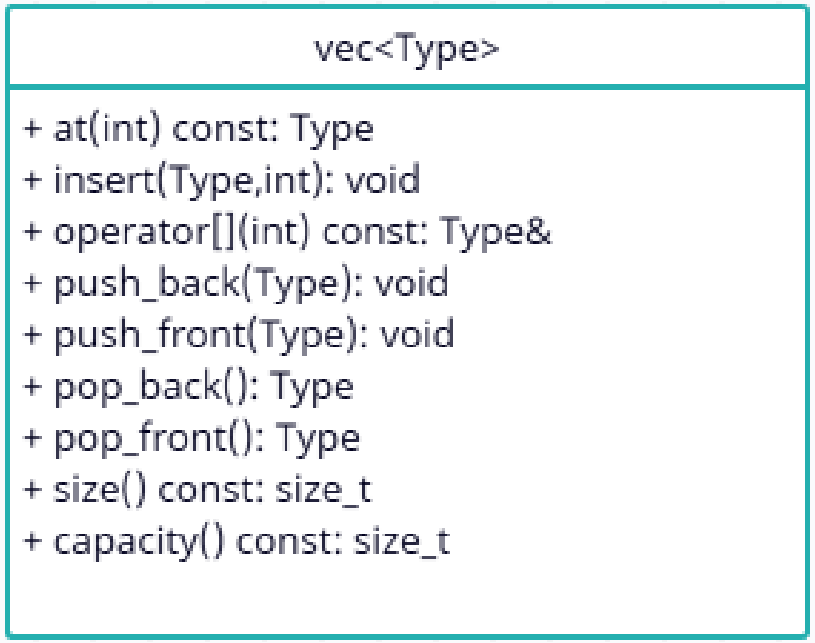
Динамический массив

Пишем собственный vector

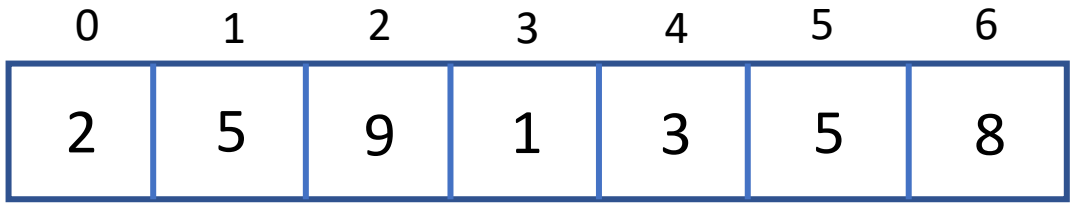
Попытка определить требования:

- Добавлять и удалять элементы в произвольные места коллекции
- Доступ к элементу по индексу
- Прозрачная адресация – у элемента можно взять адрес, к адресу элемента можно добавить 1 и получить следующий элемент и т.д.
- Детали реализации скрыты от пользователя



После разбора публичного интерфейса можно перейти к деталям реализации:

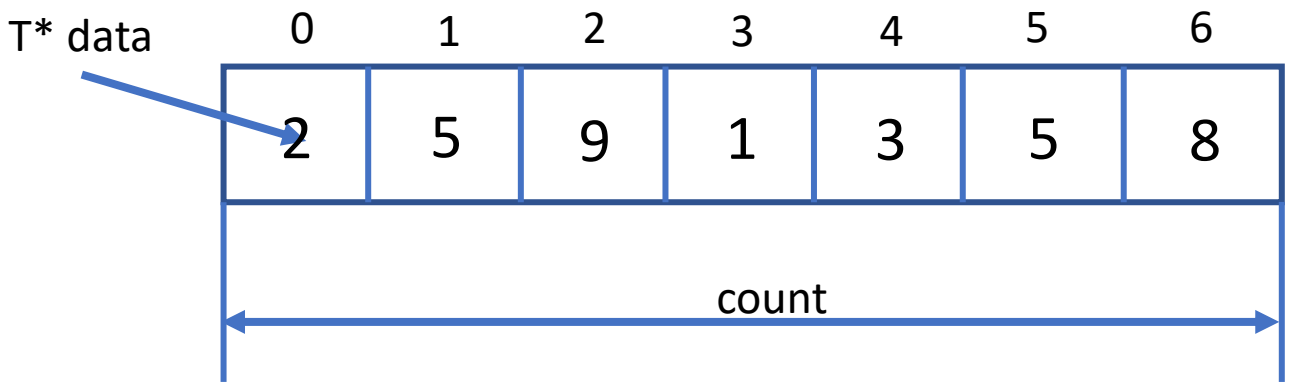
- Как именно внутри будет выглядеть коллекция?



Какие поля вы видите как члены класса vec?

