10

特殊容器
Special Containers

C++ 标准程序库不光只是包含了 STL framework 所提供的容器，还包含一些为满足特殊需求而设计的容器，它们提供简单而清晰的接口。这些容器可归类如下：

- 容器配接器（container adapters）
  这些容器配接器对标准 STL 容器予以配接（*adapt*），使之满足特殊需求。有三种标准容器配接器：
  1. Stacks（堆栈）
  2. Queues（队列）
  3. Priority queues（带优先序的队列）

  Priority queue 就是“根据排序准则，自动将元素排序”的 queue，所以在 priority queue 中所谓“下一个”元素总拥有最大值（最高优先级）。
- 一个名为 `bitset` 的特殊容器。
  所谓 bitset 就是一个位域（bitfield），其中可含任意数量的 bits，但一旦确定就固定不再变动。你可以想象它是一个 bits 容器或 Boolean 容器。注意，C++ 标准程序库同时也为 Boolean 值提供了一个长度可变的特殊容器：`vector<bool>`，本书 6.2.6 节，p158 曾经提过。

10.1 Stacks（堆栈）

class `stack<>`实作出一个 stack（也称为 LIFO，后进先出）。你可以使用 `push()` 将任意数量的元素置入 stack 中，也可以使用 `pop()` 将元素依其插入次序的反序从容器中移除（此即所谓“后进先出，LIFO”），如图 10.1 所示。

为了运用 stack，你必须先含入头文件 `<stack>`[1]：

```
#include <stack>
```

[1] 早期的 STL 中，stacks 被定义于`<stack.h>`

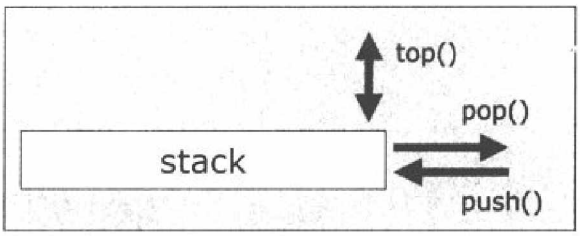

图 10.1 Stack 的接口

在头文件 <stack> 中，class stack 定义如下：

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

第一个 template 参数代表元素型别。带有默认值的第二个 template 参数用来定义 stack 内部存放元素所用的实际容器，缺省采用 deque。之所以选择 deque 而非 vector，是因为 deque 移除元素时会释放内存，并且不必在重新分配（reallocation）时复制全部元素（关于如何恰当运用各种容器，请参考 6.9 节, p226）。

例如，以下声明就定义了一个元素型别为整数的 stack[2]：

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

实际上 stack 只是很单纯地把各项操作转化为内部容器的对应调用（图 10.2）。你可以使用任何序列式容器来支持 stack，只要它们支持 back(),push_back(), pop_back()等动作就行。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
```

10.1.1 核心接口

Stacks 的核心接口就是三个成员函数 push(),top(),pop()：

- push()会将一个元素置入 stack 内。
- top()会返回 stack 内的“下一个”元素。
- pop()会从 stack 中移除元素。

---

[2] 在 STL 早期版本中，你可以把容器当做唯一的 template 参数，所以应当如下声明一个整数 stack：

```
stack< deque<int> > st;
```

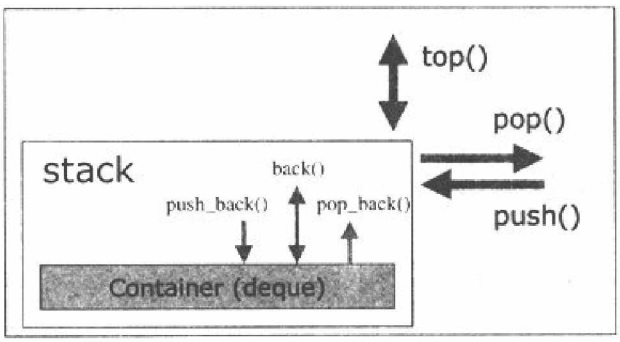

图 10.2 Stack 的内部接口

注意，pop()移除下一个元素，但是并不将它返回；top()返回下一个元素，但是并不移除它。所以如果你想移除 stack 的下一个元素同时并返回它，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，则执行 top()和 pop()会导致未定义的行为。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，轻易便可撰写一些更方便的接口。相关实例请见 10.1.4 节, p441。

10.1.2 Stacks 运用实例

下面这个程序展示 stack<>的用法：

```
// cont/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
```

10.1.3 Class stack<> 细部讨论

class stack<> 的接口非常小，你可以直接阅读其典型实现代码，从而轻松理解它：

```
namespace std {
   template <class T, class Container = deque<T> >
   class stack {
     public:
       typedef typename Container::value_type value_type;
       typedef typename Container::size_type size_type;
       typedef Container container_type;
```

