) basic_string& replace (size_type pos, size_type len, const charT* s);
basic_string& replace (const_iterator i1, const_iterator i2, const charT* s);

buffer (4) basic_string& replace (size_type pos, size_type len, const charT* s, size_type n);
basic_string& replace (const_iterator i1, const_iterator i2, const charT* s, size_type n);

fill (5) basic_string& replace (size_type pos, size_type len, size_type n, chart c);
basic_string& replace (const_iterator i1, const_iterator i2, size_type n, chart c);

template <class InputIterator>
range (6) basic_string& replace (const_iterator i1, const_iterator i2,
                                 InputIterator first, InputIterator last);

initializer list (7) basic_string& replace (const_iterator i1, const_iterator i2, initializer_list<charT> il);

string (1) basic_string& replace (size_type pos, size_type len, const basic_string& str);
basic_string& replace (const_iterator i1, const_iterator i2, const basic_string& str);

substring (2) basic_string& replace (size_type pos, size_type len, const basic_string& str,
                                         size_type subpos, size_type sublen =npos);

c-string (3) basic_string& replace (size_type pos, size_type len, const charT* s);
basic_string& replace (const_iterator i1, const_iterator i2, const charT* s);

buffer (4) basic_string& replace (size_type pos, size_type len, const charT* s, size_type n);
basic_string& replace (const_iterator i1, const_iterator i2, const charT* s, size_type n);

fill (5) basic_string& replace (size_type pos, size_type len, size_type n, chart c);
basic_string& replace (const_iterator i1, const_iterator i2, size_type n, chart c);

template <class InputIterator>
range (6) basic_string& replace (const_iterator i1, const_iterator i2,
                                 InputIterator first, InputIterator last);

initializer list (7) basic_string& replace (const_iterator i1, const_iterator i2, initializer_list<charT> il);
```

### Replace portion of string

Replaces the portion of the string that begins at character *pos* and spans *len* characters (or the part of the string in the range between [i1,i2]) by new contents:

#### (1) string

Copies *str*.

#### (2) substring

Copies the portion of *str* that begins at the character position *subpos* and spans *sublen* characters (or until the end of *str*, if either *str* is too short or if *sublen* is `basic_string::npos`).

#### (3) c-string

Copies the null-terminated character sequence (C-string) pointed by *s*.  
The length is determined by calling `traits.length(s)`.

#### (4) buffer

Copies the first *n* characters from the array of characters pointed by *s*.

#### (5) fill

Replaces the portion of the string by *n* consecutive copies of character *c*.

#### (6) range

Copies the sequence of characters in the range [first,last), in the same order.

#### (7) initializer list

Copies each of the characters in *il*, in the same order.

## Parameters

*str*

Another `basic_string` object of the same type (with the same class template arguments `chart`, `traits` and `Alloc`), whose value is copied.

*pos*

Position of the first character to be replaced.

If this is greater than the `string` `length`, it throws `out_of_range`.

*len*

Number of characters to replace (if the string is shorter, as many characters as possible are replaced).

A value of `basic_string::npos` indicates all characters until the end of the string.

*subpos*

Position of the first character in *str* that is copied to the object as replacement.

If this is greater than *str*'s `length`, it throws `out_of_range`.

*sublen*

Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).  
A value of `basic_string::npos` indicates all characters until the end of `str`.

`s` Pointer to an array of characters (such as a *c-string*).

`n` Number of characters to copy.

`c` Character value, repeated `n` times.

`first, last`

Input iterators to the initial and final positions in a range. The range used is `[first, last)`, which includes all the characters between `first` and `last`, including the character pointed by `first` but not the character pointed by `last`.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type convertible to `charT`.

`il`

An `initializer_list` object.

These objects are automatically constructed from `initializer list` declarators.

`charT` is `basic_string`'s character type (i.e., its first template parameter).  
Member type `size_type` is an unsigned integral type.

## Return Value

`*this`

## Example

```
1 // replacing in a string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string base="this is a test string.";
8     std::string str2="n example";
9     std::string str3="sample phrase";
10    std::string str4="useful.";
11
12    // replace signatures used in the same order as described above:
13
14    // Using positions:          0123456789*123456789*12345
15    std::string str=base;        // "this is a test string."
16    str.replace(9,5,str2);      // "this is an example string." (1)
17    str.replace(19,6,str3,7,6); // "this is an example phrase." (2)
18    str.replace(8,10,"just a"); // "this is just a phrase." (3)
19    str.replace(8,6,"a shorty",7); // "this is a short phrase." (4)
20    str.replace(22,1,3,'!');   // "this is a short phrase!!!" (5)
21
22    // Using iterators:          0123456789*123456789*
23    str.replace(str.begin(),str.end()-3,str3); // "sample phrase!!!" (1)
24    str.replace(str.begin(),str.begin()+6,"replace"); // "replace phrase!!!" (3)
25    str.replace(str.begin()+8,str.begin()+14,"is coolness",7); // "replace is cool!!!" (4)
26    str.replace(str.begin()+12,str.end()-4,4,'o'); // "replace is coooool!!!" (5)
27    str.replace(str.begin()+11,str.end(),str4.begin(),str4.end()); // "replace is useful." (6)
28    std::cout << str << '\n';
29    return 0;
30 }
```

Output:

```
replace is useful.
```

## Complexity

Unspecified, but generally up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If `s` does not point to an array long enough, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

If `pos` is greater than the `string` length, or if `subpos` is greater than `str`'s `length`, an `out_of_range` exception is thrown.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>basic_string::insert</code>	Insert into string (public member function )
<code>basic_string::append</code>	Append to string (public member function )
<code>basic_string::substr</code>	Generate substring (public member function )

<a href="#">basic_string::erase</a>	Erase characters from string (public member function )
<a href="#">basic_string::assign</a>	Assign content to string (public member function )

## /string/basic\_string/reserve

public member function

### std::basic\_string::reserve

<string>

`void reserve (size_type n = 0);`

#### Request a change in capacity

Requests that the [string capacity](#) be adapted to a planned change in [size](#) to a [length](#) of up to  $n$  characters.

If  $n$  is greater than the current [string capacity](#), the function causes the container to increase its [capacity](#) to  $n$  characters (or greater).

In all other cases, it is taken as a non-binding request to shrink the [string capacity](#): the container implementation is free to optimize otherwise and leave the [basic\\_string](#) with a [capacity](#) greater than  $n$ .

This function has no effect on the [string length](#) and cannot alter its content.

#### Parameters

n

Planned [length](#) for the [basic\\_string](#).

Note that the resulting [string capacity](#) may be equal or greater than  $n$ .

Member type [size\\_type](#) is an unsigned integral type.

#### Return Value

none

#### Example

```
1 // string::reserve
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5
6 int main ()
7 {
8     std::string str;
9
10    std::ifstream file ("test.txt",std::ios::in|std::ios::ate);
11    if (file) {
12        std::ifstream::streampos filesize = file.tellg();
13        str.reserve(filesize);
14
15        file.seekg(0);
16        while (!file.eof())
17        {
18            str += file.get();
19        }
20        std::cout << str;
21    }
22    return 0;
23 }
```

This example reserves enough capacity in the [basic\\_string](#) object to store an entire file, which is then read character by character. By reserving a [capacity](#) for the [basic\\_string](#) of at least the size of the entire file, we try to avoid all the automatic reallocations that the object  $str$  could suffer each time that inserting a new character would make its [length](#) surpass its [capacity](#).

#### Complexity

Unspecified, but generally constant.

#### Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

#### Data races

The object is modified.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [basic\\_string](#).

If  $n$  is greater than the [max\\_size](#), a [length\\_error](#) exception is thrown.

If the type uses the [default allocator](#), a [bad\\_alloc](#) exception is thrown if the function needs to allocate storage and fails.

#### See also

<a href="#">basic_string::capacity</a>	Return size of allocated storage (public member function )
<a href="#">basic_string::shrink_to_fit</a>	Shrink to fit (public member function )
<a href="#">basic_string::resize</a>	Resize string (public member function )

<a href="#">basic_string::max_size</a>	Return maximum size (public member function )
--	---

## /string/basic\_string/resize

public member function

### std::basic\_string::resize

<string>

```
void resize (size_type n);
void resize (size_type n, charT c);
```

#### Resize string

Resizes the string to a [length](#) of  $n$  characters.

If  $n$  is smaller than the current [string length](#), the current value is shortened to its first  $n$  character, removing the characters beyond the  $n$ th.

If  $n$  is greater than the current [string length](#), the current content is extended by inserting at the end as many characters as needed to reach a size of  $n$ . If  $c$  is specified, the new elements are initialized as copies of  $c$ , otherwise, they are [value-initialized characters](#) (null characters).

#### Parameters

n

New [string length](#), expressed in number of characters.  
Member type [size\\_type](#) is an unsigned integral type.

c

Character used to fill the new character space added to the string (in case the string is expanded).

#### Return Value

none

#### Example

```
1 // resizing string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("I like to code in C");
8     std::cout << str << '\n';
9
10    std::string::size_type sz = str.size();
11
12    str.resize (sz+2, '+');
13    std::cout << str << '\n';
14
15    str.resize (14);
16    std::cout << str << '\n';
17    return 0;
18 }
```

Output:

```
I like to code in C
I like to code in C++
I like to code
```

#### Complexity

Unspecified, but generally up to linear in the new [string length](#).

#### Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

#### Data races

The object is modified.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [basic\\_string](#).

If  $n$  is greater than [max\\_size](#), a [length\\_error](#) exception is thrown.

If the type uses the [default allocator](#), a [bad\\_alloc](#) exception is thrown if the function needs to allocate storage and fails.

#### See also

<a href="#">basic_string::size</a>	Return size (public member function )
------------------------------------	---------------------------------------

<a href="#">basic_string::clear</a>	Clear string (public member function )
-------------------------------------	--

<a href="#">basic_string::max_size</a>	Return maximum size (public member function )
--	---

## /string/basic\_string/rfind

public member function

## std::basic\_string::rfind

<string>

```
string(1) size_type rfind (const basic_string& str, size_type pos = npos) const;
c-string(2) size_type rfind (const charT* s, size_type pos = npos) const;
buffer(3) size_type rfind (const charT* s, size_type pos, size_type n) const;
character(4) size_type rfind (charT c, size_type pos = npos) const;

string(1) size_type rfind (const basic_string& str, size_type pos = npos) const noexcept;
c-string(2) size_type rfind (const charT* s, size_type pos = npos) const;
buffer(3) size_type rfind (const charT* s, size_type pos, size_type n) const;
character(4) size_type rfind (charT c, size_type pos = npos) const noexcept;
```

### Find last occurrence in string

Searches the [basic\\_string](#) for the last occurrence of the sequence specified by its arguments.

When *pos* is specified, the search only includes sequences of characters that begin at or before position *pos*, ignoring any possible match beginning after *pos*.

The function uses [traits\\_type::eq](#) to determine character equivalences.

### Parameters

str

Another [basic\\_string](#) with the subject to search for.

pos

Position of the last character in the string to be considered as the beginning of a match.

Any value greater or equal than the [string length](#) (including [basic\\_string::npos](#)) means that the entire string is searched.

Note: The first character is denoted by a value of 0 (not 1).

s

Pointer to an array of characters.

If argument *n* is specified (3), the sequence to match are the first *n* characters in the array.

Otherwise (2), a null-terminated sequence is expected: the length of the sequence to match is determined by the first occurrence of a null character.

n

Length of sequence of characters to match.

c

Individual character to be searched for.

charT is [basic\\_string](#)'s character type (i.e., its first template parameter).

Member type [size\\_type](#) is an unsigned integral type.

### Return Value

The position of the first character of the last match.

If no matches were found, the function returns [basic\\_string::npos](#).

Member type [size\\_type](#) is an unsigned integral type.

### Example

```
1 // string::rfind
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("The sixth sick sheik's sixth sheep's sick.");
8     std::string key ("sixth");
9
10    std::string::size_type found = str.rfind(key);
11    if (found!=std::string::npos)
12        str.replace (found,key.length(),"seventh");
13
14    std::cout << str << '\n';
15
16    return 0;
17 }
```

The sixth sick sheik's seventh sheep's sick.

### Complexity

Unspecified, but generally up to linear in the [string length](#) (or *pos*) times the number of characters to match (worst case).

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

If *s* does not point to an array long enough, it causes *undefined behavior*.

Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<a href="#">basic_string::find</a>	Find first occurrence in string (public member function )
<a href="#">basic_string::find_last_of</a>	Find character in string from the end (public member function )
<a href="#">basic_string::find_last_not_of</a>	Find non-matching character in string from the end (public member function )
<a href="#">basic_string::replace</a>	Replace portion of string (public member function )
<a href="#">basic_string::substr</a>	Generate substring (public member function )

## /string/basic\_string/shrink\_to\_fit

public member function

### std::basic\_string::shrink\_to\_fit

<string>

`void shrink_to_fit();`

#### Shrink to fit

Requests the `basic_string` to reduce its `capacity` to fit its `size`.

The request is non-binding, and the container implementation is free to optimize otherwise and leave the `basic_string` with a `capacity` greater than its `size`.

This function has no effect on the `string length` and cannot alter its content.

#### Parameters

none

#### Return value

none

#### Example

```
1 // string::shrink_to_fit
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str (100,'x');
8     std::cout << "1. capacity of str: " << str.capacity() << '\n';
9
10    str.resize(10);
11    std::cout << "2. capacity of str: " << str.capacity() << '\n';
12
13    str.shrink_to_fit();
14    std::cout << "3. capacity of str: " << str.capacity() << '\n';
15
16    return 0;
17 }
```

Possible output:

```
1. capacity of str: 100
2. capacity of str: 100
3. capacity of str: 10
```

#### Complexity

Unspecified, but generally constant.

#### Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

#### Data races

The object is modified.

#### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<a href="#">basic_string::capacity</a>	Return size of allocated storage (public member function )
<a href="#">basic_string::reserve</a>	Request a change in capacity (public member function )
<a href="#">basic_string::resize</a>	Resize string (public member function )
<a href="#">basic_string::clear</a>	Clear string (public member function )

## /string/basic\_string/size

public member function

**std::basic\_string::size** <string>

```
size_type size() const;
size_type size() const noexcept;
```

#### Return size

Returns the length of the string, in terms of number of characters.

This is the number of actual characters that conform the contents of the [basic\\_string](#), which is not necessarily equal to its storage capacity.

Both [basic\\_string::size](#) and [basic\\_string::length](#) are synonyms and return the same value.

#### Parameters

none

#### Return Value

The number of characters in the string.

Member type [size\\_type](#) is an unsigned integral type.

#### Example

```
1 // string::size
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "The size of str is " << str.size() << " characters.\n";
9     return 0;
10 }
```

Output:

```
The size of str is 11 characters
```

#### Complexity

Unspecified.

Constant.

#### Iterator validity

No changes.

#### Data races

The object is accessed.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

#### See also

<a href="#">basic_string::length</a>	Return length of string (public member function )
<a href="#">basic_string::resize</a>	Resize string (public member function )
<a href="#">basic_string::max_size</a>	Return maximum size (public member function )
<a href="#">basic_string::capacity</a>	Return size of allocated storage (public member function )

## /string/basic\_string/substr

public member function

**std::basic\_string::substr** <string>

```
basic_string substr (size_type pos = 0, size_type len = npos) const;
```

#### Generate substring

Returns a newly constructed [basic\\_string](#) object with its value initialized to a copy of a substring of this object.

The substring is the portion of the object that starts at character position *pos* and spans *len* characters (or until the end of the string, whichever comes first).

#### Parameters

*pos*

Position of the first character to be copied as a substring.  
If this is equal to the [string length](#), the function returns an empty string.  
If this is greater than the [string length](#), it throws [out\\_of\\_range](#).  
Note: The first character is denoted by a value of 0 (not 1).

*len*

Number of characters to include in the substring (if the string is shorter, as many characters as possible are used). A value of `basic_string::npos` indicates all characters until the end of the string.

Member type `size_type` is an unsigned integral type.

### Return Value

A `basic_string` object with a substring of this object.

### Example

```
1 // string::substr
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str="We think in generalities, but we live in details.";
8                         // (quoting Alfred N. Whitehead)
9
10    std::string str2 = str.substr (12,12);           // "generalities"
11
12    std::string::size_type pos = str.find("live"); // position of "live" in str
13
14    std::string str3 = str.substr (pos);           // get from "live" to the end
15
16    std::cout << str2 << ' ' << str3 << '\n';
17
18    return 0;
19 }
```

Output:

```
generalities live in details.
```

### Complexity

Unspecified, but generally linear in the `length` of the returned object.

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `basic_string`.

If `pos` is greater than the `string` length, an `out_of_range` exception is thrown.

If the type uses the `default allocator`, a `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

### See also

`basic_string::replace` Replace portion of string ([public member function](#))

`basic_string::data` Get string data ([public member function](#))

`basic_string::find` Find first occurrence in string ([public member function](#))

`basic_string::assign` Assign content to string ([public member function](#))

`basic_string::basic_string` Construct `basic_string` object ([public member function](#))

## /string/basic\_string/swap

public member function

### std::basic\_string::swap

`<string>`

`void swap (basic_string& str);`

#### Swap string values

Exchanges the content of the container by the content of `str`, which is another `basic_string` object of the same type. `Lengths` may differ.

After the call to this member function, the value of this object is the value `str` had before the call, and the value of `str` is the value this object had before the call.

Notice that a non-member function exists with the same name, `swap`, overloading that algorithm with an optimization that behaves like this member function.

No specifics on allocators.

Whether the container `allocators` are also swapped is not defined, unless in the case the appropriate `allocator traits` indicate explicitly that they shall `propagate`.

### Parameters

`str`

Another `basic_string` object of the same type (i.e., instantiated with the same template parameters, `charT`, `traits` and `Alloc`), whose value is swapped with that of this `basic_string`.

## Return value

none

## Example

```
1 // swap strings
2 #include <iostream>
3 #include <string>
4
5 main ()
6 {
7     std::string buyer ("money");
8     std::string seller ("goods");
9
10    std::cout << "Before the swap, buyer has " << buyer;
11    std::cout << " and seller has " << seller << '\n';
12
13    seller.swap (buyer);
14
15    std::cout << " After the swap, buyer has " << buyer;
16    std::cout << " and seller has " << seller << '\n';
17
18    return 0;
19 }
```

Output:

```
Before the swap, buyer has money and seller has goods
After the swap, buyer has goods and seller has money
```

## Complexity

Constant.

## Iterator validity

Any iterators, pointers and references related to this object and to *str* may be invalidated.

## Data races

Both the object and *str* are modified.

## Exception safety

If the allocators in both *strings* compare equal, or if their *allocator traits* indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">swap (basic_string)</a>	Exchanges the values of two strings (function template )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )

# /string/basic\_string/swap-free

function template

## std::swap (basic\_string)

<string>

```
template <class charT, class traits, class Alloc>
void swap (basic_string<charT,traits,Alloc> & x,
           basic_string<charT,traits,Alloc> & y);
```

### Exchanges the values of two strings

Exchanges the values of *basic\_string* objects *x* and *y*, such that after the call to this function, the value of *x* is the one which was on *y* before the call, and the value of *y* is that of *x*.

This is an overload of the generic algorithm *swap* that improves its performance by mutually transferring ownership over their internal data to the other object (i.e., the strings exchange references to their data, without actually copying the characters): It behaves as if *x.swap(y)* was called.

## Parameters

*x,y* *basic\_string* objects of the same type (i.e., having both the same template parameters, *charT*, *traits* and *Alloc*).

## Return value

none

## Example

```
1 // swap strings
2 #include <iostream>
3 #include <string>
4
```

```

5 main ()
6 {
7     std::string buyer ("money");
8     std::string seller ("goods");
9
10    std::cout << "Before the swap, buyer has " << buyer;
11    std::cout << " and seller has " << seller << '\n';
12
13    swap (buyer,seller);
14
15    std::cout << " After the swap, buyer has " << buyer;
16    std::cout << " and seller has " << seller << '\n';
17
18    return 0;
19 }

```

Output:

```

Before the swap, buyer has money and seller has goods
After the swap, buyer has goods and seller has money

```

## Complexity

Constant.

## Iterator validity

Any iterators, pointers and references related to both *x* and *y* may be invalidated.

## Data races

Both objects, *x* and *y*, are modified.

## Exception safety

If the allocators in both *strings* compare equal, or if their *allocator traits* indicate that the allocators shall propagate, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">basic_string::swap</a>	Swap string values ( <a href="#">public member function</a> )
<a href="#">swap</a>	Exchange values of two objects ( <a href="#">function template</a> )
<a href="#">swap_ranges</a>	Exchange values of two ranges ( <a href="#">function template</a> )

## /string/char\_traits

class template

### std::char\_traits

<string>

```

template <class charT> struct char_traits;
template <> struct char_traits<char>;
template <> struct char_traits<wchar_t>;
template <class charT> struct char_traits;
template <> struct char_traits<char>;
template <> struct char_traits<wchar_t>;
template <> struct char_traits<char16_t>;
template <> struct char_traits<char32_t>;

```

#### Character traits

Character traits classes specify character properties and provide specific semantics for certain operations on characters and sequences of characters.

The standard library includes a standard set of *character traits classes* that can be instantiated from the *char\_traits* template, and which are used by default both for the *basic\_string* objects and for the *input/output stream* objects. But any other class that follows the requirements of a *character traits class* can be used instead. This reference attempts to describe both the definition of the standard *char\_traits* and the requirements for custom *character traits classes*.

## Template parameters

charT

Character type.

The class defines the standard *character traits* for this character type.

This shall be one of the types for which an specialization is provided.

Aliased as member type *char\_traits::char\_type*.

## Template specializations

The *char\_traits* standard template supports to be instantiated with at least the following character types:

type	Description
char	Basic character set (size of 1 byte)
wchar_t	Wide character set (same size, signedness, and alignment as another integral type)

char	Basic character set (size of 1 byte)
wchar_t	Wide character set (same size, signedness, and alignment as another integral type)
char16_t	Represents 16-bit code units (same size, signedness, and alignment as <i>uint_least16_t</i> )
char32_t	Represents any of the 32-bit code points (same size, signedness, and alignment as <i>uint_least32_t</i> )

## Member types

member type	description for <i>character traits types</i>	definition	
		char	wchar_t
char_type	The template parameter ( <code>charT</code> )	char	wchar_t
int_type	Integral type that can represent all <code>charT</code> values, as well as <code>eof()</code>	int	wint_t
off_type	A type that behaves like <code>streamoff</code>	streamoff	streamoff
pos_type	A type that behaves like <code>streampos</code>	streampos	wstreampos
state_type	Multibyte transformation state type, such as <code>mbstate_t</code>	mbstate_t	mbstate_t

member type	description for <i>character traits types</i>	definition			
		char	wchar_t	char16_t	char32_t
char_type	The template parameter ( <code>charT</code> )	char	wchar_t	char16_t	char32_t
int_type	Integral type that can represent all <code>charT</code> values, as well as <code>eof()</code>	int	wint_t	uint_least16_t	uint_least32_t
off_type	A type that behaves like <code>streamoff</code>	streamoff	streamoff	streamoff	streamoff
pos_type	A type that behaves like <code>streampos</code>	streampos	wstreampos	u16streampos	u32streampos
state_type	Multibyte transformation state type, such as <code>mbstate_t</code>	mbstate_t	mbstate_t	mbstate_t	mbstate_t

## Member functions

<code>eq</code>	Compare characters for equality ( public static member function )
<code>lt</code>	Compare characters for inequality ( public static member function )
<code>length</code>	Get length of null-terminated string ( public static member function )
<code>assign</code>	Assign character ( public static member function )
<code>compare</code>	Compare sequences of characters ( public static member function )
<code>find</code>	Find first occurrence of character ( public static member function )
<code>move</code>	Move character sequence ( public static member function )
<code>copy</code>	Copy character sequence ( public static member function )
<code>eof</code>	End-of-File character ( public static member function )
<code>not_eof</code>	Not End-of-File character ( public static member function )
<code>to_char_type</code>	To char type ( public static member function )
<code>to_int_type</code>	To int type ( public static member function )
<code>eq_int_type</code>	Compare int_type values ( public static member function )

## /string/char\_traits/assign

public static member function

### std::char\_traits::assign

<string>

