
12.1. Stacks

The class `stack<>` implements a stack (also known as LIFO). With `push()`, you can insert any number of elements into the stack (Figure 12.1). With `pop()`, you can remove the elements in the opposite order in which they were inserted ("last in, first out").

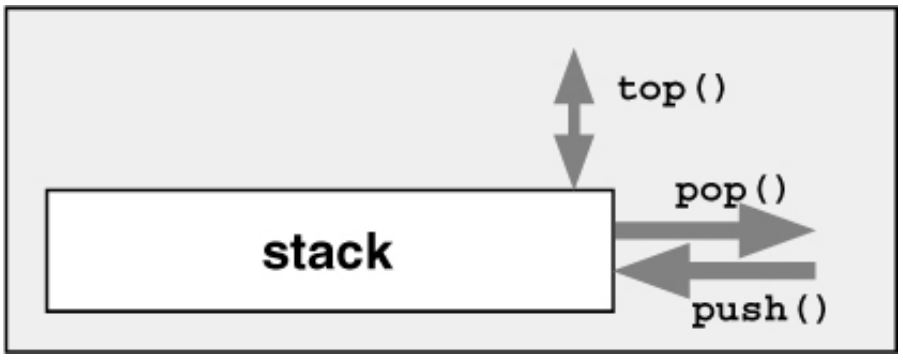

Figure 12.1. Interface of a Stack

To use a stack, you have to include the header file `<stack>`:

```
#include <stack>
```

In `<stack>`, the class `stack` is defined as follows:

```
namespace std {
    template <typename T,
              typename Container = deque<T>>
             class  stack;
}
```

The first template parameter is the type of the elements. The optional second template parameter defines the container that the stack uses internally for its elements. The default container is a deque. It was chosen because, unlike vectors, deques free their memory when elements are removed and don't have to copy all elements on reallocation (see Section 7.12, page 392, for a discussion of when to use which container).

For example, the following declaration defines a stack of integers:

```
std::stack<int> st;     // integer stack
```

The stack implementation simply maps the operations into appropriate calls of the container that is used internally (Figure 12.2). You can use any sequence container class that provides the member functions `back()`, `push_back()`, and `pop_back()`. For example, you could also use a vector or a list as the container for the elements:

```
std::stack<int,std::vector<int>> st;  // integer stack that uses a vector
```

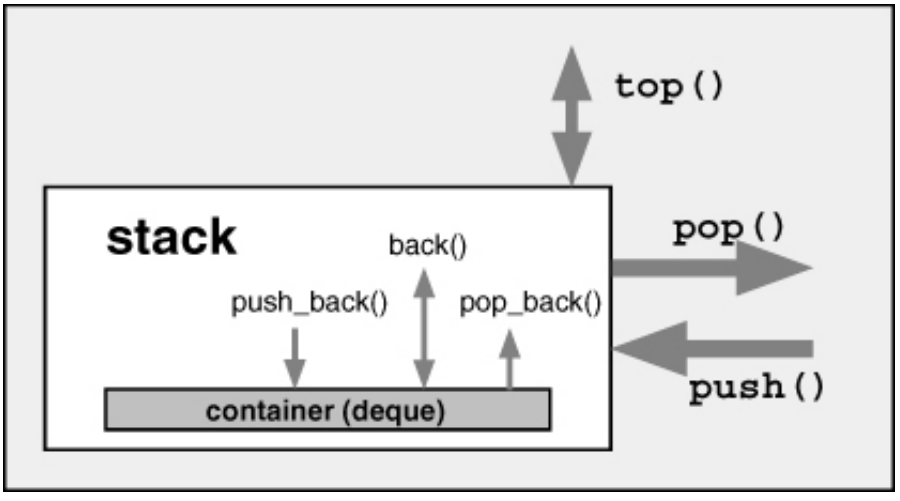

Figure 12.2. Internal Interface of a Stack

12.1.1. The Core Interface

The core interface of stacks is provided by the member functions `push()`, `top()`, and `pop()`:

- **`push()`** inserts an element into the stack.
- **`top()`** returns the next available element in the stack.
- **`pop()`** removes an element from the stack.

Note that `pop()` removes the next element but does not return it, whereas `top()` returns the next element without removing it. Thus, you must always call both functions to process and remove the next element from the stack. This interface is somewhat inconvenient, but it performs better if you want only to remove the next element without processing it. Note that the behavior of `top()` and `pop()` is undefined if the stack contains no elements. The member functions `size()` and `empty()` are provided to check whether the stack contains elements.

If you don't like the standard interface of `stack<>`, you can easily write a more convenient interface. See Section 12.1.3, page 635, for an example.