```
    protected:
      Container c; // container
    public:
      explicit stack(const Container& = Container());

      bool empty() const                  { return c.empty(); }
      size_type size() const              { return c.size(); }
      void push(const value_type& x)      { c.push_back(x); }
      void pop()                          { c.pop_back(); }
      value_type& top()                   { return c.back(); }
      const value_type& top() const       { return c.back(); }
  };

  template <class T, class Container>
    bool operator==(const stack<T, Container>&,
                    const stack<T, Container>&);
  template <class T, class Container>
    bool operator< (const stack<T, Container>&,
                    const stack<T, Container>&);
  ... // (other comparison operators)
}
```

下面是各个成员的详细说明。

型别定义

stack::**value_type**

- 元素型别
- 它和 *container*::value_type 相当。

stack::**size_type**

- 不带正负号的整数型别，用来表现大小。
- 它和 *container*::size_type 相当。

stack::**container_type**

- 内部容器的型别

各项操作（Operations）

stack::**stack** ()

- default 构造函数
- 产生一个空的 stack。

explicit *stack*::**stack** (const Container& *cont*)

- 产生一个 stack，并以容器 *cont* 内的元素为初值。
- *cont* 内的所有元素均被复制。

size_type stack::**size** () const

- 返回元素个数。
- 如果要检验容器是否为空（亦即不含任何元素），应使用 empty()，因为它可能更快。

bool *stack*::**empty** () const

- 判断 stack 是否为空（亦即不含任何元素）。
- 与 *stack*::*size*()==0 等效，但可能更快。

void *stack*::**push** (const value_type& *elem*)

- 将 *elem* 的副本安插到 stack 内，并成为其新的第一元素。

value_type& *stack*::**top** ()
const *value_type*& *stack*::**top** () const

- 以上两种形式都返回 stack 的下一个元素。所谓“下一个元素”是指最后一个（亦即在其它所有元素之后）被插入的元素。
- 调用者必须确保 stack 不为空（size()>0），否则可能导致未定义的行为。
- 第一种形式是针对 non-const stacks 设计的，返回一个 reference。所以你可以就地（*in place*）修改 stack 内的元素。这样做是否合适？由你自己决定。

void *stack*::**pop** ()

- 移除 stack 内的下一个元素。所谓“下一个元素”是指最后一个（亦即在其它所有元素之后）被插入的元素。
- 此函数没有回返值。如果想处理被移除的那个元素，你必须先调用 top()。
- 调用者必须确保 stack 不为空（size()>0），否则可能导致未定义的行为。

bool ***comparison*** (const *stack*& *stack1*, const *stack*& *stack2*)

- 返回两个同型 stack 的比较结果。

- ***comparison*** 可以是下面各运算之一：

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- 如果两个 stacks 的元素个数相等，且相同次序上的元素值也相等（也就是说所有对应元素之间的比较都得到 true），则这两个容器相等。
- stack 之间的大小比较是以“字典顺序”而定。参见 p360 对于算法 lexicographical_compare() 的描述。

10.1.4 一个使用者自定义的 Stack Class

标准的 stack<> class 将运行速度置于方便性和安全性之上。但我通常并不很重视那个。便自己写了一个 stack class，它有以下优势：

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

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

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

#include <deque>
#include <exception>

template <class 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 */
```

改用这个 stack，则先前的例子可以改写如下：

```
// cont/stack2.cpp

#include <iostream>
#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;
    }
}
```

最后新增加的一个 pop() 调用动作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，而不引发未定义行为。程序输出如下：

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

10.2 Queues（队列）

Class queue<> 实作出一个 queue（也称为 FIFO，先进先出）。你可以使用 push() 将任意数量的元素置入 queue 中（图 10.3），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区结构。

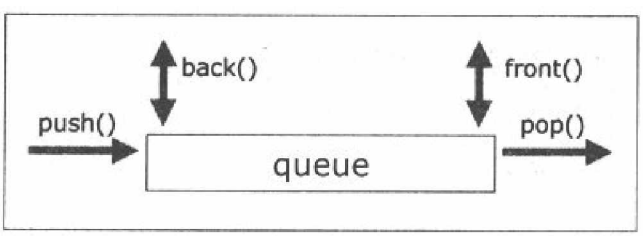

图 10.3 Queue 的接口

为了运用 queue，你必须先含入头文件 <queue>[3]：

```
#include <queue>
```

在头文件 <queue> 中，class queue 定义如下：

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

[3] 早期的 STL 中，queues 被定义于 <stack.h>

第一个 template 参数代表元素型别。带有默认值的第二个 template 参数用来定义 queue 内部存放元素用的实际容器，缺省采用 deque。

下面这个例子定义了一个内含字符串的 queue[4]：

```
std::queue<std::string> buffer;      // string queue
```

实际上 queue 只是很单纯地把各项操作转化为内部容器的对应调用（图 10.4）。你可以使用任何序列式容器来支持 queue，只要它们支持 `front()`,`back()`,`push_back()`,`pop_front()`等动作就行。例如你可以使用 list 来容纳元素：

```
std::queue<std::string, std::list<std::string> > buffer;
```

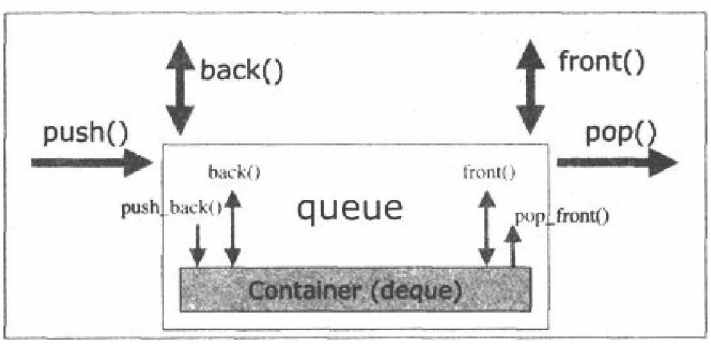

图 10.4 Queue 的内部接口

10.2.1 核心接口

Queues 的核心接口主要由成员函数 `push()`,`front()`,`back()`,`pop()`构成：

- `push()`会将一个元素置入 queue 中。
- `front()`会返回 queue 内的下一个元素（也就是第一个被置入的元素）。

[4] 在 STL 早期版本中，你可以把容器当做唯一的 template 参数，所以应当如下声明一个 string queue：

```
queue< deque<string> > buffer;
```

