Язык C++

Мещерин Илья

Лекция 14

9.1$\frac{1}{2}$) Vector<bool>

Из-за особенностей представления данных тип данных *bool* занимает 1 байт, хотя его можно уместить в 1 бит. И в *vector<bool>* как раз-таки можно все организовать так, чтобы хранить информацию об одной ячейке типа *bool* в 1 бите памяти. Т.е. можно упаковывать ячейки вектора в пачки (по 8).

```
class BoolProxy{};

BoolProxy operator[ ](int i){
   return BoolProxy(i);
}
```

Интересный момент в реализации такого вектора - реализация *operator[ ]*. Т.к. нужно обратиться не просто к ячейке массива, а к нужной пачке из переменных типа *bool* и внутри пачки к правильному месту в памяти. Вспомогательный класс будет перехватывать присваивания и правильно их обрабатывать.
Vector<bool> экономит память, но очень медленный (имеет смысл заменять его на *vector<char>*).

9.2) std::deque

```
#include <deque>
```

Представляет собой последовательный индексированный контейнер, который позволяет быстро вставлять и удалять элементы с начала и с конца. Кроме того, вставка и удаление с обоих концов оставляет действительными указатели и ссылки на остальные элементы.
По сравнению с *std::vector* добавляются методы *push_front(), pop_front(), emplace_front()*.

```
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```

9.2$\frac{1}{4}$) std::stack

```
#include <stack>
```

Есть методы *push(), pop(), top()*.

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

Реализован через *std::deque*.

```
std::stack<int, vector<int>> st;
```

Можно создать *std::stack* над *std::vector*, а не над *std::deque*. Можно создать и над другой структурой, важно чтобы у этой структуры были определены соответствующие методы.

9.2$\frac{2}{4}$) std::queue

```
#include <queue>
```

Есть методы *push()* (в конец очереди), *pop()* (из начала очереди), *front(), back()*.

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

Реализован над *std::deque*.

9.2$\frac{3}{4}$) std::priority_queue

Реализация двоичной кучи в стандартной библиотеки.
Есть методы *push()*, *pop()* (с наименьшим приоритетом), *top(), size()*.

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

Можно передать не только контейнер, но и компаратор.

9.3) std::list

```
#include <list>
```

Есть методы *push_back(), push_front(), pop_back(), pop_front(), begin()* (возвращает итератор на первый элемент), *end()* (возвращает итератор на после последнего), *size()*.

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```

Список представляет собой контейнер, который поддерживает быструю вставку и удаление элементов из любой позиции в контейнере. Быстрый произвольный доступ не поддерживается. Он реализован в виде двусвязного списка.

```
iterator insert(std::list<T>::const_iterator pos, const T &value);
```

Вставляет *value* перед элементом, на который указывает *pos*.

```
template<class InputIt>
iterator insert(std::list<T>::const_iterator pos, InputIt first, InputIt last);
```

Вставляет элементы из диапазона [first, last) перед элементом, на который указывает *pos*.

```
void splice(const_iterator pos, list &other,
            const_iterator first, const_iterator last);
```

Перемещение элементов в диапазоне от [first, last) *other* в **this*. Элементы вставляются перед элементом, на который указывает *pos*. При этом копирований не происходит.

```
template<class Compare>
void sort(Compare comp);
l.sort(std::greater<int>());
```

Стандартная сортировка не работает, т.к. для нее нужен *operator[ ]*.

```
l.unique();
```

Удаляет все последовательные эквивалентные элементы, кроме первого.

```
iterator erase(const_iterator pos);
```

Удаляет элемент в позиции *pos*.

```
iterator erase(const_iterator first, const_iterator last);
```

Удаляет элементы в диапазоне [first; last).
Указатели и итераторы к удалённым элементам становятся недействительными. Другие итераторы и указатели остаются без изменений.

```
l.reverse();
```

Меняет порядок элементов в контейнере.

Алгоритмические задачки
Задача: Дан список, проверить зацикливается ли он за линейное время и константную доп. память.
Решение: Заведем два указателя, одним будем идти с шагом 1, другим с шагом 2, один догонит второй за линейное время.
Задача: Дан список, и у каждой вершины есть еще один указатель на какую-то другую вершину списка. Необходимо скопировать структуру этого списка за линейное время и константную память.
Решение: Вставим в наш список после каждой вершины по фиктивной вершине (эти фиктивные вершины образуют в итоге нужный список). Теперь чтобы правильно расставить дополнительные указатели на какие-то другие вершины, достаточно сдвинуть их на соседние элементы. Теперь нужно вытащить новый список.

9.3$\frac{1}{2}$) std::forward_list

```
#include<forward_list>
```

Реализован в виде однонаправленного списка и в отличие от *std::list*, этот тип контейнера не поддерживает двунаправленную итерацию.

```
template<
    class T,
    class Allocator = std::allocator<T>
> class forward_list;
```

Есть методы *push_front(), pop_front(), emplace_front()*.

Нам передается аллокатор для типа T, но в списке нужно выделять память под вершины, т.е. еще и указатели на соседей.

```
template <class Type> struct rebind {
   typedef allocator<Type> other;
};
```

Получаем аллокатор для другого типа.

9.4) std::map

```
#include<map>
```

Отсортированный ассоциативный контейнер, который содержит пары ключ-значение с неповторяющимися ключами. Операции поиска, удаления и вставки имеют логарифмическую сложность. Данный тип, как правило, реализуется как красно-черное дерево.

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

Порядок ключей задается функцией сравнения *Compare*.

Если при использовании *operator[]* обратиться по ключу, которого не существовала, то создастся элемент по этому ключу, проинициализируется по умолчанию и вернется значение по умолчанию (это сделано, для того чтобы не бросать исключение). Метод *at()* в таком случае бросит исключение.
Метод *operator[]*, в отличие от метода *at()*, не является константным и не скомпилируется при работе с константным *map* (т.к. добавляет в него элемент).

```
m.insert(std::make_pair(key, value));
```

Вставляет указанную пару в *map*. Кроме того первым аргументом можно передать итератор, используемый как предположение о том, куда вставить элементы.