12.1 Stack（堆栈）

Class stack<> 实现出一个 stack（也称为LIFO，后进先出）。你可以使用 push() 将任意数量的元素放入 stack（如图12.1所示），也可以使用 pop() 将元素依其插入的相反次序从容器中移除（此即所谓“后进先出〔LIFO〕”）。

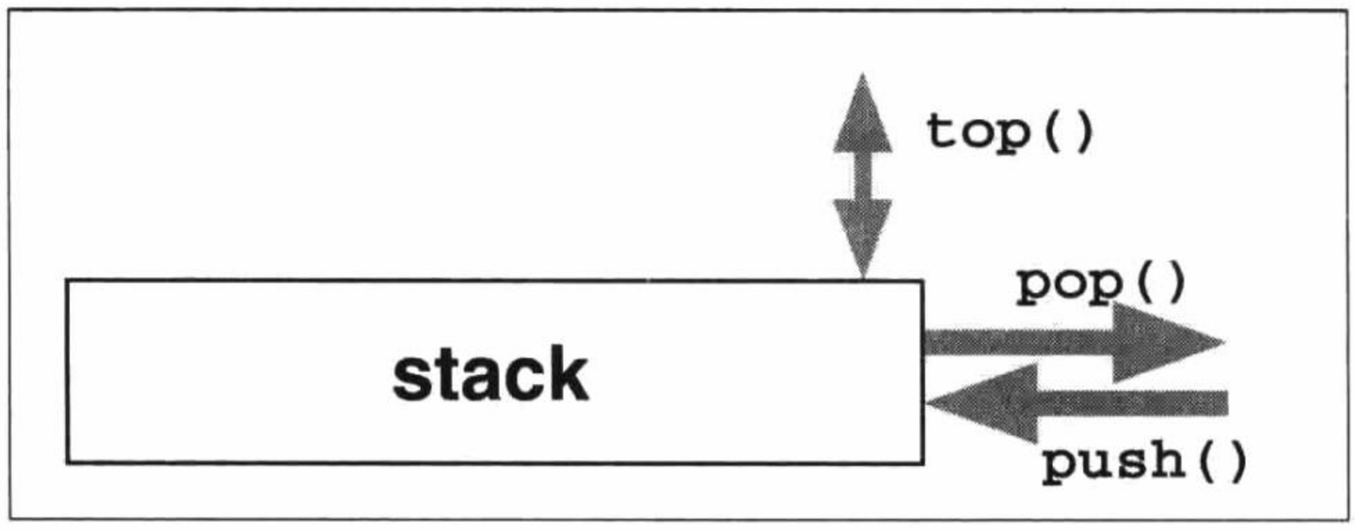

图 12.1 Stack 的接口

使用 stack 之前，必须先包含头文件 <stack>：

```
#include <stack>
```

在头文件 <stack> 中， class stack 定义如下：

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

第一个 template 参数代表元素类型。带有默认值的第二个 template 参数用来定义 stack 内部存放元素的实际容器，默认为 deque。之所以选择 deque 而非 vector，是因为 deque 移除元素时会释放内存，并且不必在重分配（reallocation）时复制全部元素（关于如何恰当运用各种容器，请参考 7.12 节第 392 页）。

例如，以下定义了一个元素类型为整数的 stack：

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

Stack 的实现中只是很单纯地把各项操作转化为内部容器的对应调用（如图12.2所示）。你可以使用任何 sequence 容器支持 stack，只要它们提供以下成员函数： back()、push_back() 和 pop_back()。例如你可以使用 vector 或 list 来容纳元素：

