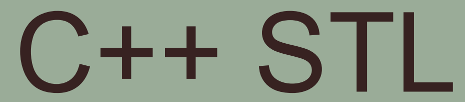


Container-Adapter

Ralf Schuchardt

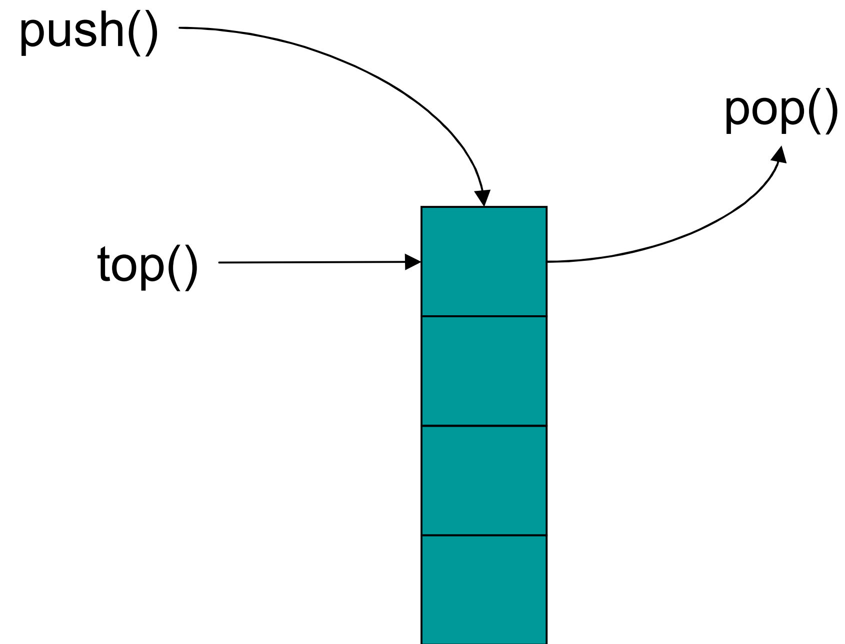
Übersicht

- **stack (LIFO)**
- queue (FIFO)
- priority_queue (SILO)

stack <stack>

```
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 typename Container container_type;
    protected:
      Container c;
    public:
      explicit stack(const Container& = Container());
      bool empty() const           { return c.empty(); }
      size_type size() const        { return c.size(); }
      value_type& top()             { return c.back(); }
      const value_type& top() const { return c.back(); }
      void push(const value_type& x) { c.push_back(x); }
      void pop()                       { c.pop_back(); }
};
```

stack::diagramm



stack::template

- ```
  template <class T,
              class Container = deque<T> >
  class stack {…};
  ```
- Anforderungen an Container:
  - back(), push_back(), pop_back()
- erfüllt von:
  - vector, deque, list
- *deque*:
  - kein Kopieren der Elemente bei Vergrößerung
  - Anfügen und Löschen von Elementen hinten ist schnell
  - Platz sparender als *list*
  ```

stack::constructor

- `explicit stack(const Container& = Container());`
  - kopiert die Elemente aus dem übergebenen Container
  - nicht überall verfügbar (VC6)
- `stack();`
  - erzeugt leeren stack

stack::operationen

- `void push(const value_type& e)`
  - legt Element *e* auf den Stack

- `value_type& top()`
  `const value_type& top() const`
  - liefert oberstes Element
  - Element muss existieren (!empty())
  - Referenz!

- `void pop()`
  - entfernt oberstes Element
  - Element muss existieren (!empty())

stack::sonstiges

- Vergleichsoperatoren für *stack* sind definiert
- Wozu ist *stack* eigentlich gut?
  - eingeschränktes Interface gegenüber *vector*, *deque* und *list*, aber keine erweiterte Funktionalität
  - keine Iteratoren

➔ Genau deswegen!

stack::top_und_pop

- *void pop()*
  T& top() nötig, wenn man das oberste Element haben will
  - Problem?!
- Nein, gut so.
  - saubereres Interface als *{ T pop(); T& top(); }*
  - flexibler als *{ T pop(); }*
  - technisch günstiger: *T&* *top()*, keine unnötigen Kopien

stack::beispiel