12.1.2. Example of Using Stacks

The following program demonstrates the use of class `stack<>`:

Click here to view code image

```
// contadapt/stack1.cpp

#include <iostream>
#include <stack>
using namespace std;

int main()
{
    stack<int> st;

    // push three elements into the stack
    st.push(1);
    st.push(2);
    st.push(3);

    // pop and print two elements from the stack
    cout << st.top() << ' ';
    st.pop();
    cout << st.top() << ' ';
    st.pop();

    // modify top element
    st.top() = 77;

    // push two new elements
    st.push(4);
    st.push(5);

    // pop one element without processing it
    st.pop();

    // pop and print remaining elements
    while (!st.empty()) {
        cout << st.top() << ' ';
        st.pop();
    }
    cout << endl;
}
```

The output of the program is as follows:

```
3 2 4 77
```

Note that when using nontrivial element types, you might consider using `std::move()` to insert elements that are no longer used or `emplace()` to let the stack internally create the element (both available since C++11):

Click here to view code image

```
stack<pair<string,string>> st;

auto p = make_pair("hello","world");
st.push(move(p));  // OK, if p is not used any more
```

```
st.emplace("nico","josuttis");
```

12.1.3. A User-Defined Stack Class

The standard class `stack<>` prefers speed over convenience and safety. This is not what I usually prefer, so I have written my own stack class, which has the following two advantages:

1. `pop()` returns the next element.
2. `pop()` and `top()` throw exceptions when the stack is empty.

In addition, I have skipped the members that are not necessary for the ordinary stack user, such as the comparison operations. My stack class is defined as follows:

Click here to view code image

```
// contadapt/Stack.hpp

/* ****************************************************************
 * Stack.hpp
 *  - safer and more convenient stack class
 * ****************************************************************/
#ifndef STACK_HPP
#define STACK_HPP

#include <deque>
#include <exception>

template <typename T>
class Stack {
  protected:
    std::deque<T> c;          // container for the elements

  public:
    // exception class for pop() and top() with empty stack
    class ReadEmptyStack : public std::exception {
      public:
        virtual const char* what() const throw() {
            return "read empty stack";
        }
    };

    // number of elements
    typename std::deque<T>::size_type size() const {
        return c.size();
    }

    // is stack empty?
    bool empty() const {
        return c.empty();
    }

    // push element into the stack
    void push (const T& elem) {
        c.push_back(elem);
    }

    // pop element out of the stack and return its value
    T pop () {
        if (c.empty()) {
            throw ReadEmptyStack();
        }
        T elem(c.back());
        c.pop_back();
        return elem;
    }

    // return value of next element
    T& top () {
        if (c.empty()) {
            throw ReadEmptyStack();
        }
        return c.back();
    }
};

#endif /* STACK_HPP */
```

With this stack class, the previous stack example could be written as follows:

```
// contadapt/stack2.cpp

#include <iostream>
#include <exception>
#include "Stack.hpp"      // use special stack class
using namespace std;

int main()
{
   try {
      Stack<int> st;

      // push three elements into the stack
      st.push(1);
      st.push(2);
      st.push(3);

      // pop and print two elements from the stack
      cout << st.pop() << ' ';
      cout << st.pop() << ' ';

      // modify top element
      st.top() = 77;

      // push two new elements
      st.push(4);
      st.push(5);

      // pop one element without processing it
      st.pop();

      // pop and print three elements
      // - ERROR: one element too many
      cout << st.pop() << ' ';
      cout << st.pop() << endl;
      cout << st.pop() << endl;
   }
   catch (const exception& e) {
      cerr << "EXCEPTION: " << e.what() << endl;
   }
}
```

The additional final call of `pop()` forces an error. Unlike the standard stack class, this one throws an exception rather than resulting in undefined behavior. The output of the program is as follows:

```
3 2 4 77
EXCEPTION: read empty stack
```

12.1.4. Class `stack<>` in Detail

The `stack<>` interface maps more or less directly to corresponding members of the container internally used. For example:

Click here to view code image

```
namespace std {
   template <typename T, typename Container = deque<T>>
   class stack {
     public:
       typedef typename Container::value_type      value_type;
       typedef typename Container::reference       reference
       typedef typename Container::const_reference const_reference;
       typedef typename Container::size_type       size_type;
       typedef          Container                  container_type;
       protected:
         Container c;     // container
       public:
         bool        empty()  const            { return c.empty(); }
         size_type   size()   const            { return c.size(); }
         void        push(const value_type& x) { c.push_back(x); }
         void        push(value_type&& x)      { c.push_back(move(x)); }
         void        pop()                     { c.pop_back(); }
         value_type& top()                     { return c.back(); }
         const value_type& top() const         { return c.back(); }
         template <typename... Args>
         void emplace(Args&&... args) {
                c.emplace_back(std::forward<Args>(args)...); }
         void swap (stack& s)  …  { swap(c,s.c); }
         …
```

```
        };
    }
```

See Section 12.4, page 645, for details of the provided members and operations.