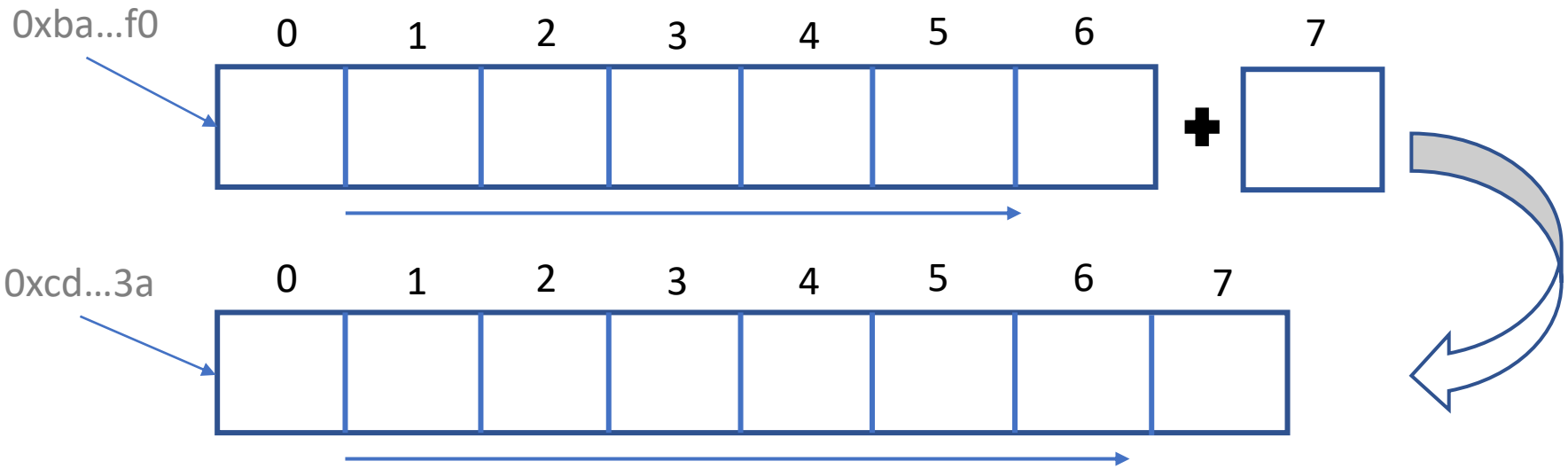
После разбора публичного интерфейса можно перейти к деталям реализации:

- Как именно внутри будет выглядеть коллекция?



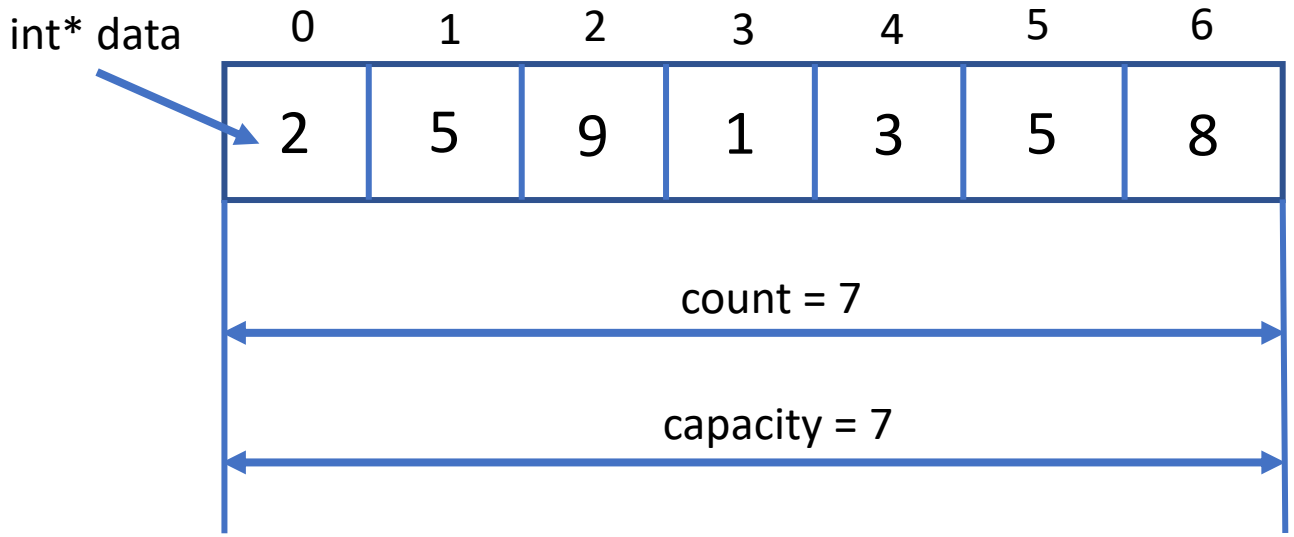
- T* data – указатель на начало массива
- int count – количество элементов в массиве

- Вспомним поведение динамического массива при добавлении нового элемента.



