STL: последовательные контейнеры

Последовательные контейнеры

Контейнер - тип данных, который обеспечивает хранение объектов других типов, а также интерфейс для доступа к ним.

Последовательный контейнер обеспечивает упорядоченный способ хранения элементов, завиясящий от времени и места их добавления, но не зависящий от их значений.

`std::array`

`std::array`

`std::array<T, N>` - аналог C-style массива из `N` элементов типа `T`.

Все элементы живут на стеке, что обеспечивает быстрый доступ к ним, но лишает возможности динамического расширения.

- Может быть быть проинициализирован так же как и обычный массив:

```
int c_style[5]{1, 2, 3};                  // [1, 2, 3, 0, 0]
std::array<int, 5> cpp_style{1, 2, 3};  // [1, 2, 3, 0, 0]
```

- При отсутствии инициализатора заполняется значениями по умолчанию:

```
std::array<int, 5> ints;  // no initialization
ints[0];  // UB

std::array<SomeClass, 5> objects;  // 5 default constructors
objects[0];  // Ok
```

std::array

- В отличие от C-style массивов могут быть скопированы.

```
// int copy[5] = c_style;  // CE
auto copy = cpp_style;     // Ok
```

- Могут быть переданы в функцию по значению.

```
void GetCArray(int arr[5]);  // <=> void GetCArray(int* arr);
void GetArray(std::array<int, 5> arr);

GetCArray(c_style);   // передается указатель на первый элемент!
GetArray(cpp_style);  // копируется в аргумент целиком
```

- Чтобы избежать копирования можно принимать по ссылке

```
void GetByRef(const std::array<int, 5>& arr);

GetByRef(cpp_style);
```

`std::array`

- Для доступа к элементам используются `[]`, `at`, `front`, `back`, `data` [O(1)]

```
void IWantPointer(int* ptr, int size);

IWantPointer(array.data(), array.size());
```

- `empty`, `size` [O(1)]

```
array.size();   // сравните с sizeof(a) / sizeof(a[0]); для C-style
array.empty();  // true, если N == 0
```

- `fill`, `swap` [O(N)]

```
array.fill(1);      // [1, 1, 1, 1, 1]
array.swap(copy);   // array == [1, 2, 3, 0, 0]; copy == [1, 1, 1, 1, 1]
```

- Лексикографическое сравнение [O(N)]

`std::vector`

`std::vector`

`std::vector<T>` - динамически расширяющийся массив элементов типа `T`.

В отличие от `std::array` имеет нефиксированный размер и хранит данные не на стеке, а в куче.

```
int n;
std::cin >> n;

auto c_style = new int[n];
// не знает свой размер
// нужно помнить о delete[]
// самостоятельно не расширяется

std::vector<int> cpp_style(n);
// знает свой размер (метод size)
// память очищается автоматически в деструкторе
// расширяется по мере необходимости
```

`std::vector`: инициализация

```cpp
// пустой массив
std::vector<T> a;

// массив из n элементов, созданных по умолчанию
std::vector<T> b(n);

// массив из n копий value
std::vector<T> c(n, value);

// списочная инициализация массива
std::vector<T> d{1, 2, 3};

// копирование [O(N)]
auto b_copy = b;

// перемещение [O(1)]
auto c_move = std::move(c);
```

std::vector: доступ к элементам

- Для доступа к элементам используются `[]`, `at`, `front`, `back`, `data` [O(1)]

```
void IWantPointer(int* ptr, int size);

IWantPointer(vector.data(), vector.size());
```

- `empty`, `size` [O(1)]

```
vector.size();
vector.empty();
```

- `fill` [O(n)], `swap` [O(1)]

```
vector.fill(1);     // [1, 1, 1, 1, 1]
vector.swap(copy);
```

- Лексикографическое сравнение [O(n)]

`std::vector`: вставка/удаление элементов

- `push_back`, `pop_back` [амортизированно O(1)]

```
std::vector<int> v{1, 2};
v.push_back(3);  // {1, 2, 3}
v.pop_back();    // {1, 2}
```

- `insert`, `erase` [O(n) в худшем случае]

```
std::vector<int> v{1, 2, 3};

v.insert(v.begin() + 1, 0);              // {1, 0, 2, 3}
v.insert(v.begin() + 3, 2, -1);          // {1, 0, 2, -1, -1, 3}

v.erase(v.begin() + 1);                  // {1, 2, -1, -1, 3}
v.erase(v.begin() + 2, v.begin() + 4);   // {1, 2, 3}
```