- back()会返回 queue 中最后一个元素（也就是第一个被插入的元素）。
- pop()会从 queue 中移除一个元素。

注意，pop()虽然移除下一个元素，但是并不返回它，front()和 back()返回下一个元素，但并不移除它。所以如果你想移除 queue 的下一个元素，又想处理它，那么就得同时调用 front()和 pop()。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 queue 之内没有元素，则 front(),back(),pop()的执行会导致未定义行为。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 queue<> 的标准接口，轻易便可撰写一些更方便的接口。相关实例请见 10.2.4 节，p450。

10.2.2 Queues 运用实例

下面这个程序展示 queue<> 的用法：

```
// cont/queue1.cpp

#include <iostream>
#include <queue>
#include <string>
using namespace std;

int main()
{
    queue<string> q;

    // insert three elements into the queue
    q.push("These ");
    q.push("are ");
    q.push("more than ");

    // read and print two elements from the queue
    cout << q.front();
    q.pop();
    cout << q.front();
    q.pop();

    // insert two new elements
    q.push("four ");
    q.push("words!");
```

```
    // skip one element
    q.pop();

    // read and print two elements
    cout << q.front();
    q.pop();
    cout << q.front() << endl;
    q.pop();

    // print number of elements in the queue
    cout << "number of elements in the queue: " << q.size()
         << endl;
}
```

程序输出如下：

```
These are four words!
number of elements in the queue: 0
```

10.2.3 Class queue<> 细部讨论

和 stack<> 相似，典型的 queue<> 实作手法也是十分清晰易懂：

```
namespace std {
    template <class T, class Container = deque<T> >
    class queue {
        public:
            typedef typename Container::value_type value_type;
            typedef typename Container::size_type size_type;
            typedef Container container_type;
        protected:
            Container c;     // container
        public:
            explicit queue(const Container& = Container());

            bool empty() const                  { return c.empty(); }
            size_type size() const              { return c.size(); }
            void push(const value_type& x) { c.push_back(x); }
            void pop()                          { c.pop_front(); }
            value_type& front()                 { return c.front(); }
            const value_type& front() const     { return c.front(); }
            value_type& back()                  { return c.back(); }
            const value_type& back() const { return c.back(); }
    };
```

```
    template <class T, class Container>
       bool operator==(const queue<T, Container>&,
                       const queue<T, Container>&);
    template <class T, class Container>
       bool operator< (const queue<T, Container>&,
                       const queue<T, Container>&);
    ... // (other comparison operators)
};
```

下面是各个成员的详细解释。

型别定义

queue::**value_type**

- 元素型别
- 此和 *container*::value_type 相当。

queue::**size_type**

- 不带正负号的整数型别，用来表现大小。
- 此和 *container*::size_type 相当。

queue::**container_type**

- 内部容器的型别

各项操作（Operations）

queue::**queue** ()

- 缺省构造函数
- 产生一个空的 queue。

explicit *queue*::**queue** (const Container& cont)

- 产生一个 queue，并以容器 *cont* 内的元素为初值。
- *cont* 内的所有元素均被复制。

size_type *queue*::**size** () const

- 返回元素个数。
- 如果要检验容器是否为空（亦即不含任何元素），应该使用 empty()，因为它可能更快。

```
bool queue::empty () const
```

- 判断 queue 是否为空（亦即不含任何元素）。
- 和 *queue*::*size*()==0 等效，但可能更快。

```
void queue::push (const value_type& elem)
```

- 将 *elem* 的副本安插到 queue 内，并成为其新的最后元素。

```
value_type& queue::front ()
const value_type& queue::front () const
```

- 两种形式都返回 queue 的下一个元素。所谓“下一个元素”是指第一个（亦即在其它所有元素之前）被置入的元素。
- 调用者必须确保 queue 不为空（size()>0），否则可能导致未定义的行为。
- 第一形式是针对 non-const queues 而设计，返回一个 reference。所以你可以就地（*in place*）修改 queue 内的元素。这样做是否合适？由你自己决定。

```
value_type& queue::back ()
const value_type& queue::back () const
```

- 两种形式都返回 queue 的最后一个元素。所谓“最后一个元素”是指最后一个（亦即在其它所有元素之后）被插入的元素。
- 调用者必须确保 queue 不为空（size()>0），否则可能导致未定义的行为。
- 第一形式是针对 non-const queues 而设计，返回一个 reference。所以你可以就地（*in place*）修改 queue 内的元素。这样做是否合适？由你自己决定。

```
void queue::pop ()
```

- 移除 queue 内的下一个元素。所谓“下一个元素”是指第一个（亦即在其它所有元素之前）被插入的元素。
- 此函数没有回返值。如果想处理被移除的那个元素，你必须先调用 front()。
- 调用者必须确保 queue 不为空（size()>0），否则可能导致未定义的行为。

```
bool comparison (const queue& queue1, const queue& queue2)
```

- 返回两个同型的 queue 的比较结果。

- ***comparison*** 可以是下面各运算之一：

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- 如果两个 queues 的元素个数相等，且相同次序上的元素值也相等（也就是说所有对应元素之间的比较都得到 true），则这两个容器相等。
- queues 之间的大小比较是以“字典顺序”而定。参见 p360 对于算法 `lexicographical_compare()` 的描述。