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

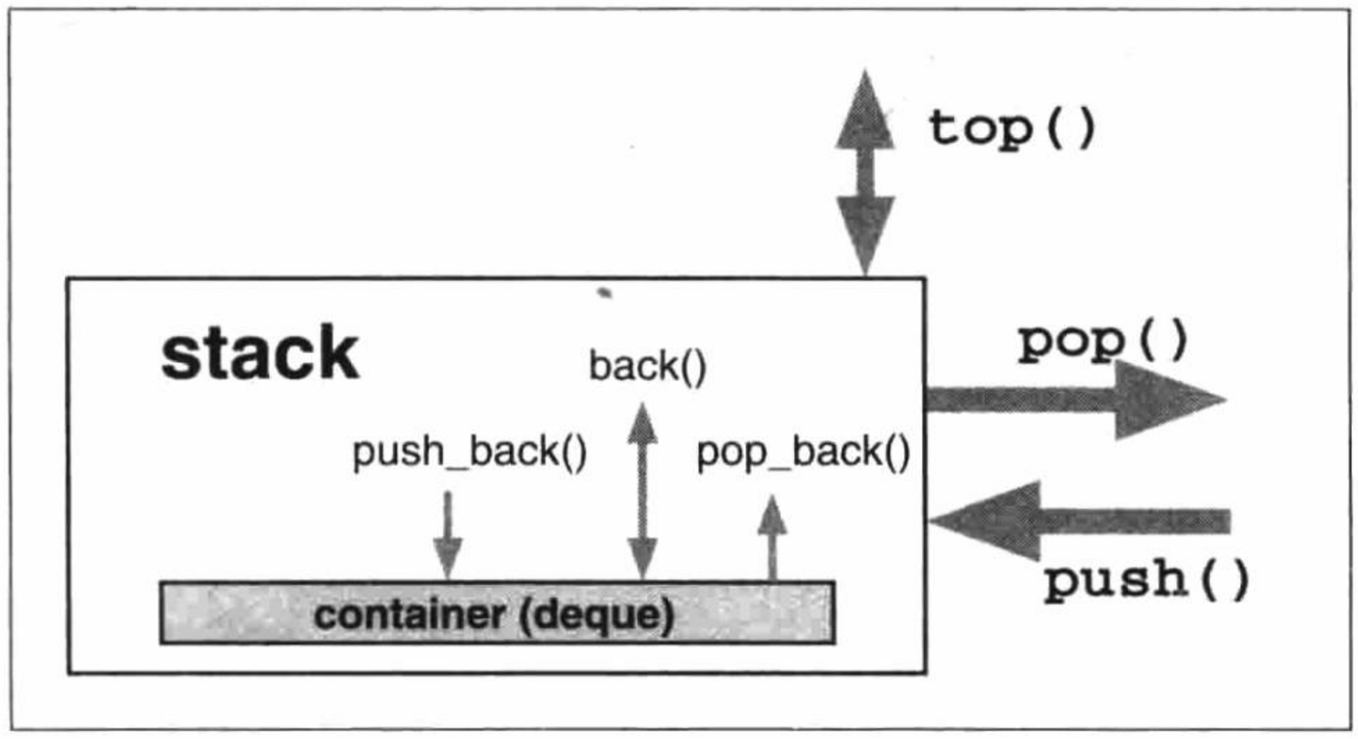

图 12.2　Stack 的内部接口

12.1.1　核心接口

Stack 的核心接口由三个成员函数提供：push()、 top() 和 pop()。

- **push()** 将一个元素放入 stack 内。
- **top()** 返回 stack 内的“下一个”元素。
- **pop()** 从 stack 中移除元素。

注意， pop() 移除下一个元素，但是并不返回它； top() 返回下一个元素，但是并不移除它。所以，如果你想移除 stack 的下一个元素同时返回它，那么这两个函数都得调用。这样的接口可能有点麻烦，但如果你只是想移除下一个元素而并不想处理它，这样的安排就比较好。注意，如果 stack 内没有元素，调用 top() 和 pop() 会导致不明确的行为。你可以采用成员函数 size() 和 empty() 来检验容器是否为空。

如果你不喜欢 stack<> 的标准接口，轻易便可写出若干更方便的接口。相关实例请见 12.1.3 节第 635 页。

12.1.2　Stack 运用实例

下面的程序展示了 stack<> 的用法：

```
// 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;
}
```

程序输出如下：

```
3 2 4 77
```

注意，当使用 nontrivial（译注：意指不凡的、复杂的）元素类型时，你可以考虑在安插“不再被使用的元素”时采用 `std::move()`，或是采用 `emplace()`，由 stack 内部创建元素（二者都始自 C++11）：

```
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 一个用户自定义的 Stack Class

标准的 stack<> class 将运作速度置于方便性和安全性之上。但我通常并不很重视这些，所以我自己写了一个 stack class，拥有以下优势：

1. pop() 会返回下一元素。
2. 如果 stack 为空，pop() 和 top() 会抛出异常（throw exception）。

此外，我把一般人不常使用的成员函数如比较动作（comparison）略去。我的 stack class 定义如下：

