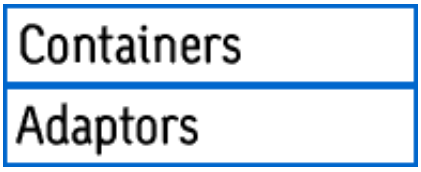
stack<T, Sequence>



Categories: containers, adaptors

Component type: type

Description

A `stack` is an adaptor that provides a restricted subset of Container functionality: it provides insertion, removal, and inspection of the element at the top of the stack. `Stack` is a "last in first out" (LIFO) data structure: the element at the top of a `stack` is the one that was most recently added. [1] `Stack` does not allow iteration through its elements. [2]

`Stack` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is `deque`, but a different type may be selected explicitly.

Example

```
int main() {
  stack<int> S;
  S.push(8);
  S.push(7);
  S.push(4);
  assert(S.size() == 3);

  assert(S.top() == 4);
  S.pop();

  assert(S.top() == 7);
  S.pop();

  assert(S.top() == 8);
  S.pop();

  assert(S.empty());
}
```

Definition

Defined in the standard header stack, and in the nonstandard backward-compatibility header stack.h.

Template parameters

Parameter	Description	Default
`T`	The type of object stored in the stack.	
`Sequence`	The type of the underlying container used to implement the stack.	`deque<T>`

Model of

Assignable, Default Constructible

Type requirements

- `T` is a model of Assignable.
- `Sequence` is a model of Back Insertion Sequence.
- `Sequence::value_type` is the same type as `T`.
- If `operator==` is used, then `T` is a model of Equality Comparable
- If `operator<` is used, then `T` is a model of LessThan Comparable.

Public base classes

None.

Members

Member	Where defined	Description
`value_type`	`stack`	See below.
`size_type`	`stack`	See below.
`stack()`	Default Constructible	The default constructor. Creates an empty `stack`.
`stack(const stack&)`	Assignable	The copy constructor.
`stack& operator=(const stack&)`	Assignable	The assignment operator.
`bool empty() const`	`stack`	See below.
`size_type size() const`	`stack`	See below.
`value_type& top()`	`stack`	See below.
`const value_type& top() const`	`stack`	See below.
`void push(const value_type&)`	`stack`	See below.
`void pop()` [3]	`stack`	See below.
`bool operator==(const stack&, const stack&)`	`stack`	See below.
`bool operator<(const stack&, const stack&)`	`stack`	See below.

New members

These members are not defined in the Assignable and Default Constructible requirements, but are specific to `stack`.

Member	Description
`value_type`	The type of object stored in the `stack`. This is the same as `T` and `Sequence::value_type`.
`size_type`	An unsigned integral type. This is the same as `Sequence::size_type`.
`bool empty() const`	Returns `true` if the `stack` contains no elements, and `false` otherwise. `S.empty()` is equivalent to `S.size() == 0`.
`size_type size() const`	Returns the number of elements contained in the `stack`.
`value_type& top()`	Returns a mutable reference to the element at the top of the stack. Precondition: `empty()` is `false`.
`const value_type& top() const`	Returns a const reference to the element at the top of the stack. Precondition: `empty()` is `false`.
`void push(const value_type& x)`	Inserts `x` at the top of the stack. Postconditions: `size()` will be incremented by `1`, and `top()` will be equal to `x`.
`void pop()`	Removes the element at the top of the stack. [3] Precondition: `empty()` is `false`. Postcondition: `size()` will be decremented by `1`.
`bool operator== (const stack&, const stack&)`	Compares two stacks for equality. Two stacks are equal if they contain the same number of elements and if they are equal element-by-element. This is a global function, not a member function.
`bool operator<(const stack&, const stack&)`	Lexicographical ordering of two stacks. This is a global function, not a member function.

Notes

[1] Stacks are a standard data structure, and are discussed in all algorithm books. See, for example, section 2.2.1 of Knuth. (D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, second edition. Addison-Wesley, 1973.)

[2] This restriction is the only reason for `stack` to exist at all. Note that any Front Insertion Sequence or Back Insertion Sequence can be used as a stack; in the case of vector, for example, the stack operations are the member functions `back`, `push_back`, and `pop_back`. The only reason to use the container adaptor `stack` instead is to make it clear that you are performing only stack operations, and no other operations.

[3] One might wonder why `pop()` returns `void`, instead of `value_type`. That is, why must one use `top()` and `pop()` to examine and remove the top element, instead of combining the two in a single member function? In fact, there is a good reason for this design. If `pop()` returned the top element, it would have to return by value rather than by reference: return by reference would create a dangling pointer. Return by value, however, is inefficient: it involves at least one redundant copy constructor call. Since it is impossible for `pop()` to return a value in such a way as to be both efficient and correct, it is more sensible for it to return no value at all and to require clients to use `top()` to inspect the value at the top of the stack.

See also

queue, priority_queue, Container, Sequence

STL Main Page

Contact Us | Site Map | Trademarks | Privacy | Using this site means you accept its Terms of Use
Copyright © 2009 - 2014 Silicon Graphics International. All rights reserved.