```
character (1) static void      assign (char_type& r, const char_type& c);
array (2) static char_type assign (char_type* p, size_t n, char_type c);

character (1) static void      assign (char_type& r, const char_type& c) noexcept;
array (2) static char_type assign (char_type* p, size_t n, char_type c);
```

#### Assign character

Assigns `c` to a character (`r`) or to an array of characters (`s`).

##### (1) character

Assigns `c` to `r`, as if using `r=c`.

##### (2) array

Assigns `c` to the first `n` characters in the array pointed by `s`, each as if using operator=.

In the standard specializations of `char_traits`, this function behaves as the built-in operator=, but custom *character traits* classes may provide an alternative behavior whenever this is consistent with the assignment operation of its character type.

## Parameters

`r` An lvalue reference to character.

`c` A character value.

`p` A pointer to the array where the characters will be written.

`n` Number of characters to fill with a value of `c`.

Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`). `size_t` is an unsigned integral type.

## Return Value

none

## Complexity

Constant.

## Exception safety

Unless *s* does not point to an array long enough, this member function never throws exceptions (no-throw guarantee) in any of the standard specializations. Otherwise, it causes *undefined behavior*.

## See also

[char\\_traits::eq](#) Compare characters for equality ( public static member function )

# /string/char\_traits/compare

public static member function

## std::char\_traits::compare

<string>

static int compare (const char\_type\* p, const char\_type\* q, size\_t n);

### Compare sequences of characters

Compares the sequence of *n* characters pointed by *p* to the sequence of *n* characters pointed by *q*.

The function performs a **lexicographical comparison** where two characters are considered equal if **member eq** returns true, and one character is considered less than another one if **member lt** returns true.

For all *character traits* types, it shall behave as if defined as:

```
1 static int compare (const char_type* p, const char_type* q, size_t n) {  
2     while (n--) {if (!eq(*p,*q)) return lt(*p,*q)?-1:1; ++p; ++q;}  
3     return 0;  
4 }
```

Although the specific signature may vary.

## Parameters

*p, q*

Pointers to arrays with a sequence of characters each.

Notice that the function will consider that the length of both sequences is *n* characters, independently on whether any of them contains or not null-characters.

Member type *char\_type* is the *character type* (i.e., the class template parameter in *char\_traits*).

*n*

Length (in characters) of the sequence of characters to compare.

*size\_t* is an unsigned integral type.

## Return Value

Returns a signed integral indicating the relation between the sequences:

value	relation
0	All characters compare equal
<0	The first character that does not compare equal is less in <i>p</i> .
>0	The first character that does not compare equal is greater in <i>p</i> .

## Example

```
1 // char_traits::compare  
2 #include <iostream>      // std::cout  
3 #include <string>        // std::basic_string, std::char_traits  
4 #include <cctype>         // std::tolower  
5 #include <cstddef>        // std::size_t  
6  
7 // case-insensitive traits:  
8 struct custom_traits: std::char_traits<char> {  
9     static bool eq (char c, char d) { return std::tolower(c)==std::tolower(d); }  
10    static bool lt (char c, char d) { return std::tolower(c)<std::tolower(d); }  
11    static int compare (const char* p, const char* q, std::size_t n) {  
12        while (n--) {if (!eq(*p,*q)) return lt(*p,*q)?-1:1; ++p; ++q;}  
13        return 0;  
14    }  
15};  
16  
17 int main ()  
18 {  
19     std::basic_string<char,custom_traits> foo,bar;  
20     foo = "Test";  
21     bar = "test";  
22     if (foo==bar) std::cout << "foo and bar are equal\n";  
23     return 0;  
24 }
```

Output:

foo and bar are equal

## Complexity

Up to linear in *n*.

## Exception safety

Unless either *p* or *q* does not point to an array long enough, this member function never throws exceptions (no-throw guarantee) in any of the standard specializations.

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">char_traits::eq</a>	Compare characters for equality ( public static member function )
<a href="#">char_traits::lt</a>	Compare characters for inequality ( public static member function )
<a href="#">char_traits::find</a>	Find first occurrence of character ( public static member function )
<a href="#">strncmp</a>	Compare characters of two strings (function )
<a href="#">lexicographical_compare</a>	Lexicographical less-than comparison (function template )

## /string/char\_traits/copy

public static member function

### std::char\_traits::copy

<string>

`static char_type* copy (char_type* dest, const char_type* src, size_t n);`

#### Copy character sequence

Copies the sequence of *n* characters pointed by *src* to the array pointed by *dest*. Ranges shall not overlap.

All *character traits* types shall implement the function as if the individual characters were assigned using member `assign`.

## Parameters

<code>dest</code>	Pointer to an array where the copied characters are written.
<code>src</code>	Pointer to an array with the <i>n</i> characters to copy.
<code>n</code>	Number of characters to copy.

Notice that the function will consider that the length of both *dest* and *src* sequences is *n* characters, independently on whether any of them contains null-characters.

Member type *char\_type* is the *character type* (i.e., the class template parameter in `char_traits`).

`size_t` is an unsigned integral type.

## Return Value

Returns *dest*.

## Example

```
1 // char_traits::copy
2 #include <iostream>    // std::cout
3 #include <string>      // std::char_traits
4
5 int main ()
6 {
7     char foo[] = "test string";
8     char bar[20];
9
10    unsigned len = std::char_traits<char>::length(foo);
11    std::char_traits<char>::copy (bar,foo,len);
12
13    bar[len] = '\0'; // append null-character
14
15    std::cout << "foo contains: " << foo << '\n';
16    std::cout << "bar contains: " << bar << '\n';
17    return 0;
18 }
```

Output:

```
foo contains: test string
bar contains: test string
```

## Complexity

Linear in *n*.

## Exception safety

Unless either *dest* or *src* does not point to an array long enough, this member function never throws exceptions (no-throw guarantee) in any of the standard specializations.

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">char_traits::move</a>	Move character sequence ( public static member function )
-----------------------------------	---

<code>char_traits::assign</code>	Assign character ( public static member function )
<code>strncpy</code>	Copy characters from string (function )
<code>memcpy</code>	Copy block of memory (function )

## /string/char\_traits/eof

public static member function

### std::char\_traits::eof

<string>

```
static int_type eof();
static constexpr int_type eof() noexcept;
```

#### End-of-File character

Returns the *End-of-File* value.

The *End-of-File* value is a special value used by many standard functions to represent an invalid character; Its value shall not compare equal to any of the values representable with `char_type` (as if transformed with member `int_type` and compared with member `eq_int_type`).

For the standard specializations of `char_traits` it returns:

specialization	value returned by <code>eof()</code>
<code>char</code>	<code>EOF</code>
<code>wchar_t</code>	<code>WEOF</code>
<code>char16_t</code>	A value that is not a valid UTF-16 code unit.
<code>char32_t</code>	A value that is not a valid Unicode code point.

#### Parameters

none

#### Return Value

The *End-of-File* value.

Member type `int_type` is an integral type that can represent this value or any valid character value.

#### Example

```
1 // char_traits::eof
2 #include <iostream>    // std::wcin, std::wcout
3 #include <string>      // std::wstring, std::char_traits
4
5 int main () {
6     std::wcout << "Please enter some text: ";
7
8     if (std::wcin.peek() == std::char_traits<wchar_t>::eof()) {
9         std::wcout << "Reading error";
10    }
11    else {
12        std::wstring ws;
13        std::getline (std::wcin,ws);
14        std::wcout << "You entered a word: " << ws << '\n';
15    }
16
17    return 0;
18 }
```

#### Complexity

Constant.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

#### See also

<code>char_traits::not_eof</code>	Not End-of-File character ( public static member function )
<code>EOF</code>	End-of-File (constant)

## /string/char\_traits/eq

public static member function

### std::char\_traits::eq

<string>

```
static bool eq (const char_type& c, const char_type& d);
static constexpr bool eq (char_type c, char_type d) noexcept;
```

#### Compare characters for equality

Returns whether characters `c` and `d` are considered equal.

In the standard specializations of `char_traits`, this function behaves as the built-in operator`==`, but custom *character traits* classes may provide an alternative behavior.

In the standard specializations of `char_traits`, this function behaves as the built-in operator`==` for type `unsigned char`, but custom *character traits* classes may provide an alternative behavior.

## Parameters

`c, d`  
Character values.  
Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`).

## Return Value

true if `c` is considered equal to `d`.

## Example

```
1 // char_traits::eq
2 #include <iostream>      // std::cout
3 #include <string>        // std::basic_string, std::char_traits
4 #include <cctype>         // std::tolower
5 #include <cstddef>        // std::size_t
6
7 // traits with case-insensitive eq:
8 struct custom_traits: std::char_traits<char> {
9     static bool eq (char c, char d) { return std::tolower(c)==std::tolower(d); }
10    // some (non-conforming) implementations of basic_string::find call this instead of eq:
11    static const char* find (const char* s, std::size_t n, char c)
12    { while( n-- && (!eq(*s,c)) ) ++s; return s; }
13 };
14
15 int main ()
16 {
17     std::basic_string<char,custom_traits> str ("Test");
18     std::cout << "T found at position " << str.find('t') << '\n';
19     return 0;
20 }
```

Output:

```
T found at position 0
```

## Complexity

Constant.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions in any of the standard specializations.

## See also

<code>char_traits::lt</code>	Compare characters for inequality ( public static member function )
<code>char_traits::compare</code>	Compare sequences of characters ( public static member function )
<code>char_traits::find</code>	Find first occurrence of character ( public static member function )
<code>char_traits::eq_int_type</code>	Compare <code>int_type</code> values ( public static member function )
<code>equal_to</code>	Function object class for equality comparison (class template )

## /string/`char_traits::eq_int_type`

public static member function

### `std::char_traits::eq_int_type`

`<string>`

```
static bool eq_int_type (const int_type& x, const int_type& y);
static constexpr bool eq_int_type (int_type x, int_type y) noexcept;
```

#### Compare `int_type` values

Returns whether `x` and `y` are considered equal.

If both `x` and `y` represent valid characters, the function returns the same as member `eq` would for their `char_type` transformations.

Also, if both `x` and `y` are copies of `eof()`, the function returns `true`. If only one of them is a copy of `eof()`, the function returns `false`.

In all other cases, the returned value is unspecified.

## Parameters

`x, y`  
Values to compare.  
Member type `int_type` is an integral type that can represent `eof()` or any valid character value.

## Return Value

true if `x` is considered equal to `y`.

## Complexity

Constant.

## Exception safety

No-throw guarantee: this member function never throws exceptions.

## See also

[char\\_traits::to\\_char\\_type](#) To char type ( public static member function )

[char\\_traits::to\\_int\\_type](#) To int type ( public static member function )

# /string/char\_traits/find

public static member function

## std::char\_traits::find

<string>

```
static const char_type* find (const char_type* p, size_t n, const char_type& c);
```

### Find first occurrence of character

Returns a pointer to the first character in the sequence of  $n$  characters pointed by  $p$  that compares equal to  $c$ .

All *character traits* types shall implement the function as if the individual characters were compared using member `eq`.

## Parameters

$p$

Pointer to an array with a sequence of characters.

Notice that the function will consider that the length of the sequences is  $n$  characters, independently on whether it contains null-characters or not.

$n$

Length (in characters) of the sequence of characters to compare.

$c$

A character value.

Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`).  
`size_t` is an unsigned integral type.

## Return Value

A pointer to the first match, if any, or a *null pointer* otherwise.

## Example

```
1 // char_traits::find
2 #include <iostream>    // std::cout
3 #include <string>      // std::char_traits
4
5 int main ()
6 {
7     const char foo[] = "test string";
8     const char* p = std::char_traits<char>::find(foo, std::char_traits<char>::length(foo), 'i');
9     if (p) std::cout << "the first 'i' in \" " << foo << "\" is at " << (p-foo) << ".\n";
10    return 0;
11 }
```

Output:

```
the first 'i' in "test string" is at 8.
```

## Complexity

Up to linear in  $n$ .

## Exception safety

Unless either  $p$  does not point to an array long enough, this member function never throws exceptions (no-throw guarantee) in any of the standard specializations.

Otherwise, it causes *undefined behavior*.

## See also

[char\\_traits::compare](#) Compare sequences of characters ( public static member function )

[char\\_traits::eq](#) Compare characters for equality ( public static member function )

[string::find](#) Find content in string (public member function )

# /string/char\_traits/length

public static member function

## std::char\_traits::length

<string>

```
static size_t length (const char_type* s);
```

### Get length of null-terminated string

Returns the length of the *null-terminated* character sequence *s*.

The behavior implemented by all *character traits* types shall be to return the position of the first character for which `member eq` returns false when compared against `charT()`.

This function is equivalent to `strlen` (for `char`) and `wcslen` (for `wchar_t`).

### Parameters

*s*

Pointer to a null-terminated character sequence.  
Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`).

### Return Value

Returns the length of *s*.

`size_t` is an unsigned integral type.

### Example

```
1 // char_traits::length
2 #include <iostream>    // std::cout
3 #include <string>      // std::char_traits
4
5 int main ()
6 {
7     const char * foo = "String literal";
8     std::cout << "foo has a length of ";
9     std::cout << std::char_traits<char>::length(foo);
10    std::cout << " characters.\n";
11    return 0;
12 }
```

Output:

```
foo has a length of 14 characters.
```

### Complexity

Linear in the returned value.

### Exception safety

If *s* points a null-terminated character sequence, this member function never throws exceptions (no-throw guarantee) in any of the standard specializations. Otherwise, it causes *undefined behavior*.

### See also

<code>char_traits::eof</code>	End-of-File character ( public static member function )
<code>strlen</code>	Get string length (function)
<code>wcslen</code>	Get wide string length (function)

## /string/`char_traits::lt`

public static member function

### `std::char_traits::lt`

`<string>`

```
static bool lt (const char_type& c, const char_type& d);
static constexpr bool lt (char_type c, char_type d) noexcept;
```

#### Compare characters for inequality

Returns whether character *c* is considered less than character *d* (i.e., whether *c* goes before *d* when ordered).

In the standard specializations of `char_traits`, this function behaves as the built-in operator`<`.

In the standard specializations of `char_traits`, this function behaves as the built-in operator`<` for type `unsigned char`.

### Parameters

*c*, *d*

Character values.  
Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`).

### Return Value

true if *c* is considered less than *d*.

### Complexity

Constant.

## Exception safety

No-throw guarantee: this member function never throws exceptions in any of the standard specializations.

## See also

<a href="#">char_traits::eq</a>	Compare characters for equality ( public static member function )
<a href="#">char_traits::compare</a>	Compare sequences of characters ( public static member function )
<a href="#">less</a>	Function object class for less-than inequality comparison (class template )

## /string/char\_traits/move

public static member function

### std::char\_traits::move

<string>

`static char_type* move (char_type* dest, const char_type* src, size_t n);`

#### Move character sequence

Copies the sequence of  $n$  characters pointed by  $src$  to the array pointed by  $dest$ , even if the ranges overlap.

All *character traits* types shall implement the function as if the individual characters were assigned using `member assign`.

## Parameters

<code>dest</code>	Pointer to an array where the copied characters are written.
<code>src</code>	Pointer to an array with the $n$ characters to copy.
<code>n</code>	Number of characters to copy.

Notice that the function will consider that the length of both  $dest$  and  $src$  sequences is  $n$  characters, independently on whether any of them contains null-characters.

Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`).

`size_t` is an unsigned integral type.

## Return Value

Returns  $dest$ .

## Example

```
1 // char_traits::move
2 #include <iostream> // std::cout
3 #include <string> // std::char_traits
4
5 int main ()
6 {
7     char foo[] = "---o.....";
8     std::cout << foo << '\n';
9     std::char_traits<char>::move (foo+3,foo,4);
10    std::cout << foo << '\n';
11    std::char_traits<char>::move (foo+6,foo,7);
12    std::cout << foo << '\n';
13    return 0;
14 }
```

Output:

```
---o.....
-----o.....
-----o....
```

## Complexity

Linear in  $n$ .

## Exception safety

Unless either  $dest$  or  $src$  does not point to an array long enough, this member function never throws exceptions (no-throw guarantee) in any of the standard specializations.

Otherwise, it causes *undefined behavior*.

## See also

<a href="#">char_traits::copy</a>	Copy character sequence ( public static member function )
<a href="#">char_traits::assign</a>	Assign character ( public static member function )
<a href="#">memmove</a>	Move block of memory (function )

## /string/char\_traits/not\_eof

public static member function

## std::char\_traits::not\_eof

<string>

```
static int_type not_eof(const int_type& c);
static constexpr int_type not_eof(int_type c) noexcept;
```

### Not End-of-File character

Returns a value that is guaranteed to not be an *End-of-File*.

If *c* is not an *End-of-File* character, it always returns *c* unchanged. Otherwise, some implementation-defined value that is not an *End-of-File* character.

All *character traits* types shall implement the function so that the returned value compares different from `eof()` using member `eq_int_type`.

### Parameters

*c*

Value.

Member type `int_type` is an integral type that can represent `eof()` or any valid character value.

### Return Value

*c* if *c* is not an *End-of-File* value, and some other value otherwise.

Member type `int_type` is an integral type that can represent `eof()` or any valid character value.

### Complexity

Constant.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

### See also

[char\\_traits::eof](#)

End-of-File character ( public static member function )

## /string/char\_traits/to\_char\_type

public static member function

## std::char\_traits::to\_char\_type

<string>

```
static char_type to_char_type (const int_type& c);
static constexpr char_type to_char_type (int_type c) noexcept;
```

### To char type

Returns the `char_type` equivalent of *c*.

If *c* has no corresponding valid character value, the function returns some implementation-defined value.

All *character traits* types shall implement the function so that the returned value transformed back to `int_type` with `to_int_type` compares equal to *c* (using member `eq_int_type`), if *c* represented a valid character value.

### Parameters

*c*

Value.

Member type `int_type` is an integral type that can represent `eof()` or any valid character value.

### Return Value

The `char_type` equivalent of *c*.

Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`).

### Complexity

Constant.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

### See also

[char\\_traits::to\\_int\\_type](#) To int type ( public static member function )

[char\\_traits::eq\\_int\\_type](#) Compare int\_type values ( public static member function )

## /string/char\_traits/to\_int\_type

public static member function

## std::char\_traits::to\_int\_type

<string>

```
static int_type to_int_type (const char_type& c);
static constexpr int_type to_int_type (char_type c) noexcept;
```

### To int type

Returns the `int_type` equivalent of `c`.

All `character traits` types shall implement the function so that the returned value can be transformed back to its `char_type` value using `char_traits::to_char_type`.

In the standard specializations of `char_traits`, this function performs the corresponding built-in integral promotion.

### Parameters

`c`

Character value.

Member type `char_type` is the *character type* (i.e., the class template parameter in `char_traits`).

### Return Value

The `int_type` equivalent of `c`.

Member type `int_type` is an integral type that can represent `eof()` or any valid character value.

### Complexity

Constant.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

### See also

`char_traits::to_char_type` To `char_type` ( public static member function )

`char_traits::eq_int_type` Compare `int_type` values ( public static member function )

## /string/stod

function template

## std::stod

<string>

```
double stod (const string& str, size_t* idx = 0);
double stod (const wstring& str, size_t* idx = 0);
```

### Convert string to double

Parses `str` interpreting its content as a floating-point number, which is returned as a value of type `double`.

If `idx` is not a null pointer, the function also sets the value of `idx` to the position of the first character in `str` after the number.

The function uses `strtod` (or `wcstod`) to perform the conversion (see `strtod` for more details on the process). Note that the format accepted by these functions depends on the current locale.

### Parameters

`str`

String object with the representation of a floating-point number.

`idx`

Pointer to an object of type `size_t`, whose value is set by the function to position of the next character in `str` after the numerical value. This parameter can also be a null pointer, in which case it is not used.

### Return Value

On success, the function returns the converted floating-point number as a value of type `double`.

### Example

```
1 // stod example
2 #include <iostream>    // std::cout
3 #include <string>      // std::string, std::stod
4
5 int main ()
6 {
7     std::string orbits ("365.24 29.53");
8     std::string::size_type sz;      // alias of size_t
9
10    double earth = std::stod (orbits,&sz);
11    double moon = std::stod (orbits.substr(sz));
12    std::cout << "The moon completes " << (earth/moon) << " orbits per Earth year.\n";
13    return 0;
14 }
```

Possible output:

The moon completes 12.3684 orbits per Earth year.

## Complexity

Unspecified, but generally linear in the number of characters interpreted.

## Data races

Modifies the value pointed by *idx* (if not zero).

## Exceptions

If no conversion could be performed, an `invalid_argument` exception is thrown.

If the value read is out of the range of representable values by a `double` (in some library implementations, this includes underflows), an `out_of_range` exception is thrown.

An invalid *idx* causes *undefined behavior*.

## See also

<code>stof</code>	Convert string to float (function template )
<code>stold</code>	Convert string to long double (function template )
<code>stoi</code>	Convert string to integer (function template )
<code>strtof</code>	Convert string to float (function )

## /string/stof

function template

### `std::stof`

`<string>`

```
float stof (const string& str, size_t* idx = 0);
float stof (const wstring& str, size_t* idx = 0);
```

#### Convert string to float

Parses *str* interpreting its content as a floating-point number, which is returned as a value of type `float`.

If *idx* is not a null pointer, the function also sets the value of *idx* to the position of the first character in *str* after the number.

The function uses `strtod` (or `wctod`) to perform the conversion (see `strtod` for more details on the process). Note that the format accepted by these functions depends on the current locale.

## Parameters

`str`

String object with the representation of a floating-point number.

`idx`

Pointer to an object of type `size_t`, whose value is set by the function to position of the next character in *str* after the numerical value.  
This parameter can also be a null pointer, in which case it is not used.

## Return Value

On success, the function returns the converted floating-point number as a value of type `float`.

## Example

```
1 // stof example
2 #include <iostream>    // std::cout
3 #include <string>      // std::string, std::stof
4
5 int main ()
6 {
7     std::string orbits ("686.97 365.24");
8     std::string::size_type sz;        // alias of size_t
9
10    float mars = std::stof (orbits,&sz);
11    float earth = std::stof (orbits.substr(sz));
12    std::cout << "One martian year takes " << (mars/earth) << " Earth years.\n";
13    return 0;
14 }
```

Possible output:

