4.3. Очередь, дек и алгоритмы поиска

4.3.1. Стек, очередь и дек

Рассмотрим новый контейнер: **очередь**. Очередь бывает из людей или запросов. Новые приходят в конец и удаляются из начала (или наоборот). Её можно реализовать с помощью вектора:

```
v.push_back(x);  // добавляем новый
v.erase(begin(v));  // удаляем из начала вектора, что очень долго
```

Элементы вектора хранятся подряд, и поэтому удаление из начала вектора будет работать за длину вектора, потому что он будет переставлять все элементы в начало, чтобы заполнить полученную пустоту: удалили нулевой, переместили первый на нулевое место, второй на первое и т.д.

Для работы с очередью есть специальный контейнер – `deque` (double-ended queue)

- Это двусторонняя очередь;
- `#include <deque>`;
- Быстрые операции:

```
d.push_back(x)  // добавление в конец
d.pop_back(x)  // удаление из конца
d.push_front(x)  // добавление в начало
d.pop_front(x)  // удаление из начала
d[i]  // обращение к элементу по индексу
```

На коде, представленном ниже, продемонстрируем скорость работы `deque`:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
  int n = 80000;
  vector<int> v(n);
  while (!v.empty()) {
    v.erase(begin(v));
```

```
  }
  cout << "Empty!" << endl;
  return 0;
}
```

Т.е. мы сделали примерно $80000^2/2$ операций (т. к. каждый раз удаляли и перемещали).

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
int main() {
  int n = 80000;
  deque<int> v(n);
  while (!v.empty()) {
    v.erase(begin(v));
  }
  cout << "Empty!" << endl;
  return 0;
}
```

Сработало моментально. И даже `pop_front` тоже сработает сразу. Дек умеет больше, но, как следствие, он менее эффективен. Если нужно работать с двумя концами, используйте дек. Но если хватает вектора, используйте вектор.

Разберём ещё одну структуру: **очередь** (`queue`).

- Если нужна только очередь, используйте `queue`;
- Основана на деке, но работает немного быстрее;
- `#include <queue>`;
- Умеет совсем немного:

```
q.push(x), q.pop(x)   // вставляем в начало и удаляем из конца
q.front(), q.back()   // ссылки на первый и последний элементы очереди
q.size(), q.empty()   // размер и проверка на пустоту
```

Кроме того, существует **стек** (`stack`).

- Позволяет лишь добавлять в конец и удалять из конца;
- `#include <stack>`;
- Как вектор, но умеет меньше:

```
st.push(x), st.pop(x)  // вставляем в конец и удаляем из конца
st.top()               // ссылка на последний элемент
st.size(), st.empty()  // размер и проверка на пустоту
```

4.3.2. Алгоритмы поиска

Рассмотрим специальный класс методов контейнеров и алгоритмов – алгоритмы поиска. Мы с ними уже сталкивались при поиске по вектору и множеству;

- Подсчёт количества:

```
count(begin(v), end(v), x);
s.count(x);
```

- Поиск:

```
find(begin(v), end(v), x);
s.find(x);
```

Рассмотрим задачу поиска элементов в контейнерах:

1. Где будем искать?
   - Неотсортированный вектор (или строка);
   - Отсортированный вектор;
   - Множество (или словарь).
2. Что будем искать и проверять?
   - Проверить существование;
   - Проверить существование и найти первое вхождение;
   - Найти первый элемент, больший или равный данному;

- Найти первый элемент, больший данного;
- Подсчитать количество;
- Перебрать все.

Как осуществляется поиск в неотсортированном векторе?

- Поиск конкретного элемента:

```
find(begin(v), end(v), x)
```

- Элемент по какому-то условию (больше, меньше, больше или равен):

```
find_if(begin(v), end(v), [](int y) {...})
```

- Посчитать количество:

```
count(begin(v), end(v), x)
```

- Перебрать все можно с помощью цикла и `find`.

Например, выведем позиции всех пробелов в строке:

```
for (auto it = find(begin(s), end(s), ' ');
     it != end(s);
     it = find(next(it), end(s), ' ')) {  // переходим в цикле к следующему пробелу
  cout << it - begin(s) << " ";  // next(it) эквивалентен it + 1
}
```

В отсортированном векторе поиск можно осуществить быстрее с помощью бинарного поиска. Количество операций равно $\log_2(N)$ – двоичному логарифму числа элементов. Столько же работает поиск во множестве и словаре.

