std::stack

Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

Template parameters

`T` - The type of the stored elements.

`Container` - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:

- `back()`
- `push_back()`
- `pop_back()`

The standard containers `std::vector`, `std::deque` and `std::list` satisfy these requirements.

Member types

Member type	Definition
`container_type`	`Container`
`value_type`	`Container::value_type`
`size_type`	`Container::size_type`
`reference`	`Container::reference`
`const_reference`	`Container::const_reference`

Member functions

(constructor)	constructs the `stack` (public member function)
(destructor)	destructs the `stack` (public member function)
`operator=`	assigns values to the container adaptor (public member function)

Element access

`top`	accesses the top element (public member function)

Capacity

`empty`	checks whether the underlying container is empty (public member function)
	returns the number of elements

size	(public member function)

Modifiers

push	inserts element at the top (public member function)
emplace (C++11)	constructs element in-place at the top (public member function)
pop	removes the top element (public member function)
swap	swaps the contents (public member function)

Member objects

`Container` **c**	the underlying container (protected member object)

Non-member functions

operator== **operator!=** **operator<** **operator<=** **operator>** **operator>=**	lexicographically compares the values in the stack (function template)
std::swap`(std::stack)`	specializes the `std::swap` algorithm (function template)

Helper classes

std::uses_allocator`<std::stack>` (C++11)	specializes the `std::uses_allocator` type trait (function template)

std::stack::stack

`explicit stack( const Container& cont = Container() );`	(1)	(until C++11)
`explicit stack( const Container& cont );`		(since C++11)
`explicit stack( Container&& cont = Container() );`	(2)	(since C++11)
`stack( const stack& other );`	(3)	
`stack( stack&& other );`	(4)	(since C++11)
`template< class Alloc >` `explicit stack( const Alloc& alloc );`	(5)	(since C++11)
`template< class Alloc >` `stack( const Container& cont, const Alloc& alloc );`	(6)	(since C++11)
`template< class Alloc >` `stack( Container&& cont, const Alloc& alloc );`	(7)	(since C++11)
`template< class Alloc >` `stack( const stack& other, const Alloc& alloc );`	(8)	(since C++11)
`template< class Alloc >` `stack( stack&& other, const Alloc& alloc );`	(9)	(since C++11)

Constructs new underlying container of the container adaptor from a variety of data sources.

1) Copy-constructs the underlying container `c` with the contents of `cont`. This is also the default constructor (until C++11)

2) Move-constructs the underlying container `c` with `std::move(cont)`. This is also the default constructor (since C++11)

3) Copy constructor. The adaptor is copy-constructed with the contents of `other.c`. (implicitly declared)

4) Move constructor. The adaptor is constructed with `std::move(other.c)`. (implicitly declared)

5-9) The following constructors are only defined if `std::uses_allocator<container_type, Alloc>::value == true`, that is, if the underlying container is an allocator-aware container (true for all standard library containers).

5) Constructs the underlying container using `alloc` as allocator. Effectively calls `c(alloc)`.

6) Constructs the underlying container with the contents of `cont` and using `alloc` as allocator. Effectively calls `c(cont, alloc)`.

7) Constructs the underlying container with the contents of `cont` using move semantics while utilising `alloc` as allocator. Effectively calls `c(std::move(cont), alloc)`.

8) Constructs the adaptor with the contents of `other.c` and using `alloc` as allocator. Effectively calls `c(athor.c, alloc)`.

9) Constructs the adaptor with the contents of `other` using move semantics while utilising `alloc` as allocator. Effectively calls `c(std::move(other.c), alloc)`.

Parameters

`alloc`	-	allocator to use for all memory allocations of the underlying container
`other`	-	another container adaptor to be used as source to initialize the underlying container
`cont`	-	container to be used as source to initialize the underlying container
`first, last`	-	range of elements to initialize with

Type requirements

- `Alloc` must meet the requirements of `Allocator`.
- `Container` must meet the requirements of `Container`. The constructors (5-10) are only defined if `Container` meets the requirements of `AllocatorAwareContainer`
- `InputIt` must meet the requirements of `InputIterator`.