```
stack<int> st;

st.push(1);
st.push(2);

assert(st.top() == 2);
st.pop();
assert(st.top() == 1);
st.pop();
assert(st.empty());

cout << "alles OK" << endl;
```

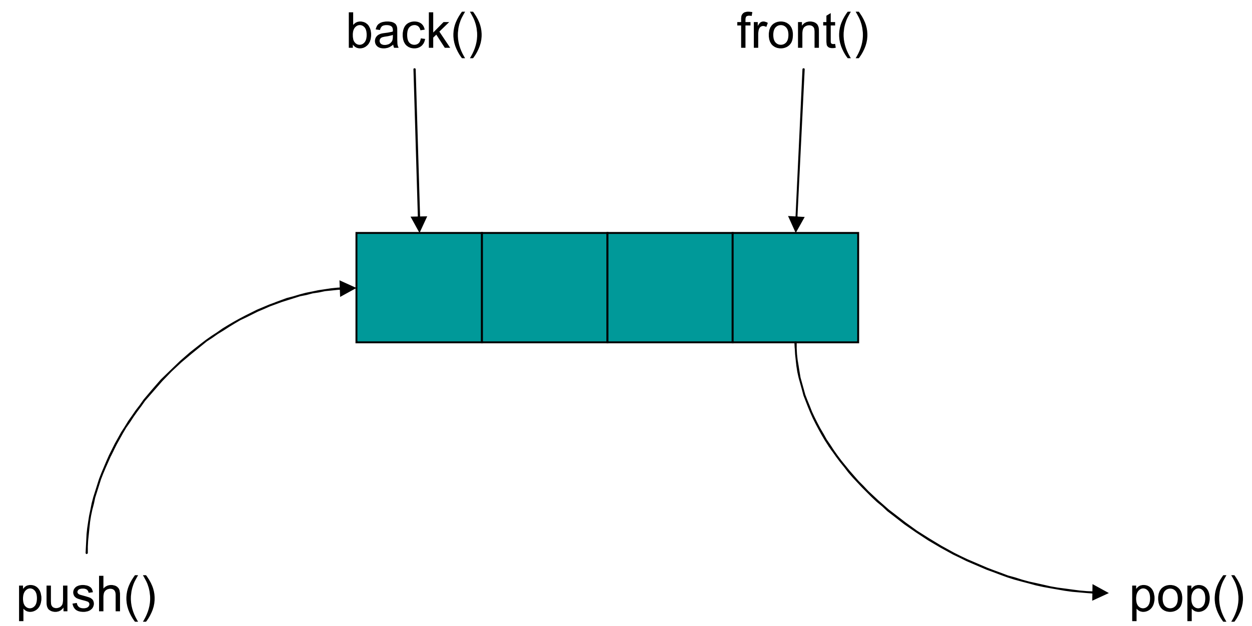
Übersicht

- stack (LIFO)
- **queue (FIFO)**
- priority_queue (SILO)

std::queue <queue>

```
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 typename Container container_type;
protected: Container c;
public:
  explicit queue(const Container& = Container());
  bool empty() const               { return c.empty(); }
  size_type size() const           {  return c.size(); }
  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(); }
  void push(const value_type& x)  {   c.push_back(x); }
  void pop()                       {    c.pop_front(); }
};
```

queue::diagramm



queue::template

- ```
  template <class T,
            class Container = deque<T> >
  class queue {…};
  ```
- Anforderungen an Container:
  - front(), back(), push_front(), pop_back()
- erfüllt von:
  - deque, list
- *deque*:
  - anfügen und löschen von Elementen hinten ist schnell
  - platzsparender als *list*
  ```

queue::constructor

- `explicit queue(const Container& = Container());`
  - kopiert die Elemente aus dem übergebenen Container
  - nicht überall verfügbar (VC6)
- `queue();`
  - erzeugt leere queue

queue::operationen [1]

- `void push(const value_type& e)`
  - fügt das Element e hinten in die Queue ein

- `value_type& back()`
  `const value_type& back() const`
  - liefert das zuletzt eingefügte Element
  - Element muss existieren (!empty())
  - Referenz!

queue::operationen [2]

- `value_type& front()`
  `const value_type& front() const`
  - liefert das nächste zu lesende (vorderste) Element
  - Element muss existieren (!empty())
  - Referenz!

- `void pop()`
  - entfernt vorderstes Element (front())
  - Element muss existieren (!empty())

queue::sonstiges

- Vergleichsoperatoren für Queues definiert
- Wozu ist *queue* eigentlich gut?
  - eingeschränktes Interface gegenüber *deque* und *list*, aber keine erweiterte Funktionalität
  - keine Iteratoren

➔ Genau deswegen!

