Языки программирования

Standard Template Library, часть 2

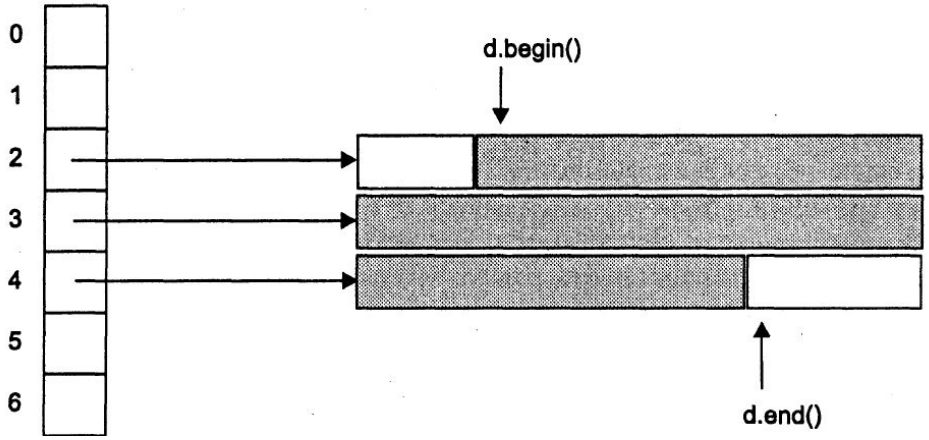
Двусторонние очереди - deque

Двусторонняя очередь – последовательный контейнер, который поддерживает:

- **произвольный доступ** к элементам, как у vector
- **эффективную вставку и удаление** с обоих концов за постоянное время

Распределение памяти выполняется автоматически.

Принцип организации двусторонней очереди



Закрашенная область блоков - занятые элементы
Не закрашенная – свободные элементы

Очередь **разбита на блоки**, доступ к которым осуществляется через **массив указателей**.

При добавлении в начало или конец выделяется память под соответствующий блок.

При выходе за границу массива указателей память перераспределяется под него так, чтобы использовались средние элементы (не занимает много времени).

Произвольный доступ к

Конструкторы создания двусторонней очереди

- По умолчанию

```
explicit deque();
```

- С заполнением `n` одинаковыми значениями, равными `value`

```
explicit deque(size_type n,
               const T& value = T());
```

- С заполнением значениями из другого контейнера с использованием диапазона итераторов `[first, last)`

```
explicit deque(InputIter first,
               InputIter last);
```

- Конструктор копирования

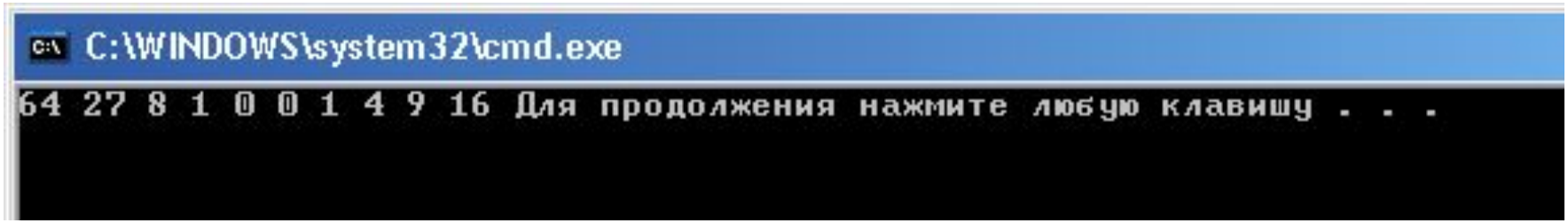
Прочие методы аналогичны vector

- Операция присваивания =, метод присвоения `assign`
- Операция произвольного доступа к элементам `[]`, метод доступа `at`
- Методы получения итераторов `begin`, `end`, `rbegin`, `rend`
- Метод вставки `insert`, метод добавления в конец `push_back`, удаления из конца `pop_back`, добавления в начало `push_front`, удаления из начала `pop_front`
- Метод удаления `erase`, очистки `clear`
- Метод получения количества элементов `size`, изменения количества `resize`

Пример использования двусторонней очереди

```
deque<int> d;

for (int i = 0; i < 5; i++)
{
  d.push_back(i * i);
  d.push_front(i * i * i);
}
for (deque<int>::iterator i = d.begin(); i != d.end(); i++)
```



Двусвязный список - list

Двусвязный список не предоставляет быстрого произвольного доступа к элементам, но реализует эффективно операции вставки и удаления.