```
One martian year takes 1.88087 Earth years.
```

## Complexity

Unspecified, but generally linear in the number of characters interpreted.

## Data races

Modifies the value pointed by *idx* (if not zero).

## Exceptions

If no conversion could be performed, an [invalid\\_argument](#) exception is thrown.

If the value read is out of the range of representable values by a `float` (in some library implementations, this includes underflows), an [out\\_of\\_range](#) exception is thrown.

An invalid `idx` causes *undefined behavior*.

## See also

<a href="#">std::stod</a>	Convert string to double (function template )
<a href="#">std::stold</a>	Convert string to long double (function template )
<a href="#">std::stoi</a>	Convert string to integer (function template )
<a href="#">std::strtof</a>	Convert string to float (function )

## /string/stoi

function template

`std::stoi`

`<string>`

```
int stoi (const string& str, size_t* idx = 0, int base = 10);
int stoi (const wstring& str, size_t* idx = 0, int base = 10);
```

### Convert string to integer

Parses `str` interpreting its content as an integral number of the specified `base`, which is returned as an `int` value.

If `idx` is not a null pointer, the function also sets the value of `idx` to the position of the first character in `str` after the number.

The function uses `strtol` (or `wctol`) to perform the conversion (see `strtol` for more details on the process).

## Parameters

`str`

String object with the representation of an integral number.

`idx`

Pointer to an object of type `size_t`, whose value is set by the function to position of the next character in `str` after the numerical value. This parameter can also be a null pointer, in which case it is not used.

`base`

Numerical base (radix) that determines the valid characters and their interpretation.

If this is 0, the base used is determined by the format in the sequence (see `strtol` for details). Notice that by default this argument is 10, not 0.

## Return Value

On success, the function returns the converted integral number as an `int` value.

## Example

```
1 // stoi example
2 #include <iostream>    // std::cout
3 #include <string>      // std::string, std::stoi
4
5 int main ()
6 {
7     std::string str_dec = "2001, A Space Odyssey";
8     std::string str_hex = "40c3";
9     std::string str_bin = "-10010110001";
10    std::string str_auto = "0x7f";
11
12    std::string::size_type sz;    // alias of size_t
13
14    int i_dec = std::stoi (str_dec,&sz);
15    int i_hex = std::stoi (str_hex,nullptr,16);
16    int i_bin = std::stoi (str_bin,nullptr,2);
17    int i_auto = std::stoi (str_auto,nullptr,0);
18
19    std::cout << str_dec << ":" << i_dec << " and [" << str_dec.substr(sz) << "]\n";
20    std::cout << str_hex << ":" << i_hex << '\n';
21    std::cout << str_bin << ":" << i_bin << '\n';
22    std::cout << str_auto << ":" << i_auto << '\n';
23
24    return 0;
25 }
```

Output:

```
2001, A Space Odyssey: 2001 and [ , A Space Odyssey]
40c3: 16579
-10010110001: -1201
0x7f: 127
```

## Complexity

Unspecified, but generally linear in the number of characters interpreted.

## Data races

Modifies the value pointed by `idx` (if not zero).

## Exceptions

If no conversion could be performed, an `invalid_argument` exception is thrown.

If the value read is out of the range of representable values by an `int`, an `out_of_range` exception is thrown.

An invalid `idx` causes *undefined behavior*.

## See also

<code>stol</code>	Convert string to long int (function template )
<code>stoul</code>	Convert string to unsigned integer (function template )
<code>strtol</code>	Convert string to long integer (function )

## /string/stol

function template

### `std::stol`

`<string>`

```
long stol (const string& str, size_t* idx = 0, int base = 10);
long stol (const wstring& str, size_t* idx = 0, int base = 10);
```

#### Convert string to long int

Parses `str` interpreting its content as an integral number of the specified `base`, which is returned as a value of type `long int`.

If `idx` is not a null pointer, the function also sets the value of `idx` to the position of the first character in `str` after the number.

The function uses `strtol` (or `wctol`) to perform the conversion (see `strtol` for more details on the process).

## Parameters

<code>str</code>	String object with the representation of an integral number.
<code>idx</code>	Pointer to an object of type <code>size_t</code> , whose value is set by the function to position of the next character in <code>str</code> after the numerical value. This parameter can also be a null pointer, in which case it is not used.
<code>base</code>	Numerical base (radix) that determines the valid characters and their interpretation. If this is 0, the base used is determined by the format in the sequence (see <code>strtol</code> for details). Notice that by default this argument is 10, not 0.

## Return Value

On success, the function returns the converted integral number as a value of type `long int`.

## Example

```
1 // stol example
2 #include <iostream>    // std::cout
3 #include <string>      // std::string, std::stol
4
5 int main ()
6 {
7     std::string str_dec = "1987520";
8     std::string str_hex = "2f04e009";
9     std::string str_bin = "-11101001100100111010";
10    std::string str_auto = "0x7fffff";
11
12    std::string::size_type sz;    // alias of size_t
13
14    long li_dec = std::stol (str_dec,&sz);
15    long li_hex = std::stol (str_hex,nullptr,16);
16    long li_bin = std::stol (str_bin,nullptr,2);
17    long li_auto = std::stol (str_auto,nullptr,0);
18
19    std::cout << str_dec << ":" << li_dec << '\n';
20    std::cout << str_hex << ":" << li_hex << '\n';
21    std::cout << str_bin << ":" << li_bin << '\n';
22    std::cout << str_auto << ":" << li_auto << '\n';
23
24    return 0;
25 }
```

Output:

```
1987520: 1987520
2f04e009: 788848649
-11101001100100111010: -956730
0x7fffff: 8388607
```

## Complexity

Unspecified, but generally linear in the number of characters interpreted.

## Data races

Modifies the value pointed by *idx* (if not zero).

## Exceptions

If no conversion could be performed, an `invalid_argument` exception is thrown.

If the value read is out of the range of representable values by a `long int`, either an `invalid_argument` or an `out_of_range` exception is thrown.

An invalid *idx* causes *undefined behavior*.

## See also

<code>stoi</code>	Convert string to integer (function template )
<code>stoll</code>	Convert string to long long (function template )
<code>stoul</code>	Convert string to unsigned integer (function template )
<code>strtol</code>	Convert string to long integer (function )

## /string/stold

function template

### `std::stold`

`<string>`

```
long double stold (const string& str, size_t* idx = 0);
long double stold (const wstring& str, size_t* idx = 0);
```

#### Convert string to long double

Parses *str* interpreting its content as a floating-point number, which is returned as a value of type `long double`.

If *idx* is not a null pointer, the function also sets the value of *idx* to the position of the first character in *str* after the number.

The function uses `strtold` (or `wctold`) to perform the conversion (see `strtod` for more details on the process).

## Parameters

`str`

String object with the representation of a floating-point number.

`idx`

Pointer to an object of type `size_t`, whose value is set by the function to position of the next character in *str* after the numerical value.  
This parameter can also be a null pointer, in which case it is not used.

## Return Value

On success, the function returns the converted floating-point number as a value of type `long double`.

## Example

```
1 // stold example
2 #include <iostream>    // std::cout
3 #include <string>      // std::string, std::stod
4
5 int main ()
6 {
7     std::string orbits ("90613.305 365.24");
8     std::string::size_type sz;        // alias of size_t
9
10    long double pluto = std::stod (orbits,&sz);
11    long double earth = std::stod (orbits.substr(sz));
12    std::cout << "Pluto takes " << (pluto/earth) << " years to complete an orbit.\n";
13    return 0;
14 }
```

Possible output:

```
Pluto takes 248.093 years to complete an orbit.
```

## Complexity

Unspecified, but generally linear in the number of characters interpreted.

## Data races

Modifies the value pointed by *idx* (if not zero).

## Exceptions

If no conversion could be performed, an `invalid_argument` exception is thrown.

If the value read is out of the range of representable values by a `long double` (in some library implementations, this includes underflows), an `out_of_range` exception is thrown.

An invalid *idx* causes *undefined behavior*.

## See also

<code>stof</code>	Convert string to float (function template )
-------------------	--

<b>std::stod</b>	Convert string to double (function template )
<b>std::stoi</b>	Convert string to integer (function template )
<b>std::strtol</b>	Convert string to long double (function )

## /string/stoll

function template

### std::stoll

<string>

```
long long stoll (const string& str, size_t* idx = 0, int base = 10);
long long stoll (const wstring& str, size_t* idx = 0, int base = 10);
```

#### Convert string to long long

Parses *str* interpreting its content as an integral number of the specified *base*, which is returned as a value of type `long long`.

If *idx* is not a null pointer, the function also sets the value of *idx* to the position of the first character in *str* after the number.

The function uses `strtoll` (or `wctoll`) to perform the conversion (see `strtol` for more details on the process).

### Parameters

<b>str</b>	String object with the representation of an integral number.
<b>idx</b>	Pointer to an object of type <code>size_t</code> , whose value is set by the function to position of the next character in <i>str</i> after the numerical value. This parameter can also be a null pointer, in which case it is not used.
<b>base</b>	Numerical base (radix) that determines the valid characters and their interpretation. If this is 0, the base used is determined by the format in the sequence (see <code>strtol</code> for details). Notice that by default this argument is 10, not 0.

### Return Value

On success, the function returns the converted integral number as a value of type `long long`.

### Example

```
1 // stoll example
2 #include <iostream>      // std::cout
3 #include <string>        // std::string, std::stoll
4
5 int main ()
6 {
7     std::string str = "8246821 0xfffff 020";
8
9     std::string::size_type sz = 0;    // alias of size_t
10
11    while (!str.empty()) {
12        long long ll = std::stoll (str,&sz,0);
13        std::cout << str.substr(0,sz) << " interpreted as " << ll << '\n';
14        str = str.substr(sz);
15    }
16
17    return 0;
18 }
```

Output:

```
8246821 interpreted as 8246821
0xfffff interpreted as 65535
020 interpreted as 16
```

### Complexity

Unspecified, but generally linear in the number of characters interpreted.

### Data races

Modifies the value pointed by *idx* (if not zero).

### Exceptions

If no conversion could be performed, an `invalid_argument` exception is thrown.

If the value read is out of the range of representable values by a `long long`, an `out_of_range` exception is thrown.

An invalid *idx* causes *undefined behavior*.

### See also

<b>stoi</b>	Convert string to integer (function template )
<b>stol</b>	Convert string to long int (function template )
<b>stoull</b>	Convert string to unsigned long long (function template )
<b>strtol</b>	Convert string to long long integer (function )

## /string/stoul

function template

### std::stoul

<string>

```
unsigned long stoul (const string& str, size_t* idx = 0, int base = 10);
unsigned long stoul (const wstring& str, size_t* idx = 0, int base = 10);
```

#### Convert string to unsigned integer

Parses *str* interpreting its content as an integral number of the specified *base*, which is returned as an *unsigned long* value.

If *idx* is not a null pointer, the function also sets the value of *idx* to the position of the first character in *str* after the number.

The function uses `strtoul` (or `wcstoul`) to perform the conversion (see `strtol` for more details on the process).

#### Parameters

*str*

String object with the representation of an integral number.

*idx*

Pointer to an object of type `size_t`, whose value is set by the function to position of the next character in *str* after the numerical value. This parameter can also be a null pointer, in which case it is not used.

*base*

Numerical base (radix) that determines the valid characters and their interpretation.

If this is 0, the base used is determined by the format in the sequence (see `strtol` for details). Notice that by default this argument is 10, not 0.

#### Return Value

On success, the function returns the converted integral number as an *unsigned long* value.

#### Example

```
1 // stoul example
2 #include <iostream>    // std::cin, std::cout
3 #include <string>      // std::string, std::stoul, std::getline
4
5 int main ()
6 {
7     std::string str;
8     std::cout << "Enter an unsigned number: ";
9     std::getline (std::cin,str);
10    unsigned long ul = std::stoul (str,nullptr,0);
11    std::cout << "You entered: " << ul << '\n';
12    return 0;
13 }
```

#### Complexity

Unspecified, but generally linear in the number of characters interpreted.

#### Data races

Modifies the value pointed by *idx* (if not zero).

#### Exceptions

If no conversion could be performed, an `invalid_argument` exception is thrown.

If the value read is out of the range of representable values by an *unsigned long*, an `out_of_range` exception is thrown.

An invalid *idx* causes *undefined behavior*.

#### See also

<a href="#">stoi</a>	Convert string to integer (function template )
<a href="#">stol</a>	Convert string to long int (function template )
<a href="#">stoull</a>	Convert string to unsigned long long (function template )
<a href="#">strtoul</a>	Convert string to unsigned long integer (function )

## /string/stoull

function template

### std::stoull

<string>

```
unsigned long long stoull (const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull (const wstring& str, size_t* idx = 0, int base = 10);
```

#### Convert string to unsigned long long

Parses *str* interpreting its content as an integral number of the specified *base*, which is returned as a value of type *unsigned long long*.

If *idx* is not a null pointer, the function also sets the value of *idx* to the position of the first character in *str* after the number.

The function uses `strtoull` (or `wcstoull`) to perform the conversion (see `strtol` for more details on the process).

## Parameters

<code>str</code>	String object with the representation of an integral number.
<code>idx</code>	Pointer to an object of type <code>size_t</code> , whose value is set by the function to position of the next character in <code>str</code> after the numerical value. This parameter can also be a null pointer, in which case it is not used.
<code>base</code>	Numerical base (radix) that determines the valid characters and their interpretation. If this is 0, the base used is determined by the format in the sequence (see <code>strtol</code> for details). Notice that by default this argument is 10, not 0.

## Return Value

On success, the function returns the converted integral number as a value of type `unsigned long long`.

## Example

```
1 // stoull example
2 #include <iostream>    // std::cout
3 #include <string>      // std::string, std::stoull
4
5 int main ()
6 {
7     std::string str = "8246821 0xfffff 020 -1";
8
9     std::string::size_type sz = 0;    // alias of size_t
10
11    while (!str.empty()) {
12        unsigned long long ull = std::stoull (str,&sz,0);
13        std::cout << str.substr(0,sz) << " interpreted as " << ull << '\n';
14        str = str.substr(sz);
15    }
16
17    return 0;
18 }
```

Possible output:

```
8246821 interpreted as 8246821
0xfffff interpreted as 65535
020 interpreted as 16
-1 interpreted as 18446744073709551615
```

## Complexity

Unspecified, but generally linear in the number of characters interpreted.

## Data races

Modifies the value pointed by `idx` (if not zero).

## Exceptions

If no conversion could be performed, an `invalid_argument` exception is thrown.

If the value read is out of the range of representable values by a `unsigned long long`, an `out_of_range` exception is thrown.

An invalid `idx` causes *undefined behavior*.

## See also

<code>stoi</code>	Convert string to integer (function template )
<code>stoul</code>	Convert string to unsigned integer (function template )
<code>stoll</code>	Convert string to long long (function template )
<code>strtoull</code>	Convert string to unsigned long long integer (function )

## /string/string

class

### `std::string`

`<string>`

```
typedef basic_string<char> string;
```

#### String class

Strings are objects that represent sequences of characters.

The standard `string` class provides support for such objects with an interface similar to that of a `standard container` of bytes, but adding features specifically designed to operate with strings of single-byte characters.

The `string` class is an instantiation of the `basic_string` class template that uses `char` (i.e., bytes) as its *character type*, with its default `char_traits` and `allocator` types (see `basic_string` for more info on the template).

Note that this class handles bytes independently of the encoding used: If used to handle sequences of multi-byte or variable-length characters (such as UTF-8), all members of this class (such as `length` or `size`), as well as its iterators, will still operate in terms of bytes (not actual encoded characters).

## Member types

member type	definition
<code>value_type</code>	<code>char</code>
<code>traits_type</code>	<code>char_traits&lt;char&gt;</code>
<code>allocator_type</code>	<code>allocator&lt;char&gt;</code>
<code>reference</code>	<code>char&amp;</code>
<code>const_reference</code>	<code>const char&amp;</code>
<code>pointer</code>	<code>char*</code>
<code>const_pointer</code>	<code>const char*</code>
<code>iterator</code>	a random access iterator to <code>char</code> (convertible to <code>const_iterator</code> )
<code>const_iterator</code>	a random access iterator to <code>const char</code>
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>
<code>difference_type</code>	<code>ptrdiff_t</code>
<code>size_type</code>	<code>size_t</code>

## Member functions

<b>(constructor)</b>	Construct string object (public member function )
<b>(destructor)</b>	String destructor (public member function )
<b>operator=</b>	String assignment (public member function )

### Iterators:

<code>begin</code>	Return iterator to beginning (public member function )
<code>end</code>	Return iterator to end (public member function )
<code>rbegin</code>	Return reverse iterator to reverse beginning (public member function )
<code>rend</code>	Return reverse iterator to reverse end (public member function )
<code>cbegin</code>	Return const_iterator to beginning (public member function )
<code>cend</code>	Return const_iterator to end (public member function )
<code>crbegin</code>	Return const_reverse_iterator to reverse beginning (public member function )
<code>crend</code>	Return const_reverse_iterator to reverse end (public member function )

### Capacity:

<code>size</code>	Return length of string (public member function )
<code>length</code>	Return length of string (public member function )
<code>max_size</code>	Return maximum size of string (public member function )
<code>resize</code>	Resize string (public member function )
<code>capacity</code>	Return size of allocated storage (public member function )
<code>reserve</code>	Request a change in capacity (public member function )
<code>clear</code>	Clear string (public member function )
<code>empty</code>	Test if string is empty (public member function )
<code>shrink_to_fit</code>	Shrink to fit (public member function )

### Element access:

<b>operator[]</b>	Get character of string (public member function )
<b>at</b>	Get character in string (public member function )
<b>back</b>	Access last character (public member function )
<b>front</b>	Access first character (public member function )

### Modifiers:

<b>operator+=</b>	Append to string (public member function )
<b>append</b>	Append to string (public member function )
<b>push_back</b>	Append character to string (public member function )
<b>assign</b>	Assign content to string (public member function )
<b>insert</b>	Insert into string (public member function )
<b>erase</b>	Erase characters from string (public member function )
<b>replace</b>	Replace portion of string (public member function )
<b>swap</b>	Swap string values (public member function )
<b>pop_back</b>	Delete last character (public member function )

### String operations:

<code>c_str</code>	Get C string equivalent (public member function )
<code>data</code>	Get string data (public member function )
<code>get_allocator</code>	Get allocator (public member function )
<code>copy</code>	Copy sequence of characters from string (public member function )
<code>find</code>	Find content in string (public member function )
<code>rfind</code>	Find last occurrence of content in string (public member function )
<code>find_first_of</code>	Find character in string (public member function )

<b>find_last_of</b>	Find character in string from the end (public member function )
<b>find_first_not_of</b>	Find absence of character in string (public member function )
<b>find_last_not_of</b>	Find non-matching character in string from the end (public member function )
<b>substr</b>	Generate substring (public member function )
<b>compare</b>	Compare strings (public member function )

## Member constants

<b>npos</b>	Maximum value for size_t (public static member constant )
-------------	---

## Non-member function overloads

<b>operator+</b>	Concatenate strings (function )
<b>relational operators</b>	Relational operators for string (function )
<b>swap</b>	Exchanges the values of two strings (function )
<b>operator&gt;&gt;</b>	Extract string from stream (function )
<b>operator&lt;&lt;</b>	Insert string into stream (function )
<b>getline</b>	Get line from stream into string (function )

# /string/string/append

public member function

## std::string::append

<string>

```

string (1) string& append (const string& str);
substring (2) string& append (const string& str, size_t subpos, size_t sublen);
c-string (3) string& append (const char* s);
buffer (4) string& append (const char* s, size_t n);
fill (5) string& append (size_t n, char c);
range (6) template <class InputIterator>
           string& append (InputIterator first, InputIterator last);

string (1) string& append (const string& str);
substring (2) string& append (const string& str, size_t subpos, size_t sublen);
c-string (3) string& append (const char* s);
buffer (4) string& append (const char* s, size_t n);
fill (5) string& append (size_t n, char c);
range (6) template <class InputIterator>
           string& append (InputIterator first, InputIterator last);
initializer list(7) string& append (initializer_list<char> il);

string (1) string& append (const string& str);
substring (2) string& append (const string& str, size_t subpos, size_t sublen = npos);
c-string (3) string& append (const char* s);
buffer (4) string& append (const char* s, size_t n);
fill (5) string& append (size_t n, char c);
range (6) template <class InputIterator>
           string& append (InputIterator first, InputIterator last);
initializer list(7) string& append (initializer_list<char> il);

```

### Append to string

Extends the `string` by appending additional characters at the end of its current value:

#### (1) `string`

Appends a copy of `str`.

#### (2) `substring`

Appends a copy of a substring of `str`. The substring is the portion of `str` that begins at the character position `subpos` and spans `sublen` characters (or until the end of `str`, if either `str` is too short or if `sublen` is `string::npos`).

#### (3) `c-string`

Appends a copy of the string formed by the null-terminated character sequence (C-string) pointed by `s`.

#### (4) `buffer`

Appends a copy of the first `n` characters in the array of characters pointed by `s`.

#### (5) `fill`

Appends `n` consecutive copies of character `c`.

#### (6) `range`

Appends a copy of the sequence of characters in the range `[first, last)`, in the same order.

#### (7) `initializer list`

Appends a copy of each of the characters in `il`, in the same order.

## Parameters

### `str`

Another `string` object, whose value is appended.

### `subpos`

Position of the first character in `str` that is copied to the object as a substring.

If this is greater than `str`'s `length`, it throws `out_of_range`.

Note: The first character in `str` is denoted by a value of 0 (not 1).

**sublen**  
Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).  
A value of `string::npos` indicates all characters until the end of `str`.

**s**  
Pointer to an array of characters (such as a *c-string*).

**n**  
Number of characters to copy.

**c**  
Character value, repeated *n* times.

**first, last**  
Input iterators to the initial and final positions in a range. The range used is `[first, last)`, which includes all the characters between `first` and `last`, including the character pointed by `first` but not the character pointed by `last`.  
The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type convertible to `char`.  
If `InputIterator` is an integral type, the arguments are casted to the proper types so that signature (5) is used instead.

**il**  
An `initializer_list` object.  
These objects are automatically constructed from *initializer list* declarators.

`size_t` is an unsigned integral type.

## Return Value

`*this`

## Example

```

1 // appending to string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str;
8     std::string str2="Writing ";
9     std::string str3="print 10 and then 5 more";
10
11    // used in the same order as described above:
12    str.append(str2);           // "Writing "
13    str.append(str3,6,3);       // "10 "
14    str.append("dots are cool",5); // "dots "
15    str.append("here: ");       // "here: "
16    str.append(10u,'.');// "....."
17    str.append(str3.begin()+8,str3.end()); // " and then 5 more"
18    str.append<int>(5,0x2B); // "...."
19
20    std::cout << str << '\n';
21    return 0;
22 }
```

Output:

Writing 10 dots here: ..... and then 5 more.....

## Complexity

Unspecified, but generally up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `string`.

If `s` does not point to an array long enough, or if the range specified by `[first,last)` is not valid, it causes *undefined behavior*.

If `subpos` is greater than `str`'s `length`, an `out_of_range` exception is thrown.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

A `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>string::operator+=</code>	Append to string (public member function )
<code>string::assign</code>	Assign content to string (public member function )
<code>string::insert</code>	Insert into string (public member function )
<code>string::replace</code>	Replace portion of string (public member function )

# /string/string/assign

public member function

## std::string::assign

<string>