- gleiche *void pop()*-Problematik wie bei *stack*

queue::beispiel

```
queue<int> qu;

qu.push(1);
qu.push(2);

assert(qu.front() == 1 && qu.back() == 2);
qu.pop();
assert(qu.front() == 2 && qu.back() == 2);
qu.pop();
assert(qu.empty());

cout << "Alles OK" << endl;
```

Übersicht

- stack (LIFO)
- queue (FIFO)
- **priority_queue (SILO)**

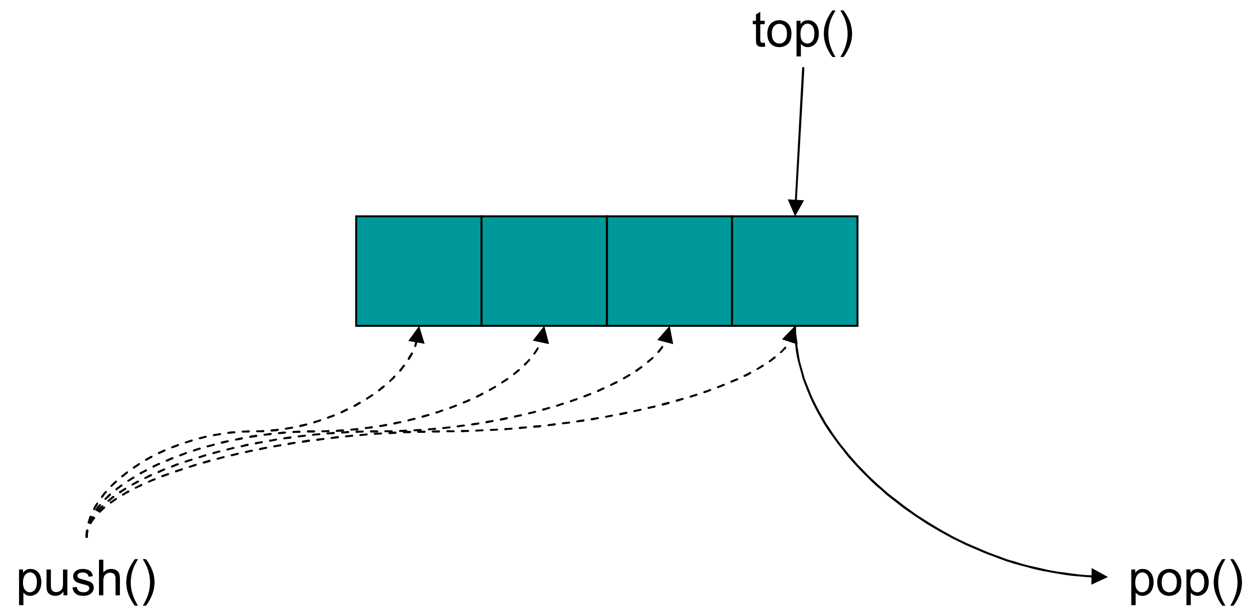
std::priority_queue <queue>

```
template <class T, class Container = vector<T>,
          class Compare = less<Container::value_type> >
class priority_queue {
public:
  typedef typename Container::value_type value_type;
  typedef typename Container::size_type size_type;
  typedef typename Container container_type;
protected:
  Container c;
  Compare comp;
…
};
```

std::priority_queue <queue>

```
template <class T, class Container = vector<T>,
          class Compare = less<Container::value_type> >
class priority_queue {
…
public:
  explicit priority_queue(const Compare& x = Compare(),
                          const Container& = Container());
  template <class InputIterator>
  priority_queue(InputIterator first,
                 InputIterator last,
                 const Compare& x = Compare(),
                 const Container& = Container());
  bool empty() const              { return c.empty(); }
  size_type size() const          { return c.size();  }
  const value_type& top() const   { return c.front(); }
  void push(const value_type& x);
  void pop();
};
```

priority_queue::diagramm



priority_queue::template

- ```
  template <
    class T, class Container = vector<T>,
    class Compare = less<Container::value_type>
  >
  class priority_queue {…};
  ```
- Anforderungen an Container:
  - front(), push_back(), pop_back(), Random-Access-Iteratoren
- erfüllt von:
  - vector, deque
- *vector*
  - einfacher als deque, hält Elemente mehr zusammen (Speicherseiten)
  - zusätzliche Kopien bei Vergrößerung spielen keine Rolle, da sowieso viel kopiert wird
  - Random-Access-Iteratoren bei *vector* am einfachsten, deshalb sehr schnell
- Compare definiert Strict-Weak-Ordering
  ```

priority_queue::constructor

