

Lecture 23

Контейнеры

Лучший контейнер - вектор. Самый часто используемый.

Последовательные контейнеры

- Контейнеры
 - **vector** — массив с переменным размером и гарантией непрерывности памяти*
 - **array** — массив с фиксированным размером, известным в момент компиляции
 - **deque** — массив с переменным размером без гарантий по памяти
 - **list** — двусвязный список
 - **forward_list** — односвязный список
- Адаптеры
 - **stack** — FIFO контейнер, чаще всего на базе deque
 - **queue** — LIFO контейнер, чаще всего на базе deque
 - **priority_queue** — очередь с приоритетами, чаще всего на базе vector

**When choosing a container, remember vector is best. Leave a comment to explain if you choose from the rest*

3

(c) Tony van Eerd

Белые кружки - гарантии по памяти.

Адаптеры - содержат в себе один из контейнеров с специфичным интерфейсом.

Deque

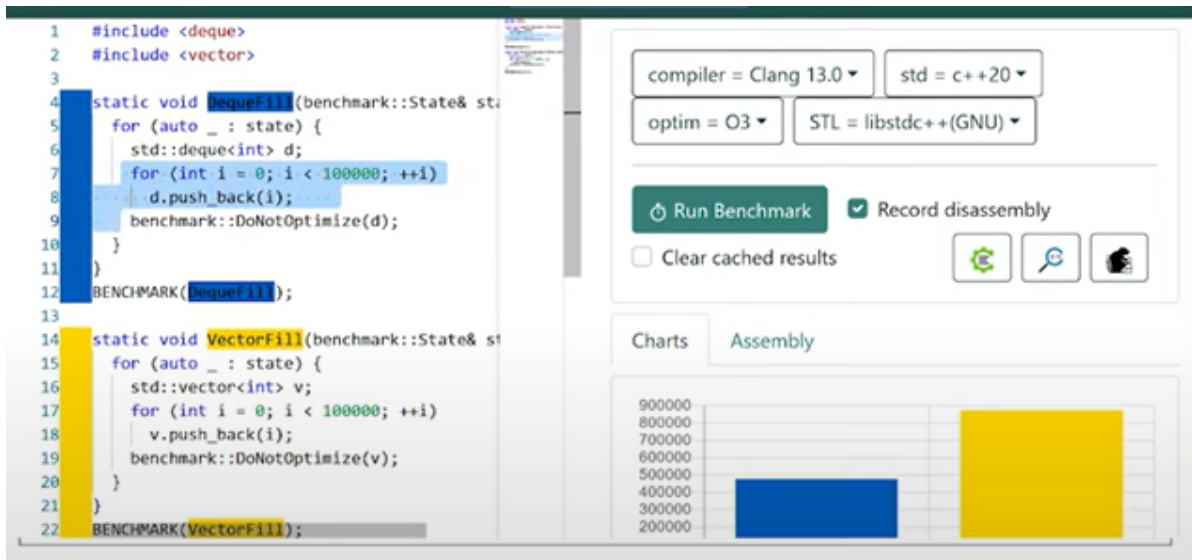
Что может смущать в этом коде?

```
• std::deque<int> d; // подумайте если бы это был vector?  
  
for (int i = 0; i != N; ++i) {  
    d.push_front(i);  
    d.push_back(i);  
}
```

- **deque** — массив с переменным размером без гарантий по памяти
- Поэтому ответ: всё хорошо.
- Вставка в начало и в конец дека имеет всегда честную константную сложность $O(1)$.