Complexity

1, 3, 5, 6, 8: linear in `cont` or `other`

2, 4, 7, 9: constant

This section is incomplete

Example

Run this code

```
#include <stack>
#include <deque>
#include <iostream>

int main()
{
    std::stack<int> c1;
    c1.push(5);
    std::cout << c1.size() << '\n';

    std::stack<int> c2(c1);
    std::cout << c2.size() << '\n';

    std::deque<int> deq {3, 1, 4, 1, 5};
    std::stack<int> c3(deq);
    std::cout << c3.size() << '\n';
}
```

Output:

```
1
1
5
```

See also

operator=	assigns values to the container adaptor (public member function)

std::stack::~stack

```
~stack();
```

Destructs the container adaptor. The destructors of the elements are called and the used storage is deallocated. Note, that if the elements are pointers, the pointed-to objects are not destroyed.

Complexity

Linear in the size of the container adaptor.

std::stack::operator=

```
stack operator=( const stack& other );      (1)
stack operator=( stack&& other );           (2)  (since C++11)
```

Replaces the contents of the container adaptor with those of `other`.

1) Copy assignment operator. Replaces the contents with a copy of the contents of `other`. Effectively calls `c = other.c;` . (implicitly declared)
2) Move assignment operator. Replaces the contents with those of `other` using move semantics. Effectively calls `c = std::move(other.c);` (implicitly declared)

Parameters

other - another container adaptor to be used as source

Return value

`*this`

Complexity

Equivalent to that of `operator=` of the underlying container.

See also

(constructor)	constructs the `stack` (public member function)

std::stack::top

```
reference top();
const_reference top() const;
```

Returns reference to the top element in the stack. This is the most recently pushed element. This element will be removed on a call to `pop()`. Effectively calls `c.back()`.

Parameters

(none)

Return value

Reference to the last element

Complexity

Constant

Example

Run this code

```
#include <stack>
#include <iostream>

int main()
{
    std::stack<int>   s;

    s.push( 2 );
    s.push( 6 );
    s.push( 51 );

    std::cout << s.size() << " elements on stack\n";
    std::cout << "Top element: "
              << s.top()         // Leaves element on stack
              << "\n";
    std::cout << s.size() << " elements on stack\n";
    s.pop();
    std::cout << s.size() << " elements on stack\n";
    std::cout << "Top element: " << s.top() << "\n";

    return 0;
}
```

Output:

```
3 elements on stack
Top element: 51
3 elements on stack
2 elements on stack
Top element: 6
```

See also

push	inserts element at the top (public member function)
pop	removes the top element (public member function)

std::stack::empty

```
bool empty() const;
```

Checks if the underlying container has no elements, i.e. whether `c.empty()`.

Parameters

(none)

Return value

`true` if the underlying container is empty, `false` otherwise

Complexity

Constant

See also

`size`	returns the number of elements (public member function)

std::stack::size

```
size_type size() const;
```

Returns the number of elements in the underlying container, that is, `c.size()`.

Parameters

(none)

Return value

The number of elements in the container.

Complexity

Constant.

See also

empty	checks whether the underlying container is empty (public member function)

std::stack::push

```
void push( const T& value );
void push( T&& value );        (since C++11)
```

Pushes the given element `value` to the top of the stack.

1) Effectively calls `c.push_back(value)`
2) Effectively calls `c.push_back(std::move(value))`

Parameters

`value` - the value of the element to push

Return value

(none)

Complexity

Equal to the complexity of `Container::push_back`.