Отсюда **для итераторов:**

- `++`, `--` выполняются за постоянное время
- `+k`, `-k` – за время, пропорциональное k

Основные методы двусвязного списка (I)

- Получение первого элемента

```
T& front();
```

- Получение последнего элемента

```
T& back();
```

- Методы добавления и удаления в начало и конец

```
void push_front(const& T value);
void pop_front();
void push_back(const& T value);
void pop_back();
```

- Получение и изменение размера

```
size_type size();
void resize(size_type sz);
```

Основные методы двусвязного списка (II)

- Методы изменения списка

```
iterator insert(iterator pos,
                const T& value);
  void insert(iterator pos,int n,
              const T& value);
  void insert(iterator pos,
       InputIter first,
       InputIter last);
iterator erase(iterator pos);
iterator erase(iterator first,
          iterator last);
void swap(list<T>& x);
void clear();
```

Основные методы двусвязного списка (II)

- Методы сцепки списков – служат для перемещения элементов из одного списка в другой без перераспределения памяти, только за счет изменения указателей
  - Перенос всех элементов `x` перед `pos`

```
void splice(iterator pos, list<T>& x);
```

  - Перенос одного элемента из `x`, на который указывает `i` в позицию перед `pos`

```
void splice(iterator pos, list<T>& x,
            iterator i);
```

  - Перенос всех элементов `x` из диапазона `[first, last)`

```
void splice(iterator pos, list<T>& x,
            iterator first,
```

Основные методы двусвязного списка (III)

- Методы удаления элементов
  - По значению элемента (удаляются все вхождения)

```
void remove(const& T value);
```

  - По условию, задаваемому предикатом

```
void remove_if(Predicate pred);
```

- Методы упорядочивания элементов
  - По возрастанию (для T должна быть определена операция <)

```
void sort();
```

  - С помощью функционального объекта сравнения (см. примеры для priority_queue)

```
void sort(Compare comp);
```

Основные методы двусвязного списка (IV)

- Методы удаления дубликатов элементов в отсортированном списке
  - При наличии операции ==, определенной для T

```
void unique();
```

  - С помощью бинарного предиката, возвращающего значение типа bool (истина ⇔ два элемента равны)

```
void unique(BinaryPredicate pred);
```

- Метод слияния упорядоченных списков

```
void merge(list<T>& x, Compare comp);
```

- Метод изменения порядка элементов на обратный

```
void reverse();
```

Пример использования двусвязного списка

```
//Шаблонная функция печати элементов списка
template <class T>
ostream& operator<< (ostream& out, const list<T>& l) {
  for (list<T>::const_iterator i = l.begin(); i != l.end(); i++)
   out << *i << " ";
  cou
  ret
}
…….
list<int> l;
for (int i = 0; i < 20; i++)
  l.push_back(rand() % 10);
cout << "Random list     : " << l;
l.remove(0);
cout << "After remove(0) : " << l;
l.sort();
cout << "After sort     : " << l;
l.unique();
```

Стеки - stack

Стек – не самостоятельный контейнер, он является адаптером одного из существующих контейнеров (`vector`, `deque`, `list`).