4

Fill bench



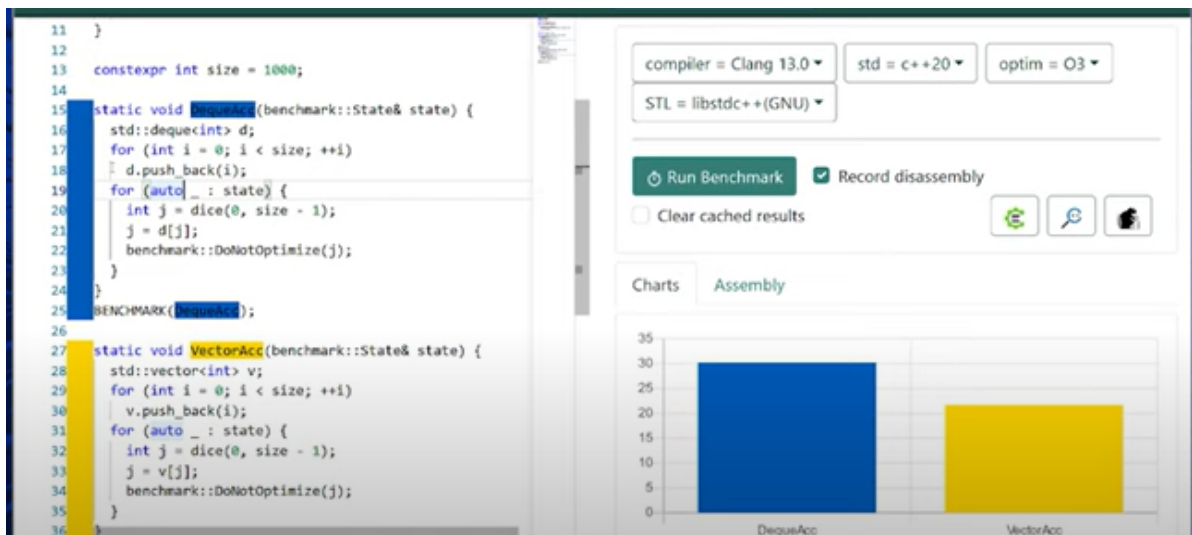
Видно, что из-за релокаций заполнение вектора идет медленнее! Поэтому при заполнении вектора нужно делать `reserve`.

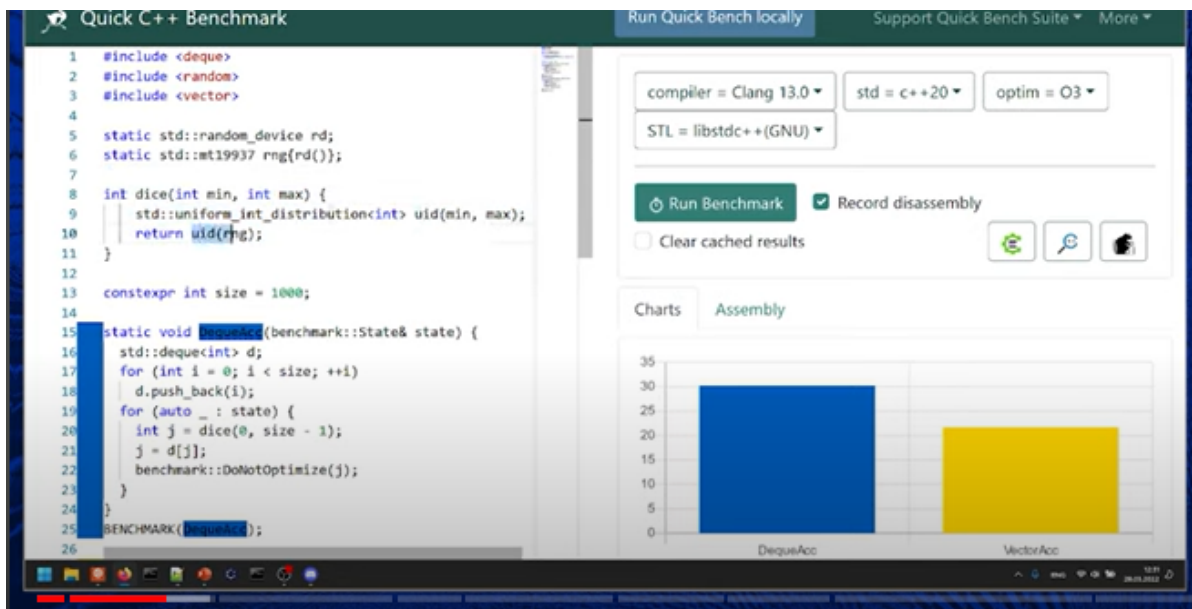
Рассмотрите deque вместо vector*

- Эффективно растёт в обоих направлениях
- Не требует больших реаллокаций с перемещениями, так как разбит на блоки
- Гораздо меньше фрагментирует кучу