Будем ли мы при каждом добавлении или удалении элемента реаллоцировать коллекцию?

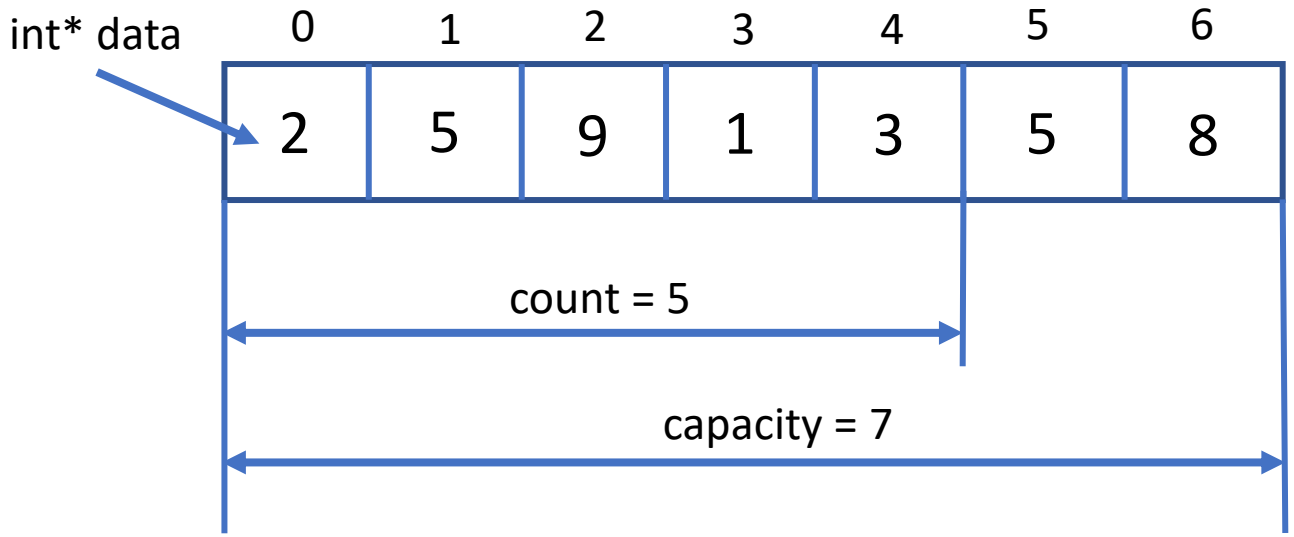
- Каждое изменение коллекции не обязательно должно сопровождаться реаллокацией.
- Нам ничего не мешает держать массив бОльшего размера, чем количество элементов.
- Поэтому нам нужна еще переменная для хранения количества выделенной памяти – capacity.