`std::vector`: вставка/удаление элементов

- `resize`

```
std::vector<int> v{1, 2};
v.resize(4);       // {1, 2, ?, ?}
v.resize(5, -1);   // {1, 2, ?, ?, -1}
v.resize(2);       // {1, 2}
```

- `clear`

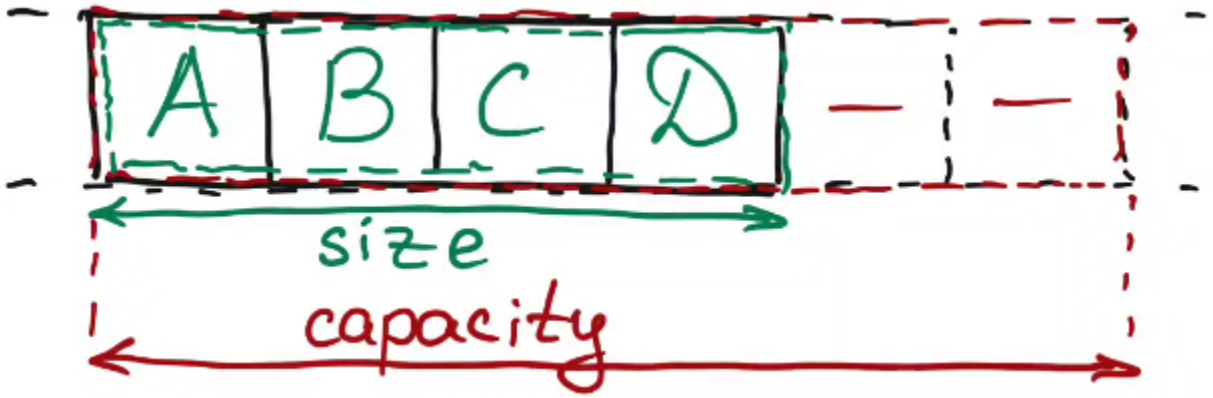
```
std::vector<int> v{1, 2};
v.clear();  // {}
```

`std::vector`: управление хранилищем

Для обеспечения амортизационной константы при добавлении элементов в конец используется мультипликативная схема расширения массива.

В частности, это означает, что в каждый момент времени памяти выделено несколько больше, чем нужно для хранения всех объектов.

В связи с этим у вектора есть 2 разных атрибута: `size` и `capacity` (`capacity` $\geq$ `size`).



[demo: size_vs_capacity]

`std::vector`: управление хранилищем

- `reserve(count)` - увеличивает вместимость массива *минимум* до *count*

```
std::vector<int> v(5);  // v.size() == 5, v.capacity() >= 5
v.reserve(10);          // v.size() == 5, v.capacity() >= 10
v.reserve(2);           // ничего не происходит
```

- `shrink_to_fit` - уменьшает `capacity` до `size`

```
std::vector<int> v(1000);  // v.size() == 1000, v.capacity() >= 1000
v.resize(1);               // v.size() == 1   , v.capacity() >= 1000
v.shrink_to_fit();         // v.size() == 1   , v.capacity() == 1
```

`shrink_to_fit` - единственный метод, который может уменьшить `capacity` (`resize` и `reserve` до меньших размеров не приводят к реаллокации).

Загадочный std::vector

```
// создаем вектор из 100 векторов размера 1'000'000
std::vector<std::vector<int>> v(100, std::vector<int>(1'000'000, 1));

v.resize(10);  // v.size() == 10, v.capacity() >= 100
v.clear();     // v.size() == 0,  v.capacity() >= 100
```

Хотим ли мы, чтобы ненужные векторы занимали память?

```
std::vector<A> v;
v.reserve(100);
```

А что если у `A` нет конструктора по умолчанию?

Загадочный std::vector

Сколько работает метод `clear`?

Сколько работает `resize` при уменьшении размера?