10.2.4 一个使用者自定义的 Queue Class

标准的 `queue<>` class 将运行速度置于方便性和安全性之上。但我通常并不很重视那个。便自己写了一个 queue class，它有以下优势：

1. `pop()` 会返回下一元素。
2. 如果 queue 为空，`pop()` 和 `front()` 会抛出异常。

此外，我把一般人不常使用的那些成员函数如比较操作（comparison）和 `back()` 略去。我的 queue class 定义如下：

```
// cont/Queue.hpp
/* ************************************************************
* Queue.hpp
* - safer and more convenient queue class
* ************************************************************/
#ifndef QUEUE_HPP
#define QUEUE_HPP

#include <deque>
#include <exception>

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

   public:
      /* exception class for pop() and top() with empty queue
      */
```

```
        class ReadEmptyQueue : public std::exception {
            public:
                virtual const char* what() const throw() {
                    return "read empty queue";
                }
        };

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

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

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

    // read element from the queue and return its value
    T pop () {
        if (c.empty()) {
            throw ReadEmptyQueue();
        }
        T elem(c.front());
        c.pop_front();
        return elem;
    }

    // return value of next element
    T& front () {
        if (c.empty()) {
            throw ReadEmptyQueue();
        }
        return c.front();
    }
};
#endif /* QUEUE_HPP */
```

改用这个 queue，则先前的例子可以改写如下：

```
// cont/queue2.cpp

#include <iostream>
#include <string>
#include "Queue.hpp"       // use special queue class
using namespace std;

int main()
{
   try {
      Queue<string> q;

      // insert three elements into the queue
      q.push("These ");
      q.push("are ");
      q.push("more than ");

      // read and print two elements from the queue
      cout << q.pop();
      cout << q.pop();

      // push two new elements
      q.push("four ");
      q.push("words!");

      // skip one element
      q.pop();

      // read and print two elements from the queue
      cout << q.pop();
      cout << q.pop() << endl;
```

```
        // print number of remaining elements
        cout << "number of elements in the queue: " << q.size()
             << endl;

        // read and print one element
        cout << q.pop() << endl;
    }
    catch (const exception& e) {
        cerr << "EXCEPTION: " << e.what() << endl;
    }
}
```

最后新增的一个 pop() 调用操作是为了刻意引发错误。和标准 queue class 不同的是，我这个版本会抛出异常，而不引发未定义行为。程序输出如下：

```
These are four words!
number of elements in the queue: 0
EXCEPTION: read empty queue
```

10.3 Priority Queues（优先队列）

class priority_queue<> 实作出一个 queue，其中的元素根据优先级被读取。它的接口和 queues 非常相近。也就是说，push() 对着 queue 置入一个元素，top()/pop() 存取/移除下一个元素（图 10.5）。然而这所谓“下一个元素”并非第一个置入的元素，而是“优先级最高”的元素。换句话说 priority queue 内的元素已经根据其值进行了排序。和往常一样，你可以透过 template 参数指定一个排序准则。缺省的排序准则是利用 operator< 形成降序排列，那么所谓“下一个元素”就是“数值最大的元素”。如果同时存在若干个数值最大的元素，无法确知究竟哪一个会入选。

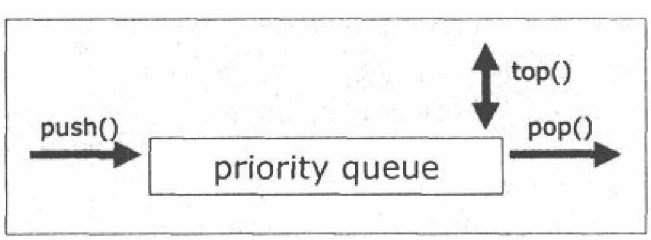

图 10.5 Priority Queue 的接口

priority queue 和一般的 queue 都定义于头文件<queue>[5]：

```
#include <queue>
```

在头文件 <queue> 中，class priority_queue 定义如下：

```
namespace std {
    template <class T,
              class Container = vector<T>,
              class Compare = less<typename Container::value_type>
    class priority_queue;
}
```

第一个 template 参数是元素型别，带有默认值的第二个 template 参数定义了 priority queue 内部用来存放元素的容器。缺省容器是 vector。带有默认值的第三个 template 参数定义出“用以搜寻下一个最高优先元素”的排序准则。缺省情况下是以 operator<作为比较标准。

下面这个例子定义了一个元素型别为 float 的 priority queue[6]：

```
std::priority_queue<float> pbuffer; // priority queue for floats
```

实际上 priority queue 只是很单纯地把各项操作转化为内部容器的对应调用。你可以使用任何序列式容器来支持 priority queue，只要它们支持随机存取迭代器和 front(),push_back(),pop_back()等动作就行。由于 priority queue 需要用到 STL heap 算法（参见 9.9.4 节, p406），所以其内部容器必须支持随机存取迭代器。例如你可以使用 deque 来容纳元素：

```
std::priority_queue< float, std::deque<float> > pbuffer;
```

如果需要定义自己的排序准则，就必须传递一个函数（或仿函数）作为二元判断式（binary predicate），用以比较两个元素并以此作为排序准则（关于排序准则的更多介绍，请见 6.5.2 节, p178 和 8.1.1 节, p294）。例如以下式子定义了一个逆向的（降序的）priority queue：

```
std::priority_queue<float,std::vector<float>,
                    std::greater<float> > pbuffer;
```

在此 priority queue 中，“下一个元素”始终拥有最小元素值。

---

[5] 早期的 STL 中，priority queue 被定义于 <stack.h>

[6] 早期的 STL 中，你必须以 template 参数传入内部容器。所以一个元素型别为 float 的 priority queue 声明如下：

```
priority_queue< vector<float>,less<float> > buffer;
```

10.3.1 核心接口

priority queue 的核心接口主要由成员函数 push()，top()，pop()组成：

- push() 会将一个元素置入 priority queue 中。
- top() 会返回 priority queue 中的“下一个元素”。
- pop() 会从 priority queue 中移除一个元素。