```
string(1) string& assign (const string& str);
substring(2) string& assign (const string& str, size_t subpos, size_t sublen);
c-string(3) string& assign (const char* s);
buffer(4) string& assign (const char* s, size_t n);
fill(5) string& assign (size_t n, char c);
range(6) template <class InputIterator>
          string& assign (InputIterator first, InputIterator last);

string(1) string& assign (const string& str);
substring(2) string& assign (const string& str, size_t subpos, size_t sublen);
c-string(3) string& assign (const char* s);
buffer(4) string& assign (const char* s, size_t n);
fill(5) string& assign (size_t n, char c);
range(6) template <class InputIterator>
          string& assign (InputIterator first, InputIterator last);
initializer list(7) string& assign (initializer_list<char> il);
move(8) string& assign (string&& str) noexcept;

string(1) string& assign (const string& str);
substring(2) string& assign (const string& str, size_t subpos, size_t sublen = npos);
c-string(3) string& assign (const char* s);
buffer(4) string& assign (const char* s, size_t n);
fill(5) string& assign (size_t n, char c);
range(6) template <class InputIterator>
          string& assign (InputIterator first, InputIterator last);
initializer list(7) string& assign (initializer_list<char> il);
move(8) string& assign (string&& str) noexcept;
```

### Assign content to string

Assigns a new value to the string, replacing its current contents.

#### (1) string

Copies *str*.

#### (2) substring

Copies the portion of *str* that begins at the character position *subpos* and spans *sublen* characters (or until the end of *str*, if either *str* is too short or if *sublen* is `string::npos`).

#### (3) c-string

Copies the null-terminated character sequence (C-string) pointed by *s*.

#### (4) buffer

Copies the first *n* characters from the array of characters pointed by *s*.

#### (5) fill

Replaces the current value by *n* consecutive copies of character *c*.

#### (6) range

Copies the sequence of characters in the range `[first, last)`, in the same order.

#### (7) initializer list

Copies each of the characters in *il*, in the same order.

#### (8) move

Acquires the contents of *str*.

*str* is left in an unspecified but valid state.

## Parameters

*str*

Another `string` object, whose value is either copied or moved.

*subpos*

Position of the first character in *str* that is copied to the object as a substring.

If this is greater than *str*'s `length`, it throws `out_of_range`.

Note: The first character in *str* is denoted by a value of 0 (not 1).

*sublen*

Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).

A value of `string::npos` indicates all characters until the end of *str*.

*s*

Pointer to an array of characters (such as a `c-string`).

*n*

Number of characters to copy.

*c*

Character value, repeated *n* times.

*first*, *last*

`Input iterators` to the initial and final positions in a range. The range used is `[first, last)`, which includes all the characters between *first* and *last*, including the character pointed by *first* but not the character pointed by *last*.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type convertible to `char`.

If `InputIterator` is an integral type, the arguments are casted to the proper types so that signature (5) is used instead.

*il*

An `initializer_list` object.

These objects are automatically constructed from `initializer_list` declarators.

`size_t` is an unsigned integral type.

## Return Value

\*this

## Example

```
1 // string::assign
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str;
8     std::string base="The quick brown fox jumps over a lazy dog.";
9
10    // used in the same order as described above:
11
12    str.assign(base);
13    std::cout << str << '\n';
14
15    str.assign(base,10,9);
16    std::cout << str << '\n';           // "brown fox"
17
18    str.assign("pangrams are cool",7);
19    std::cout << str << '\n';           // "pangram"
20
21    str.assign("c-string");
22    std::cout << str << '\n';           // "c-string"
23
24    str.assign(10,'*');
25    std::cout << str << '\n';           // "*****"
26
27    str.assign<int>(10,0x2D);
28    std::cout << str << '\n';           // "-----"
29
30    str.assign(base.begin()+16,base.end()-12);
31    std::cout << str << '\n';           // "fox jumps over"
32
33    return 0;
34 }
```

Output:

```
The quick brown fox jumps over a lazy dog.
brown fox
pangram
c-string
*****
-----
fox jumps over
```

## Complexity

Unspecified.

Unspecified, but generally linear in the new [string length](#) (and constant for the *move version*).

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

The *move assign* form (8), modifies *str*.

## Exception safety

For the *move assign* (8), the function does not throw exceptions (no-throw guarantee).

In all other cases, there are no effects in case an exception is thrown (strong guarantee).

If *s* does not point to an array long enough, or if the range specified by [*first*,*last*) is not valid, it causes *undefined behavior*.

If *subpos* is greater than *str*'s *length*, an *out\_of\_range* exception is thrown.

If the resulting *string* length would exceed the *max\_size*, a *length\_error* exception is thrown.

A *bad\_alloc* exception is thrown if the function needs to allocate storage and fails.

## See also

<a href="#">string::operator=</a>	String assignment ( <a href="#">public member function</a> )
<a href="#">string::append</a>	Append to string ( <a href="#">public member function</a> )
<a href="#">string::insert</a>	Insert into string ( <a href="#">public member function</a> )
<a href="#">string::replace</a>	Replace portion of string ( <a href="#">public member function</a> )

## /string/string/at

public member function

**std::string::at**

<string>

```
char& at (size_t pos);
const char& at (size_t pos) const;
```

### Get character in string

Returns a reference to the character at position *pos* in the [string](#).

The function automatically checks whether *pos* is the valid position of a character in the string (i.e., whether *pos* is less than the [string length](#)), throwing an [out\\_of\\_range](#) exception if it is not.

### Parameters

*pos*

Value with the position of a character within the string.

Note: The first character in a [string](#) is denoted by a value of 0 (not 1).

If it is not the position of a character, an [out\\_of\\_range](#) exception is thrown.

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

### Return value

The character at the specified position in the string.

If the [string](#) object is const-qualified, the function returns a [const char&](#). Otherwise, it returns a [char&](#).

### Example

```
1 // string::at
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for (unsigned i=0; i<str.length(); ++i)
9     {
10         std::cout << str.at(i);
11     }
12     return 0;
13 }
```

This code prints out the content of a string character by character using the [at](#) member function:

```
Test string
```

### Complexity

Unspecified.

### Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

### Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The reference returned can be used to access or modify characters.

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The object is accessed (neither the const nor the non-const versions modify it).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [string](#).

If *pos* is not less than the [string length](#), an [out\\_of\\_range](#) exception is thrown.

### See also

<a href="#">string::operator[]</a>	Get character of string ( <a href="#">public member function</a> )
<a href="#">string::substr</a>	Generate substring ( <a href="#">public member function</a> )
<a href="#">string::find</a>	Find content in string ( <a href="#">public member function</a> )
<a href="#">string::replace</a>	Replace portion of string ( <a href="#">public member function</a> )

## /string/string/back

public member function

### std::string::back

<string>

```
char& back();
const char& back() const;
```

#### Access last character

Returns a reference to the last character of the string.

This function shall not be called on [empty](#) strings.

#### Parameters

none

#### Return value

A reference to the last character in the string.

If the string object is const-qualified, the function returns a const char&. Otherwise, it returns a char&.

#### Example

```
1 // string::back
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("hello world.");
8     str.back() = '!';
9     std::cout << str << '\n';
10    return 0;
11 }
```

Output:

```
hello world!
```

#### Complexity

Constant.

#### Iterator validity

No changes.

#### Data races

The container is accessed (neither the const nor the non-const versions modify the container).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

#### Exception safety

If the string is not [empty](#), the function never throws exceptions (no-throw guarantee).

Otherwise, it causes [undefined behavior](#).

#### See also

<a href="#">string::front</a>	Access first character ( <a href="#">public member function</a> )
<a href="#">string::push_back</a>	Append character to string ( <a href="#">public member function</a> )
<a href="#">string::pop_back</a>	Delete last character ( <a href="#">public member function</a> )
<a href="#">string::at</a>	Get character in string ( <a href="#">public member function</a> )
<a href="#">string::operator[]</a>	Get character of string ( <a href="#">public member function</a> )

## /string/string/begin

public member function

### std::string::begin

<string>

```
iterator begin();
const_iterator begin() const;
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

#### Return iterator to beginning

Returns an iterator pointing to the first character of the string.

#### Parameters

none

## Return Value

An iterator to the beginning of the string.

If the `string` object is const-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are [random access iterator](#) types (pointing to a character and to a const character, respectively).

## Example

```
1 // string::begin/end
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for ( std::string::iterator it=str.begin(); it!=str.end(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
Test string
```

## Complexity

Unspecified.

## Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

## Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed (neither the const nor the non-const versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>string::end</code>	Return iterator to end ( <a href="#">public member function</a> )
<code>string::rbegin</code>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )

# /string/string/capacity

public member function

## std::string::capacity

`<string>`

```
size_t capacity() const;
size_t capacity() const noexcept;
```

### Return size of allocated storage

Returns the size of the storage space currently allocated for the `string`, expressed in terms of bytes.

This `capacity` is not necessarily equal to the `string length`. It can be equal or greater, with the extra space allowing the object to optimize its operations when new characters are added to the `string`.

Notice that this `capacity` does not suppose a limit on the `length` of the `string`. When this `capacity` is exhausted and more is needed, it is automatically expanded by the object (reallocating its storage space). The theoretical limit on the `length` of a `string` is given by member `max_size`.

The *capacity* of a [string](#) can be altered any time the object is modified, even if this modification implies a reduction in size or if the capacity has not been exhausted (this is in contrast with the guarantees given to *capacity* in [vector containers](#)).

The *capacity* of a [string](#) can be explicitly altered by calling member [reserve](#).

## Parameters

none

## Return Value

The size of the storage capacity currently allocated for the [string](#).

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Example

```
1 // comparing size, length, capacity and max_size
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "size: " << str.size() << "\n";
9     std::cout << "length: " << str.length() << "\n";
10    std::cout << "capacity: " << str.capacity() << "\n";
11    std::cout << "max_size: " << str.max_size() << "\n";
12    return 0;
13 }
```

A possible output for this program could be:

```
size: 11
length: 11
capacity: 15
max_size: 429496729
```

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">string::reserve</a>	Request a change in capacity ( <a href="#">public member function</a> )
<a href="#">string::length</a>	Return length of string ( <a href="#">public member function</a> )
<a href="#">string::size</a>	Return length of string ( <a href="#">public member function</a> )
<a href="#">string::max_size</a>	Return maximum size of string ( <a href="#">public member function</a> )

# /string/string/cbegin

public member function

## std::string::cbegin

<string>

```
const_iterator cbegin() const noexcept;
```

### Return const\_iterator to beginning

Returns a [const\\_iterator](#) pointing to the first character of the string.

A [const\\_iterator](#) is an iterator that points to [const](#) content. This iterator can be increased and decreased (unless it is itself [const](#)), just like the iterator returned by [string::begin](#), but it cannot be used to modify the contents it points to, even if the [string](#) object is not [const](#).

## Parameters

none

## Return Value

A [const\\_iterator](#) to the beginning of the string.

Member type [const\\_iterator](#) is a [random access iterator](#) type that points to a [const](#) character.

## Example

```

1 // string::cbegin/cend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Lorem ipsum");
8     for (auto it=str.cbegin(); it!=str.cend(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

LoREM ipsum

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<b>string::begin</b>	Return iterator to beginning (public member function )
<b>string::cend</b>	Return const_iterator to end (public member function )
<b>string::crbegin</b>	Return const_reverse_iterator to reverse beginning (public member function )

## /string/string/cend

public member function

### std::string::cend

<string>

`const_iterator cend() const noexcept;`

#### Return const\_iterator to end

Returns a `const_iterator` pointing to the *past-the-end* character of the string.

A `const_iterator` is an iterator that points to `const` content. This iterator can be increased and decreased (unless it is itself `const`), just like the iterator returned by `string::end`, but it cannot be used to modify the contents it points to, even if the `string` object is not `const`.

If the object is an `empty string`, this function returns the same as `string::cbegin`.

The value returned shall not be dereferenced.

## Parameters

none

## Return Value

A `const_iterator` to the past-the-end of the string.

Member type `const_iterator` is a `random access iterator` type that points to a `const` character.

## Example

```

1 // string::cbegin/cend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("LoREM ipsum");
8     for (auto it=str.cbegin(); it!=str.cend(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

LoREM ipsum

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>string::end</code>	Return iterator to end ( <a href="#">public member function</a> )
<code>string::cbegin</code>	Return const_iterator to beginning ( <a href="#">public member function</a> )
<code>string::crend</code>	Return const_reverse_iterator to reverse end ( <a href="#">public member function</a> )

# /string/string/clear

public member function

<string>

## std::string::clear

```
void clear();
void clear() noexcept;
```

### Clear string

Erases the contents of the [string](#), which becomes an [empty string](#) (with a [length](#) of 0 characters).

## Parameters

none

## Return value

none

## Example

```
1 // string::clear
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     char c;
8     std::string str;
9     std::cout << "Please type some lines of text. Enter a dot (.) to finish:\n";
10    do {
11        c = std::cin.get();
12        str += c;
13        if (c=='\n')
14        {
15            std::cout << str;
16            str.clear();
17        }
18    } while (c!='.');
19    return 0;
20 }
```

This program repeats every line introduced by the user until a line contains a dot ('.'). Every newline character ('\n') triggers the repetition of the line and the clearing of the current string content.

## Complexity

Unspecified, but generally constant.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">string::erase</a>	Erase characters from string (public member function )
<a href="#">string::resize</a>	Resize string (public member function )
<a href="#">string::empty</a>	Test if string is empty (public member function )

# /string/string/compare

public member function

## std::string::compare

<string>

string (1)	int compare (const string& str) const;
	int compare (size_t pos, size_t len, const string& str) const;
substrings (2)	int compare (size_t pos, size_t len, const string& str, size_t subpos, size_t sublen) const;
c-string (3)	int compare (const char* s) const;
	int compare (size_t pos, size_t len, const char* s) const;
buffer (4)	int compare (size_t pos, size_t len, const char* s, size_t n) const;
string (1)	int compare (const string& str) const noexcept;
	int compare (size_t pos, size_t len, const string& str) const;
substrings (2)	int compare (size_t pos, size_t len, const string& str, size_t subpos, size_t sublen) const;
c-string (3)	int compare (const char* s) const;
	int compare (size_t pos, size_t len, const char* s) const;
buffer (4)	int compare (size_t pos, size_t len, const char* s, size_t n) const;
string (1)	int compare (const string& str) const noexcept;
	int compare (size_t pos, size_t len, const string& str) const;
substrings (2)	int compare (size_t pos, size_t len, const string& str, size_t subpos, size_t sublen = npos) const;
c-string (3)	int compare (const char* s) const;
	int compare (size_t pos, size_t len, const char* s) const;
buffer (4)	int compare (size_t pos, size_t len, const char* s, size_t n) const;

### Compare strings

Compares the value of the [string](#) object (or a substring) to the sequence of characters specified by its arguments.

The *compared string* is the value of the [string](#) object or -if the signature used has a *pos* and a *len* parameters- the substring that begins at its character in position *pos* and spans *len* characters.

This string is compared to a *comparing string*, which is determined by the other arguments passed to the function.

### Parameters

str	Another <a href="#">string</a> object, used entirely (or partially) as the <i>comparing string</i> .
pos	Position of the first character in the <i>compared string</i> . If this is greater than the <a href="#">string length</a> , it throws <a href="#">out_of_range</a> . Note: The first character is denoted by a value of 0 (not 1).
len	Length of <i>compared string</i> (if the string is shorter, as many characters as possible). A value of <a href="#">string::npos</a> indicates all characters until the end of the string.
subpos, sublen	Same as <i>pos</i> and <i>len</i> above, but for the <i>comparing string</i> .
s	Pointer to an array of characters. If argument <i>n</i> is specified (4), the first <i>n</i> characters in the array are used as the <i>comparing string</i> . Otherwise (3), a null-terminated sequence is expected: the length of the sequence with the characters to use as <i>comparing string</i> is determined by the first occurrence of a null character.
n	Number of characters to compare.

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

### Return Value

Returns a signed integral indicating the relation between the strings:

value	relation between <i>compared string</i> and <i>comparing string</i>
0	They compare equal
<0	Either the value of the first character that does not match is lower in the <i>compared string</i> , or all compared characters match but the <i>compared string</i> is shorter.
>0	Either the value of the first character that does not match is greater in the <i>compared string</i> , or all compared characters match but the <i>compared string</i> is longer.

### Example

```
1 // comparing apples with apples
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str1 ("green apple");
```

```

8 std::string str2 ("red apple");
9
10 if (str1.compare(str2) != 0)
11   std::cout << str1 << " is not " << str2 << '\n';
12
13 if (str1.compare(6,5,"apple") == 0)
14   std::cout << "still, " << str1 << " is an apple\n";
15
16 if (str2.compare(str2.size()-5,5,"apple") == 0)
17   std::cout << "and " << str2 << " is also an apple\n";
18
19 if (str1.compare(6,5,str2,4,5) == 0)
20   std::cout << "therefore, both are apples\n";
21
22 return 0;
23 }
```

Output:

```
green apple is not red apple
still, green apple is an apple
and red apple is also an apple
therefore, both are apples
```

## Complexity

Unspecified, but generally up to linear in both the *compared* and *comparing string's lengths*.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [string](#) (except (1), which is guaranteed to not throw).

If *s* does not point to an array long enough, it causes *undefined behavior*.

If *pos* is greater than the [string length](#), or if *subpos* is greater than *str*'s [length](#), an [out\\_of\\_range](#) exception is thrown.

## See also

<a href="#">string::find</a>	Find content in string ( <a href="#">public member function</a> )
<a href="#">string::replace</a>	Replace portion of string ( <a href="#">public member function</a> )
<a href="#">string::substr</a>	Generate substring ( <a href="#">public member function</a> )
<a href="#">relational operators (string)</a>	Relational operators for string ( <a href="#">function</a> )

# /string/string/copy

public member function

## std::string::copy

<[string](#)>

```
size_t copy (char* s, size_t len, size_t pos = 0) const;
```

### Copy sequence of characters from string

Copies a substring of the current value of the [string](#) object into the array pointed by *s*. This substring contains the *len* characters that start at position *pos*.

The function does not append a null character at the end of the copied content.

## Parameters

*s*

Pointer to an array of characters.  
The array shall contain enough storage for the copied characters.

*len*

Number of characters to copy (if the string is shorter, as many characters as possible are copied).

*pos*

Position of the first character to be copied.  
If this is greater than the [string length](#), it throws [out\\_of\\_range](#).  
Note: The first character in the [string](#) is denoted by a value of 0 (not 1).

## Return value

The number of characters copied to the array pointed by *s*. This may be equal to *len* or to *length()*-*pos* (if the string value is shorter than *pos+len*).

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Example

```

1 // string::copy
2 #include <iostream>
```

```

3 #include <string>
4
5 int main ()
6 {
7     char buffer[20];
8     std::string str ("Test string...");
9     std::size_t length = str.copy(buffer,6,5);
10    buffer[length]='\0';
11    std::cout << "buffer contains: " << buffer << '\n';
12    return 0;
13 }

```

Output:

```
buffer contains: string
```

## Complexity

Linear in the number of characters copied.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `string`.

If `s` does not point to an array long enough, it causes *undefined behavior*.

If `pos` is greater than the `string length`, an `out_of_range` exception is thrown.

## See also

<code>string::substr</code>	Generate substring ( <a href="#">public member function</a> )
<code>string::assign</code>	Assign content to string ( <a href="#">public member function</a> )
<code>string::c_str</code>	Get C string equivalent ( <a href="#">public member function</a> )
<code>string::replace</code>	Replace portion of string ( <a href="#">public member function</a> )
<code>string::insert</code>	Insert into string ( <a href="#">public member function</a> )
<code>string::append</code>	Append to string ( <a href="#">public member function</a> )

## /string/string/crbegin

public member function

### std::string::crbegin

`<string>`

`const_reverse_iterator crbegin() const noexcept;`

**Return const\_reverse\_iterator to reverse beginning**

Returns a `const_reverse_iterator` pointing to the last character of the string (i.e., its *reverse beginning*).

## Parameters

none

## Return Value

A `const_reverse_iterator` to the *reverse beginning* of the string.

Member type `const_reverse_iterator` is a reverse `random access iterator` type that points to a `const character`.

## Example

```

1 // string::crbegin/crend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("lorem ipsum");
8     for (auto rit=str.crbegin(); rit!=str.crend(); ++rit)
9         std::cout << *rit;
10    std::cout << '\n';
11
12    return 0;
13 }

```

Output:

```
muspi merol
```

## Complexity

Unspecified, but generally constant.

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">string::begin</a>	Return iterator to beginning (public member function )
<a href="#">string::crend</a>	Return const_reverse_iterator to reverse end (public member function )
<a href="#">string::rbegin</a>	Return reverse iterator to reverse beginning (public member function )

## /string/string/crend

public member function

### std::string::crend

<string>

`const_reverse_iterator crend() const noexcept;`

#### Return const\_reverse\_iterator to reverse end

Returns a const\_reverse\_iterator pointing to the theoretical character preceding the first character of the string (which is considered its *reverse end*).

### Parameters

none

### Return Value

A const\_reverse\_iterator to the *reverse end* of the string.

Member type const\_reverse\_iterator is a reverse random access iterator type that points to a const character.

### Example

```
1 // string::crbegin/crend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("lorem ipsum");
8     for (auto rit=str.crbegin(); rit!=str.crend(); ++rit)
9         std::cout << *rit;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
muspi merol
```

### Complexity

Unspecified, but generally constant.

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### See also

<a href="#">string::begin</a>	Return iterator to beginning (public member function )
<a href="#">string::crend</a>	Return const_reverse_iterator to reverse end (public member function )
<a href="#">string::rbegin</a>	Return reverse iterator to reverse beginning (public member function )

# /string/string/c\_str

public member function

## std::string::c\_str

<string>

```
const char* c_str() const;
const char* c_str() const noexcept;
```

### Get C string equivalent

Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the `string` object.

This array includes the same sequence of characters that make up the value of the `string` object plus an additional terminating null-character ('\0') at the end.

A program shall not alter any of the characters in this sequence.

The pointer returned points to the internal array currently used by the `string` object to store the characters that conform its value.

Both `string::data` and `string::c_str` are synonyms and return the same value.

The pointer returned may be invalidated by further calls to other member functions that modify the object.

### Parameters

none

### Return Value

A pointer to the c-string representation of the `string` object's value.

### Example

```
1 // strings and c-strings
2 #include <iostream>
3 #include <cstring>
4 #include <string>
5
6 int main ()
7 {
8     std::string str ("Please split this sentence into tokens");
9
10    char * cstr = new char [str.length()+1];
11    std::strcpy (cstr, str.c_str());
12
13    // cstr now contains a c-string copy of str
14
15    char * p = std::strtok (cstr, " ");
16    while (p!=0)
17    {
18        std::cout << p << '\n';
19        p = std::strtok(NULL, " ");
20    }
21
22    delete[] cstr;
23    return 0;
24 }
```

Output:

```
Please
split
this
sentence
into
tokens
```

### Complexity, iterator, access, exceptions

Unspecified or contradictory specifications.