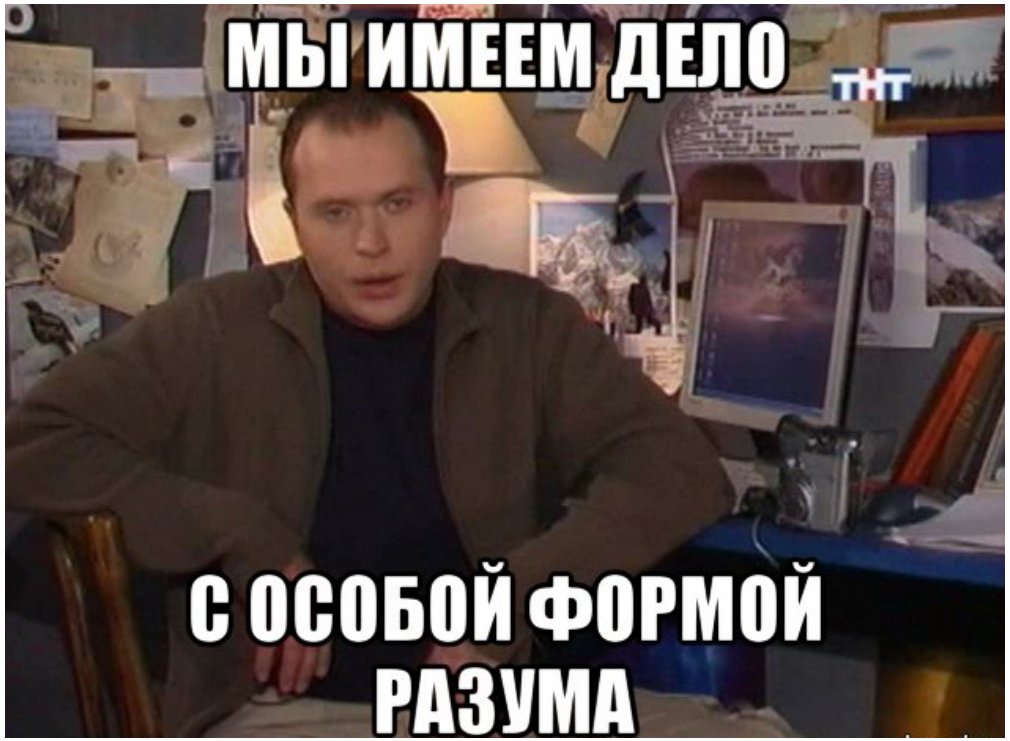
Сколько работает `reserve` при увеличении вместимости?

Загадочный std::vector

`clear()` работает за $\Theta(size)$.

`resize(count)` работает за $\Omega(|size - count|)$

`reserve(count)` работает за $O(size)$



[demo: vector_memory_management]

`std::vector`: `emplace`, `emplace_back`

Помимо `insert` и `push_back` у `std::vector` есть так называемые "размещающие" методы.

Проблема:

```
std::vector<std::vector<int>> vv;
vv.push_back(std::vector<int>(10, 1));
vv.insert(v.begin(), std::vector<int>(5, -1));
...
```

При каждом вызове `push_back` или `insert` создается временный объект, который затем перемещается (либо копируется) в нужное место вектора (2 действия).

Хотелось бы создать объект в нужном месте сразу в 1 действие.

`std::vector`: `emplace`, `emplace_back`

Помимо `insert` и `push_back` у `std::vector` есть так называемые "размещающие" методы.

Решение:

```
std::vector<std::vector<int>> vv;
vv.emplace_back(10, 1);
vv.emplace(v.begin(), 5, -1);
...
```

Данные методы принимают параметры конструктора объекта и создают его непосредственно в нужной ячейке памяти без лишних трат на копирование/перемещение + выглядит более лаконично.

`std::vector` : полезные советы

Совет 1: используйте `reserve`

Классика: на вход приходит число n, а затем следует n строк с данными, которые нужно сохранить в вектор:

```
const auto n = ReadInt();
std::vector<int> v;
for (int i = 0; i < n; ++i) {
  v.push_back(ReadInt());  // периодические реаллокации памяти
}
```

Если максимальное число элементов известно заранее, то стоит сразу выделить достаточное количество памяти.

```
const auto n = ReadInt();
std::vector<int> v;
v.reserve(n);  // сразу выделяем память под n элементов
for (int i = 0; i < n; ++i) {
  v.push_back(ReadInt());  // реаллокаций не происходит
}
```

Совет 2: быстрое удаление

Задача: необходимо удалить элемент из произвольного места вектора

```
std::vector<A> v;
...
v.erase(v.begin() + pos);  // потенциально O(n)
```

Решение: если порядок элементов в векторе не важен, то можно воспользоваться следующим трюком:

```
std::vector<A> v;
...
v[pos] = std::move(v.back());  // как правило, O(1)
v.pop_back();                  // O(1)
```

Совет 3: erase-remove идиома

Задача: нужно удалить элементы, удовлетворяющие некоторому критерию, с *сохранением относительного порядка элементов*.

```
// хотим удалить все 0
for (int i = 0; i < v.size();) {  // O(n^2)
  if (v[i] == 0) {
    v.erase(v.begin() + i);
  } else {
    ++i;
  }
}
```

Решение:

```
auto zeros_begin = std::remove(v.begin(), v.end(), 0);  // O(n)
v.erase(zeros_begin, v.end());  // O(n)
```

`std::deque`

`std::deque`

`std::deque<T>` - шаблонный класс двунаправленной очереди.