和其它容器配接器一样，pop()会移除下一个元素，但是不返回它；top()会返回下一个元素，但并不移除它。所以如果你想移除 priority queue 内的下一个元素，又想处理它，就得同时调用这两个函数。注意，如果 priority queue 内没有元素，执行 top()和 pop()会导致未定义行为。你可以采用成员函数 size()和 empty()来检验容器是否为空。

10.3.2 Priority Queues 运用实例

以下程序展示了 class priority_queue<> 的用法：

```
// cont/pqueue1.cpp

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

int main()
{
    priority_queue<float> q;

    // insert three elements into the priority queue
    q.push(66.6);
    q.push(22.2);
    q.push(44.4);

    // read and print two elements
    cout << q.top() << ' ';
    q.pop();
    cout << q.top() << endl;
    q.pop();

    // insert three more elements
    q.push(11.1);
    q.push(55.5);
    q.push(33.3);
```

```
    // skip one element
    q.pop();

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

程序输出如下：

```
66.6 44.4
33.3 22.2 11.1
```

由上可见，在元素 66.6, 22.2, 44.4 被置入后，程序打印出优先级最高的元素 66.6 和 44.4。另三个元素被置入后，priority queue 内含 22.2, 11.1, 55.5, 33.3（按照插入次序）。下一个元素被 pop() 忽略掉了，所以最后一个循环依次打印 33.3, 22.2, 11.1。

10.3.3 Class priority_queue<> 细部讨论

和 stack<> 以及 queue<> 一样，priority_queue<> 的大部分操作都非常清晰明确，足以顾名思义：

```
namespace std {
    template <class T, class Container = vector<T>,
              class Compare = less<typename Container::value_type> >
    class priority_queue {
        public:
            typedef typename Container::value_type value_type;
            typedef typename Container::size_type size_type;
            typedef Container container_type;

        protected:
            Compare comp;     // sorting criterion
            Container c;      // container

        public:
            // constructors
            explicit priority_queue(const Compare& cmp = Compare(),
                                    const Container& cont = Container()
              : comp(cmp), c(cont) {
                make_heap(c.begin(),c.end(),comp);
            }
```

```
        template <class InputIterator>
        priority_queue(InputIterator first, InputIterator last,
                       const Compare& cmp = Compare(),
                       const Container& cont = Container())
          : comp(cmp), c(cont) {
            c.insert(c.end(),first,last);
            make_heap(c.begin(),c.end(),comp);
        }

        void push(const value_type& x); {
           c.push_back(x);
           push_heap(c.begin(),c.end(),comp);
        }

        void pop() {
           pop_heap(c.begin(),c.end(),comp);
           c.pop_back();
        }

        bool empty() const { return c.empty(); }
        size_type size() const { return c.size(); }
        const value_type& top() const { return c.front(); }
    };
}
```

可见 priority queue 使用了 STL heap 算法。这些算法在 9.9.4 节, p406 中介绍过。注意，和前述另两个容器配接器不同的是，这里没有比较（comparison）操作符。

下面是各个成员的详细说明。

型别定义

priority_queue::**value_type**

- 元素型别
- 此和 *container*::value_type 相当。

priority_queue::**size_type**

- 不带正负号的整数型别，用来表现大小。
- 此和 *container*::size_type 相当。

priority_queue::**container_type**

- 内部容器的型别

构造函数

priority_queue::**priority_queue** ()

- default 构造函数
- 产生一个空的 priority queue。

explicit *priority_queue*::**priority_queue** (const CompFunc& *op*)

- 产生一个 priority queue，以 *op* 为排序准则。
- 如何将排序准则当做构造函数参数传入，请见 p191 和 p213。

```
priority_queue::priority_queue (const CompFunc& op,
                                const Container& cont)
```

- 产生一个 priority queue，以 *op* 为排序准则，并以容器 *cont* 内的元素为初值。
- *cont* 中的所有元素都会被复制。

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end)
```

- 产生一个 priority queue，以区间 [*beg*, *end*] 内的元素为初值。
- 此构造函数是一个 template member（见 p11），也就是说只要源区间内的元素型别可以转化为本容器内的元素型别，此构造函数即可运行。

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end,
                                const CompFunc& op)
```

- 产生一个 priority queue，以 *op* 为排序准则，并以区间 [*beg*, *end*) 内的元素为初值。
- 此构造函数是一个 template member（见 p11），也就是说只要源区间内的元素型别可以转化为本容器内的元素型别，此构造函数即可运作。
- 如何将排序准则当做构造函数参数传入，请见 p191 和 p213。

```
priority_queue::priority_queue (InputIterator beg,
                                InputIterator end,
                                const CompFunc& op,
                                const Container& cont)
```

- 产生一个 priority queue，以 *op* 为排序准则，并以区间 [*beg*, *end*) 内的元素以及容器 *cont* 内的元素为初值。
- 此构造函数是一个 template member（见 p11），也就是说只要源区间内的元素型别可以转化为本容器内的元素型别，此一构造函数即可运作。

其它操作

```
size_type priority_queue::size () const
```

- 返回元素数量。
- 如果要检验容器是否为空（亦即不含任何元素），应使用 `empty()`，因为它可能更快。

```
bool priority_queue::empty () const
```

- 判断 priority queue 是否为空（亦即不含任何元素）。
- 与 `priority_queue::size()==0` 等效，但可能更快。

```
void priority_queue::push (const value_type& elem)
```

- 将 *elem* 的副本安插到 priority queue 内。

