

## Тема 8. Библиотека STL

В язык C++ была включена так называемая библиотека стандартных шаблонов (Standard Template Library, STL). Библиотека содержит универсальные шаблонные классы и функции, которые реализуют широкий спектр алгоритмов и структур данных. Библиотека основана на механизме шаблонов `template` и тем самым позволяет работать с самыми разными объектами и типами данных.

Библиотека вводит следующие понятия, которые реализуются в виде программного кода и составляю компоненты этой библиотеки:

- контейнеры (`container`)
- итераторы (`iterator`)
- алгоритмы (`algorithm`)
- функциональные объекты или функторы (`functor`).

### Контейнеры.

Контейнер — это объект, используемый для хранения других (однотипных) объектов, которые могут быть как базовыми типами, так и экземплярами разнообразных классов.

В библиотеке STL реализованы следующие контейнеры:

- `bitset` — битовое множество;
- `deque` — дек;
- `list` — линейный двунаправленный список;
- `map` — ассоциативный контейнер, построенный по принципу `key:value`, в котором каждому ключу `key` соответствует значение `value`;
- `multimap` — ассоциативный контейнер, в котором одному значению (`key`) соответствует несколько значений;
- `multiset` — множество, в котором один и тот же элемент может встречаться несколько раз;
- `priority_queue` — очередь с приоритетами;
- `queue` — обычная очередь;
- `set` — множество, в котором каждый элемент встречается только один раз;
- `stack` — стек;
- `vector` — динамический массив.

Практически все эти структуры рассматривались нами на лекциях. Поэтому не будем останавливаться на их допустимых операциях, поскольку о них можно прочитать в любой справочной литературе. Основная идея работы с контейнерами состоит в использовании итераторов для доступа к элементам. Так, функции поиска элементов возвращают итераторы, по которым далее можно получить доступ к элементам.

Простой пример.

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

int main()
{
```

```

multimap<int, char> m;
m.insert(std::pair<int, char>(1, 'a'));
m.insert(std::pair<int, char>(2, 'b'));
m.insert(std::pair<int, char>(2, 'c'));
m.insert(std::pair<int, char>(2, 'd'));
m.insert(std::pair<int, char>(1, 'e'));
m.insert(std::pair<int, char>(3, 'f'));

for (auto i = m.find(2); i != m.end(); ++i) {
    cout << "find " << i->first << " -> " << i->second << endl;
}