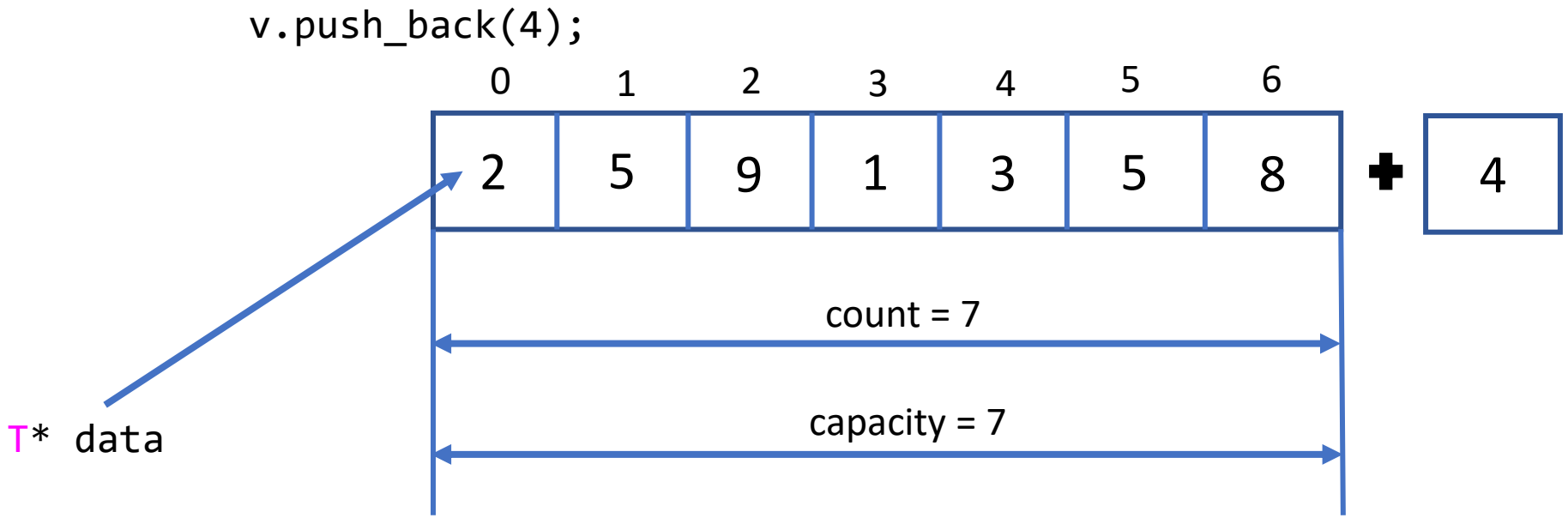

```
v.pop_back();
v.pop_back();
```

- Как выглядит коллекция теперь? Чему равны count и capacity?

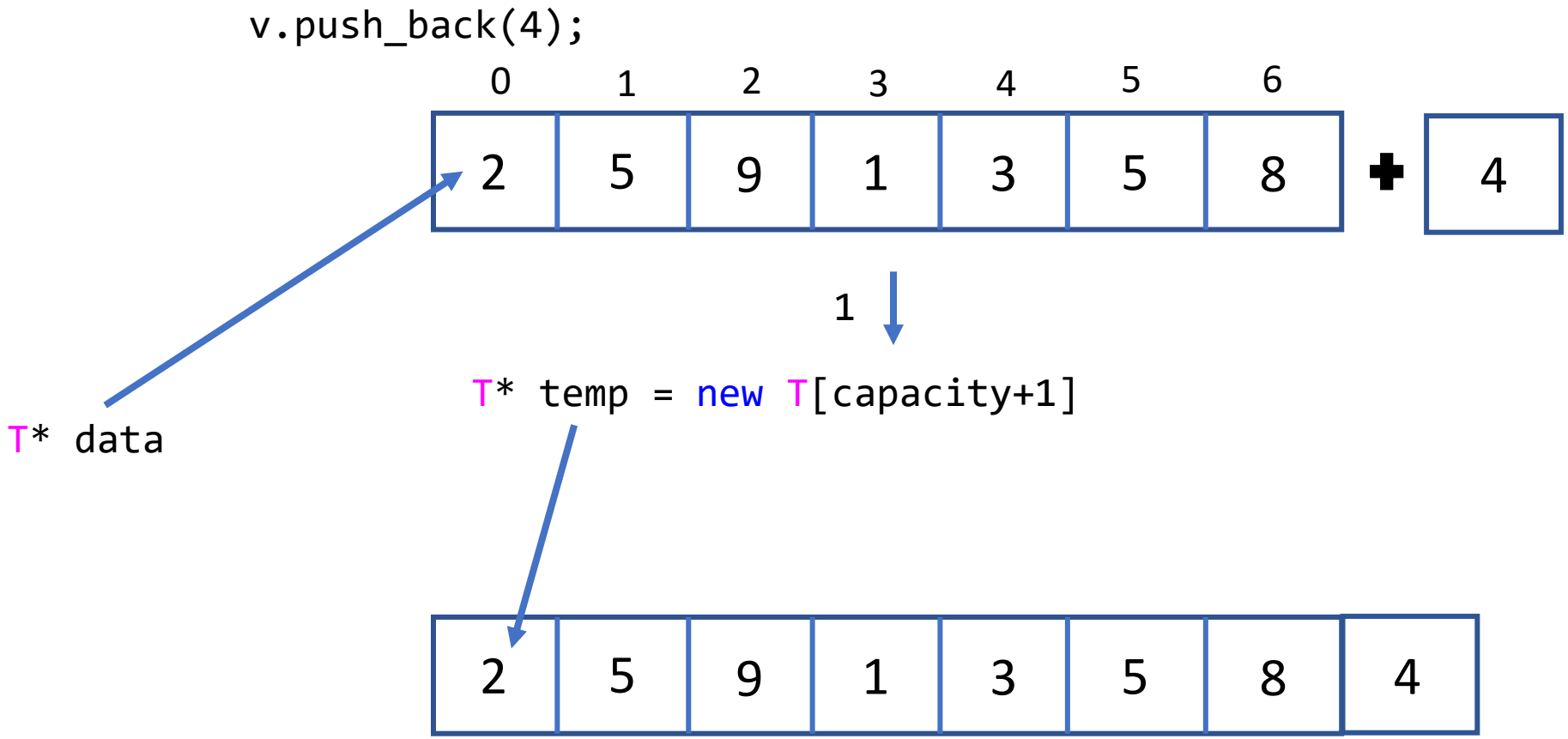
- Каждое изменение коллекции не обязательно должно сопровождаться реаллокацией.
- Нам ничего не мешает держать массив бОльшего размера, чем количество элементов.
- Поэтому нам нужна еще переменная для хранения количества выделенной памяти – capacity.