```
const value_type& priority_queue::top () const
```

- 返回 priority queue 内的“下一个元素”。所谓下一个元素是指所有元素中数值最大的那个。如果同时存在若干相等的最大元素，则无法确知会返回哪一个。
- 调用者必须确保 priority queue 不为空（`size()>0`），否则导致未定义行为。

```
void priority_queue::pop ()
```

- 移除 priority queue 内的“下一个元素”。所谓下一个元素是指所有元素中数值最大的那个。如果同时存在若干相等的最大元素，则无法确知会返回哪一个。
- 这个函数无回返值。如果想处理“下一个元素”，必须先调用 `top()`。
- 调用者必须确保 priority queue 不为空（`size()>0`），否则导致未定义行为。

10.4 Bitsets

Bitsets 造出一个内含位（bits）或布尔（boolean）值且大小固定的 array。当你需要管理各式标志（flags），并以标志的任意组合来表现变量时，就可运用 bitsets。C 程序和传统 C++ 程序通常使用型别 long 来作为 bits array，再通过 &,|,~ 等位操作符（bit operators）操作各个位。Class bitset 的优点在于可容纳任意个数的位（译注：但不能动态改变），并提供各项操作。例如你可以对某个特定位置赋值一个位，也可以将 bitsets 作为由 0 和 1 组成的序列，进行读写。

注意，你不可以改变 bitset 内位的数量。这个数量的具体值是由 template 参数决定的。如果你需要一个可变长度的位容器，可考虑使用 vector<bool>（参见 6.2.6 节，p158）。

Class bitset 定义于头文件 <bitset> 之中：

```
#include <bitset>
```

其中的 class bitset 是个 template class，有一个 template 参数，用来指定位的数量：

```
namespace std {
    template <size_t Bits>
    class bitset;
}
```

在这里，template 参数并不是一个型别，而是一个不带正负号的整数（此一语言特性请参考 p10）。

注意，如果 template 参数不同，具现化所得的 template 型别就不同。换句话说你只能针对位个数相同的 bitsets 进行比较和组合。

10.4.1 Bitsets 运用实例

将 Bitsets 当做一组标志

第一个例子展示如何运用 bitsets 来管理一组标志。每个标志都有一个由列举型别（enum）定义出来的值。该列举值就表示位在 bitset 中的位置。举个例子，这些 bits 可以代表颜色，那么每一个列举值都代表一种颜色。通过运用 bitset，你可以管理一个变量，其中包含颜色的任意组合：

```
// cont/bitset1.cpp

#include <bitset>
#include <iostream>
using namespace std;
```

```
int main()
{
  /* enumeration type for the bits
  * - each bit represents a color
  */
  enum Color { red, yellow, green, blue, white, black, ...,
               numColors };

  // create bitset for all bits/colors
  bitset<numColors> usedColors;

  // set bits for two colors
  usedColors.set(red);
  usedColors.set(blue);

  // print some bitset data
  cout << "bitfield of used colors: " << usedColors
       << endl;
  cout << "number of used colors: " << usedColors.count()
       << endl;
  cout << "bitfield of unused colors: " << ~usedColors
       << endl;

  // if any color is used
  if (usedColors.any()) {
     // loop over all colors
     for (int c = 0; c < numColors; ++c) {
        // if the actual color is used
        if (usedColors[(Color)c]) {
           ...
        }
     }
  }
}
```

在 I/O 中利用 Bitsets 表示二进制

Bitsets 一个强有力的特性就是可以在整数值和位序列之间相互转化，只要很简单地产生一个临时的 bitset 就可以办到：

```
// cont/bitset2.cpp

#include <bitset>
#include <iostream>
#include <string>
#include <limits>
using namespace std;

int main()
{
  /* print some numbers in binary representation
  */
  cout << "267 as binary short: "
       << bitset<numeric_limits<unsigned short>::digits>(267)
       << endl;

  cout << "267 as binary long: "
       << bitset<numeric_limits<unsigned long>::digits>(267)
       << endl;

  cout << "10,000,000 with 24 bits: "
       << bitset<24>(1e7) << endl;

  /* transform binary representation into integral number
  */
  cout << "\"1000101011\" as number: "
       << bitset<100>(string("1000101011")).to_ulong() << endl;
}
```

程序可能输出如下（具体内容视平台上的 short 和 long 的位数量而定）

```
267 as binary short:       0000000100001011
267 as binary long:        00000000000000000000000100001011
10,000,000 with 24 bits:   100110001001011010000000
"1000101011" as number:    555
```

此例中的：

```
bitset<numeric_limits<unsigned short>::digits>(267)
```

将数值 267 转换成一个 bitset，其位数与型别 unsigned short 相符（关于数值极限的讨论，请见 4.3 节, p60）。针对 bitset 而设计的 output 操作符（operator<<）能够将这组位翻译成一个“0/1 序列”并打印出来。

同样道理，以下式子：

```
bitset<100>(string("1000101011"))
```

将一个二进制字符透过 to_ulong()转化为一个整数值，然后转化至一个 bitset。注意，bitset 内的位数应小于 sizeof(unsigned long)，因为如果该 bitset 无法按照 unsigned long 来表现这个值，它会抛出一个异常[7]。

10.4.2 Class bitset 细部讨论

bitset class 提供了以下操作。

生成，拷贝和销毁

Bitsets 定义了一些特殊的构造函数，但是没有定义特殊的 copy 构造函数、assignment 操作符和析构函数。所以 bitsets 的赋值和复制行为都是采用缺省行为，也就是按位次序逐一拷贝（所谓 bitwise copy）。

```
bitset<bits>::bitset ()
```

- default 构造函数。
- 生成一个 bitset，所有位均初始化为零。
- 例如：

```
bitset<50> flags;    // flags: 0000...000000
                     // thus, 50 unset bits
```