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

### See also

[string::data](#)

Get string data (public member function )

<b>string::copy</b>	Copy sequence of characters from string (public member function )
<b>string::operator[]</b>	Get character of string (public member function )
<b>string::front</b>	Access first character (public member function )

## /string/string/data

public member function

### std::string::data

<string>

```
const char* data() const;
const char* data() const noexcept;
```

#### Get string data

Returns a pointer to an array that contains the same sequence of characters as the characters that make up the value of the [string](#) object.

Accessing the value at `data() + size()` produces *undefined behavior*: There are no guarantees that a null character terminates the character sequence pointed by the value returned by this function. See [string::c\\_str](#) for a function that provides such guarantee.

A program shall not alter any of the characters in this sequence.

Returns a pointer to an array that contains a null-terminated sequence of characters (i.e., a C-string) representing the current value of the [string](#) object.

This array includes the same sequence of characters that make up the value of the [string](#) object plus an additional terminating null-character ('\0') at the end.

The pointer returned points to the internal array currently used by the [string](#) object to store the characters that conform its value.

Both `string::data` and `string::c_str` are synonyms and return the same value.

The pointer returned may be invalidated by further calls to other member functions that modify the object.

#### Parameters

none

#### Return Value

A pointer to the c-string representation of the [string](#) object's value.

#### Example

```
1 // string::data
2 #include <iostream>
3 #include <string>
4 #include <cstring>
5
6 int main ()
7 {
8     int length;
9
10    std::string str = "Test string";
11    char* cstr = "Test string";
12
13    if ( str.length() == std::strlen(cstr) )
14    {
15        std::cout << "str and cstr have the same length.\n";
16
17        if ( memcmp (cstr, str.data(), str.length() ) == 0 )
18            std::cout << "str and cstr have the same content.\n";
19    }
20    return 0;
21 }
```

Output:

```
str and cstr have the same length.
str and cstr have the same content.
```

#### Complexity, iterator, access, exceptions

Unspecified or contradictory specifications.

#### Complexity

Constant.

#### Iterator validity

No changes.

#### Data races

The object is accessed.

#### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">string::c_str</a>	Get C string equivalent (public member function )
<a href="#">string::copy</a>	Copy sequence of characters from string (public member function )
<a href="#">string::operator[]</a>	Get character of string (public member function )
<a href="#">string::front</a>	Access first character (public member function )

## /string/string/empty

public member function

### std::string::empty

<string>

```
bool empty() const;
bool empty() const noexcept;
```

#### Test if string is empty

Returns whether the `string` is empty (i.e. whether its `length` is 0).

This function does not modify the value of the string in any way. To clear the content of a `string`, see `string::clear`.

## Parameters

none

## Return Value

true if the `string` length is 0, false otherwise.

## Example

```
1 // string::empty
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string content;
8     std::string line;
9     std::cout << "Please introduce a text. Enter an empty line to finish:\n";
10    do {
11        getline(std::cin,line);
12        content += line + '\n';
13    } while (!line.empty());
14    std::cout << "The text you introduced was:\n" << content;
15    return 0;
16 }
```

This program reads the user input line by line and stores it into string `content` until an empty line is introduced.

## Complexity

Unspecified, but generally constant.

Constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">string::clear</a>	Clear string (public member function )
<a href="#">string::size</a>	Return length of string (public member function )
<a href="#">string::length</a>	Return length of string (public member function )

## /string/string/end

public member function

### std::string::end

<string>

```
iterator end();
const_iterator end() const;
iterator end() noexcept;
const_iterator end() const noexcept;
```

## **Return iterator to end**

Returns an iterator pointing to the *past-the-end* character of the string.

The *past-the-end* character is a theoretical character that would follow the last character in the string. It shall not be dereferenced.

Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with `string::begin` to specify a range including all the characters in the string.

If the object is an [empty string](#), this function returns the same as `string::begin`.

### **Parameters**

none

### **Return Value**

An iterator to the *past-the-end* of the string.

If the `string` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `iterator` and `const_iterator` are [random access iterator](#) types (pointing to a character and to a `const` character, respectively).

### **Example**

```
1 // string::begin/end
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for ( std::string::iterator it=str.begin(); it!=str.end(); ++it)
9         std::cout << *it;
10    std::cout << '\n';
11
12    return 0;
13 }
```

Output:

```
Test string
```

### **Complexity**

Unspecified.

### **Iterator validity**

Generally, no changes.

On some implementations, the non-`const` version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

### **Data races**

The object is accessed, and in some implementations, the non-`const` version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

### **Complexity**

Unspecified, but generally constant.

### **Iterator validity**

No changes.

### **Data races**

The object is accessed (neither the `const` nor the non-`const` versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

### **Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

### **See also**

<a href="#">string::begin</a>	Return iterator to beginning ( <a href="#">public member function</a> )
<a href="#">string::rbegin</a>	Return reverse iterator to reverse beginning ( <a href="#">public member function</a> )
<a href="#">string::rend</a>	Return reverse iterator to reverse end ( <a href="#">public member function</a> )

## **/string/string/erase**

public member function

## std::string::erase

&lt;string&gt;

```
sequence (1) string& erase (size_t pos = 0, size_t len = npos);
character (2) iterator erase (iterator p);
range (3)     iterator erase (iterator first, iterator last);

sequence (1) string& erase (size_t pos = 0, size_t len = npos);
character (2) iterator erase (const_iterator p);
range (3)     iterator erase (const_iterator first, const_iterator last);
```

### Erase characters from string

Erases part of the [string](#), reducing its length:

#### (1) sequence

Erases the portion of the string value that begins at the character position *pos* and spans *len* characters (or until the *end of the string*, if either the content is too short or if *len* is `string::npos`.  
Notice that the default argument erases all characters in the string (like member function `clear`).

#### (2) character

Erases the character pointed by *p*.

#### (3) range

Erases the sequence of characters in the range [*first*,*last*].

## Parameters

*pos*

Position of the first character to be erased.  
If this is greater than the [string length](#), it throws `out_of_range`.  
Note: The first character in *str* is denoted by a value of 0 (not 1).

*len*

Number of characters to erase (if the string is shorter, as many characters as possible are erased).  
A value of `string::npos` indicates all characters until the end of the string.

*p*

Iterator to the character to be removed.

*first*, *last*

Iterators specifying a range within the [string](#)] to be removed: [*first*,*last*). i.e., the range includes all the characters between *first* and *last*, including the character pointed by *first* but not the one pointed by *last*.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

Member types `iterator` and `const_iterator` are [random access iterator](#) types that point to characters of the [string](#).

## Return value

The *sequence version (1)* returns `*this`.

The others return an iterator referring to the character that now occupies the position of the first character erased, or `string::end` if no such character exists.

Member type `iterator` is a [random access iterator](#) type that points to characters of the [string](#).

## Example

```
1 // string::erase
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("This is an example sentence.");
8     std::cout << str << '\n';
9     // "This is an example sentence."
10    str.erase (10,8);
11    std::cout << str << '\n';
12    // "This is an sentence."
13    str.erase (str.begin() + 9);
14    std::cout << str << '\n';
15    // "This is a sentence."
16    str.erase (str.begin() + 5, str.end() - 9);
17    std::cout << str << '\n';
18    // "This sentence."
19
20 }
```

Output:

```
This is an example sentence.
This is an sentence.
This is a sentence.
This sentence.
```

## Complexity

Unspecified, but generally up to linear in the new [string](#) length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `string`.

If `pos` is greater than the `string length`, an `out_of_range` exception is thrown.

An invalid `p` in (2), or an invalid range in (3), causes *undefined behavior*.

For (1) and (3), if an exception is thrown, there are no changes in the `string` (strong guarantee).  
For (2), it never throws exceptions (no-throw guarantee).

If `pos` is greater than the `string length`, an `out_of_range` exception is thrown.

An invalid range in (3), causes *undefined behavior*.

## See also

<code>string::clear</code>	Clear string (public member function)
<code>string::replace</code>	Replace portion of string (public member function)
<code>string::insert</code>	Insert into string (public member function)
<code>string::assign</code>	Assign content to string (public member function)
<code>string::append</code>	Append to string (public member function)

# /string/string/find

public member function

## std::string::find

<string>

```
string (1) size_t find (const string& str, size_t pos = 0) const;
C-string (2) size_t find (const char* s, size_t pos = 0) const;
buffer (3) size_t find (const char* s, size_t pos, size_t n) const;
character (4) size_t find (char c, size_t pos = 0) const;

string (1) size_t find (const string& str, size_t pos = 0) const noexcept;
C-string (2) size_t find (const char* s, size_t pos = 0) const;
buffer (3) size_t find (const char* s, size_t pos, size_type n) const;
character (4) size_t find (char c, size_t pos = 0) const noexcept;
```

### Find content in string

Searches the `string` for the first occurrence of the sequence specified by its arguments.

When `pos` is specified, the search only includes characters at or after position `pos`, ignoring any possible occurrences that include characters before `pos`.

Notice that unlike member `find_first_of`, whenever more than one character is being searched for, it is not enough that just one of these characters match, but the entire sequence must match.

## Parameters

<code>str</code>	Another <code>string</code> with the subject to search for.
<code>pos</code>	Position of the first character in the string to be considered in the search. If this is greater than the <code>string length</code> , the function never finds matches. Note: The first character is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.
<code>s</code>	Pointer to an array of characters. If argument <code>n</code> is specified (3), the sequence to match are the first <code>n</code> characters in the array. Otherwise (2), a null-terminated sequence is expected: the length of the sequence to match is determined by the first occurrence of a null character.
<code>n</code>	Length of sequence of characters to match.
<code>c</code>	Individual character to be searched for.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

## Return Value

The position of the first character of the first match.

If no matches were found, the function returns `string::npos`.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

## Example

```
1 // string::find
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4
5 int main ()
6 {
7     std::string str ("There are two needles in this haystack with needles.");
```

```

8 std::string str2 ("needle");
9
10 // different member versions of find in the same order as above:
11 std::size_t found = str.find(str2);
12 if (found!=std::string::npos)
13     std::cout << "first 'needle' found at: " << found << '\n';
14
15 found=str.find("needles are small",found+1,6);
16 if (found!=std::string::npos)
17     std::cout << "second 'needle' found at: " << found << '\n';
18
19 found=str.find("haystack");
20 if (found!=std::string::npos)
21     std::cout << "'haystack' also found at: " << found << '\n';
22
23 found=str.find('.');
24 if (found!=std::string::npos)
25     std::cout << "Period found at: " << found << '\n';
26
27 // let's replace the first needle:
28 str.replace(str.find(str2),str2.length(),"preposition");
29 std::cout << str << '\n';
30
31 return 0;
32 }

```

Notice how parameter *pos* is used to search for a second instance of the same search string. Output:

```

first 'needle' found at: 14
second 'needle' found at: 44
'haystack' also found at: 30
Period found at: 51
There are two prepositions in this haystack with needles.

```

## Complexity

Unspecified, but generally up to linear in `length()-pos` times the length of the sequence to match (worst case).

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If *s* does not point to an array long enough, it causes *undefined behavior*. Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<code>string::rfind</code>	Find last occurrence of content in string (public member function )
<code>string::find_first_of</code>	Find character in string (public member function )
<code>string::find_last_of</code>	Find character in string from the end (public member function )
<code>string::find_first_not_of</code>	Find absence of character in string (public member function )
<code>string::find_last_not_of</code>	Find non-matching character in string from the end (public member function )
<code>string::replace</code>	Replace portion of string (public member function )
<code>string::substr</code>	Generate substring (public member function )

## /string/string/find\_first\_not\_of

public member function

### std::string::find\_first\_not\_of

`<string>`

```

string (1) size_t find_first_not_of (const string& str, size_t pos = 0) const;
c-string (2) size_t find_first_not_of (const char* s, size_t pos = 0) const;
buffer (3) size_t find_first_not_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_first_not_of (char c, size_t pos = 0) const;

string (1) size_t find_first_not_of (const string& str, size_t pos = 0) const noexcept;
c-string (2) size_t find_first_not_of (const char* s, size_t pos = 0) const;
buffer (3) size_t find_first_not_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_first_not_of (char c, size_t pos = 0) const noexcept;

```

#### Find absence of character in string

Searches the `string` for the first character that does not match any of the characters specified in its arguments.

When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences before that character.

## Parameters

`str`

Another `string` with the set of characters to be used in the search.

**pos**  
 Position of the first character in the string to be considered in the search.  
 If this is greater than the [string length](#), the function never finds matches.  
 Note: The first character is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.

**s**  
 Pointer to an array of characters.  
 If argument *n* is specified (3), the first *n* characters in the array are used in the search.  
 Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters used in the search is determined by the first occurrence of a null character.

**n**  
 Number of character values to search for.

**c**  
 Individual character to be searched for.

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

### Return Value

The position of the first character that does not match.  
 If no such characters are found, the function returns [string::npos](#).

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

### Example

```

1 // string::find_first_not_of
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <cstddef>            // std::size_t
5
6 int main ()
7 {
8     std::string str ("look for non-alphabetic characters...");
9
10    std::size_t found = str.find_first_not_of("abcdefghijklmnopqrstuvwxyz ");
11
12    if (found!=std::string::npos)
13    {
14        std::cout << "The first non-alphabetic character is " << str[found];
15        std::cout << " at position " << found << '\n';
16    }
17
18    return 0;
19 }
```

The first non-alphabetic character is - at position 12

### Complexity

Unspecified, but generally up to linear in [length\(\)](#)-pos times the number of characters to match (worst case).

### Iterator validity

No changes.

### Data races

The object is accessed.

### Exception safety

If *s* does not point to an array long enough, it causes *undefined behavior*.  
 Otherwise, the function never throws exceptions (no-throw guarantee).

### See also

<a href="#">string::find</a>	Find content in string (public member function )
<a href="#">string::find_first_of</a>	Find character in string (public member function )
<a href="#">string::find_last_not_of</a>	Find non-matching character in string from the end (public member function )
<a href="#">string::replace</a>	Replace portion of string (public member function )
<a href="#">string::substr</a>	Generate substring (public member function )

## /string/string/find\_first\_of

public member function

### std::string::find\_first\_of

<string>

```

string (1) size_t find_first_of (const string& str, size_t pos = 0) const;
c-string (2) size_t find_first_of (const char* s, size_t pos = 0) const;
buffer (3) size_t find_first_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_first_of (char c, size_t pos = 0) const;

string (1) size_t find_first_of (const string& str, size_t pos = 0) const noexcept;
```

```

c-string (2) size_t find_first_of (const char* s, size_t pos = 0) const;
buffer (3) size_t find_first_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_first_of (char c, size_t pos = 0) const noexcept;

```

## Find character in string

Searches the [string](#) for the first character that matches **any** of the characters specified in its arguments.

When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences before *pos*.

Notice that it is enough for one single character of the sequence to match (not all of them). See [string::find](#) for a function that matches entire sequences.

## Parameters

<i>str</i>	Another <a href="#">string</a> with the characters to search for.
<i>pos</i>	Position of the first character in the string to be considered in the search. If this is greater than the <a href="#">string length</a> , the function never finds matches. Note: The first character is denoted by a value of 0 (not 1): A value of 0 means that the entire string is searched.
<i>s</i>	Pointer to an array of characters. If argument <i>n</i> is specified (3), the first <i>n</i> characters in the array are searched for. Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters to match is determined by the first occurrence of a null character.
<i>n</i>	Number of character values to search for.
<i>c</i>	Individual character to be searched for.

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Return Value

The position of the first character that matches.

If no matches are found, the function returns [string::npos](#).

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Example

```

1 // string::find_first_of
2 #include <iostream>           // std::cout
3 #include <cstring>            // std::string
4 #include <cstddef>             // std::size_t
5
6 int main ()
7 {
8     std::string str ("Please, replace the vowels in this sentence by asterisks.");
9     std::size_t found = str.find_first_of("aeiou");
10    while (found!=std::string::npos)
11    {
12        str[found]='*';
13        found=str.find_first_of("aeiou",found+1);
14    }
15
16    std::cout << str << '\n';
17
18    return 0;
19 }

```

Pl\*\*s\*, r\*p\*pl\*c\* th\* v\*w\*ls \*n th\*s s\*nt\*nc\* by \*st\*r\*sks.

## Complexity

Unspecified, but generally up to linear in [length\(\)](#)-*pos* times the number of characters to match (worst case).

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If *s* does not point to an array long enough, it causes *undefined behavior*.

Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<a href="#">string::find</a>	Find content in string ( <a href="#">public member function</a> )
<a href="#">string::find_last_of</a>	Find character in string from the end ( <a href="#">public member function</a> )
<a href="#">string::find_first_not_of</a>	Find absence of character in string ( <a href="#">public member function</a> )
<a href="#">string::replace</a>	Replace portion of string ( <a href="#">public member function</a> )

## /string/string/find\_last\_not\_of

public member function

### std::string::find\_last\_not\_of

&lt;string&gt;

```

string(1) size_t find_last_not_of (const string& str, size_t pos = npos) const;
c-string(2) size_t find_last_not_of (const char* s, size_t pos = npos) const;
buffer(3) size_t find_last_not_of (const char* s, size_t pos, size_t n) const;
character(4) size_t find_last_not_of (char c, size_t pos = npos) const;

string(1) size_t find_last_not_of (const string& str, size_t pos = npos) const noexcept;
c-string(2) size_t find_last_not_of (const char* s, size_t pos = npos) const;
buffer(3) size_t find_last_not_of (const char* s, size_t pos, size_t n) const;
character(4) size_t find_last_not_of (char c, size_t pos = npos) const noexcept;

```

#### Find non-matching character in string from the end

Searches the [string](#) for the last character that does not match any of the characters specified in its arguments.

When *pos* is specified, the search only includes characters at or before position *pos*, ignoring any possible occurrences after *pos*.

#### Parameters

*str*

Another [string](#) with the set of characters to be used in the search.

*pos*

Position of the last character in the string to be considered in the search.

Any value greater than, or equal to, the [string length](#) (including [string::npos](#)) means that the entire string is searched.

Note: The first character is denoted by a value of 0 (not 1).

*s*

Pointer to an array of characters.

If argument *n* is specified (3), the first *n* characters in the array are used in the search.

Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters used in the search is determined by the first occurrence of a null character.

*n*

Number of character values to search for.

*c*

Individual character to be searched for.

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

#### Return Value

The position of the first character that does not match.

If no such characters are found, the function returns [string::npos](#).

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

#### Example

```

1 // string::find_last_not_of
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <cstddef>            // std::size_t
5
6 int main ()
7 {
8     std::string str ("Please, erase trailing white-spaces  \n");
9     std::string whitespaces (" \t\f\v\n\r");
10
11    std::size_t found = str.find_last_not_of(whitespaces);
12    if (found!=std::string::npos)
13        str.erase(found+1);
14    else
15        str.clear();          // str is all whitespace
16
17    std::cout << '[' << str << "]\n";
18
19    return 0;
20 }

```

[Please, erase trailing white-spaces]

#### Complexity

Unspecified, but generally up to linear in the [string length](#) (or *pos*) times the number of characters to match (worst case).

#### Iterator validity

No changes.

#### Data races

The object is accessed.

## Exception safety

If `s` does not point to an array long enough, it causes *undefined behavior*. Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<code>string::find</code>	Find content in string (public member function )
<code>string::find_last_of</code>	Find character in string from the end (public member function )
<code>string::find_first_not_of</code>	Find absence of character in string (public member function )
<code>string::replace</code>	Replace portion of string (public member function )
<code>string::substr</code>	Generate substring (public member function )

## /string/string/find\_last\_of

public member function

### std::string::find\_last\_of

<string>

```
string (1) size_t find_last_of (const string& str, size_t pos = npos) const;
c-string (2) size_t find_last_of (const char* s, size_t pos = npos) const;
buffer (3) size_t find_last_of (const char* s, size_t pos, size_t n) const;
character (4) size_t find_last_of (char c, size_t pos = npos) const;

string (1) size_t find_last_of (const string& str, size_t pos = npos) const noexcept;
c-string (2) size_t find_last_of (const char* s, size_t pos = npos) const noexcept;
buffer (3) size_t find_last_of (const char* s, size_t pos, size_t n) const noexcept;
character (4) size_t find_last_of (char c, size_t pos = npos) const noexcept;
```

#### Find character in string from the end

Searches the `string` for the last character that matches **any** of the characters specified in its arguments.

When `pos` is specified, the search only includes characters at or before position `pos`, ignoring any possible occurrences after `pos`.

## Parameters

<code>str</code>	Another <code>string</code> with the characters to search for.
<code>pos</code>	Position of the last character in the string to be considered in the search. Any value greater than, or equal to, the <code>string</code> length (including <code>string::npos</code> ) means that the entire string is searched. Note: The first character is denoted by a value of 0 (not 1).
<code>s</code>	Pointer to an array of characters. If argument <code>n</code> is specified (3), the first <code>n</code> characters in the array are searched for. Otherwise (2), a null-terminated sequence is expected: the length of the sequence with the characters to match is determined by the first occurrence of a null character.
<code>n</code>	Number of character values to search for.
<code>c</code>	Individual character to be searched for.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

## Return Value

The position of the last character that matches.

If no matches are found, the function returns `string::npos`.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

## Example

```
1 // string::find_last_of
2 #include <iostream>           // std::cout
3 #include <string>              // std::string
4 #include <cstddef>             // std::size_t
5
6 void SplitFilename (const std::string& str)
7 {
8     std::cout << "Splitting: " << str << '\n';
9     std::size_t found = str.find_last_of("\\\\");
10    std::cout << " path: " << str.substr(0,found) << '\n';
11    std::cout << " file: " << str.substr(found+1) << '\n';
12 }
13
14 int main ()
15 {
16     std::string str1 ("/usr/bin/man");
17     std::string str2 ("c:\\windows\\winhelp.exe");
18
19     SplitFilename (str1);
20     SplitFilename (str2);
```

```
21     return 0;
22 }
23 }
```

```
Splitting: /usr/bin/man
path: /usr/bin
file: man
Splitting: c:\windows\winhelp.exe
path: c:\windows
file: winhelp.exe
```

## Complexity

Unspecified, but generally up to linear in the [string length](#) (or *pos*) times the number of characters to match (worst case).

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If *s* does not point to an array long enough, it causes *undefined behavior*. Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<a href="#">string::find</a>	Find content in string (public member function )
<a href="#">string::rfind</a>	Find last occurrence of content in string (public member function )
<a href="#">string::find_first_of</a>	Find character in string (public member function )
<a href="#">string::find_last_not_of</a>	Find non-matching character in string from the end (public member function )
<a href="#">string::replace</a>	Replace portion of string (public member function )
<a href="#">string::substr</a>	Generate substring (public member function )

# /string/string/front

public member function

## std::string::front

<string>

```
char& front();
const char& front() const;
```

### Access first character

Returns a reference to the first character of the [string](#).

Unlike member [string::begin](#), which returns an iterator to this same character, this function returns a direct reference.

This function shall not be called on [empty strings](#).