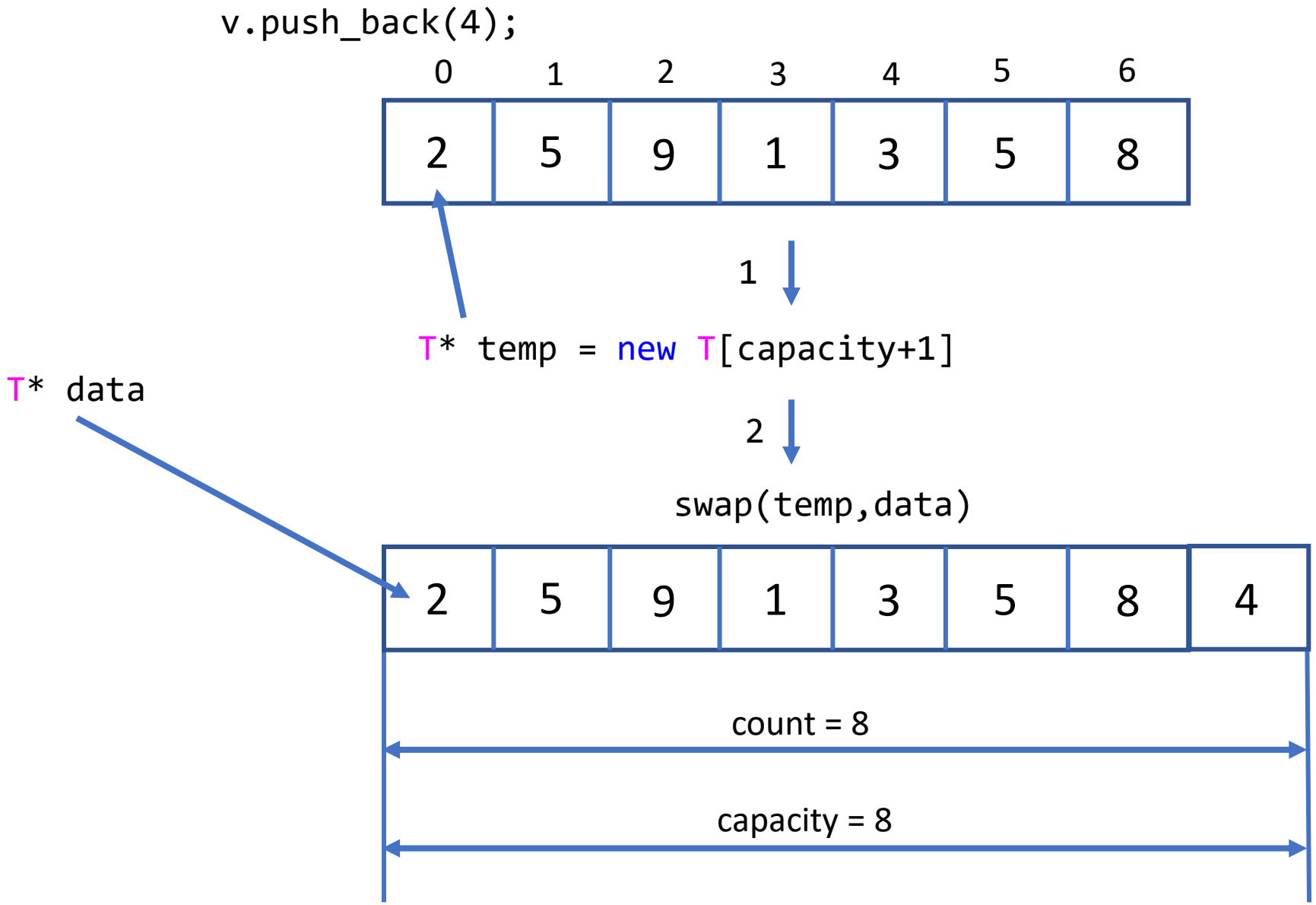
- Но если необходимо добавить элемент, а выделенное место кончилось?



