C++ Memoire

[<iostream> 2](#_Toc74075360)

[printf() vs cout 2](#_Toc74075361)

[<istream> 3](#_Toc74075362)

[<string.h> 4](#_Toc74075363)

[definition 4](#_Toc74075364)

[member functions 4](#_Toc74075365)

[iterators 4](#_Toc74075366)

[capacity 4](#_Toc74075367)

[element access 4](#_Toc74075368)

[modifiers 5](#_Toc74075369)

[string operations 5](#_Toc74075370)

[constants 6](#_Toc74075371)

[overloads 6](#_Toc74075372)

[member types 6](#_Toc74075373)

[<regex> 7](#_Toc74075374)

[Templates 9](#_Toc74075375)

# Standard C libraries are included in C++

See *C-memoire* for usage.

## C libraries included in C++ include:

<stdio.h>

<…>

# pointers

address operator:

int \*yPtr = &y;

dereference operator:

\*y = 23;

int z = \*y;

const value: *const* keyword before type

const pointer: *const* keyword between asterisk and varname

trick for deciphering pointer declarations: read it backwards, i.e: countPtr is a const pointer to an int that is const

nonconstant pointer to nonconstant data:

int\* countPtr

nonconstant pointer to constant data:

const int\* countPtr

constant pointer to nonconstant data:

int\* const countPtr

constant pointer to constant data:

const int\* const countPtr

# references

int& y = x;

References cannot be null, must be initialized when they are created, and cannot be changed to refer to another object.

A reference serves as a second label attached to the same memory location as the original variable, and once created can be used in the same way as that variable (that is they don't need to be dereferenced before use) - it is the same variable, only by another name.

References are 'safer' than pointers, but are also less powerful

References should be preferred over pointers, where possible

# [exceptions](https://en.cppreference.com/w/cpp/error/exception)

logic\_error

Faulty logic within program

invalid\_argument

Argument value has not been accepted

domain\_error

Input is outside of domain for which operation is defined

length\_error

Attempt to exceed maximum allow size

out\_of\_range

Attempt to access element outside defined range

future\_error

bad\_optional\_access

runtime\_error

Error due to events beyond scope of the program and not easily predicated

range\_error

Result cannot be represented by the destination type

overflow\_error

Result is too large for destination type

underflow\_error

Result is a subnormal floating-point value

regex\_error

system\_error

ios\_base::failure

filesystem::filesystem\_error

tx\_exception

nonexistant\_local\_time

ambiguous\_local\_time

format\_error

bad\_typeid

bad\_cast

bad\_any\_cast

bad\_weak\_ptr

bad\_function\_call

bad\_alloc

bad\_array\_new\_length

bad\_exception

ios\_base::failure

bad\_variant\_access

## methods

(Constructor)

(Destructor)

operator=

Copies exception object

what()

Returns an explanatory string

# <iostream>

Standard I/O stream objects header. Mostly defined in *<istream>*.

cin

Object of class *istream*, representing *stdin*, using *char* (narrow, single-byte) characters, providing input from a file, pipe, or keyboard (see: *istream* for operations supported by *cin*). *cin* is tied to *cout*, meaning the *cout* buffer is flushed before each *cin* operation. A program should mix operations on *cin* with operations on *wcin* (or other wide-oriented inputs – when an input operation is performed on a stream, it becomes either *narrow*, or *wide*, unless changed by call to *freopen()*.

cout

cerr

clog

wcin

wcout

wcerr

wclog

*scanf()* / *printf()* operations can be intermixed with *cin* / *cout*, disabling this can increase performance of *cin* / *cout*:

std::ios::sync\_with\_stdio(false);

*endl* is considerably slower than using *'\n'*.

## printf() vs cout

<https://stackoverflow.com/questions/2872543/printf-vs-cout-in-c/20238349>

*printf()* is considerably faster than *cout* (but still fast enough that io is likely to be a bottleneck)

*printf()* cannot (easily) be used to print the null byte.

*cout* provides simpler handling of type.

*cout* (being a class) can be extended.

*cout* provides a ‘simpler’ syntax.