Отсюда следствие: если вы просто хотите быстро искать по набору элементов, но не хотите добавлять новые или удалять какие-то, вам достаточно отсортированного вектора. Это будет оптимальнее, чем если вы используете множество. В отсортированном векторе можно искать так:

- Проверка на существование:

```
binary_search(begin(v), end(v), x)
```

- Первый больший или равный данному:

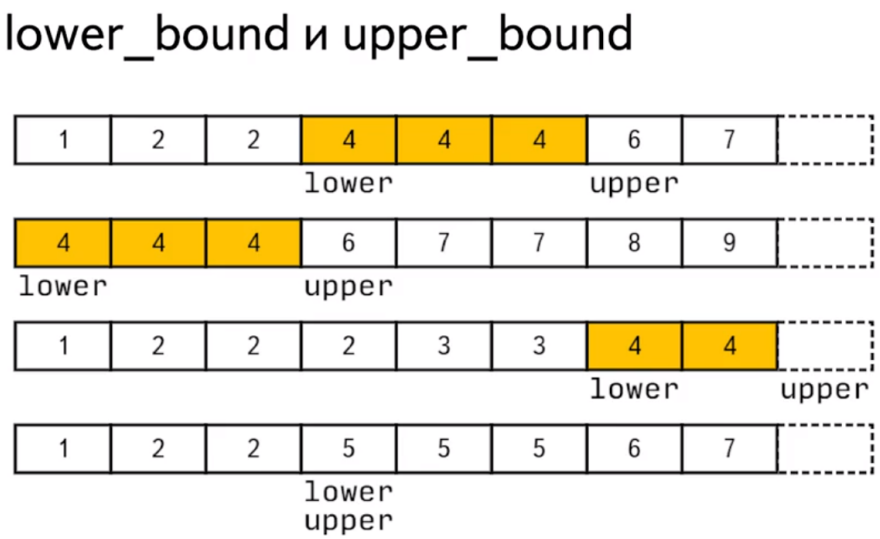
```
lower_bound(begin(v), end(v), x)
```

- Первый элемент, больший данного:

```
upper_bound(begin(v), end(v), x)
```

- Диапазон элементов, равных данному (аналог `minmax`):

```
equal_range(begin(v), end(v), x) ==
make_pair(lower_bound(...), upper_bound(...))
```



Ещё про `equal_range`.

- Если элемент есть, то `equal_range = [lower_bound, upper_bound)` – диапазон всех вхождений;
- Если же элемента нет, то `lower_bound == upper_bound` – позиция, куда можно вставить элемент без нарушения порядка сортировки;
- Количество вхождений `== upper_bound - lower_bound`;
- А перебрать все элементы, равные данному, можно просто проитерировавшись от `lower_bound` до `upper_bound`.

Поиск во множестве мы уже знаем:

- `s.count(x)`;
- `s.find(x)`;
- `s.lower_bound(x)`;
- `s.upper_bound(x)`;
- `s.equal_range(x)`.

4.3.3. Анализ распространённых ошибок

Первый пример распространённых ошибок – вычитание итераторов множества:

```
int main() {
  set<int> s = {1, 2, 7};
  end(s) - begin(s);
  return 0;
}
// no match for 'operator-'
```

Это ошибка простая, а вот если мы возьмём алгоритм, принимающий `Random` итераторы (например, `partial_sort`) и передадим ему итераторы множества:

```
int main() {
  set<int> s = {1, 2, 7};
  partial_sort(begin(s), end(s), end(s))
  return 0;
}
// no match for 'operator-', 'operator+', 'operator<'
```

Всё по той же причине. Итератор множества – не `Random` итератор, по нему нельзя сравнивать или перемещать элементы.

Теперь попробуем вызвать `remove`:

```
int main() {
  set<int> s = {1, 2, 7};
  remove(begin(s), end(s), 0);
  return 0;
}
// assignment of read-only location ...
```

Т. е. присваивание в итератор, содержимое которого нельзя менять. Ссылка под итераторами константная.

Теперь одна из самых страшных ошибок (не ловится на этапе компиляции) – передача диапазона, у которого итераторы от разных контейнеров:

```
int main() {
  vector<int> s1 = {1, 2, 7};
  vector<int> s2 = {2, 7};
  sort(begin(s1), end(s2));  // диапазон от начала одного вектора до конца другого
  return 0;
}
```

Запускаем код – и программа упала. В обратном порядке она, скорее всего, зациклится. Проверить это можно, закомментировав код, и если он заработает, проблема в нём.

Если же мы передадим итераторы разных типов, то:

```
int main() {
  vector<int> s1 = {1, 2, 7};
  vector<int> s2 = {2, 7};
  sort(begin(s1), rend(s2));
  return 0;
}
// deduced conflicting types for parameter random access iterator
```

Итераторы должны иметь один тип. А у нас в данном случае тип разный и не получается вызвать функцию `sort`.