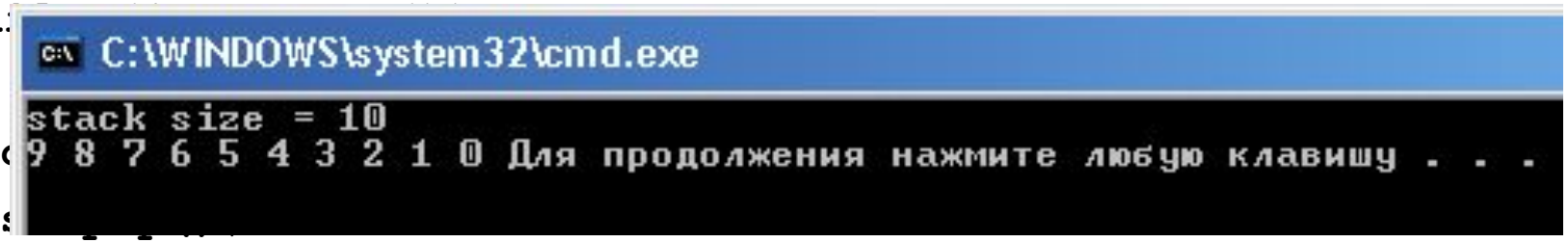
По умолчанию – `deque`.

Реализация стека в STL

```
template <class T, class Container = deque<T> >
class stack {
  protected:
   Container c;
  public:
   explicit stack(const Container& _c = Container()) {
  c = _c; }
   bool empty() const { return c.empty(); }
   size_type size() const { return c.size(); }
   T& top() { return c.back(); }
   void push(const T& x) { c.push_back(x); }
   void pop() { c.pop_back(); }
};
```

Пример использования стека

```
stack<int> st;
for (int i = 0; i < 10; i++)
  st.push(i);
cout << "stack size = " << st.size() << endl;
wh
{
  c
  s
}
```



Очереди - queue

Очередь – не самостоятельный контейнер, он является адаптером одного из существующих контейнеров (`deque`, `list`).

По умолчанию – `deque`.

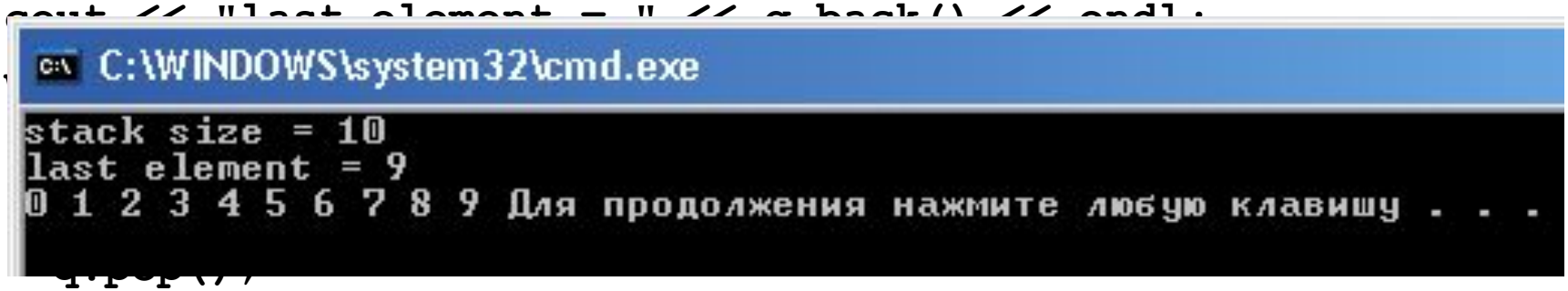
`vector` не подходит, т.к. нет операции удаления из начала.

Реализация очереди в STL

```
template <class T, class Container = deque<T> >
class queue{
  protected:
   Container c;
  public:
   explicit queue(const Container& _c = Container()) {
  c = _c; }
   bool empty() const { return c.empty(); }
   size_type size() const { return c.size(); }
   T& front() { return c.front(); }
   T& back() { return c.back(); }
   void push(const T& x) { c.push_back(x); }
   void pop() { c.pop_front(); }
};
```

Пример использования очереди

```
queue<int> q;
for (int i = 0; i < 10; i++)
  q.push(i);
cout << "queue size = " << q.size() << endl;
```




```
}
```

Очереди с приоритетами – priority_queue

Очередь с приоритетами – очередь, в которой каждому элементу соответствует приоритет, определяющий порядок выборки из очереди.

