

STL

Тема 5. Адаптеры

Адаптеры

- Адаптеры контейнеров
- Функциональные адаптеры

Адаптеры контейнеров. Stack

```
template < class T, class Container = deque<T> > class stack;
```

- `bool empty () const;`
- `void pop ();`
- `void push (const T& x);`
- `size_type size () const;`
- `value_type& top ();`
- `const value_type& top () const;`

```
Stack<T>
```

```
Stack<T, vector<T>>
```

```
Stack<T, list<T>>
```

```
Stack<T, deque<T>>
```

Адаптеры контейнеров. Queue

```
template < class T, class Container = deque<T> > class queue;
```

- `bool empty () const;`
- `void pop ();`
- `void push (const T& x);`
- `size_type size () const;`
- `value_type& front ();`
- `const value_type& front () const;`
- `value_type& back ();`
- `const value_type& back () const;`

```
queue <T>
```

```
queue <T, list<T>>
```

```
queue <T, deque<T>>
```

Адаптеры контейнеров. Priority_queue

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

- bool empty () const;
- void pop ();
- void push (const T& x);
- size_type size () const;
- value_type& top ();
- const value_type& top () const;

Функциональные адаптеры

bind1st, bind2nd

Not1, not2

Функциональные адаптеры. bind2nd

```
template <class Operation, class T>
binder2nd<Operation> bind2nd (const Operation& op, const T& x)
{
    return binder2nd<Operation>(
        op, typename
        Operation::second_argument_type(x)
        );
}
```

Функциональные адаптеры. bind2nd

```
template <class Operation>
class binder2nd : public unary_function <typename Operation::first_argument_type, typename
    Operation::result_type>
{
protected:
    Operation op;
    typename Operation::second_argument_type value;

public:
    binder2nd ( const Operation& x, const typename Operation::second_argument_type& y)
        : op (x), value(y)
    {}

    typename Operation::result_type operator() (const typename Operation::first_argument_type& x) const
    {
        return op(x,value);
    }
};
```


Функциональные адаптеры. bind2nd

```
int main ()
{
    int numbers[] = {10, 200, 300, 400, 500};
    int* wh = find_if ( numbers, numbers+5, bind2nd(greater<int>(),200));

    cout << "There are " << cx << " negative elements.\n"; return 0;
}
```

```
struct Greater200 : unary_function<int,int>
{
    bool operator()(int x) const { return x > 200;}
}
```

Функциональные адаптеры. bind1st

```
template <class Operation, class T>
binder1st<Operation> bind1st (const Operation& op, const T& x)
{
    return binder1st<Operation>(
        op,
        typename Operation::first_argument_type(x));
}
```

```
int main ()
{
    int numbers[] = {10,20,30,40,50,10};
    int cx = count_if (numbers, numbers+6, bind1st(equal_to<int>(),10) );
}
```

Функциональные адаптеры. Not1

```
template <class Predicate>
class unary_negate :
    public unary_function <typename Predicate::argument_type, bool>
{
protected:
    Predicate fn_;
public:
    explicit unary_negate (const Predicate& pred) : fn_ (pred) {}
    bool operator() (const typename Predicate::argument_type& x) const
    {
        return !fn_(x);
    }
};
```

Функциональные адаптеры. Not1

```
template <class Predicate>
unary_negate<Predicate> not1 (const Predicate& pred)
{
    return unary_negate<Predicate>(pred);
}
```

```
int main ()
{
    int numbers[] = {100, 200, 300, 400, 500};
    int* wh = find_if ( numbers, numbers+5,
                        not1(bind2nd(greater<int>(),200)));
}
```

Функциональные адаптеры. Not2

```
template <class Predicate>
class binary_negate :
    public binary_function <typename Predicate::first_argument_type, typename
    Predicate::second_argument_type, bool>
{
protected:
    Predicate fn_;
public:
    explicit binary_negate ( const Predicate& pred ) : fn_ (pred) {}
    bool operator() (const typename Predicate::first_argument_type& x,
                     const typename Predicate::second_argument_type& y) const
    {
        return !fn_(x,y);
    }
};
```

Функциональные адаптеры. Not2

```
template <class Predicate>
binary_negate<Predicate> not2 (const Predicate& pred)
{
    return binary_negate<Predicate>(pred);
}
```

```
int main ()
{
    int foo[] = {10,20,30,40,50};
    int bar[] = {0,15,30,45,60};
    pair<int*,int*> firstmatch, firstmismatch;
    firstmismatch = mismatch (foo, foo+5, bar, equal_to<int>());
    firstmatch = mismatch (foo, foo+5, bar, not2(std::equal_to<int>()));
}
```

Практическое задание

- Для произвольного целочисленного массива данных написать функцию которая уменьшает все элементы в 2 раза с использованием функциональных объектов и адаптеров
- В массиве символов убрать все двойные пробелы, заменив их на одинарные