## Parameters

none

## Return value

A reference to the first character in the [string](#).

If the [string](#) object is [const](#)-qualified, the function returns a [const char&](#). Otherwise, it returns a [char&](#).

## Example

```
1 // string::front
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("test string");
8     str.front() = 'T';
9     std::cout << str << '\n';
10    return 0;
11 }
```

Output:

```
Test string
```

## Complexity

Constant.

## **Iterator validity**

No changes.

## **Data races**

The container is accessed (neither the const nor the non-const versions modify the container).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## **Exception safety**

If the [string](#) is not [empty](#), the function never throws exceptions (no-throw guarantee).

Otherwise, it causes [undefined behavior](#).

## **See also**

<a href="#">string::back</a>	Access last character (public member function )
<a href="#">string::at</a>	Get character in string (public member function )
<a href="#">string::operator[]</a>	Get character of string (public member function )

# /string/string/get\_allocator

public member function

## **std::string::get\_allocator**

<string>

```
allocator_type get_allocator() const;
allocator_type get_allocator() const noexcept;
```

### **Get allocator**

Returns a copy of the allocator object associated with the [string](#).

[string](#) uses the default [allocator<char>](#) type, which has no state (thus, the value returned is the same as a default-constructed allocator).

## **Parameters**

none

## **Return Value**

An allocator object.

In [string](#), member type [allocator\\_type](#) is an alias of [allocator<char>](#).

## **Complexity**

Unspecified, but generally constant.

## **Iterator validity**

No changes.

## **Data races**

The object is accessed.

## **Exception safety**

**No-throw guarantee:** this member function never throws exceptions.

Copying any instantiation of the [default allocator](#) is also guaranteed to never throw.

## **See also**

<a href="#">allocator</a>	Default allocator (class template )
---------------------------	-------------------------------------

# /string/string/getline

function

## **std::getline (string)**

<string>

```
(1) istream& getline (istream& is, string& str, char delim);
(2) istream& getline (istream& is, string& str);

(1) istream& getline (istream& is, string& str, char delim);
    istream& getline (istream& is, string& str, char delim);
(2) istream& getline (istream& is, string& str);
    istream& getline (istream& is, string& str);
```

### **Get line from stream into string**

Extracts characters from *is* and stores them into *str* until the delimitation character *delim* is found (or the newline character, '\n', for (2)).

The extraction also stops if the end of file is reached in *is* or if some other error occurs during the input operation.

If the delimiter is found, it is extracted and discarded (i.e. it is not stored and the next input operation will begin after it).

Note that any content in *str* before the call is replaced by the newly extracted sequence.

Each extracted character is appended to the *string* as if its member `push_back` was called.

## Parameters

<i>is</i>	istream object from which characters are extracted.
<i>str</i>	string object where the extracted line is stored. The contents in the string before the call (if any) are discarded and replaced by the extracted line.

## Return Value

The same as parameter *is*.

A call to this function may set any of the internal state flags of *is* if:

flag	error
<code>eofbit</code>	The end of the source of characters is reached during its operations.
<code>failbit</code>	The input obtained could not be interpreted as a valid textual representation of an object of this type. In this case, <i>distr</i> preserves the parameters and internal data it had before the call. Notice that some <code>eofbit</code> cases will also set <code>failbit</code> .
<code>badbit</code>	An error other than the above happened. (see <code>ios_base::iostate</code> for more info on these)

Additionally, in any of these cases, if the appropriate flag has been set with *is*'s member function `ios::exceptions`, an exception of type `ios_base::failure` is thrown.

## Example

```
1 // extract to string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string name;
8
9     std::cout << "Please, enter your full name: ";
10    std::getline (std::cin, name);
11    std::cout << "Hello, " << name << "!\n";
12
13    return 0;
14 }
```

## Complexity

Unspecified, but generally linear in the resulting length of *str*.

## Iterator validity

Any iterators, pointers and references related to *str* may be invalidated.

## Data races

Both objects, *is* and *str*, are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, both *is* and *str* end up in a valid state.

## See also

<code>istream::getline</code>	Get line (public member function )
<code>operator&gt;&gt; (string)</code>	Extract string from stream (function )

# /string/string/insert

public member function

## std::string::insert

<string>

```
    string (1)   string& insert (size_t pos, const string& str);
    substring (2) string& insert (size_t pos, const string& str, size_t subpos, size_t sublen);
    c-string (3) string& insert (size_t pos, const char* s);
    buffer (4)   string& insert (size_t pos, const char* s, size_t n);
    fill (5)     string& insert (size_t pos, size_t n, char c);
    single character (6) iterator insert (iterator p, char c);
    range (7)    template <class InputIterator>
                  void insert (iterator p, InputIterator first, InputIterator last);
```

```

string (1) string& insert (size_t pos, const string& str);
substring (2) string& insert (size_t pos, const string& str, size_t subpos, size_t sublen);
c-string (3) string& insert (size_t pos, const char* s);
buffer (4) string& insert (size_t pos, const char* s, size_t n);
fill (5) string& insert (size_t pos, size_t n, char c);
single character (6) iterator insert (const_iterator p, size_t n, char c);
range (7) template <class InputIterator>
            iterator insert (iterator p, InputIterator first, InputIterator last);
initializer list (8) string& insert (const_iterator p, initializer_list<char> il);

string (1) string& insert (size_t pos, const string& str);
substring (2) string& insert (size_t pos, const string& str, size_t subpos, size_t sublen = npos);
c-string (3) string& insert (size_t pos, const char* s);
buffer (4) string& insert (size_t pos, const char* s, size_t n);
fill (5) string& insert (size_t pos, size_t n, char c);
single character (6) iterator insert (const_iterator p, char c);
range (7) template <class InputIterator>
            iterator insert (iterator p, InputIterator first, InputIterator last);
initializer list (8) string& insert (const_iterator p, initializer_list<char> il);

```

## Insert into string

Inserts additional characters into the [string](#) right before the character indicated by *pos* (or *p*):

### (1) [string](#)

Inserts a copy of *str*.

### (2) [substring](#)

Inserts a copy of a substring of *str*. The substring is the portion of *str* that begins at the character position *subpos* and spans *sublen* characters (or until the end of *str*, if either *str* is too short or if *sublen* is [npos](#)).

### (3) [c-string](#)

Inserts a copy of the string formed by the null-terminated character sequence (C-string) pointed by *s*.

### (4) [buffer](#)

Inserts a copy of the first *n* characters in the array of characters pointed by *s*.

### (5) [fill](#)

Inserts *n* consecutive copies of character *c*.

### (6) [single character](#)

Inserts character *c*.

### (7) [range](#)

Inserts a copy of the sequence of characters in the range `[first, last)`, in the same order.

### (8) [initializer list](#)

Inserts a copy of each of the characters in *il*, in the same order.

`size_t` is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Parameters

### pos

Insertion point: The new contents are inserted before the character at position *pos*.  
If this is greater than the object's [length](#), it throws [out\\_of\\_range](#).

Note: The first character is denoted by a value of 0 (not 1).

### str

Another [string](#) object.

### subpos

Position of the first character in *str* that is inserted into the object as a substring.  
If this is greater than *str*'s [length](#), it throws [out\\_of\\_range](#).

Note: The first character in *str* is denoted by a value of 0 (not 1).

### sublen

Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).  
A value of [npos](#) indicates all characters until the end of *str*.

### s

Pointer to an array of characters (such as a [c-string](#)).

### n

Number of characters to insert.

### c

Character value.

### p

Iterator pointing to the insertion point: The new contents are inserted before the character pointed by *p*.  
*iterator* is a member type, defined as a [random access iterator](#) type that points to characters of the [string](#).

### first, last

[Input iterators](#) to the initial and final positions in a range. The range used is `[first, last)`, which includes all the characters between *first* and *last*, including the character pointed by *first* but not the character pointed by *last*.  
The function template argument [InputIterator](#) shall be an [input iterator](#) type that points to elements of a type convertible to [char](#).

### il

An [initializer\\_list](#) object.

These objects are automatically constructed from [initializer list](#) declarators.

## Return value

The signatures returning a reference to [string](#), return `*this`.

Those returning an [iterator](#), return an iterator pointing to the first character inserted.

Member type `iterator` is a [random access iterator](#) type that points to characters of the `string`.

## Example

```
1 // inserting into a string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str="to be question";
8     std::string str2="the ";
9     std::string str3="or not to be";
10    std::string::iterator it;
11
12    // used in the same order as described above:
13    str.insert(6,str2);           // to be (the )question
14    str.insert(6,str3,3,4);      // to be (not )the question
15    str.insert(10,"that is cool",8); // to be not (that is )the question
16    str.insert(10,"to be ");      // to be not (to be )that is the question
17    str.insert(15,1,':');        // to be not to be(:) that is the question
18    it = str.insert(str.begin()+5,' '); // to be(,) not to be: that is the question
19    str.insert (str.end(),3,'.');// to be, not to be: that is the question...
20    str.insert (it+2,str3.begin(),str3.begin()+3); // (or )
21
22    std::cout << str << '\n';
23    return 0;
24 }
```

Output:

```
to be, or not to be: that is the question...
```

## Complexity

Unspecified, but generally up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `string`.

If `s` does not point to an array long enough, or if either `p` or the range specified by `[first,last)` is not valid, it causes *undefined behavior*.

If `pos` is greater than the `string` `length`, or if `subpos` is greater than `str`'s `length`, an `out_of_range` exception is thrown.

If the resulting `string` `length` would exceed the `max_size`, a `length_error` exception is thrown.

A `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>string::append</code>	Append to string ( <a href="#">public member function</a> )
<code>string::replace</code>	Replace portion of string ( <a href="#">public member function</a> )
<code>string::substr</code>	Generate substring ( <a href="#">public member function</a> )
<code>string::operator=</code>	String assignment ( <a href="#">public member function</a> )
<code>string::operator+=</code>	Append to string ( <a href="#">public member function</a> )

## /string/string/length

public member function

### `std::string::length`

`<string>`

```
size_t length() const;
size_t length() const noexcept;
```

#### Return length of string

Returns the length of the string, in terms of bytes.

This is the number of actual bytes that conform the contents of the `string`, which is not necessarily equal to its storage `capacity`.

Note that `string` objects handle bytes without knowledge of the encoding that may eventually be used to encode the characters it contains. Therefore, the value returned may not correspond to the actual number of encoded characters in sequences of multi-byte or variable-length characters (such as UTF-8).

Both `string::size` and `string::length` are synonyms and return the exact same value.

## Parameters

none

## Return Value

The number of bytes in the string.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

## Example

```
1 // string::length
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "The size of str is " << str.length() << " bytes.\n";
9     return 0;
10 }
```

Output:

```
The size of str is 11 bytes
```

## Complexity

Unspecified.

Constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<code>string::size</code>	Return length of string (public member function )
<code>string::resize</code>	Resize string (public member function )
<code>string::max_size</code>	Return maximum size of string (public member function )
<code>string::capacity</code>	Return size of allocated storage (public member function )

# /string/string/max\_size

public member function

## std::string::max\_size

`<string>`

```
size_t max_size() const;
size_t max_size() const noexcept;
```

### Return maximum size of string

Returns the maximum length the `string` can reach.

This is the maximum potential `length` the string can reach due to known system or library implementation limitations, but the object is not guaranteed to be able to reach that length: it can still fail to allocate storage at any point before that length is reached.

## Parameters

none

## Return Value

The maximum length the `string` can reach.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

## Example

```
1 // comparing size, length, capacity and max_size
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "size: " << str.size() << "\n";
9     std::cout << "length: " << str.length() << "\n";
10    std::cout << "capacity: " << str.capacity() << "\n";
11    std::cout << "max_size: " << str.max_size() << "\n";
12    return 0;
13 }
```

A possible output for this program could be:

```
size: 11
length: 11
capacity: 15
max_size: 4294967291
```

## Complexity

Unspecified, but generally constant.  
Constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">string::capacity</a>	Return size of allocated storage (public member function )
<a href="#">string::size</a>	Return length of string (public member function )
<a href="#">string::resize</a>	Resize string (public member function )

## /string/string/npos

public static member constant

<string>

### std::string::npos

`static const size_t npos = -1;`

#### Maximum value for size\_t

`npos` is a static member constant value with the greatest possible value for an element of type `size_t`.

This value, when used as the value for a `len` (or `sublen`) parameter in `string`'s member functions, means "until the end of the string".

As a return value, it is usually used to indicate no matches.

This constant is defined with a value of `-1`, which because `size_t` is an unsigned integral type, it is the largest possible representable value for this type.

## /string/string/operator<<

function

### std::operator<< (string)

<string>

`ostream& operator<< (ostream& os, const string& str);`

#### Insert string into stream

Inserts the sequence of characters that conforms value of `str` into `os`.

This function overloads `operator<<` to behave as described in `ostream::operator<<` for c-strings, but applied to `string` objects.

## Parameters

`os`

ostream object where characters are inserted.

`str`

string object with the content to insert.

## Return Value

The same as parameter `os`.

If some error happens during the output operation, the stream's `badbit` flag is set, and if the appropriate flag has been set with `ios::exceptions`, an exception is thrown.

## Example

```
1 // inserting strings into output streams
2 #include <iostream>
3 #include <string>
4
5 main ()
6 {
7     std::string str = "Hello world!";
```

```

8   std::cout << str << '\n';
9
10 } 
```

## Complexity

Unspecified, but generally linear in `str`'s length.

## Iterator validity

No changes.

## Data races

Objects `os` is modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, both `is` and `str` end up in a valid state.

## See also

<code>ostream::operator&lt;&lt;</code>	Insert formatted output (public member function )
<code>operator&gt;&gt; (string)</code>	Extract string from stream (function )

## /string/string/operator=

public member function

### std::string::operator=

`<string>`

```

string (1) string& operator= (const string& str);
c-string (2) string& operator= (const char* s);
character (3) string& operator= (char c);

string (1) string& operator= (const string& str);
c-string (2) string& operator= (const char* s);
character (3) string& operator= (char c);
initializer list (4) string& operator= (initializer_list<char> il);
move (5) string& operator= (string&& str) noexcept; 
```

#### String assignment

Assigns a new value to the string, replacing its current contents.

(See member function `assign` for additional assignment options).

#### Parameters

<code>str</code>	A <code>string</code> object, whose value is either copied (1) or moved (5) if different from <code>*this</code> (if moved, <code>str</code> is left in an unspecified but valid state).
<code>s</code>	Pointer to a null-terminated sequence of characters. The sequence is copied as the new value for the string.
<code>c</code>	A character. The string value is set to a single copy of this character (the string length becomes 1).
<code>il</code>	An <code>initializer_list</code> object. These objects are automatically constructed from <code>initializer_list</code> declarators. The characters are copied, in the same order.

#### Return Value

`*this`

#### Example

```

1 // string assigning
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str1, str2, str3;
8     str1 = "Test string: ";    // c-string
9     str2 = 'x';              // single character
10    str3 = str1 + str2;      // string
11
12    std::cout << str3 << '\n';
13
14 } 
```

Output: \_\_\_\_\_

Test string: x

## Complexity

Unspecified.

Unspecified, but generally linear in the new `string` length (and constant for the move version).

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

The move assignment (5) modifies `str`.

## Exception safety

For the move assignment (5), the function does not throw exceptions (no-throw guarantee).  
In all other cases, there are no effects in case an exception is thrown (strong guarantee).

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.  
A `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>string::assign</code>	Assign content to string (public member function )
<code>string::operator+=</code>	Append to string (public member function )
<code>string::insert</code>	Insert into string (public member function )
<code>string::replace</code>	Replace portion of string (public member function )
<code>string::string</code>	Construct string object (public member function )
<code>string::compare</code>	Compare strings (public member function )

# /string/string/operator>>

function  
**std::operator>> (string)** <string>  
`istream& operator>> (istream& is, string& str);`

### Extract string from stream

Extracts a string from the input stream `is`, storing the sequence in `str`, which is overwritten (the previous value of `str` is replaced).

This function overloads `operator>>` to behave as described in `istream::operator>>` for c-strings, but applied to `string` objects.

Each extracted character is appended to the `string` as if its member `push_back` was called.

Notice that the `istream` extraction operations use whitespaces as separators; Therefore, this operation will only extract what can be considered a word from the stream. To extract entire lines of text, see the `string` overload of global function `getline`.

## Parameters

`is`      `istream` object from which characters are extracted.  
`str`      `string` object where the extracted content is stored.

## Return Value

The same as parameter `is`.

A call to this function may set any of the internal state flags of `is` if:

flag	error
<code>eofbit</code>	The end of the source of characters is reached during its operations.
<code>failbit</code>	The input obtained could not be interpreted as a valid textual representation of an object of this type. In this case, <code>distr</code> preserves the parameters and internal data it had before the call. Notice that some <code>eofbit</code> cases will also set <code>failbit</code> .
<code>badbit</code>	An error other than the above happened.

(see `ios_base::iostate` for more info on these)

Additionally, in any of these cases, if the appropriate flag has been set with `is`'s member function `ios::exceptions`, an exception of type `ios_base::failure` is thrown.

## Example

```
1 // extract to string
2 #include <iostream>
3 #include <string>
4
5 main ()
6 {
```

```

7 std::string name;
8 std::cout << "Please, enter your name: ";
9 std::cin >> name;
10 std::cout << "Hello, " << name << "!\n";
11
12 return 0;
13 }
14 }
```

## Complexity

Unspecified, but generally linear in the resulting length of *str*.

## Iterator validity

Any iterators, pointers and references related to *str* may be invalidated.

## Data races

Both objects, *is* and *str*, are modified.

## Exception safety

**Basic guarantee:** if an exception is thrown, both *is* and *str* end up in a valid state.

## See also

<a href="#">istream::operator&gt;&gt;</a>	Extract formatted input (public member function )
<a href="#">getline (string)</a>	Get line from stream into string (function )
<a href="#">operator&lt;&lt; (string)</a>	Insert string into stream (function )

## /string/string/operator[]

public member function

### std::string::operator[]

<string>

```
char& operator[] (size_t pos);
const char& operator[] (size_t pos) const;
```

#### Get character of string

Returns a reference to the character at position *pos* in the *string*.

If *pos* is equal to the *string* length and the *string* is const-qualified, the function returns a reference to a null character ('\0').

If *pos* is equal to the *string* length, the function returns a reference to the null character that follows the last character in the string (which should not be modified).

## Parameters

*pos*

Value with the position of a character within the string.

Note: The first character in a *string* is denoted by a value of 0 (not 1).

*size\_t* is an unsigned integral type (the same as member type *string::size\_type*).

## Return value

The character at the specified position in the string.

If the *string* object is const-qualified, the function returns a *const char&*. Otherwise, it returns a *char&*.

## Example

```

1 // string::operator[]
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     for (int i=0; i<str.length(); ++i)
9     {
10         std::cout << str[i];
11     }
12     return 0;
13 }
```

This code prints out the content of a string character by character using the offset operator on *str*:

Test string

## Complexity

Unspecified.

## Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

## Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The reference returned can be used to access or modify characters.

## Exception safety

If *pos* is less than the `string` length, the function never throws exceptions (no-throw guarantee).

If *pos* is equal to the `string` length, the const-version never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

Note that using the reference returned to modify elements that are out of bounds (including the character at *pos*) also causes *undefined behavior*.

## Complexity

Constant.

## Iterator validity

No changes.

## Data races

The object is accessed (neither the const nor the non-const versions modify it).

The reference returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## Exception safety

If *pos* is less or equal to the `string` length, the function never throws exceptions (no-throw guarantee).

Otherwise, it causes *undefined behavior*.

Note that using the reference returned to modify elements that are out of bounds (including the character at *pos*) also causes *undefined behavior*.

## See also

<code>string::at</code>	Get character in string (public member function )
<code>string::substr</code>	Generate substring (public member function )
<code>string::find</code>	Find content in string (public member function )
<code>string::replace</code>	Replace portion of string (public member function )

# /string/string/operator+

function <code>std::operator+ (string)</code>		<string>
string (1)	<code>string operator+ (const string&amp; lhs, const string&amp; rhs);</code>	
c-string (2)	<code>string operator+ (const string&amp; lhs, const char* rhs);</code>	
character (3)	<code>string operator+ (const char* lhs, const string&amp; rhs);</code> <code>string operator+ (const string&amp; lhs, char rhs);</code> <code>string operator+ (char lhs, const string&amp; rhs);</code>	
string (1)	<code>string operator+ (const string&amp; lhs, const string&amp; rhs);</code> <code>string operator+ (string&amp;&amp; lhs, string&amp;&amp; rhs);</code> <code>string operator+ (string&amp;&amp; lhs, const string&amp; rhs);</code> <code>string operator+ (const string&amp; lhs, string&amp;&amp; rhs);</code> <code>string operator+ (const string&amp; lhs, const char* rhs);</code>	
c-string (2)	<code>string operator+ (string&amp;&amp; lhs, const char* rhs);</code> <code>string operator+ (const char* lhs, const string&amp; rhs);</code> <code>string operator+ (const char* lhs, string&amp;&amp; rhs);</code>	
character (3)	<code>string operator+ (const string&amp; lhs, char rhs);</code> <code>string operator+ (string&amp;&amp; lhs, char rhs);</code> <code>string operator+ (char lhs, const string&amp; rhs);</code> <code>string operator+ (char lhs, string&amp;&amp; rhs);</code>	

## Concatenate strings

Returns a newly constructed `string` object with its value being the concatenation of the characters in *lhs* followed by those of *rhs*.

In the signatures taking at least one *rvalue reference* as argument, the returned object is *move-constructed* by passing this argument, which is left in an unspecified but valid state. If both arguments are *rvalue references*, only one of them is moved (it is unspecified which), with the other one preserving its value.

## Parameters

*lhs*, *rhs*

Arguments to the left- and right-hand side of the operator, respectively.

If of type `char*`, it shall point to a null-terminated character sequence.

## Example

```
1 // concatenating strings
2 #include <iostream>
```

```

3 #include <string>
4
5 main ()
6 {
7     std::string firstlevel ("com");
8     std::string secondlevel ("cplusplus");
9     std::string scheme ("http://");
10    std::string hostname;
11    std::string url;
12
13    hostname = "www." + secondlevel + '.' + firstlevel;
14    url = scheme + hostname;
15
16    std::cout << url << '\n';
17
18    return 0;
19 }

```

Output:

`http://www.cppplus.com`

## Return Value

A `string` whose value is the concatenation of *lhs* and *rhs*.

## Complexity

Unspecified, but generally linear in the resulting `string` length (and linear in the length of the non-moved argument for signatures with *rvalue references*).

## Iterator validity

The signatures with *rvalue references* may invalidate iterators, pointers and references related to the moved `string`.

## Data races

The signatures with *rvalue references* modify the moved `string`.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in either `string` objects.