auto r = m.equal_range(2);
for (auto i = r.first; i != r.second; i++) {
    cout << "range find " << i->first << " -> " << i->second << endl;
}
return 0;
}

```

## Итераторы.

Итератор — это абстракция, формализующая последовательный доступ к элементам контейнера через “позицию” элемента в контейнере. Итераторы для разных контейнеров могут различаться по своей функциональности, но все они так или иначе реализуют операции доступа к значению (\* или ->), перемещения (например, ++ или --) и сравнения с другими итераторами (в простейшем случае на равно или не равно). Итераторы могут быть константными и неконстантными.

В C++ различают 5 видов итераторов:

`RandomIter` — итератор произвольного доступа, записывает и извлекает значение из контейнера. Обеспечивает произвольный доступ к элементам контейнера;

`BiIter` — двунаправленный итератор, записывает и извлекает значения. Перемещается вперед и назад;

`ForIter` — прямой итератор — перемещается только вперед. Записывает и извлекает значения;

`InIter` — итератор ввода, только извлекает значения. Перемещается только вперед;

`OutIter` — итератор вывода, только записывает значения. Перемещается только вперед.

Первый итератор работает, например, с массивами `vector`. Двунаправленный итератор работает с большинством остальных контейнеров. Последние два итератора предназначены для классов потокового ввода/вывода, которые реализуют последовательный доступ к данным.

В C++ также имеется понятие интервала, которое определяется как два итератора, задающих позицию первого элемента и позицию после последнего элемента.

Также есть специальные итераторы (адаптеры). Например, так называемый итератор вставки. Он позволяет более эффективно организовать добавление элементов, сокращая количество промежуточных объектов.

## Функторы.

Абстракции контейнера и итератора как объектов, позволяющих работать с любыми типами данных, далее развивается по понятию функтора, как некоторой абстракции для операций, которые, возможно, будут выполняться с этими типами.

Функтор — это класс, в котором реализована операторная функция `operator()`. Таким образом, класс может выступать как некоторая функция, которая теперь может не только вычислять значение, но и сохранять свое внутреннее состояние.

Библиотека STL определяет некоторое количество функторов, которые делятся на две категории:

бинарные — содержат два аргумента;

унарные — содержат один аргумент.

STL содержит следующие бинарные функторы:

`plus` — суммирует (+) два аргумента;

`minus` — вычитает (-) аргументы;

`multiplies` — умножает (\*) два аргумента;

`divides` — делит (/) аргументы;

`modulus` — возвращает результат операции % для двух аргументов;

`equal_to` — сравнивает аргументы на равенство (==);

`not_equal_to` — сравнивает аргументы на неравенство (!=);

`greater` — определяет, больше ли первый аргумент чем второй аргумент (>);

`greater_equal` — определяет, первый аргумент есть больше или равен второму аргументу (>=);

`less` — определяет, меньше ли первый аргумент чем второй аргумент (<);

`less_equal` — определяет, первый аргумент меньше или равен второму аргументу (<=);

`logical_and` — применяет к аргументам логическое «И» (AND);

`logical_or` — применяет к двум аргументам логическое «ИЛИ» (OR).

Также определены два унарных функтора:

`logical_not` — применяет к аргументу логическое «НЕТ» (NOT);

`negate` — изменяет знак своего аргумента на противоположный.

Такие функторы используются в стандартных алгоритмах (см. далее). Они определены для базовых типов, а пользователь может определить их для своих типов и тем самым получить возможность применять стандартные алгоритмы к своим наборам данных.

## Алгоритмы.

Алгоритмы позволяют обрабатывать данные из контейнеров. Например, инициализировать, сортировать, преобразовывать, реализовывать различные виды поиска и т.п.

Все алгоритмы являются шаблонными функциями. Они могут быть применены к любому типу контейнера. В STL есть, например, такие алгоритмы:

`adjacent_find` — осуществляет поиск пары соседних элементов, которые совпадают между собой;

`binary_search` — выполняет бинарный поиск в упорядоченной последовательности;

`copy` — копирует одну последовательность в другую;

`copy_backward` — делает то же, что и `copy`, только результирующая последовательность записывается в обратном порядке;  
`count` — подсчитывает количество вхождений заданного элемента в последовательности;  
`count_if` — вычисляет количество вхождений элемента в последовательности, соответствующей заданному условию;  
`equal` — определяет, совпадают ли элементы двух диапазонов;  
`equal_range` — возвращает диапазон, который допускает вставку элементов без нарушения порядка;  
`fill`, `fill_n` — заполняют диапазон нужными значениями;  
`find` — осуществляет поиск элемента, возвращает позицию (итератор) первого вхождения элемента в последовательности;  
(и еще несколько вариантов поиска . . .)  
`for_each` — применяет указанную функцию к заданному диапазону элементов;  
`generate`, `generate_n` — присваивают элементам диапазона значения, возвращаемые функцией-генератором;  
`include` — определяет, содержит ли одна последовательность другую;  
`inplace_merge` — объединяет два диапазона;  
`make_heap` — создает пирамиду на основе заданной последовательности;  
`max` — возвращает максимум из двух значений;  
`max_element` — возвращает позицию (итератор) на максимальный элемент последовательности;  
`min`, `min_element`  
`merge` — объединяет две упорядоченные последовательности. Результат записывается в третью последовательность;  
`next_permutation` — формирует следующую перестановку элементов последовательности;  
`remove` — удаляет элементы, которые имеют определенное значение;  
`remove_if` — удаляет элементы по заданному предикату;  
`replace` — заменяет элементы заданного диапазона из одного значения на другое;  
`reverse` — изменяет порядок элементов последовательности на противоположный в пределах заданного диапазона;  
`rotate` — выполняет циклический сдвиг влево элементов последовательности в заданных пределах;  
`search` — выполняет поиск подпоследовательности в последовательности;  
`set_difference`, `set_intersection`, `set_symmetric_difference`, `set_union` — операции как с множествами;  
`sort`, `stable_sort` — сортировки;  
`swap` — меняет местами значения, заданные ссылками;  
`transform` — для заданной последовательности применяет функцию к каждому элементу;  
`unique` — удаляет дубликаты из указанной последовательности;  
и кое-какие другие . . .  
Алгоритмы обычно требуют указания диапазона элементов контейнера (в виде итераторов) и требуемой функции (в виде функтора).  
Простой пример — сортировка.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <class T>
ostream & operator<<(ostream &os, const vector<T> &v)
{
    for (auto i = v.cbegin(); i != v.cend(); ++i) {
        os << *i << " ";
    }
    os << endl;
    return os;
}

/*
struct {
    bool operator()(int a, int b) const { return a < b; }
} MyLess;
*/

int main()
{
    vector<int> a = {3, 4, 1, 2, 8, 6, 7, 9};
    cout << a;

    //sort(a.begin(), a.end(), less<int>()); // стандартное сравнение для сортировки
    sort(a.begin(), a.end());
    cout << a;

    sort(a.begin(), a.end(), greater<int>());
    cout << a;

    struct {
        bool operator()(int a, int b) const { return a < b; }
    } MyLess;
    sort(a.begin(), a.end(), MyLess);
    cout << a;

    return 0;
}
```