See also

`emplace` (C++11)	constructs element in-place at the top (public member function)
`pop`	removes the top element (public member function)

std::stack::emplace

```
template< class... Args >
void emplace( Args&&... args );
```
(since C++11)

Pushes new element on top of the stack. The element is constructed in-place, i.e. no copy or move operations are performed. The constructor of the element is called with exactly the same arguments as supplied to the function.

Effectively calls `c.emplace_back(std::forward<Args>(args)...)`

Parameters

args - arguments to forward to the constructor of the element

Return value

(none)

Complexity

Identical to the complexity of `Container::emplace_back`.

See also

push	inserts element at the top (public member function)
pop	removes the top element (public member function)

std::stack::pop

```
void pop();
```

Removes the top element from the stack. Effectively calls `c.pop_back()`

Parameters

(none)

Return value

(none)

Complexity

Equal to the complexity of `Container::pop_back`.

See also

emplace (C++11)	constructs element in-place at the top (public member function)
push	inserts element at the top (public member function)

std::stack::swap

```
void swap( stack& other );    (since C++11)
```

Exchanges the contents of the container adaptor with those of `other`. Effectively calls `using std::swap; swap(c, other.c);`

Parameters

other - container adaptor to exchange the contents with

Return value

(none)

Exceptions

`noexcept` specification:

```
noexcept(noexcept(std::swap(c, other.c)))
```

Complexity

Same as underlying container (typically constant)

See also

std::swap(std::stack)	specializes the `std::swap` algorithm (function template)

operator==,!=,<,<=,>,>=(std::stack)

```
template< class T, class Container >
bool operator==( stack<T,Container>& lhs,      (1)
                 stack<T,Container>& rhs );
template< class T, class Container >
bool operator!=( stack<T,Container>& lhs,      (2)
                 stack<T,Container>& rhs );
template< class T, class Container >
bool operator<( stack<T,Container>& lhs,       (3)
                stack<T,Container>& rhs );
template< class T, class Container >
bool operator<=( stack<T,Container>& lhs,      (4)
                 stack<T,Container>& rhs );
template< class T, class Container >
bool operator>( stack<T,Container>& lhs,       (5)
                stack<T,Container>& rhs );
template< class T, class Container >
bool operator>=( stack<T,Container>& lhs,      (6)
                 stack<T,Container>& rhs );
```

Compares the contents of the underlying containers of two container adaptors. The comparison is done by applying the corresponding operator to the underlying containers.

Parameters

lhs, rhs - container adaptors whose contents to compare
- `T` must meet the requirements of `EqualityComparable`.

Return value

`true` if the corresponding comparison yields `true`, `false` otherwise.

Complexity

Linear in the size of the container

std::swap(std::stack)

```
template< class T, class Container >
void swap( stack<T,Container>& lhs,
           stack<T,Container>& rhs );
```

Specializes the `std::swap` algorithm for `std::stack`. Swaps the contents of `lhs` and `rhs`. Calls `lhs.swap(rhs)`.

Parameters

`lhs, rhs` - containers whose contents to swap

Return value

(none)

Complexity

Same as swapping the underlying container.

Exceptions

(since C++17)

`noexcept` specification:

```
noexcept(noexcept(lhs.swap(rhs)))
```

See also

`swap`	swaps the contents (public member function)

std::uses_allocator<std::stack>

```
template< class T, class Container, class Alloc >
struct uses_allocator<stack<T,Container>,Alloc> :        (since C++11)
    std::uses_allocator<Container, Alloc>::type { };
```

Provides a transparent specialization of the `std::uses_allocator` type trait for `std::stack` : the container adaptor uses allocator if and only if the underlying container does.

Inherited from std::integral_constant

Member constants

value [static]	`true` (public static member constant)

Member functions

operator bool	converts the object to `bool` , returns `value` (public member function)
operator() (C++14)	returns `value` (public member function)

Member types

Type	Definition
`value_type`	`bool`
`type`	`std::integral_constant<bool, value>`

See also

uses_allocator (C++11)	checks if the specified type supports uses-allocator construction (class template)