Но оставьте комментарий в коде если вы его действительно **выберете*

Deque плохо вставляет в середину. Лучше использовать при частых вставках в начало или конец. Также deque плох при обращении к элементам.





Обратим внимание на интерфейс random. Rand() использует линейный конгруэнтный генератор, который статистически намного хуже Мейсен-твистора.

Деки против векторов

Вектора

- Доступ к элементу $O(1)$
- Вставка в конец аморт. $O(1)+$
- Вставка в начало $O(N)$
- Вставка в середину $O(N)$
- Вычисление размера $O(1)$
- Есть гарантии по памяти
- Есть `reserve / capacity`

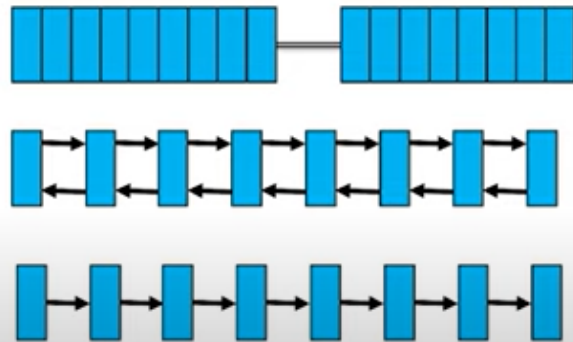
Деки

- Доступ к элементу $O(1)$
- Вставка в конец $O(1)$
- Вставка в начало $O(1)$
- Вставка в середину $O(N)$
- Вычисление размера $O(1)$
- Нет гарантий по памяти
- Нет необходимости в `reserve/capacity`

Lists

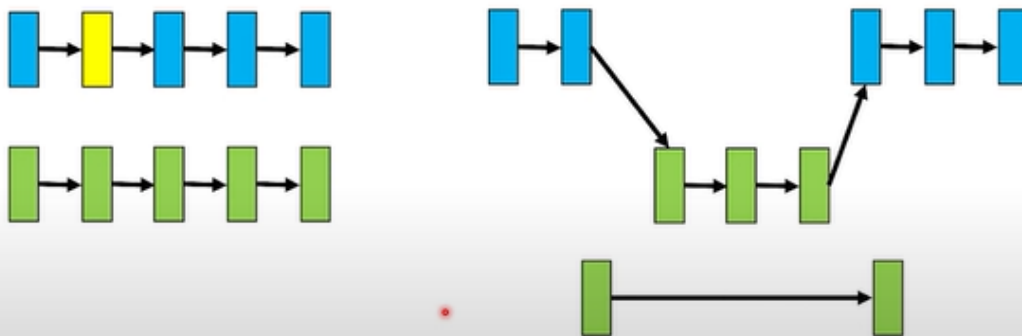
Узловые контейнеры

- **deque** произвольный доступ, быстрая вставка в начало и в конец.
- **forward_list** последовательный доступ, быстрая вставка в любое место.
- **list** последовательный доступ, быстрая вставка в любое место, итерация в обе стороны.



8

Особая возможность списков: сплайс



9

Сплайс может занимать **$O(N)$** в самом сложном случае и $O(1)$ при вставке в начало или конец. И причина не в перевязке указателей. Причина в пересчете размера контейнера.