- ```
  explicit priority_queue
     (const Compare& x = Compare(),
      const Container& = Container());
  ```
  - kopiert die Elemente aus dem übergebenen Container
  - Container-Parameter nicht überall verfügbar (VC6)
- ```
  template <class InputIterator>
  priority_queue(InputIterator first,
                 InputIterator last,
                const Compare& x = Compare(),
                const Container& = Container());
  ```
  - kopiert die Elemente aus dem übergebenen Container und hängt [first, last) an
  - Container-Parameter nicht überall verfügbar (VC6)
  - VC6 verlangt Zeiger

priority_queue::operationen

- `void push(const value_type& e)`
  - fügt ein Element in die priority_queue ein
- `const value_type& top() const`
  - liefert das größte Element
  - Element muss existieren (!empty())
  - *const* Referenz!
- `void pop()`
  - entfernt größtes Element (top())
  - Element muss existieren (!empty())

priority_queue::sonstiges

- keine Vergleichsoperatoren für priority_queues definiert
- Kopieren und Vergleichen der Elemente sollte effizient sein
- größtes Element ist vorne (less<T>)
- gleiche *void pop()*-Problematik wie bei *stack*
- nicht stabil (keine Reihenfolge von gleichen Elementen definiert)

priority_queue::implementation

- mittels Heap-Struktur
  - größtes Element immer vorne () $\forall 1 \le \frac{j}{2} < j \le N : K_{\frac{j}{2}} \ge K_j$
  - Einfügen und Löschen in O(log N)
- priority_queue meist mit den heap-Funktionen aus <algorithm> implementiert:
  - make_heap
  - push_heap
  - pop_heap
  - sort_heap

heap [1]

- ```
  void make_heap(RandomAccessIterator first,
                 RandomAccessIterator last
                 [, Compare comp])
  ```
  - konstruiert einen Heap aus dem Bereich [first, last)
  - maximal 3 * N Vergleiche
- ```
  void sort_heap(RandomAccessIterator first,
                 RandomAccessIterator last
              [, Compare comp])
  ```
  - sortiert die Elemente des Heaps [first, last), klein zu groß
  - zerstört den Heap dadurch
  - maximal N log N Vergleiche

heap [2]

- ```
  void push_heap(RandomAccessIterator first,
                 RandomAccessIterator last
              [, Compare comp])
  ```
  - fügt das Element last-1 in den Heap [first, last-1) ein
  - maximal log N Vergleiche
- ```
  void pop_heap(RandomAccessIterator first,
                RandomAccessIterator last
             [, Compare comp])
  ```
  - vertauscht die Elemente first und last-1
  - erzeugt dann aus [first, last-1) einen gültigen Heap
  - maximal 2 * log N Vergleiche

heap::beispiel [1]

```
1.  int feld[10];
2.  for (int i = 0; i < 10; ++i) feld[i] = i;
3.  std::make_heap(feld, feld+10);
4.  for (int d = 10; d > 0; --d) {
5.      std::cout << "pre pop:   ";
6.      for (int i = 0; i < d; ++i)
            std::cout << feld[i] << " ";
7.      std::cout << std::endl << "after pop: ";
8.      pop_heap(feld, feld+d);
9.      for (int i = 0; i < d; ++i)
            std::cout << feld[i] << " ";
10.     std::cout << std::endl << std::endl;
11.     --d;
12. }
```

heap::beispiel [2]

```
$ ./show_heap2
pre pop:    9 8 6 7 4 5 2 0 3 1
after pop:  8 7 6 3 4 5 2 0 1 9

pre pop:    8 7 6 3 4 5 2 0 1
after pop:  7 4 6 3 1 5 2 0 8

pre pop:    7 4 6 3 1 5 2 0
after pop:  6 4 5 3 1 0 2 7

pre pop:    6 4 5 3 1 0 2
after pop:  5 4 2 3 1 0 6

pre pop:    5 4 2 3 1 0
after pop:  4 3 2 0 1 5

pre pop:    4 3 2 0 1
after pop:  3 1 2 0 4

pre pop:    3 1 2 0
after pop:  2 1 0 3

pre pop:    2 1 0
after pop:  1 0 2

pre pop:    1 0
after pop:  0 1

pre pop:    0
after pop:  0
```

priority_queue::stable [1]

- priority_queue nicht stabil
  - ```
    struct Message {
        int priority;
        string msg;
    };
    operator < (const Message& m1,
                const Message& m2)
    { return m1.priority < m2.priortity; }
    ```
  - bei zwei Nachrichten mit gleicher Priorität: keine Aussage über die Reihenfolge
  - FIFO-Reihenfolge aber manchmal sinnvoll
    ```

priority_queue::stable [2]

- ```
  struct Message {
      int priority;
      unsigned insert_num;
      string msg;
      Message& queueInsert() {
          static unsigned insert_counter = 0;
          insert_num = ++insert_counter;
          return *this;
      }
  };
  ```
- ```
  operator < (const Message& m1,
              const Message& m2)
  { if (m1.priority == m2.priortity)
          return m1.insert_num > m2.insert_num;
    else
          return m1.priority < m2.priority; }
  ```
- ```
  priority_queue<Message> pqueue;
  pqueue.push(message.queueInsert());
  ```
  ```

Literatur

- Nicolai Josuttis: Die C++ Standardbibliothek. Addison-Wesley
- Bjarne Stroustrup: Die C++ Programmiersprache. Addison-Wesley
- http://www.sgi.com/tech/stl/
- C++ Standard Draft
- include/…