If *s* is not a null-terminated character sequence, it causes *undefined behavior*.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

A `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>string::append</code>	Append to string ( <a href="#">public member function</a> )
<code>string::insert</code>	Insert into string ( <a href="#">public member function</a> )
<code>string::operator+=</code>	Append to string ( <a href="#">public member function</a> )

## /string/string/operator+=

public member function

### std::string::operator+=

`<string>`

<code>string (1)</code>	<code>string&amp; operator+=(const string&amp; str);</code>
<code>c-string (2)</code>	<code>string&amp; operator+=(const char* s);</code>
<code>character (3)</code>	<code>string&amp; operator+=(char c);</code>
<code>string (1)</code>	<code>string&amp; operator+=(const string&amp; str);</code>
<code>c-string (2)</code>	<code>string&amp; operator+=(const char* s);</code>
<code>character (3)</code>	<code>string&amp; operator+=(char c);</code>
<code>initializer list (4)</code>	<code>string&amp; operator+=(initializer_list&lt;char&gt; il);</code>

#### Append to string

Extends the `string` by appending additional characters at the end of its current value:

(See member function `append` for additional appending options).

## Parameters

`str` A `string` object, whose value is copied at the end.

`s` Pointer to a null-terminated sequence of characters.  
The sequence is copied at the end of the string.

`c` A character, which is appended to the current value of the string.

`il` An `initializer_list` object.  
These objects are automatically constructed from `initializer_list` declarators.  
The characters are appended to the string, in the same order.

## Return Value

\*this

## Example

```
1 // string::operator+=
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string name ("John");
8     std::string family ("Smith");
9     name += " K. ";           // c-string
10    name += family;         // string
11    name += '\n';            // character
12
13    std::cout << name;
14    return 0;
15 }
```

Output:

```
John K. Smith
```

## Complexity

Unspecified, but generally up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `string`.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

A `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>string::append</code>	Append to string (public member function )
<code>string::assign</code>	Assign content to string (public member function )
<code>string::operator=</code>	String assignment (public member function )
<code>string::insert</code>	Insert into string (public member function )
<code>string::replace</code>	Replace portion of string (public member function )

# /string/string/operators

function

## relational operators (string)

<string>

```
(1) bool operator== (const string& lhs, const string& rhs);
    bool operator== (const char* lhs, const string& rhs);
    bool operator== (const string& lhs, const char* rhs);
    bool operator!= (const string& lhs, const string& rhs);
(2) bool operator!= (const char* lhs, const string& rhs);
    bool operator!= (const string& lhs, const char* rhs);
    bool operator< (const string& lhs, const string& rhs);
(3) bool operator< (const char* lhs, const string& rhs);
    bool operator< (const string& lhs, const char* rhs);
    bool operator<= (const string& lhs, const string& rhs);
(4) bool operator<= (const char* lhs, const string& rhs);
    bool operator<= (const string& lhs, const char* rhs);
    bool operator> (const string& lhs, const string& rhs);
(5) bool operator> (const char* lhs, const string& rhs);
    bool operator> (const string& lhs, const char* rhs);
    bool operator>= (const string& lhs, const string& rhs);
(6) bool operator>= (const char* lhs, const string& rhs);
    bool operator>= (const string& lhs, const char* rhs);

    bool operator== (const string& lhs, const string& rhs) noexcept;
(1) bool operator== (const char* lhs, const string& rhs);
    bool operator== (const string& lhs, const char* rhs);
    bool operator!= (const string& lhs, const string& rhs) noexcept;
(2) bool operator!= (const char* lhs, const string& rhs);
    bool operator!= (const string& lhs, const char* rhs);
    bool operator< (const string& lhs, const string& rhs) noexcept;
(3) bool operator< (const char* lhs, const string& rhs);
    bool operator< (const string& lhs, const char* rhs);
```

```

(4) bool operator<= (const string& lhs, const string& rhs) noexcept;
bool operator<= (const char* lhs, const string& rhs);
bool operator<= (const string& lhs, const char* rhs);
(5) bool operator> (const string& lhs, const string& rhs) noexcept;
bool operator> (const char* lhs, const string& rhs);
bool operator> (const string& lhs, const char* rhs);
bool operator>= (const string& lhs, const string& rhs) noexcept;
(6) bool operator>= (const char* lhs, const string& rhs);
bool operator>= (const string& lhs, const char* rhs);

```

## Relational operators for string

Performs the appropriate comparison operation between the `string` objects *lhs* and *rhs*.

The functions use `string::compare` for the comparison.

These operators are overloaded in header `<string>`.

## Parameters

*lhs*, *rhs*  
 Arguments to the left- and right-hand side of the operator, respectively.  
 If of type `char*`, it shall point to a null-terminated character sequence.

## Example

```

1 // string comparisons
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::string foo = "alpha";
8     std::string bar = "beta";
9
10    if (foo==bar) std::cout << "foo and bar are equal\n";
11    if (foo!=bar) std::cout << "foo and bar are not equal\n";
12    if (foo< bar) std::cout << "foo is less than bar\n";
13    if (foo> bar) std::cout << "foo is greater than bar\n";
14    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
15    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
16
17    return 0;
18 }

```

Output:

```

foo and bar are not equal
foo is less than bar
foo is less than or equal to bar

```

## Return Value

true if the condition holds, and false otherwise.

## Complexity

Unspecified, but generally up to linear in both *lhs* and *rhs*'s lengths.

## Iterator validity

No changes.

## Data races

Both objects, *lhs* and *rhs*, are accessed.

## Exception safety

If an argument of type `char*` does not point to null-terminated character sequence, it causes *undefined behavior*.  
 Otherwise, if an exception is thrown, there are no changes in the `string` (strong guarantee).

If an argument of type `char*` does not point to null-terminated character sequence, it causes *undefined behavior*.  
 For operations between `string` objects, exceptions are never thrown (no-throw guarantee).  
 For other cases, if an exception is thrown, there are no changes in the `string` (strong guarantee).

## See also

<code>string::compare</code>	Compare strings (public member function )
<code>string::find</code>	Find content in string (public member function )
<code>string::operator=</code>	String assignment (public member function )
<code>string::swap</code>	Swap string values (public member function )

## /string/string/pop\_back

public member function

`<string>`

## std::string::pop\_back

void pop\_back();

### Delete last character

Erases the last character of the [string](#), effectively reducing its [length](#) by one.

### Parameters

none

### Return value

none

### Example

```
1 // string::pop_back
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("hello world!");
8     str.pop_back();
9     std::cout << str << '\n';
10    return 0;
11 }
```

hello world

### Complexity

Unspecified, but generally constant.

### Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

### Data races

The object is modified.

### Exception safety

If the [string](#) is [empty](#), it causes [undefined behavior](#).

Otherwise, the function never throws exceptions (no-throw guarantee).

### See also

<a href="#">string::back</a>	Access last character ( <a href="#">public member function</a> )
<a href="#">string::push_back</a>	Append character to string ( <a href="#">public member function</a> )
<a href="#">string::erase</a>	Erase characters from string ( <a href="#">public member function</a> )

## /string/string/push\_back

public member function

## std::string::push\_back

<string>

void push\_back (char c);

### Append character to string

Appends character c to the end of the [string](#), increasing its [length](#) by one.

### Parameters

c

Character added to the [string](#).

### Return value

none

### Example

```
1 // string::push_back
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5
6 int main ()
7 {
8     std::string str;
9     std::ifstream file ("test.txt",std::ios::in);
```

```

10 if (file) {
11     while (!file.eof()) str.push_back(file.get());
12 }
13 std::cout << str << '\n';
14 return 0;
15 }

```

This example reads an entire file character by character, appending each character to a string object using `push_back`.

## Complexity

Unspecified; Generally amortized constant, but up to linear in the new `string` length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `string`.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.  
A `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>string::back</code>	Access last character ( <a href="#">public member function</a> )
<code>string::pop_back</code>	Delete last character ( <a href="#">public member function</a> )
<code>string::append</code>	Append to string ( <a href="#">public member function</a> )
<code>string::insert</code>	Insert into string ( <a href="#">public member function</a> )
<code>string::end</code>	Return iterator to end ( <a href="#">public member function</a> )

## /string/string/rbegin

public member function

### std::string::rbegin

`<string>`

```

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;

```

#### Return reverse iterator to reverse beginning

Returns a `reverse iterator` pointing to the last character of the string (i.e., its *reverse beginning*).

*Reverse iterators* iterate backwards: increasing them moves them towards the beginning of the string.

`rbegin` points to the character right before the one that would be pointed to by member `end`.

## Parameters

none

## Return Value

A reverse iterator to the *reverse beginning* of the string.

If the `string` object is `const`-qualified, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are reverse `random access iterator` types (pointing to a character and to a `const character`, respectively).

## Example

```

1 // string::rbegin/rend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("now step live...");
8     for (std::string::reverse_iterator rit=str.rbegin(); rit!=str.rend(); ++rit)
9         std::cout << *rit;
10    return 0;
11 }

```

This code prints out the reversed content of a string character by character using a reverse iterator that iterates between `rbegin` and `rend`. Notice how even though the reverse iterator is increased, the iteration goes backwards through the string (this is a feature of reverse iterators).

The actual output is:

...evil pets won

## Complexity

Unspecified.

## Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

## Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed (neither the const nor the non-const versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<a href="#">string::rend</a>	Return reverse iterator to reverse end (public member function )
<a href="#">string::begin</a>	Return iterator to beginning (public member function )
<a href="#">string::end</a>	Return iterator to end (public member function )

# /string/string/rend

public member function

## std::string::rend

<string>

```
reverse_iterator rend();
const_reverse_iterator rend() const;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

### Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element preceding the first character of the string (which is considered its *reverse end*).

The range between `string::rbegin` and `string::rend` contains all the characters of the `string` (in reverse order).

## Parameters

none

## Return Value

A reverse iterator to the *reverse end* of the string.

If the `string` object is *const-qualified*, the function returns a `const_iterator`. Otherwise, it returns an `iterator`.

Member types `reverse_iterator` and `const_reverse_iterator` are reverse *random access iterator* types (pointing to a character and to a *const character*, respectively).

## Example

```
1 // string::rbegin/rend
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("now step live...");
8     for (std::string::reverse_iterator rit=str.rbegin(); rit!=str.rend(); ++rit)
9         std::cout << *rit;
10    return 0;
11 }
```

This code prints out the reversed content of a string character by character using a reverse iterator that iterates between `rbegin` and `rend`. Notice how even

though the reverse iterator is increased, the iteration goes backwards through the string (this is a feature of reverse iterators).

The actual output is:

```
...evil pets won
```

## Complexity

Unspecified.

## Iterator validity

Generally, no changes.

On some implementations, the non-const version may invalidate all iterators, pointers and references on the first access to string characters after the object has been constructed or modified.

## Data races

The object is accessed, and in some implementations, the non-const version modifies it on the first access to string characters after the object has been constructed or modified.

The iterator returned can be used to access or modify characters.

## Complexity

Unspecified, but generally constant.

## Iterator validity

No changes.

## Data races

The object is accessed (neither the const nor the non-const versions modify it).

The iterator returned can be used to access or modify characters. Concurrently accessing or modifying different characters is safe.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

The copy construction or assignment of the returned iterator is also guaranteed to never throw.

## See also

<code>string::rbegin</code>	Return reverse iterator to reverse beginning (public member function )
<code>string::begin</code>	Return iterator to beginning (public member function )
<code>string::end</code>	Return iterator to end (public member function )

# /string/string/replace

public member function

## std::string::replace

<string>

```
string (1) string& replace (size_t pos, size_t len, const string& str);
string& replace (iterator i1, iterator i2, const string& str);

substring (2) string& replace (size_t pos, size_t len, const string& str,
                           size_t subpos, size_t sublen);

c-string (3) string& replace (size_t pos, size_t len, const char* s);
string& replace (iterator i1, iterator i2, const char* s);

buffer (4) string& replace (size_t pos, size_t len, const char* s, size_t n);
string& replace (iterator i1, iterator i2, const char* s, size_t n);

fill (5) string& replace (size_t pos, size_t len, size_t n, char c);
string& replace (iterator i1, iterator i2, size_t n, char c);

template <class InputIterator>
range (6) string& replace (iterator i1, iterator i2,
                           InputIterator first, InputIterator last);

string (1) string& replace (size_t pos, size_t len, const string& str);
string& replace (const_iterator i1, const_iterator i2, const string& str);

substring (2) string& replace (size_t pos, size_t len, const string& str,
                           size_t subpos, size_t sublen);

c-string (3) string& replace (size_t pos, size_t len, const char* s);
string& replace (const_iterator i1, const_iterator i2, const char* s);

buffer (4) string& replace (size_t pos, size_t len, const char* s, size_t n);
string& replace (const_iterator i1, const_iterator i2, const char* s, size_t n);

fill (5) string& replace (size_t pos, size_t len, size_t n, char c);
string& replace (const_iterator i1, const_iterator i2, size_t n, char c);

template <class InputIterator>
range (6) string& replace (const_iterator i1, const_iterator i2,
                           InputIterator first, InputIterator last);

initializer list (7) string& replace (const_iterator i1, const_iterator i2, initializer_list<char> il);

string (1) string& replace (size_t pos, size_t len, const string& str);
string& replace (const_iterator i1, const_iterator i2, const string& str);

substring (2) string& replace (size_t pos, size_t len, const string& str,
                           size_t subpos, size_t sublen = npos);

c-string (3) string& replace (size_t pos, size_t len, const char* s);
string& replace (const_iterator i1, const_iterator i2, const char* s);

buffer (4) string& replace (size_t pos, size_t len, const char* s, size_t n);
string& replace (const_iterator i1, const_iterator i2, const char* s, size_t n);
```

```

    fill(5) string& replace (size_t pos,           size_t len,           size_t n, char c);
    string& replace (const_iterator i1, const_iterator i2, size_t n, char c);
template <class InputIterator>
range(6)   string& replace (const_iterator i1, const_iterator i2,
                           InputIterator first, InputIterator last);
initializer list(7) string& replace (const_iterator i1, const_iterator i2, initializer_list<char> il);

```

## Replace portion of string

Replaces the portion of the string that begins at character *pos* and spans *len* characters (or the part of the string in the range between *[i1,i2]*) by new contents:

### (1) **string**

Copies *str*.

### (2) **substring**

Copies the portion of *str* that begins at the character position *subpos* and spans *sublen* characters (or until the end of *str*, if either *str* is too short or if *sublen* is `string::npos`).

### (3) **c-string**

Copies the null-terminated character sequence (C-string) pointed by *s*.

### (4) **buffer**

Copies the first *n* characters from the array of characters pointed by *s*.

### (5) **fill**

Replaces the portion of the string by *n* consecutive copies of character *c*.

### (6) **range**

Copies the sequence of characters in the range *[first, last]*, in the same order.

### (7) **initializer list**

Copies each of the characters in *il*, in the same order.

---

## Parameters

**str** Another `string` object, whose value is copied.

**pos** Position of the first character to be replaced.  
If this is greater than the `string` length, it throws `out_of_range`.

**len** Number of characters to replace (if the string is shorter, as many characters as possible are replaced).  
A value of `string::npos` indicates all characters until the end of the string.

**subpos** Position of the first character in *str* that is copied to the object as replacement.  
If this is greater than *str*'s `length`, it throws `out_of_range`.

**sublen** Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).  
A value of `string::npos` indicates all characters until the end of *str*.

**s** Pointer to an array of characters (such as a `c-string`).

**n** Number of characters to copy.

**c** Character value, repeated *n* times.

**first, last** `Input iterators` to the initial and final positions in a range. The range used is *[first, last]*, which includes all the characters between *first* and *last*, including the character pointed by *first* but not the character pointed by *last*.  
The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type convertible to `char`.

**il** An `initializer_list` object.  
These objects are automatically constructed from `initializer list` declarators.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

---

## Return Value

`*this`

---

## Example

```

1 // replacing in a string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string base="this is a test string.";
8     std::string str2="n example";
9     std::string str3="sample phrase";
10    std::string str4="useful.";
11
12    // replace signatures used in the same order as described above:
13
14    // Using positions:          0123456789*123456789*12345
15    std::string str=base;        // "this is a test string."
16    str.replace(9,5,str2);      // "this is an example string." (1)
17    str.replace(19,6,str3,7,6); // "this is an example phrase." (2)
18    str.replace(8,10,"just a"); // "this is just a phrase."      (3)
19    str.replace(8,6,"a shorty",7); // "this is a short phrase." (4)

```

```

20 str.replace(22,1,3,'!');           // "this is a short phrase!!!" (5)
21
22 // Using iterators:
23 str.replace(str.begin(),str.end()-3,str3);          0123456789*123456789*
24 str.replace(str.begin(),str.begin()+6,"replace");    // "sample phrase!!!" (1)
25 str.replace(str.begin()+8,str.begin()+14,"is coolness",7); // "replace phrase!!!" (3)
26 str.replace(str.begin()+12,str.end()-4,'o');         // "replace is cool!!!" (4)
27 str.replace(str.begin()+11,str.end(),str4.begin(),str4.end()); // "replace is useful." (5)
28 std::cout << str << '\n';
29 return 0;
30 }

```

Output:

```
replace is useful.
```

## Complexity

Unspecified, but generally up to linear in the new [string length](#).

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [string](#).

If [s](#) does not point to an array long enough, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

If [pos](#) is greater than the [string length](#), or if [subpos](#) is greater than [str's length](#), an [out\\_of\\_range](#) exception is thrown.

If the resulting [string length](#) would exceed the [max\\_size](#), a [length\\_error](#) exception is thrown.

A [bad\\_alloc](#) exception is thrown if the function needs to allocate storage and fails.

## See also

<a href="#">string::insert</a>	Insert into string (public member function )
<a href="#">string::append</a>	Append to string (public member function )
<a href="#">string::substr</a>	Generate substring (public member function )
<a href="#">string::erase</a>	Erase characters from string (public member function )
<a href="#">string::assign</a>	Assign content to string (public member function )

# /string/string/reserve

public member function

## std::string::reserve

<string>

```
void reserve (size_t n = 0);
```

### Request a change in capacity

Requests that the [string capacity](#) be adapted to a planned change in [size](#) to a [length](#) of up to [n](#) characters.

If [n](#) is greater than the current [string capacity](#), the function causes the container to increase its [capacity](#) to [n](#) characters (or greater).

In all other cases, it is taken as a non-binding request to shrink the [string capacity](#): the container implementation is free to optimize otherwise and leave the [string](#) with a [capacity](#) greater than [n](#).

This function has no effect on the [string length](#) and cannot alter its content.

## Parameters

n

Planned [length](#) for the [string](#).

Note that the resulting [string capacity](#) may be equal or greater than [n](#).

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Return Value

none

## Example

```

1 // string::reserve
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5
6 int main ()
7 {
8     std::string str;
9
10    std::ifstream file ("test.txt",std::ios::in|std::ios::ate);

```

```

11 if (file) {
12     std::ifstream::streampos filesize = file.tellg();
13     str.reserve(filesize);
14
15     file.seekg(0);
16     while (!file.eof())
17     {
18         str += file.get();
19     }
20     std::cout << str;
21 }
22 return 0;
23 }
```

This example reserves enough capacity in the `string` object to store an entire file, which is then read character by character. By reserving a `capacity` for the `string` of at least the size of the entire file, we try to avoid all the automatic reallocations that the object `str` could suffer each time that inserting a new character would make its `length` surpass its `capacity`.

## Complexity

Unspecified, but generally constant.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the `string`.

If  $n$  is greater than the `max_size`, a `length_error` exception is thrown.

A `bad_alloc` exception is thrown if the function needs to allocate storage and fails.

## See also

<code>string::capacity</code>	Return size of allocated storage ( <a href="#">public member function</a> )
<code>string::shrink_to_fit</code>	Shrink to fit ( <a href="#">public member function</a> )
<code>string::resize</code>	Resize string ( <a href="#">public member function</a> )
<code>string::max_size</code>	Return maximum size of string ( <a href="#">public member function</a> )

## /string/string/resize

public member function

### std::string::resize

`<string>`

```
void resize (size_t n);
void resize (size_t n, char c);
```

#### Resize string

Resizes the string to a `length` of  $n$  characters.

If  $n$  is smaller than the current `string length`, the current value is shortened to its first  $n$  character, removing the characters beyond the  $n$ th.

If  $n$  is greater than the current `string length`, the current content is extended by inserting at the end as many characters as needed to reach a size of  $n$ . If  $c$  is specified, the new elements are initialized as copies of  $c$ , otherwise, they are *value-initialized characters* (null characters).

## Parameters

`n`  
New `string length`, expressed in number of characters.  
`size_t` is an unsigned integral type (the same as member type `string::size_type`).

`c`  
Character used to fill the new character space added to the string (in case the string is expanded).

## Return Value

none

## Example

```

1 // resizing string
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("I like to code in C");
8     std::cout << str << '\n';
9
10    unsigned sz = str.size();
```

```

12 str.resize (sz+2, '+');
13 std::cout << str << '\n';
14
15 str.resize (14);
16 std::cout << str << '\n';
17 return 0;
18 }

```

Output:

```
I like to code in C
I like to code in C++
I like to code
```

## Complexity

Unspecified, but generally up to linear in the new [string](#) length.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [string](#).

If *n* is greater than [max\\_size](#), a [length\\_error](#) exception is thrown.

A [bad\\_alloc](#) exception is thrown if the function needs to allocate storage and fails.

## See also

<a href="#">string::size</a>	Return length of string ( <a href="#">public member function</a> )
<a href="#">string::clear</a>	Clear string ( <a href="#">public member function</a> )
<a href="#">string::max_size</a>	Return maximum size of string ( <a href="#">public member function</a> )

# /string/string/rfind

public member function

## std::string::rfind

<string>

```

string (1) size_t rfind (const string& str, size_t pos = npos) const;
c-string (2) size_t rfind (const char* s, size_t pos = npos) const;
buffer (3) size_t rfind (const char* s, size_t pos, size_t n) const;
character (4) size_t rfind (char c, size_t pos = npos) const;

string (1) size_t rfind (const string& str, size_t pos = npos) const noexcept;
c-string (2) size_t rfind (const char* s, size_t pos = npos) const;
buffer (3) size_t rfind (const char* s, size_t pos, size_t n) const;
character (4) size_t rfind (char c, size_t pos = npos) const noexcept;

```

### Find last occurrence of content in string

Searches the [string](#) for the last occurrence of the sequence specified by its arguments.

When *pos* is specified, the search only includes sequences of characters that begin at or before position *pos*, ignoring any possible match beginning after *pos*.

## Parameters

<i>str</i>	Another <a href="#">string</a> with the subject to search for.
<i>pos</i>	Position of the last character in the string to be considered as the beginning of a match. Any value greater or equal than the <a href="#">string</a> <a href="#">length</a> (including <a href="#">string::npos</a> ) means that the entire string is searched. Note: The first character is denoted by a value of 0 (not 1).
<i>s</i>	Pointer to an array of characters. If argument <i>n</i> is specified (3), the sequence to match are the first <i>n</i> characters in the array. Otherwise (2), a null-terminated sequence is expected: the length of the sequence to match is determined by the first occurrence of a null character.
<i>n</i>	Length of sequence of characters to match.
<i>c</i>	Individual character to be searched for.

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Return Value

The position of the first character of the last match.  
If no matches were found, the function returns [string::npos](#).

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

## Example

```
1 // string::rfind
2 #include <iostream>
3 #include <string>
4 #include <cstddef>
5
6 int main ()
7 {
8     std::string str ("The sixth sick sheik's sixth sheep's sick.");
9     std::string key ("sixth");
10
11    std::size_t found = str.rfind(key);
12    if (found!=std::string::npos)
13        str.replace (found,key.length(),"seventh");
14
15    std::cout << str << '\n';
16
17    return 0;
18 }
```