Позволяет эффективно (амортизированно за O(1)) добавлять элементы как в начало, так и в конец (`push_back` , `push_front` , `emplace_back` , `emplace_front`).

В остальном похож на вектор (`[]` , `at` , `resize` , `clear` , ...).

В силу внутреннего устройства не предоставляет методы `reserve` , `capacity` , но позволяет вызывать `shrink_to_fit` .

`std::deque` инвалидация ссылок и указателей

Что может пойти не так?

```
std::vector<int> v;
...
int& x = v[5];
v.push_back(10);
x = 0;
```

`std::deque` инвалидация ссылок и указателей

Что может пойти не так?

```
std::vector<int> v;
...
int& x = v[5];
v.push_back(10);  // произошла реаллокация
x = 0;  // BOOM! (ссылка больше невалидна)
```

Аналогичное может произойти и с указателем

```
std::vector<int> v;
...
int* p = &v[5];
v.push_back(10);  // произошла реаллокация
*p = 0;  // BOOM! (указатель провис)
```

`std::deque` инвалидация ссылок и указателей

У `std::deque` такой проблемы нет

(если изменения не затрагивают целевой элемент)

```
std::deque<int> d;
...
int& x = d[5];
d.push_back(10);  // произошла реаллокация
x = 0;  // Всегда Ok!
```

```
std::deque<int> d;
...
int* p = &d[5];
v.push_front(10);  // произошла реаллокация
*p = 0;  // Всегда Ok!
```

Получается, `std::deque` - `std::vector` на максималках?

Зачем тогда нужен `std::vector`?

`std::deque`: реализация

`std::deque` vs `std::vector`

В общем случае, операции над `std::vector` эффективнее чем над `std::deque`.

Если речь идет о возможности добавления в начало и инвалидации ссылок, то стоит предпочесть `std::deque`.

И помните главный принцип C++: "не плати за то, что не используешь".

`std::list`

`std::list<T>` - шаблонный класс двусвязного списка

- `push_back`, `emplace_back`, `pop_back`, `push_front`, `emplace_front`, `pop_front`
- `front`, `back`, доступ к остальным элементам через итераторы (нет `[]`)
- `insert`, `erase` за O(1)
- Не инвалидирует ссылки и указатели на элементы

`std::list`: `splice`

Преимуществом списков является то, что можно быстро (за O(1)) перенести элементы одного списка в другой или переставить элементы списка внутри него самого.

```
std::list<int> l{...};
std::list<int> other{...};
```

1. Splice

```
l.splice(pos, other);  // переносим все узлы из other в l, в позицию pos
```

2. Single-element splice

```
l.splice(pos, other, elem);  // переносим узел, на который указывает elem
```

`std::list`: `splice`

```
std::list<int> l{...};
std::list<int> other{...};
```

3. Range splice

```
l.splice(pos, other, begin, end);  // переносим узлы с begin до end
```

Вопросы:

- Какова сложность этой версии `splice`?
- Какова сложность метода `size()`?

`std::list`: `splice`

```
std::list<int> l{...};
std::list<int> other{...};
```

3. Range splice

```
l.splice(pos, other, begin, end);  // переносим узлы с begin до end
```

Эта версия `splice` работает за $O(1)$, если $other == size$, и $O(distance(begin, end))$ иначе.

Иначе нельзя бы было гарантировать `size` за $O(1)$.

`std::forward_list`

`std::forward_list<T>` - шаблонный класс односвязного списка

Ясно, что с односвязным списком можно делать те же операции, что и над двусвязным, за исключением `push_back`, `pop_back`.

А есть еще ограничения?

`std::forward_list`: `*_after`

Имея итератор на элемент двусвязного списка, можно свободно его удалить, или вставить перед ним элемент:

```
std::list<int> l{...};
l.insert(pos, 0);
l.erase(pos);
```

Для односвязного списка указателя на узел не достаточно для его удаления или вставки перед ним.

Можно лишь вставить или удалить элемент *после* него.

`std::forward_list`: `*_after`

```
std::forward_list<int> l{...};
std::forward_list<int> other{...};
```

- `insert_after`

```
l.insert_after(pos, 0);
```

- `erase_after`

```
l.erase_after(pos);
```

- `splice_after`

```
l.splice_after(pos, other);  // O(|other|)
l.splice_after(pos, other, before_element);  // O(1)
l.splice_after(pos, other, before_begin, end); // O(d(before_begin, end))
```

А еще у `std::forward_list` нет `size`.