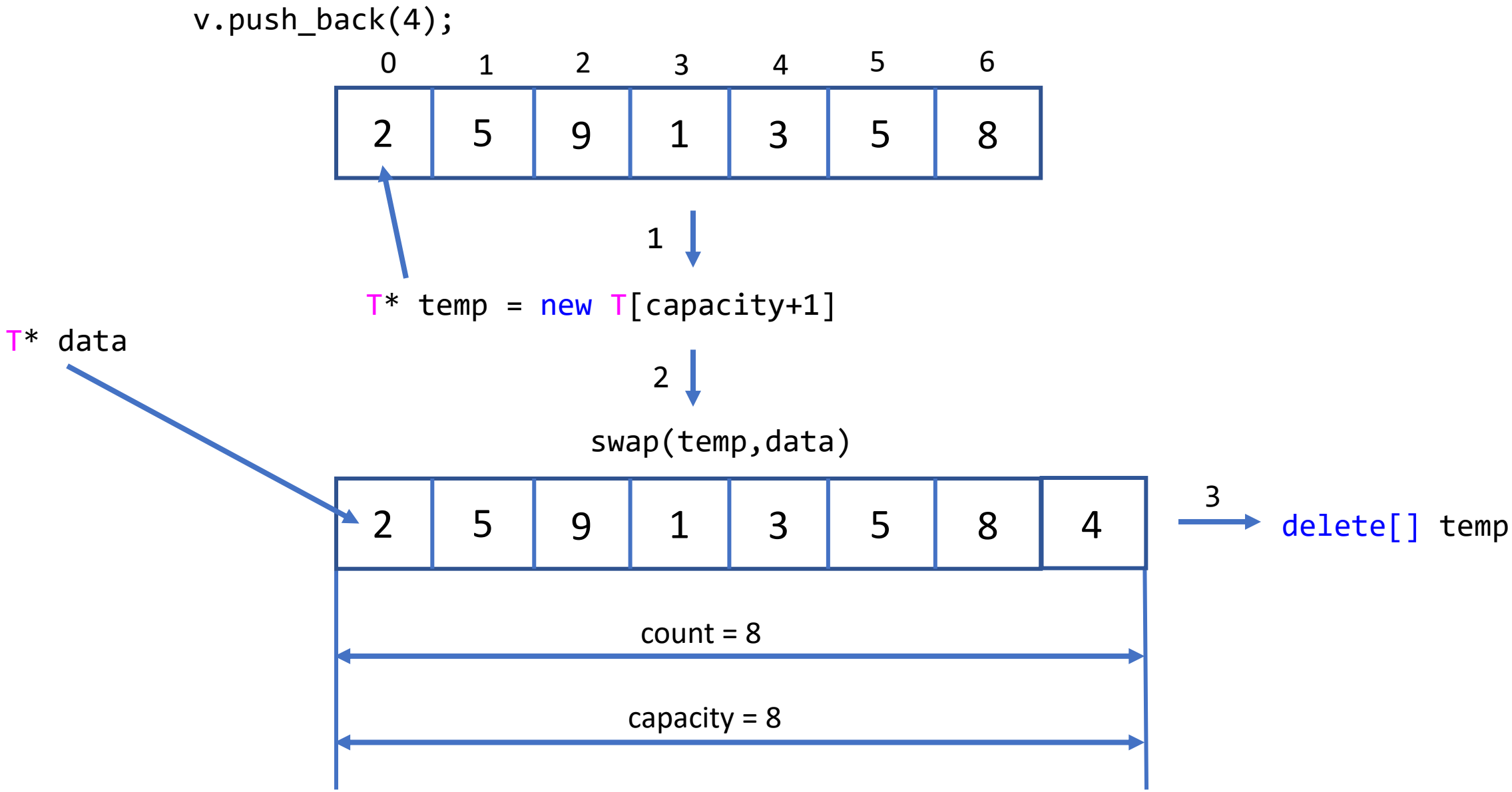
- Но если необходимо добавить элемент, а выделенное место кончилось?