Магия сплайсов

В листах вставка производится ровно на то место, на которое указывает итератор

```
x86-64 gcc 12.2  -O0
Program returned: 0
Program stdout
1
0
2
3
4
5

1
2
3
4
5
6
7
8
9
10
11
12
#include <iostream>
#include <list>

int main() {
    std::list<int> l{1, 2, 3, 4, 5};
    auto &it = ++l.begin();
    l.insert(it, 0);
    for (auto &elem: l) {
        std::cout << elem << std::endl;
    }
}
```

Однако в `forward_list` это не так: при указании на `begin` вставка листа в лист не будет производиться в начало.

```
Program returned: 0
Program stdout
1
99
2
3
4
5

2 #include <iostream>
3 #include <forward_list>
4
5 int main() {
6     std::forward_list<int> l{1, 2, 3, 4,
7     auto &&it = l.begin();
8     l.insert_after(it, 99);
9     for (auto &&elem: l) {
10         std::cout << elem << std::endl;
11     }
12 }
```

Нужно использовать `before_begin`.

```
x86-64 gcc 12.2
Program returned: 0
Program stdout
99
1
2
3
4
5

1 #include <iostream>
2 #include <forward_list>
3
4
5 int main() {
6     std::forward_list<int> l{1, 2, 3, 4,
7     auto &&it = l.before_begin();
8     l.insert_after(it, 99);
9     for (auto &&elem: l) {
10         std::cout << elem << std::endl;
11     }
12 }
```

Сплайс для списков: простая форма

```
forward_list<int> fst = { 1, 2, 3 };
forward_list<int> snd = { 10, 20, 30 };
auto it = fst.begin(); // указывает на 1

// перемещаем second в начало first, it указывает на 1
fst.splice_after(fst.before_begin(), snd);
```



При этом `it` не изменился: он продолжает указывать на число 1. Он не инвалидировался.

Сплайс для списков: сложная форма

```
// forward_list<int> fst = {10, 20, 30, 1, 2, 3 };
// forward_list<int> snd = {};
// it указывает на 1

// перекидываем элементы со второго по it в список second
snd.splice_after(snd.before_begin(), fst, fst.begin(), it);
```



11

Теперь `second` пустой. Перекинем в него элементы из первого **со второго** (т.к. `begin`, а не `before_begin`) **по итератор на число 1 (не включительно)**, т.к. все интервалы в стандартной библиотеке полуоткрытые)

Сплайс для списков: средняя форма

```
// forward_list<int> fst = { 10, 1, 2, 3 };
// forward_list<int> snd = { 20, 30 };
// it указывает на 1

// все элементы второго списка начиная со второго в первый
fst.splice_after(fst.before_begin(), snd, snd.begin());
```



12

<https://godbolt.org/z/PgMgarjTd>

Здесь мы вставляем луч элементов второго списка в первый в самое начало.

Таким образом,

- Простая форма - перемещение одного в другой $O(1)$
- Средняя форма - откуда-то и до конца $O(1)$
- Сложная форма - перемещение части в конкретное место $O(N)$

Лучше проверять на эксперименте `splice`, ибо все время забывается интерфейс.

Применение:

1. В многопоточной среде итераторы не инвалидируются
2. Перевязка, например в LRU cache

Адаптеры

Идея контейнерных адаптеров

"Стек"



"Очередь"



14

Виды адаптеров

- **stack** – LIFO стек над последовательным контейнером

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

- **queue** – FIFO очередь над последовательным контейнером

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

- **priority_queue** – очередь с приоритетами (как binary heap) над последовательным контейнером

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

15

Адаптеры - ужатые с точки зрения интерфейса контейнеры - для удобства.

priority_queue построена на векторе, т.к. по алгоритмы в бинарной куче нужно много прыгать, т.е. random_access требуется за $O(1)$.

Применение:

- алгоритм Прима для построения остоного дерева.

Остовное дерево - все вершины соединены минимальным числом ребер.

- UNIX sort при слянии отсортированных частей

Почему адаптер стэк не использует контейнер односвязный список?

Нету интерфейса `push_back` / `pop_back`.

Контейнеро-подобные классы

Битовые маски

Никогда не используется `std::array<bool>` . Он не оптимизирован.

Используйте `std::bitset<mask_size>`.