```
// 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 */
```

如果改用这个 stack，先前的例子可改写如下：

```
// 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;
    }
}
```

最后一个（多出来的） pop() 调用是为了刻意引发错误。和标准 stack class 不同的是，我这个版本会抛出异常，而不是引发不明确行为。程序输出如下：

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

12.1.4 细究 Class stack<>

Class stack<> 的接口或多或少直接映射了容器内部所用的相应成员。例如：

```
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); }
    ...
  };
}
```

12.4 节第 645 页有对各个成员和操作的详细说明。

12.2 Queue（队列）

Class queue<> 实现出一个 queue（也称为FIFO〔先进先出〕）。你可以使用 push() 将任意数量的元素放入 queue 中（如图12.3所示），也可以使用 pop() 将元素依其插入次序从容器中移除（此即所谓“先进先出〔FIFO〕”）。换句话说，queue 是一个典型的数据缓冲构造。

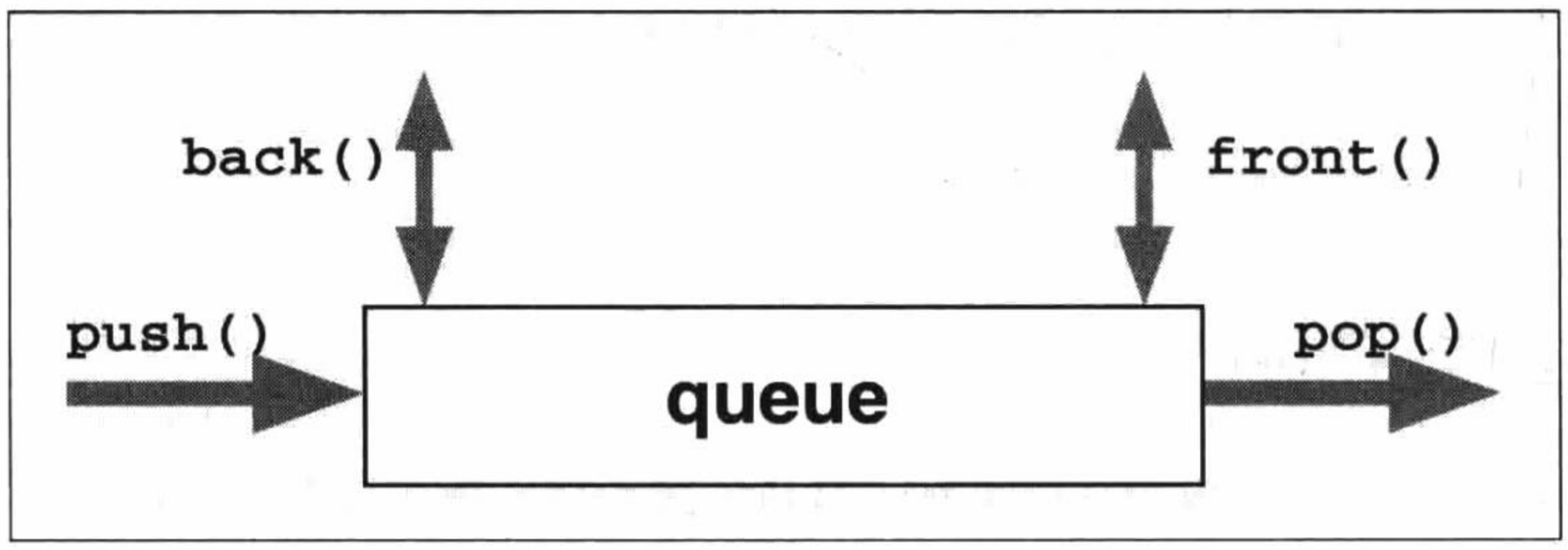

图 12.3 Queue 的接口

为了运用 queue，你必须先包含头文件 <queue>：

```
#include <queue>
```