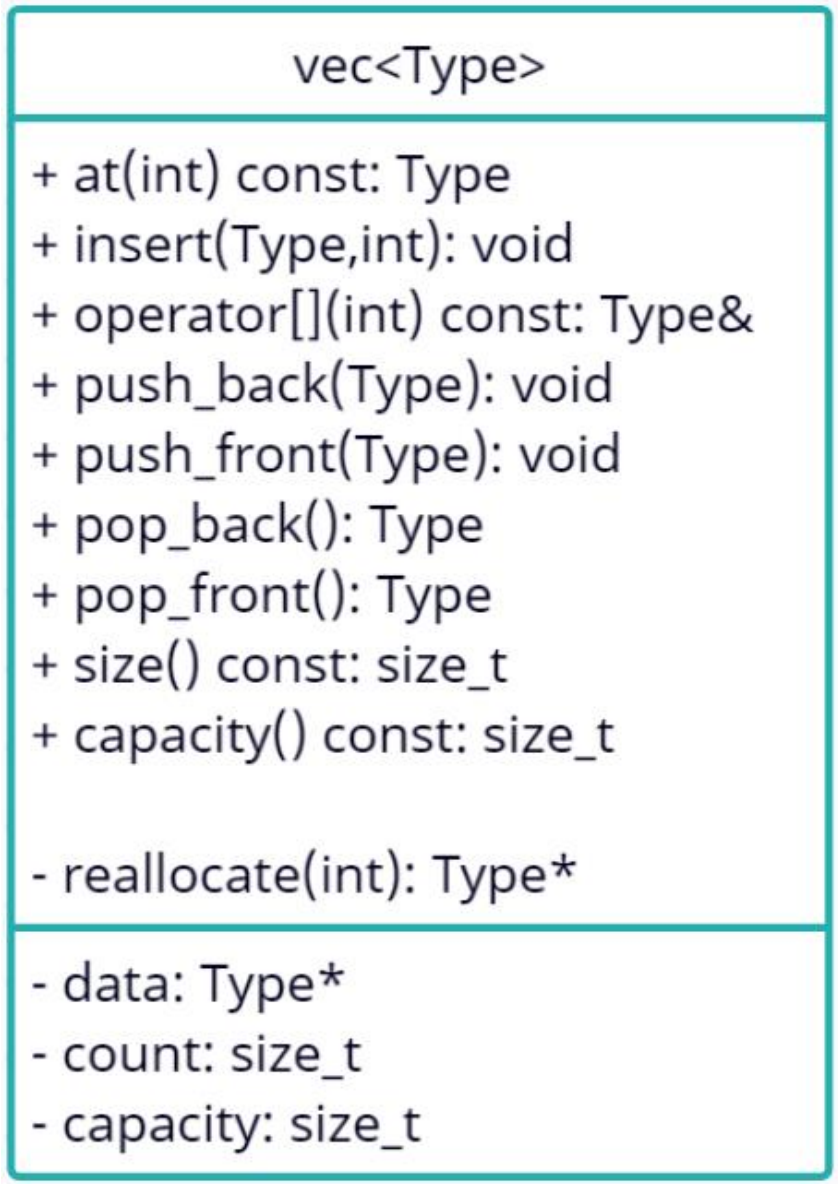
- Но если необходимо добавить элемент, а выделенное место кончилось?



- Но если необходимо добавить элемент, а выделенное место кончилось?



Итог:



- ✓ Добавление и удаление элементов
- ✓ Доступ к элементу по индексу
- ✓ Получение размера коллекции
- ✓ Сохранена прозрачная адресация
- ✓ Реализация скрыта от пользователя

Начнем с полей:

```
template <typename T> class vec
{
    T* m_data;                  // указатель на начало массива
    size_t m_data_size;         // текущее выделенное пространство
    size_t m_count;             // текущее кол-во элементов

    ...
};
```

Чем дополним?

Конструктор, деструктор, что-то еще?

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    vec()
    : m_data(nullptr), m_data_size(0), m_count(0)
    { }

    virtual ~vec() { if (m_data) delete[] m_data; }
};
```

Поскольку класс владеет ресурсом – реализуем правило пяти.

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    vec()
    : m_data(nullptr), m_data_size(0), m_count(0)
    { }

    vec(const vec& rhs) { ... }

    vec(vec&& rhs) { ... }

    vec& operator=(const vec& rhs) { ... }

    vec& operator=(vec&& rhs) { ... }

    virtual ~vec() { if (m_data) delete[] m_data; }
    ...
};
```

Добавим методы is_full, clear, size и capacity – они нам пригодятся

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    ...
    bool is_full() {                          }

    void clear() {              }

    size_t size() const {                 }

    size_t capacity() const {                     }
};
```

Добавим методы is_full, clear, size и capacity – они нам пригодятся

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    ...
    bool is_full() { return m_count == m_data_size; }

    void clear() { m_count = 0; }

    size_t size() const { return m_count; }

    size_t capacity() const { return m_data_size; }
};
```

Реализуем реаллокацию

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
private:
    void reallocate(int new_size) {









    }
};
```

Реализуем реаллокацию

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
private:
    void reallocate(int new_size) {
        if (!new_size) return;









    }
};
```

Реализуем реаллокацию

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
private:
    void reallocate(int new_size) {
        if (!new_size) return;

        T* tmp = new T[new_size];
        if (m_data) {




        }
        else



    }
};
```

Реализуем реаллокацию

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
private:
    void reallocate(int new_size) {
        if (!new_size) return;

        T* tmp = new T[new_size];
        if (m_data) {
            m_count = std::min<size_t>(m_count, new_size);
            std::copy(m_data, m_data + m_count, tmp);
            delete[] m_data;
        }
        else m_count = 0;



    }
};
```