```
The sixth sick sheik's seventh sheep's sick.
```

## Complexity

Unspecified, but generally up to linear in the `string length` (or `pos`) times the number of characters to match (worst case).

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

If `s` does not point to an array long enough, it causes *undefined behavior*.  
Otherwise, the function never throws exceptions (no-throw guarantee).

## See also

<code>string::find</code>	Find content in string (public member function )
<code>string::find_last_of</code>	Find character in string from the end (public member function )
<code>string::find_last_not_of</code>	Find non-matching character in string from the end (public member function )
<code>string::replace</code>	Replace portion of string (public member function )
<code>string::substr</code>	Generate substring (public member function )

## /string/string/shrink\_to\_fit

public member function

### `std::string::shrink_to_fit`

`<string>`

`void shrink_to_fit();`

#### Shrink to fit

Requests the `string` to reduce its `capacity` to fit its `size`.

The request is non-binding, and the container implementation is free to optimize otherwise and leave the `string` with a `capacity` greater than its `size`.

This function has no effect on the `string length` and cannot alter its content.

## Parameters

none

## Return value

none

## Example

```
1 // string::shrink_to_fit
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str (100,'x');
8     std::cout << "1. capacity of str: " << str.capacity() << '\n';
9
10    str.resize(10);
11    std::cout << "2. capacity of str: " << str.capacity() << '\n';
```

```

12     str.shrink_to_fit();
13     std::cout << "3. capacity of str: " << str.capacity() << '\n';
14 }
15
16 return 0;
17 }
```

Possible output:

```

1. capacity of str: 100
2. capacity of str: 100
3. capacity of str: 10
```

## Complexity

Unspecified, but generally constant.

## Iterator validity

Any iterators, pointers and references related to this object may be invalidated.

## Data races

The object is modified.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [string](#).

A [bad\\_alloc](#) exception is thrown if the function needs to allocate storage and fails.

## See also

<a href="#">string::capacity</a>	Return size of allocated storage ( <a href="#">public member function</a> )
<a href="#">string::reserve</a>	Request a change in capacity ( <a href="#">public member function</a> )
<a href="#">string::resize</a>	Resize string ( <a href="#">public member function</a> )
<a href="#">string::clear</a>	Clear string ( <a href="#">public member function</a> )

# /string/string/size

public member function

## std::string::size

<[string](#)>

```

size_t size() const;
size_t size() const noexcept;
```

### Return length of string

Returns the length of the string, in terms of bytes.

This is the number of actual bytes that conform the contents of the [string](#), which is not necessarily equal to its storage [capacity](#).

Note that [string](#) objects handle bytes without knowledge of the encoding that may eventually be used to encode the characters it contains. Therefore, the value returned may not correspond to the actual number of encoded characters in sequences of multi-byte or variable-length characters (such as UTF-8).

Both [string::size](#) and [string::length](#) are synonyms and return the same value.

## Parameters

none

## Return Value

The number of bytes in the string.

[size\\_t](#) is an unsigned integral type (the same as member type [string::size\\_type](#)).

## Example

```

1 // string::size
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str ("Test string");
8     std::cout << "The size of str is " << str.size() << " bytes.\n";
9     return 0;
10 }
```

Output:

```
The size of str is 11 bytes
```

## Complexity

Unspecified.

Constant.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**No-throw guarantee:** this member function never throws exceptions.

## See also

<a href="#">string::length</a>	Return length of string (public member function )
<a href="#">string::resize</a>	Resize string (public member function )
<a href="#">string::max_size</a>	Return maximum size of string (public member function )
<a href="#">string::capacity</a>	Return size of allocated storage (public member function )

# /string/string/string

public member function

## std::string::string

<string>

```
default (1) string();
copy (2) string (const string& str);
substring (3) string (const string& str, size_t pos, size_t len = npos);
from c-string (4) string (const char* s);
from sequence (5) string (const char* s, size_t n);
fill (6) string (size_t n, char c);
range (7) string (InputIterator first, InputIterator last);

default (1) string();
copy (2) string (const string& str);
substring (3) string (const string& str, size_t pos, size_t len = npos);
from c-string (4) string (const char* s);
from buffer (5) string (const char* s, size_t n);
fill (6) string (size_t n, char c);
range (7) string (InputIterator first, InputIterator last);
initializer list (8) string (initializer_list<char> il);
move (9) string (string& str) noexcept;
```

### Construct string object

Constructs a [string](#) object, initializing its value depending on the constructor version used:

#### (1) empty string constructor (default constructor)

Constructs an empty string, with a length of zero characters.

#### (2) copy constructor

Constructs a copy of str.

#### (3) substring constructor

Copies the portion of str that begins at the character position pos and spans len characters (or until the end of str, if either str is too short or if len is `string::npos`).

#### (4) from c-string

Copies the null-terminated character sequence (C-string) pointed by s.

#### (5) from buffer

Copies the first n characters from the array of characters pointed by s.

#### (6) fill constructor

Fills the string with n consecutive copies of character c.

#### (7) range constructor

Copies the sequence of characters in the range [first, last), in the same order.

#### (8) initializer list

Copies each of the characters in il, in the same order.

#### (9) move constructor

Acquires the contents of str.

str is left in an unspecified but valid state.

All constructors above support an object of member type allocator\_type as additional optional argument at the end, which for `string` is not relevant (not shown above, see [basic\\_string's constructor](#) for full signatures).

## Parameters

str

Another `string` object, whose value is either copied or acquired.

pos

Position of the first character in *str* that is copied to the object as a substring.

If this is greater than *str*'s `length`, it throws `out_of_range`.

Note: The first character in *str* is denoted by a value of 0 (not 1).

`len`

Length of the substring to be copied (if the string is shorter, as many characters as possible are copied).  
A value of `string::npos` indicates all characters until the end of *str*.

`s`

Pointer to an array of characters (such as a *c-string*).

`n`

Number of characters to copy.

`c`

Character to fill the string with. Each of the *n* characters in the string will be initialized to a copy of this value.

`first, last`

`Input iterators` to the initial and final positions in a range. The range used is `[first, last)`, which includes all the characters between *first* and *last*, including the character pointed by *first* but not the character pointed by *last*.

The function template argument `InputIterator` shall be an `input iterator` type that points to elements of a type convertible to `char`.

If `InputIterator` is an integral type, the arguments are casted to the proper types so that signature (5) is used instead.

`il`

An `initializer_list` object.

These objects are automatically constructed from *initializer list* declarators.

`size_t` is an unsigned integral type.

## Example

```
1 // string constructor
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string s0 ("Initial string");
8
9     // constructors used in the same order as described above:
10    std::string s1;
11    std::string s2 (s0);
12    std::string s3 (s0, 8, 3);
13    std::string s4 ("A character sequence");
14    std::string s5 ("Another character sequence", 12);
15    std::string s6a (10, 'x');
16    std::string s6b (10, 42);      // 42 is the ASCII code for '*'
17    std::string s7 (s0.begin(), s0.begin() + 7);
18
19    std::cout << "s1: " << s1 << "\ns2: " << s2 << "\ns3: " << s3;
20    std::cout << "\ns4: " << s4 << "\ns5: " << s5 << "\ns6a: " << s6a;
21    std::cout << "\ns6b: " << s6b << "\ns7: " << s7 << '\n';
22    return 0;
23 }
```

Output:

```
s1:
s2: Initial string
s3: str
s4: A character sequence
s5: Another char
s6a:xxxxxxxxxx
s6b: ****
s7: Initial
```

## Complexity

Unspecified.

Unspecified, but generally linear in the resulting `string` length (and constant for *move constructors*).

## Iterator validity

The *move constructors* (9) may invalidate iterators, pointers and references related to *str*.

## Data races

The *move constructors* (9) modify *str*.

## Exception safety

The *move constructor* with no allocator argument (9, *first*) never throws exceptions (no-throw guarantee).  
In all other cases, there are no effects in case an exception is thrown (strong guarantee).

If *s* is a null pointer, if *n* == `npos`, or if the range specified by `[first, last)` is not valid, it causes *undefined behavior*.

If *pos* is greater than *str*'s `length`, an `out_of_range` exception is thrown.

If *n* is greater than the array pointed by *s*, it causes *undefined behavior*.

If the resulting `string` length would exceed the `max_size`, a `length_error` exception is thrown.

A `bad_alloc` exception is thrown if the function fails when attempting to allocate storage.

## See also

`string::operator=` | String assignment (public member function )

<b>string::assign</b>	Assign content to string (public member function )
<b>string::resize</b>	Resize string (public member function )
<b>string::clear</b>	Clear string (public member function )

## /string/string/~string

public member function

### std::string::~string

<string>

`~string();`

#### String destructor

Destroys the `string` object.

This deallocates all the storage `capacity` allocated by the `string` using its `allocator`.

#### Complexity

Unspecified, but generally constant.

#### Iterator validity

All iterators, pointers and references are invalidated.

#### Data races

The object is modified.

#### Exception safety

**No-throw guarantee:** never throws exceptions.

## /string/string/substr

public member function

### std::string::substr

<string>

`string substr (size_t pos = 0, size_t len = npos) const;`

#### Generate substring

Returns a newly constructed `string` object with its value initialized to a copy of a substring of this object.

The substring is the portion of the object that starts at character position `pos` and spans `len` characters (or until the end of the string, whichever comes first).

#### Parameters

`pos`

Position of the first character to be copied as a substring.

If this is equal to the `string length`, the function returns an empty string.

If this is greater than the `string length`, it throws `out_of_range`.

Note: The first character is denoted by a value of 0 (not 1).

`len`

Number of characters to include in the substring (if the string is shorter, as many characters as possible are used).

A value of `string::npos` indicates all characters until the end of the string.

`size_t` is an unsigned integral type (the same as member type `string::size_type`).

#### Return Value

A `string` object with a substring of this object.

#### Example

```

1 // string::substr
2 #include <iostream>
3 #include <string>
4
5 int main ()
6 {
7     std::string str="We think in generalities, but we live in details.";
8                         // (quoting Alfred N. Whitehead)
9
10    std::string str2 = str.substr (3,5);      // "think"
11
12    std::size_t pos = str.find("live");      // position of "live" in str
13
14    std::string str3 = str.substr (pos);      // get from "live" to the end
15
16    std::cout << str2 << ' ' << str3 << '\n';
17
18    return 0;
19 }
```

Output:

```
think live in details.
```

## Complexity

Unspecified, but generally linear in the [length](#) of the returned object.

## Iterator validity

No changes.

## Data races

The object is accessed.

## Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the [string](#).

If *pos* is greater than the [string length](#), an [out\\_of\\_range](#) exception is thrown.

A [bad\\_alloc](#) exception is thrown if the function needs to allocate storage and fails.

## See also

<a href="#">string::replace</a>	Replace portion of string (public member function )
<a href="#">string::data</a>	Get string data (public member function )
<a href="#">string::find</a>	Find content in string (public member function )
<a href="#">string::assign</a>	Assign content to string (public member function )
<a href="#">string::string</a>	Construct string object (public member function )

# /string/string/swap

public member function

## std::string::swap

<string>

```
void swap (string& str);
```

### Swap string values

Exchanges the content of the container by the content of *str*, which is another [string](#) object. [Lengths](#) may differ.

After the call to this member function, the value of this object is the value *str* had before the call, and the value of *str* is the value this object had before the call.

Notice that a non-member function exists with the same name, [swap](#), overloading that algorithm with an optimization that behaves like this member function.

## Parameters

*str*

Another [string](#) object, whose value is swapped with that of this [string](#).

## Return value

none

## Example

```
1 // swap strings
2 #include <iostream>
3 #include <string>
4
5 main ()
6 {
7     std::string buyer ("money");
8     std::string seller ("goods");
9
10    std::cout << "Before the swap, buyer has " << buyer;
11    std::cout << " and seller has " << seller << '\n';
12
13    seller.swap (buyer);
14
15    std::cout << " After the swap, buyer has " << buyer;
16    std::cout << " and seller has " << seller << '\n';
17
18    return 0;
19 }
```

Output:

```
Before the swap, buyer has money and seller has goods
After the swap, buyer has goods and seller has money
```

## Complexity

Constant.

## Iterator validity

Any iterators, pointers and references related to this object and to *str* may be invalidated.

## Data races

Both the object and *str* are modified.

## Exception safety

No-throw guarantee: this member function never throws exceptions.

## See also

<a href="#">swap (string)</a>	Exchanges the values of two strings (function )
<a href="#">swap_ranges</a>	Exchange values of two ranges (function template )

## /string/string/swap-free

function  
**std::swap (string)** <string>

**void swap (string& x, string& y);**

### Exchanges the values of two strings

Exchanges the values of **string** objects *x* and *y*, such that after the call to this function, the value of *x* is the one which was on *y* before the call, and the value of *y* is that of *x*.

This is an overload of the generic algorithm **swap** that improves its performance by mutually transferring ownership over their internal data to the other object (i.e., the strings exchange references to their data, without actually copying the characters): It behaves as if *x.swap(y)* was called.

## Parameters

**x,y** string objects to swap.

## Return value

none

## Example

```
1 // swap strings
2 #include <iostream>
3 #include <string>
4
5 main ()
6 {
7     std::string buyer ("money");
8     std::string seller ("goods");
9
10    std::cout << "Before the swap, buyer has " << buyer;
11    std::cout << " and seller has " << seller << '\n';
12
13    swap (buyer,seller);
14
15    std::cout << " After the swap, buyer has " << buyer;
16    std::cout << " and seller has " << seller << '\n';
17
18    return 0;
19 }
```

Output:

```
Before the swap, buyer has money and seller has goods
After the swap, buyer has goods and seller has money
```

## Complexity

Constant.

## Iterator validity

Any iterators, pointers and references related to both *x* and *y* may be invalidated.

## Data races

Both objects, *x* and *y*, are modified.

## Exception safety

No-throw guarantee: this member function never throws exceptions.

## See also

<a href="#">string::swap</a>	Swap string values (public member function )
------------------------------	--

<b>swap</b>	Exchange values of two objects (function template )
<b>swap_ranges</b>	Exchange values of two ranges (function template )

## /string/to\_string

function  
**std::to\_string** <string>

```
string to_string ( int val);
string to_string ( long val);
string to_string ( long long val);
string to_string ( unsigned val);
string to_string ( unsigned long val);
string to_string ( unsigned long long val);
string to_string ( float val);
string to_string ( double val);
string to_string ( long double val);
```

### Convert numerical value to string

Returns a `string` with the representation of `val`.

The format used is the same that `printf` would print for the corresponding type:

type of <code>val</code>	<code>printf</code> equivalent	description
int	"%d"	Decimal-base representation of <code>val</code> .
long	"%ld"	The representations of negative values are preceded with a minus sign (-).
long long	"%lld"	
unsigned	"%u"	
unsigned long	"%lu"	Decimal-base representation of <code>val</code> .
unsigned long long	"%llu"	
float	"%f"	As many digits are written as needed to represent the integral part, followed by the decimal-point character and six decimal digits.
double	"%f"	inf (or infinity) is used to represent infinity. nan (followed by an optional sequence of characters) to represent NaNs (Not-a-Number).
long double	"%Lf"	The representations of negative values are preceded with a minus sign (-).

### Parameters

`val`  
Numerical value.

### Return Value

A `string` object containing the representation of `val` as a sequence of characters.

### Example

```
1 // to_string example
2 #include <iostream>      // std::cout
3 #include <string>        // std::string, std::to_string
4
5 int main ()
6 {
7     std::string pi = "pi is " + std::to_string(3.1415926);
8     std::string perfect = std::to_string(1+2+4+7+14) + " is a perfect number";
9     std::cout << pi << '\n';
10    std::cout << perfect << '\n';
11    return 0;
12 }
```

Possible output:

```
pi is 3.141593
28 is a perfect number
```

### Exceptions

The `string` constructor may throw.

### See also

<b>sprintf</b>	Write formatted data to string (function )
<b>to_wstring</b>	Convert numerical value to wide string (function )

## /string/to\_wstring

function  
**std::to\_wstring** <string>

```
wstring to_wstring ( int val);
wstring to_wstring ( long val);
wstring to_wstring ( long long val);
```

```
wstring to_wstring (unsigned val);
wstring to_wstring (unsigned long val);
wstring to_wstring (unsigned long long val);
wstring to_wstring (float val);
wstring to_wstring (double val);
wstring to_wstring (long double val);
```

#### Convert numerical value to wide string

Returns a [wstring](#) with the representation of *val*.

The format used is the same that [wprintf](#) would print for the corresponding type:

type of <i>val</i>	wprintf equivalent	description
int	L"%d"	Decimal-base representation of <i>val</i> .
long	L"%ld"	The representations of negative values are preceded with a minus sign (-).
long long	L"%lld"	
unsigned	L"%u"	
unsigned long	L"%lu"	Decimal-base representation of <i>val</i> .
unsigned long long	L"%llu"	
float	L"%f"	As many digits are written as needed to represent the integral part, followed by the decimal-point character and six decimal digits.
double	L"%f"	inf (or infinity) is used to represent infinity.
long double	L"%Lf"	nan (followed by an optional sequence of characters) to represent NaNs ( <i>Not-a-Number</i> ). The representations of negative values are preceded with a minus sign (-).

## Parameters

*val*

Numerical value.

## Return Value

A [wstring](#) object containing the representation of *val* as a sequence of characters.

## Example

```
1 // to_wstring example
2 #include <iostream>    // std::wcout
3 #include <string>      // std::wstring, std::to_wstring
4
5 int main ()
6 {
7     std::wstring pi = L"pi is " + std::to_wstring(3.1415926);
8     std::wstring perfect = std::to_wstring(1+2+4+7+14) + L" is a perfect number";
9     std::wcout << pi << L'\n';
10    std::wcout << perfect << L'\n';
11    return 0;
12 }
```

Possible output:

```
pi is 3.141593
28 is a perfect number
```

## Exceptions

The [wstring](#) constructor may throw.

## See also

<a href="#">swprintf</a>	Write formatted data to wide string ( <a href="#">function</a> )
--------------------------	--

<a href="#">to_string</a>	Convert numerical value to string ( <a href="#">function</a> )
---------------------------	--

## /string/u16string

class

### std::u16string

<string>

`typedef basic_string<char16_t> u16string;`

#### String of 16-bit characters

String class for 16-bit characters.

This is an instantiation of the [basic\\_string](#) class template that uses `char16_t` as the character type, with its default `char_traits` and `allocator` types (see [basic\\_string](#) for more info on the template).

## Member types

member type	definition
<code>value_type</code>	<code>char16_t</code>
<code>traits_type</code>	<code>char_traits&lt;char16_t&gt;</code>
<code>allocator_type</code>	<code>allocator&lt;char16_t&gt;</code>
<code>reference</code>	<code>char16_t&amp;</code>
<code>const_reference</code>	<code>const char16_t&amp;</code>

<code>pointer</code>	<code>char16_t*</code>
<code>const_pointer</code>	<code>const char16_t*</code>
<code>iterator</code>	a random access iterator to <code>char16_t</code> (convertible to <code>const_iterator</code> )
<code>const_iterator</code>	a random access iterator to <code>const char16_t</code>
<code>reverse_iterator</code>	<code>reverse_iterator&lt;iterator&gt;</code>
<code>const_reverse_iterator</code>	<code>reverse_iterator&lt;const_iterator&gt;</code>
<code>difference_type</code>	<code>ptrdiff_t</code>
<code>size_type</code>	<code>size_t</code>

## Member functions

Note: The references to the members of its basic template (`basic_string`) are linked here.

<b>(constructor)</b>	Construct <code>basic_string</code> object (public member function )
<b>(destructor)</b>	String destructor (public member function )
<b>operator=</b>	String assignment (public member function )

### Iterators:

<code>begin</code>	Return iterator to beginning (public member function )
<code>end</code>	Return iterator to end (public member function )
<code>rbegin</code>	Return reverse iterator to reverse beginning (public member function )
<code>rend</code>	Return reverse iterator to reverse end (public member function )
<code>cbegin</code>	Return <code>const_iterator</code> to beginning (public member function )
<code>cend</code>	Return <code>const_iterator</code> to end (public member function )
<code>crbegin</code>	Return <code>const_reverse_iterator</code> to reverse beginning (public member function )
<code>crend</code>	Return <code>const_reverse_iterator</code> to reverse end (public member function )

### Capacity:

<code>size</code>	Return size (public member function )
<code>length</code>	Return length of string (public member function )
<code>max_size</code>	Return maximum size (public member function )
<code>resize</code>	Resize string (public member function )
<code>capacity</code>	Return size of allocated storage (public member function )
<code>reserve</code>	Request a change in capacity (public member function )
<code>clear</code>	Clear string (public member function )
<code>empty</code>	Test whether string is empty (public member function )
<code>shrink_to_fit</code>	Shrink to fit (public member function )

### Element access:

<b>operator[]</b>	Get character of string (public member function )
<code>at</code>	Get character of string (public member function )
<code>back</code>	Access last character (public member function )
<code>front</code>	Access first character (public member function )

### Modifiers:

<b>operator+=</b>	Append to string (public member function )
<code>append</code>	Append to string (public member function )
<code>push_back</code>	Append character to string (public member function )
<code>assign</code>	Assign content to string (public member function )
<code>insert</code>	Insert into string (public member function )
<code>erase</code>	Erase characters from string (public member function )
<code>replace</code>	Replace portion of string (public member function )
<code>swap</code>	Swap string values (public member function )
<code>pop_back</code>	Delete last character (public member function )

### String operations:

<code>c_str</code>	Get C-string equivalent
<code>data</code>	Get string data (public member function )
<code>get_allocator</code>	Get allocator (public member function )
<code>copy</code>	Copy sequence of characters from string (public member function )
<code>find</code>	Find first occurrence in string (public member function )
<code>rfind</code>	Find last occurrence in string (public member function )
<code>find_first_of</code>	Find character in string (public member function )
<code>find_last_of</code>	Find character in string from the end (public member function )
<code>find_first_not_of</code>	Find non-matching character in string (public member function )
<code>find_last_not_of</code>	Find non-matching character in string from the end (public member function )
<code>substr</code>	Generate substring (public member function )
<code>compare</code>	Compare strings (public member function )

## Member constants

Note: The references to the members of its basic template ([basic\\_string](#)) are linked here.

<a href="#">npos</a>	Maximum value of size_type ( <a href="#">public static member constant</a> )
----------------------	--

## **Non-member functions overloads**

---

Note: The references to the general overloads of its basic template ([basic\\_string](#)) are shown here.

<a href="#">operator+</a>	Concatenate strings ( <a href="#">function template</a> )
---------------------------	---

<a href="#">relational operators</a>	Relational operators for <a href="#">basic_string</a> ( <a href="#">function template</a> )
--------------------------------------	---

<a href="#">swap</a>	Exchanges the values of two strings ( <a href="#">function template</a> )
----------------------	---

<a href="#">operator&gt;&gt;</a>	Extract string from stream ( <a href="#">function template</a> )
----------------------------------	--

<a href="#">operator&lt;&lt;</a>
----------------------------------