```
bitset<bits>::bitset (unsigned long value)
```

- 产生一个 bitset，以整数值 *value* 的位作为初值。
- 如果 *value* 的值太小，前面不足的位被设为 0。
- 例如：

```
bitset<50> flags(7);      // flags: 0000...000111
```

---

[7] 注意，你必须先将这个初始值显式转化为 string。这恐怕是标准规格中的一个失误，因为标准规格早期版本允许我们这么做：

```
bitset<100>("1000101011")
```

但是此构造函数被用于其它不同的 string 型别时，会意外导致隐式型别转换被排除。这个问题目前已经在标准委员会中有了解决提案。

```
explicit bitset<bits>::bitset (const string& str)
bitset<bits>::bitset (const string& str, string::size_type str_idx)
bitset<bits>::bitset (const string& str, string::size_type str_idx,
                      string::size_type str_num)
```

- 所有形式都用来产生 bitset，以字符串 str 或其子字符串加以初始化。
- 该字符串或子字符串中只能包含字符 "0" 和 "1"。
- *str_idx* 是 *str* 中用于初始化的第一个字符。
- 如果省略 *str_num*，从 *str_idx* 开始到 *str* 结束的所有字符都将用于初始化。
- 如果该字符串或子字符串中的字符数量少于所需，则前面多余的位将设初值 0。
- 如果该字符串的子字符串中的字符数量多于所需，则多出来的字符会被忽略。
- 如果 *str_idx* > *str.size()*，抛出 out_of_range 异常。
- 如果有个字符既不是 "0" 也不是 "1"，抛出 invalid_argument 异常。
- 这个构造函数是个 template member（参见 p11）。因此就第一参数而言，不会发生从 *const char** 向 *string* 的隐式型别转换[8]。
- 举个例子：

```
bitset<50> flags(string("1010101"));
// flags: 0000...0001010101
bitset<50> flags(string("1111000"),2,3);
// flags: 0000...0000000110
```

非更易性操作（Nonmanipulating Operations）

```
size_t bitset<bits>::size () const
```

- 返回位的个数。

```
size_t bitset<bits>::count () const
```

- 返回 “位值为 1” 的位个数。

```
bool bitset<bits>::any () const
```

- 判断是否有任何位被设立（数值为 1）。

```
bool bitset<bits>::none () const
```

- 判断是否没有任何一个位被设立（亦即所有的位值皆为 0）。

---

[8] 这可能是标准规格中的一个失误，因为早期标准规格允许我们这么做：

```
bitset<50> flags("1010101")
```

但是此一构造函数被用于其它不同的 string 型别时，会意外导致隐式型别转换被排除。这个问题目前已经在标准委员会中有了解决提案。

```
bool bitset<bits>::test (size_t idx) const
```

- 判断 *idx* 位置上的位是否被设立（数值为 1）。
- 如果 idx >= size()则抛出 out_of_range 异常。

```
bool bitset<bits>::operator == (const bitset<bits>& bits) const
```

- 判断 *this 和 *bits* 的所有位是否都相等。

```
bool bitset<bits>::operator != (const bitset<bits>& bits) const
```

- 判断 *this 和 *bits* 之中是否有些位不相等。

更易性操作（manipulating Operations）

```
bitset<bits>& bitset<bits>::set ()
```

- 将所有位设为 true（1）。
- 返回更动后的 bitset。

```
bitset<bits>& bitset<bits>::set (size_t idx)
```

- 将位置 *idx* 上的位设为 true（1）。
- 返回更动后的 bitset。
- 如果 *idx* >= size()则抛出 out_of_range 异常。

```
bitset<bits>& bitset<bits>::set (size_t idx, int value)
```

- 根据 *value* 上的值设定 *idx* 位置的位值。
- 返回更动后的 bitset。
- *value* 将被当做 boolean 值处理。如果 *value* 等于 0，则 *idx* 位置上的位值被设为 false；其它的 *value* 值都会使该位被设为 true。
- 如果 *idx* >= size()则抛出 out_of_range 异常。

```
bitset<bits>& bitset<bits>::reset ()
```

- 将所有位设为 false（0）。
- 返回更动后的 bitset。

```
bitset<bits>& bitset<bits>::reset (size_t idx)
```

- 将位置 *idx* 上的位设为 false（0）。
- 返回更动后的 bitset。
- 如果 *idx* >= size()则抛出 *out_of_range* 异常。

```
bitset<bits>& bitset<bits>::flip ()
```

- 反转所有位（原本 1 者转为 0，原本 0 者转为 1）。
- 返回更动后的 bitset。

```
bitset<bits>& bitset<bits>::flip (size_t idx)
```

- 反转 *idx* 位置上的位。
- 返回更动后的 bitset。
- 如果 *idx* >= size() 则抛出 out_of_range 异常。

```
bitset<bits>& bitset<bits>::operator ^= (const bitset<bits>& bits)
```

- 对每个位逐一进行 exclusive-or 运算。
- 将 *this 之中所有和“*bits* 内数值为 1 的位”的对应位都翻转，其它位保持不动。
- 返回更动后的 bitset。

```
bitset<bits>& bitset<bits>::operator |= (const bitset<bits>& bits)
```

- 位逐一进行 or 运算。
- 将*this 之中所有和“*bits* 内数值为 1 的位”的对应位都设为 1，其它位保持不动。
- 返回更动后的 bitset。

```
bitset<bits>& bitset<bits>::operator &= (const bitset<bits>& bits)
```

- 位逐一进行 and 运算。
- 将*this 之中所有和“*bits* 内数值为 0 的位”的对应位都设为 0，其它位保持不动。
- 返回更动后的 bitset。