Реализуем реаллокацию

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
private:
    void reallocate(int new_size) {
        if (!new_size) return;

        T* tmp = new T[new_size];
        if (m_data) {
            m_count = std::min<size_t>(m_count, new_size);
            std::copy(m_data, m_data + m_count, tmp);
            delete[] m_data;
        }
        else m_count = 0;

        m_data = tmp;
        m_data_size = new_size;
    }
};
```

Добавим оператор [] и at(int) для доступа к элементам

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    ...
    T& operator[](int index) const& {
        if (index >= m_count || index < 0)
                throw std::out_of_range("index out of range");

        return m_data[index];
    }

    T at(int index) const {
        return operator[](index);
    }
    ...
};
```

Добавление элемента вставкой

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    void insert(T&& value, int index) {









    }
...
};
```

Добавление элемента вставкой

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    void insert(T&& value, int index) {
        if (index > m_count || index < 0)
            throw std::out_of_range("index out of range");









    }
...
};
```

Добавление элемента вставкой

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    void insert(T&& value, int index) {
        if (index > m_count || index < 0)
            throw std::out_of_range("index out of range");

        if (is_full())
            reallocate(m_data_size+1);









    }
...
};
```

Добавление элемента вставкой

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    void insert(T&& value, int index) {
        if (index > m_count || index < 0)
            throw std::out_of_range("index out of range");

        if (is_full())
            reallocate(m_data_size+1);

        for (int i = m_count - 1; i >= index; --i) {
            m_data[i + 1] = m_data[i];
        }

        m_data[index] = std::forward<T>(value);
        m_count++;
    }
...
};
```

Добавление элемента вставкой

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    void insert(T&& value, int index) {
        if (index > m_count || index < 0)
            throw std::out_of_range("index out of range");

        if (is_full())
            reallocate(std::max<int>(m_data_size, 1) * 2);

        for (int i = m_count - 1; i >= index; --i) {
            m_data[i + 1] = m_data[i];
        }

        m_data[index] = std::forward<T>(value);
        m_count++;
    }
...
};
```

Добавление в конец и в начало

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    void push_back(T&& value)

    void push_front(T&& value)

    ...
};
```

Добавление в конец и в начало

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    void push_back(T&& value) {
        insert(std::forward<T>(value), m_count);
    }



    void push_front(T&& value) {
        insert(std::forward<T>(value), 0);
    }



    ...
};
```

Выборка элемента с конца

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_back() {



    }



...
};
```

Выборка элемента с конца

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_back() {
        if (!m_count)
            throw std::out_of_range("tryed pop element from empty collecton");


    }




...
};
```

Выборка элемента с конца

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_back() {
        if (!m_count)
            throw std::out_of_range("tryed pop element from empty collecton");

        return m_data[--m_count];
    }




...
};
```

Выборка элемента с начала

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_front() {









    }

...
};
```

Выборка элемента с начала

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_front() {
        if (!m_count)
            throw std::out_of_range("tryed pop element from empty collecton");









    }

...
};
```

Выборка элемента с начала

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_front() {
        if (!m_count)
            throw std::out_of_range("tryed pop element from empty collecton");

        T tmp = m_data[0];









    }

...
};
```

Выборка элемента с начала

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_front() {
        if (!m_count)
            throw std::out_of_range("tryed pop element from empty collecton");

        T tmp = m_data[0];
        for (int i = 1; i < m_count; ++i) {
            m_data[i - 1] = m_data[i];
        }



    }

...
};
```

Выборка элемента с начала

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T pop_front() {
        if (!m_count)
            throw std::out_of_range("tryed pop element from empty collecton");

        T tmp = m_data[0];
        for (int i = 1; i < m_count; ++i) {
            m_data[i - 1] = m_data[i];
        }

        m_count--;
        return tmp;
    }

...
};
```

И финальный штрих:

- На какой элемент будет указывать `ptr`?

```
int* ptr = &vec<int>{1,2,3}[0];
```

И финальный штрих:

- На какой элемент будет указывать `ptr`?

```
int* ptr = &vec<int>{1,2,3}[0];
```

- Аналогично и ссылки – они просто провиснут...

```
int& ref = vec<int>{1,2,3}[0];
```

И финальный штрих:

- На какой элемент будет указывать ptr?

```
int* ptr = &vec<int>{1,2,3}[0];
```

- Аналогично и ссылки – они просто провиснут...

```
int& ref = vec<int>{1,2,3}[0];
```

- Поэтому безжалостно запретим использовать operator[] для rvalue.

```
template <typename T> class vec
{
    T* m_data;
    size_t m_data_size;
    size_t m_count;
public:
    T& operator[](int) && = delete;
...
};
```

vec.cpp
godbolt.org