По умолчанию порядок определяется операцией < - из очереди выбирается максимальный элемент.

`priority_queue` – адаптер контейнера с произвольным доступом к элементам, например, `vector` или `deque` (по умолчанию – `vector`).

Реализация очереди с приоритетами в STL

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

Функциональные объекты сравнения (Бинарные предикаты)

Параметр `Compare` является функциональным объектом, задающим операцию сравнения на меньше. Можно задать стандартные значения:

- `less<тип>` - сравнение на меньше
- `greater<тип>` - сравнение на больше
- `less_equal<тип>` - сравнение на меньше или равно
- `greater_equal<тип>` - сравнение на больше или равно

Данные стандартные функциональные объекты определены в заголовочном файле `<functional>`

Очереди с приоритетами и функциональные объекты

По умолчанию в `priority_queue` используется функциональный объект `less<T>`, который требует, чтобы у типа `T` была реализована операция `<`.

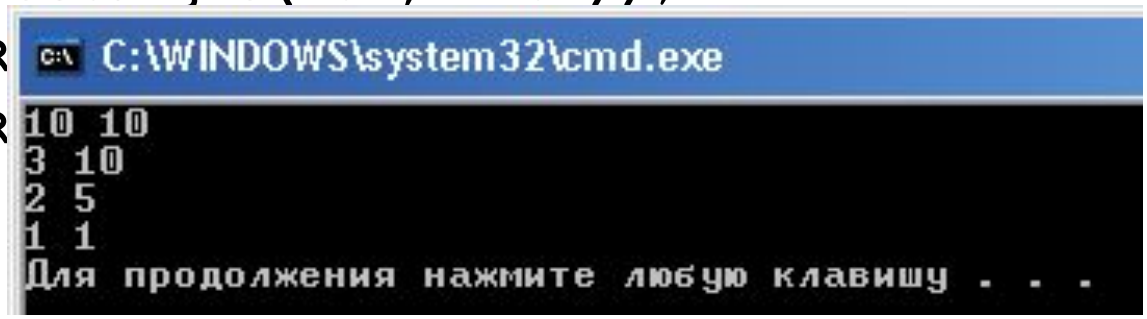
При необходимости можно реализовать собственный функциональный объект сравнения, в нем должна быть определена операция `()` с двумя параметрами (сравниваемыми объектами) результат сравнения (тип `bool`) истина ⇔ первый параметр меньше второго

Пример использования очереди с приоритетами и реализованной собственной операцией < - сравнение прямоугольников по площади (I)

```
class Rectangle
{
public:
  double A, B;
  Rectangle(double _A = 0.0, double _B = 0.0) :
  A(_A), B(_B) {}
  double getArea() const { return A * B; };
  bool operator < (const Rectangle& x) const {
   return getArea() < x.getArea();
  }
};
```

Пример использования очереди с приоритетами и реализованной собственной операцией < - сравнение прямоугольников по площади (II)

```
priority_queue<Rectangle> q;
q.push(Rectangle(1.0, 1.0));
q.push(Rectangle(3.0, 10.0));
q.push(R
q.push(R
while (!
{
  cout << q.top().A << " " << q.top().B << endl;
  q.pop();
}
```



Пример использования очереди с приоритетами и собственным функциональным объектом сравнения

```
……//Определение класса Rectangle идет выше
class RectangleCompare {
public:
  bool operator() (const Rectangle& x, const
  Rectangle& y)
  {
   return x.getArea() < y.getArea();
  }
};
……
priority_queue<Rectangle,
  vector<Rectangle>,RectangleCompare> q;
…… //далее идет использование q
```

Домашнее задание

- Самостоятельно разобраться с контейнерами map, set, multimap, multiset. Выписать их основные методы.