```
bitset<bits>& bitset<bits>::operator <<= (size_t num)
```

- 将所有位向左移动 *num* 个位置。
- 返回更动后的 bitset。
- 空出来的位设为 false（0）。

```
bitset<bits>& bitset<bits>::operator >>= (size_t num)
```

- 将所有位向右移动 *num* 个位置。
- 返回更动后的 bitset。
- 空出来的位设为 false（0）。

使用 Operator[] 存取位

```
bitset<bits>::reference bitset<bits>::operator[] (size_t idx)
bool bitset<bits>::operator[] (size_t idx) const
```

- 这两种形式都返回 *idx* 位置上的位值。
- 第一种形式针对 non-const bitsets，使用了一个 **proxy**（代理人、替身）型别，使得回返值成为一个可被更动的值（左值，lvalue）。详见下一段文字叙述。
- 调用者必须确保 *idx* 有效，否则会导致未定义的行为。

当我们针对 non-const bitsets 调用 operator[]时，返回的是一个特殊的临时对象 bitset<>::reference。这个对象称为 **proxy**（代理人，替身）[9]，它的存在使我们得以对 operator[]所存取的位做适当的内容变动（译注：**proxy** 是一种设计模式，可参考《*Design Patterns*》或《*More Effective C++*》条款 30）。

具体地说，对于上述 references，允许以下五种操作：

1. reference& operator= (bool)

   根据传来的值设置该位。

2. reference& operator= (const reference&)

   根据另一个 reference 的内容设定该位。

3. reference& flip ()

   翻转该位。

4. operator bool () const

   （自动地）将该位转换为布尔值。

5. bool operator~ () const

   返回该位的补码（翻转值）。

举个例子，你可以这么做:

```
bitset<50> flags;
...
flags[42] = true;          // set bit 42
flags[13] = flags[42];     // assign value of bit 42 to bit 13
flags[42].flip();          // toggle value of bit 42
if (flags[13]) {           // if bit 13 is set,
   flags[10] = ~flags[42]; // then assign complement of bit 42 to bit
}
```

产生新的（经修改的）bitset

```
bitset<bits> bitset<bits>::operator ~ () const
```

- 产生一个新的 bitset 并返回；以*this 的位翻转值作为初值。

```
bitset<bits> bitset<bits>::operator << (size_t num) const
```

- 产生一个新的 bitset 并返回；以*this 的位向左移动 *num* 个位置作为初值。

```
bitset<bits> bitset<bits>::operator >> (size_t num) const
```

- 产生一个新的 bitset 并返回；以*this 的位向右移动 *num* 个位置作为初值。

---

[9] **proxy**（替身，代理人）使你能对通常力不能及的问题加以控制。用来获取更高的安全性。本例中虽然回返值理论上如同 bool，但 **proxy** 使我们得以进行某些特殊操作。

```
bitset<bits> operator & (const bitset<bits>& bits1,
                         const bitset<bits>& bits2)
```

- 对 *bits1* 和 *bits2* 两者进行“各位逐一 and 运算”并返回结果。
- 返回的新 bitset 中，只有“*bits1* 和 *bits2* 的位值都为 1”的那些位才会被设为 1。

```
bitset<bits> operator | (const bitset<bits>& bits1,
                         const bitset<bits>& bits2)
```

- 对 *bits1* 和 *bits2* 两者进行“各位逐一 or 运算”并返回结果。
- 返回的新 bitset 中，只有“*bits1* 或 *bits2* 的位值为 1”的那些位才会被设为 1。

```
bitset<bits> operator ^ (const bitset<bits>& bits1,
                         const bitset<bits>& bits2)
```

- 对 *bits1* 和 *bits2* 两者进行“各位逐一 exclusive or 运算”并返回结果。
- 返回的新 bitset 中，只有“*bits1* 或 *bits2* 的位值相异”的那些位才会被设为 1。

型别转换

```
unsigned long bitset<bits>::to_ulong () const
```

- 返回 bitset 所有位所代表的整数
- 如果 unsigned long 不足以表现这个整数，抛出 overflow_error 异常。

```
string bitset<bits>::to_string () const
```

- 返回一个 string，以字符串形式表现该 bitset 的二进制值（不是 0 就是 1）。
- 字符顺序按照 bitset 的索引高低排列。
- 这是一个 template 函数，其回返值型别被参数化了。根据 C++ 语言规则，你必须这么使用：

```
bitset<50> b;
...
b.template to_string<char,char_traits<char>,allocator<char> >()
```

I/O 操作

```
istream& operator >> (istream& strm, bitset<bits>& bits)
```

- 将一个包含 '0' 和 '1' 的字符序列转换为对应位，读入 *bits*。
- 读取行为一直进行下去，直到发生以下数种情况之一：
  - 读取结束（多半是这种情况）。
  - *strm* 中出现 *end-of-file* 符号。
  - 下一个字符既不是 '0' 也不是 '1'。

- 返回 *strm*。
- 如果读入的位少于 bitset 的位数量，前面不足的位填 0。
- 如果此一操作无法读取任何字符，则 *strm* 的 `ios::failbit` 会被设立，导致相关异常被抛出来（参见 13.4.4 节, p602）。

`ostream& operator<< (ostream& strm, const bitset<bits>& bits)`

- 将 *bits* 的二进制形式转换为字符串（成为一个包含 '0' 和 '1' 的序列）。
- 使用 `to_string()` 来产生输出字符（参见 p468）。
- 返回 *strm*。
- 参见 p462 的范例。