*printf()* provides precise (yet relatively simple) formatting

*cout* is 'safer'

# [<iomanip>](https://www.cplusplus.com/reference/iomanip/)

*fill character* is the character used to pad fields of a specified width.

setiosflags(*ios\_base::fmtflags mask*)

Set the format flags specified by *mask*. Behaves as if *setf()* were called with *mask* as argument on the stream on which it is inserted/extracted.

resetiosflags(*ios\_base::fmtflags mask*)

Unsets the format flags specified by *mask*. Behaves as if *setf()* were called with *mask* as argument on the stream on which it is inserted/extracted.

setbase(*int base*)

Sets the *basefield* to one of its possible values: *dec*, *hex*, *oct*, or *zero* according to *base* (*10*, *16*, *8*, *other* respectively). Can be inserted/extracted on input and output streams.

setfill(*char\_type c*)

Sets *c* as the streams *fill character*. Behaves as if *fill()* were called with *c* as argument on output stream.

setprecision(*int n*)

Sets the decimal precision to be used to format floating point values on output. Behaves as if *precision()* were called with *n* as argument on stream on which it is inserted/extracted.

setw(*int n*)

Sets the *field width* to be used on output operations. Behaves as if *width()* were called with *n* as argument on stream on which it is inserted/extracted.

get\_money(*moneyT& mon, bool intl = false*)

Extracts characters from the input stream it is applied to, interpreting them as a monetary expression, which is stored in *mon* (which is either *long double* or *basic\_string*). *intl* should be true for international representations.

put\_money(*const moneyT& mon, bool intl = false*)

Inserts the representation of *mon* (either *long double* or *basic\_string*) as a monetary value into the output stream it is applied to.

get\_time(*struct tm\* tmb, const charT\* fmt*)

Extracts characters from the input stream it applied to, interpreting them as a time and date, of format *fmt*, storing it in *tmb* (pointer to *struct tm*).

put\_time(*const struct tm\* tmb, const charT\* fmt*)

Inserts representation of the datetime *tmb*, with format *fmt*.

# [<istream>](http://www.cplusplus.com/reference/istream/istream/)

# [<string.h>](http://www.cplusplus.com/reference/string/string/)

*Use <cstring> for null terminated C string library in C++.*

## definition

typdef basic\_string<char> string

The C++ string is an object that represents a sequence of characters, similar to that of a standard container of bytes, but with added features specifically for single-byte characters. Specifically, *string* is an instantiation of the *basic\_string* class template that uses *char* as its character type, with its default *char\_traits* and allocator types (see: *basic\_string*). Note: the string is stored/processed/iterated in terms of bytes, regardless of text encoding being used.

## member functions

(constructor)

(destructor)

operator=

## iterators

begin()

end()

rbegin()

rend()

cbegin()

cend()

crbegin()

crend()

## capacity

size()

length()

max\_size()

resize()

capacity()

reserve()

clear()

empty()

shrink\_to\_fit()

## element access

operator[]

append()

push\_back()

assign()

insert()

erase()

replace()

swap()

pop\_back()

## modifiers

operator+=

append()

push\_back()

assign()

insert()

erase()

replace()

swap()

pop\_back()

## string operations

c\_str()

data()

get\_allocator()

copy()

find()

rfind()

find\_first\_of()

find\_last\_of()

find\_first\_not\_of()

find\_last\_not\_of()

substr()

compare()

## constants

npos

## overloads

operator+

relational operators

swap()

operator>>

operator<<

getline()

## member types

|  |  |
| --- | --- |
| **member type** | **definition** |
| value\_type | char |
| traits\_type | [char\_traits](http://www.cplusplus.com/char_traits)<char> |
| allocator\_type | [allocator](http://www.cplusplus.com/allocator)<char> |
| reference | char& |
| const\_reference | const char& |
| pointer | char\* |
| const\_pointer | const char\* |
| iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to char (convertible to const\_iterator) |
| const\_iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to const char |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |
| difference\_type | [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |
| size\_type | [size\_t](http://www.cplusplus.com/size_t) |

# (others)

<iomanip>

<cmah>

<cstdlib>

<ctime>

<vector>

<list>

<deque>

<queue>

<stack>

<map>

<set>

<bitset>

<cctype>

<cstring>

<typeinfo>

<exception>

<stdexcept>

<memory>

<fstream>

<sstream>

<functional>

<iterator>

<algorithm>

<cassert>

<cfloat>

<climits>

<cstdio>

<locale>

<limits>

<utility>

# <vector>

template < class T, class Alloc = allocator<T> > class vector;

Vectors in C++ are arrays that can change in size. Internally implemented as a dynamically sized array with additional space allocated for new data (meaning a slow resize is not necessary for each insertion at the end, all insertions in the middle are slow), accessing elements is extremely fast.

*T* gives the type of elements, aliased as member type *vector::value\_type*.

*Alloc* is the type of allocator used to define storage allocation model, aliased as member type *vector::allocator\_type*.

## methods

### constructor - default

explicit vector(*const allocator\_type& alloc = allocator\_type()*)

Empty container with no elements

### constructor - fill

explicit vector(*size\_type n*);

vector(*size\_type n, const value\_type& val, const allocator\_type& alloc = allocator\_type()*)

Container with *n* elements, each of which is a copy of *val* (if given).

### constructor - range

template <class InputIterator>

vector(*InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type()*)

Container with as many elements as the range *[first, last)*, with each element emplace-constructed from its corresponding element in that range.

### constructor – copy

vector(*const vector& x*)

vector(*const vector& x, const allocator\_type& alloc*)

Constructs a container with a copy of each element in *x*, in the same order.

### constructor – move

vector(*vector&& x*)

vector(*vector&& x, const allocator\_type& alloc*)

Container that acquires the elements of *x*. If *alloc* is specified and is different from the allocator of *x*, the elements are moved, otherwise no elements are constructed (their ownership is directly transferred). *x* is left in an unspecified but valid state.

### constructor – initializer list

vector(*initializer\_list<value\_type> il, const allocator\_type& alloc = allocator\_type()*)

Constructs a container with a copy of each element in *il*, in the same order.

### destructor

~vector()

Calls *allocator\_traits::destroy* on each of the contained elements, and deallocates storage allocated by vector.

### operator= - copy

vector& operator= (*const vector& x*)

Copies elements from *x* as new contents of container, with *x* preserving its contents.

### operator= - move

vector& operator= (*vector&& x*)

Moves the elements of *x* as the new contents of container, with *x* left in an unspecified valid state.

### operator = - initializer list

vector& operator= (*initalizer\_list<value\_type> li*)

Copies the elements of *li* into the container

### iterators

begin()

Return a random-access-iterator pointing to the first element. If the vector object is const-qualified, this is a *const\_iterator*.

end()

Returns an iterator referencing the past-the-end element, which is the end index for functions where ranges are exclusive of the end element. If the vector object is const-qualified, this is a *const\_iterator*.

rbegin()

Returns a reverse-random-access-iterator pointing to the last element in the vector. If the vector object is const-qualified, this is a const\_iterator.

rend()

Returns a reverse-iterator pointing to the element preceding the first. If the vector object is const-qualified, this is a const\_iterator.

cbegin()

Returns a const-random-access-iterator pointing to the first element in the container.

cend()

Returns a const-random-access-iterator pointing to the past-the-end element.

crbegin()

Returns a const-reverse-iterator to the last element in the vector.

crend()

Returns a const-reverse-iterator to the element preceding the first.

### capacity

size()

Number of elements in the vector, which may be different from storage capacity.

max\_size()

Maximum vector size defined by system implementation.

resize(*size\_type n[, const value\_type& val]*)

Resizes the container so that it contains *n* elements. *val* is copied to new elements where the vector is enlarged (or use default constructor if not given).

capacity()

Returns size of storage space currently allocated, in terms of elements, which may be different from *size()*.

empty()

true if vector *size()* is 0, false otherwise.

reserve(*size\_type n*)

Request that storage capacity be increased to at least enough to contain *n* elements.

shrink\_to\_fit()

Request the container reduce its capacity to fit its current *size()*.

### element access

operator[]

Returns a reference to the element at position *n*. Do not call with *n* out-of-range. If the vector is const-qualified, this reference is a *const\_reference*.

at()

Returns a reference to the element at position *n*. Checks bounds and throws *out\_of\_range* exception accordingly. If the vector is const-qualified, this reference is a *const\_reference*.

front()

Returns a reference to the first element in the vector. Behaviour is undefined for an empty vector. If the vector is const-qualified, this reference is a *const\_reference*.

back()

Returns a reference to the last element in the vector. Behaviour is undefined for an empty vector. If the vector is const-qualified, this reference is a *const\_reference*.

data()

Returns a pointer to the internal array used by vector. Elements in this array are guaranteed to be contiguous, that is, this vector can be offset to access any element in the array.

### modifiers

*range*

template <class InputIterator>

void assign(*InputIterator first, InputIterator last*)

Replace contents of vector, new contents are elements constructed from each element in range between *first* and *last*, in the same order.

*fill*

void assign(*size\_type n, const value\_type& val*)

Replace contents of vector, new contents are *n* elements each initialized to a copy of *val*.

*initializer list*

void assign(*initializer\_list<value\_type> il*);

Replace contents of vector, new contents are copies of the values passed as initializer list, in same order.

push\_back(*const value\_type& val*)

push\_back(*value\_type&& val*)

Add a new element to the end of the vector. The content of *val* is copied (or moved) to the new element.

pop\_back()

Removes the last element in the vector. Behaviour is undefined for an empty vector.

*single element*

iterator insert(*const\_iterator position, const value\_type& val*)

*fill*

iterator insert(*const\_iterator position, size\_type n, const value\_type& val*)

*range*

template <class InputIterator>

iterator insert(*const\_iterator position, InputIterator first, InputIterator last*)

*move*

iterator insert(*const\_iterator position, value\_type&& val*)

*initializer list*

iterator insert(*const\_iterator position, initializer\_list<value\_type> il*)

Vector is extended by inserting new elements before the specified *position*. Returns an iterator pointing to the first of the newely inserted elements.

iterator erase(*const\_iterator position*)

iterator erase(*const\_iterator first, const\_iterator last*)

Remove from the vector either a single element (position) or a range of elements *[first, last)*. Returns an iterator pointing to the new location of the element that followed the last element erased (or the container end if the last element was erased).

swap(*vector& x*)

Exchanges the contents of vector with that of *x*, which is another vector of the same type.

clear()

Remove all elements from the vector (which are destroyed).

template <class... Args>

iterator emplace(*const\_iterator position, Args&&... args*)

The container is extended by inserting a new element at *position*. This element is constructed in place using *args* as arguments for its construction. Returns an iterator that points to the newly emplaced element.

emplace\_back()

Inserts a new element at the end of the vector. This element is constructed in place using *args* as arguments for its construction. Returns an iterator that points to the newly emplaced element.

### allocator

get\_allocator()

Returns a copy of the allocator object associated with the vector.

### non-member function overloads

relational operators: == != < <= > >=

First compares LHS and RHS vector sizes, then compares elements sequentially.

template <class T, class Alloc>

void swap(*vector<T,Alloc>& x, vector<T,Alloc>& y*);

The contents of the container *x* are exchanged with those of *y*. Both containers must be of the same type. This is an overload of the utility function *swap()*.

### template specializations

vector<bool>

Specialized version of vector, with elements of type bool, stored with a single bit per value to optimise space.

## member types

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| value\_type | The first template parameter (T) |  |
| allocator\_type | The second template parameter (Alloc) | defaults to: [allocator](https://www.cplusplus.com/allocator)<value\_type> |
| reference | value\_type& |  |
| const\_reference | const value\_type& |  |
| pointer | [allocator\_traits](https://www.cplusplus.com/allocator_traits)<allocator\_type>::pointer | for the default [allocator](https://www.cplusplus.com/allocator): value\_type\* |
| const\_pointer | [allocator\_traits](https://www.cplusplus.com/allocator_traits)<allocator\_type>::const\_pointer | for the default [allocator](https://www.cplusplus.com/allocator): const value\_type\* |
| iterator | a [random access iterator](https://www.cplusplus.com/RandomAccessIterator) to value\_type | convertible to const\_iterator |
| const\_iterator | a [random access iterator](https://www.cplusplus.com/RandomAccessIterator) to const value\_type |  |
| reverse\_iterator | [reverse\_iterator](https://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](https://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | a signed integral type, identical to: iterator\_traits<iterator>::difference\_type | usually the same as [ptrdiff\_t](https://www.cplusplus.com/ptrdiff_t) |
| size\_type | an unsigned integral type that can represent any non-negative value of difference\_type | usually the same as [size\_t](https://www.cplusplus.com/size_t) |

# <regex>

# <memory>

unique\_ptr

# [<filesystem>](https://en.cppreference.com/w/cpp/filesystem)

classes

path

filesystem\_error

directory\_entry

directory\_iterator

recursive\_directory\_iterator

file\_status

space\_info

file\_type

perms

perm\_options

copy\_options

directory\_options

file\_time\_type

non-member functions

absolute

canonical

weakly\_canonical

relative

proximate

copy

copy\_file

copy\_symlink

create\_directory

create\_directories

create\_hard\_link

create\_symlink

create\_directory\_symlink

current\_path

exists

equivalent

file\_size

hard\_link\_count

last\_write\_time

permissions

read\_symlink

remove

remove\_all

rename

resize\_file

space

status

symlinks\_status

temp\_directory\_path

file types

is\_block\_file

is\_character\_file

is\_directory

is\_empty

is\_fifo

is\_other

is\_regular\_file

is\_socket

is\_symlink

status\_known

# Templates

Function overloading

cannot be overloaded

1. Function declarations that differ only in return type
2. Member functions if any of them is a static member functions
3. Functions with argument declarations that differ only in pointer/array (since an array as function argument in C/C++ is converted to a pointer as argument)
4. Functions with argument declarations that differ only in that one is a function type, and the other is a pointer to the same function type (that is: *‘void h(int ())’* vs *‘**void h(int (\*)())’*
5. Functions with argument declarations that differ only in present or absence of *const* / *volatile* specifiers.
6. Functions with argument declarations that differ only in their default values

d