

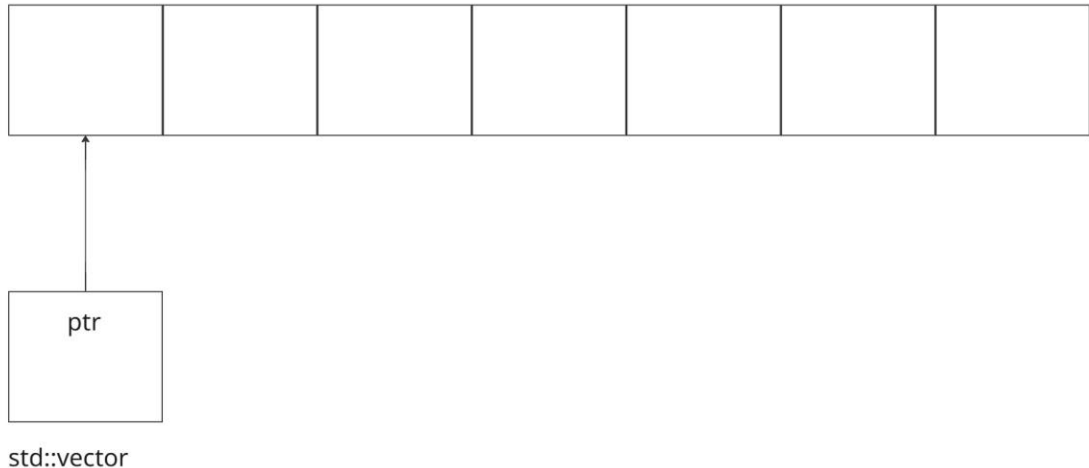
Принципы работы контейнера `std::vector` из библиотеки `<vector>`

Представление в памяти

При создании вектора выделяется блок памяти фиксированного размера, все элементы вектора хранятся в этом блоке непрерывно. Сам вектор содержит указатель на выделенный в куче буфер.

Схема представления вектора в памяти:

Выделенный в куче буфер:



Основные методы вектора для работы с памятью:

- `.size()` - это количество реальных объектов (т. е размер).
- `.capacity()` - это количество объектов, под которые зарезервирована память (т. е емкость).
- `.reserve(n)` - выделит память под `n` элементов (но `size` при этом будет равно 0).
- `.shrink_to_fit()` – уменьшает `capacity` до `size`.

Вставка

При вставке элемента он копируется в выделенную память, и `size` увеличивается. Мы можем продолжать вставлять элементы до тех пор, пока размер не станет равен ёмкости, что означает, что вектор заполнен. Поэтому пока размер меньше емкости временная сложность составляет $O(1)$ при вставке в конец (при других вставках должен учитываться сдвиг элементов), а когда он становится больше емкости, происходит перераспределение. При перераспределении весь блок копируется в новый блок большей емкости, что занимает $O(n)$.

Основные методы вставки в векторе:

- `.push_back()` - добавляет новый элемент в конец вектора.
- `.insert()` - вставляет новые элементы в заданную позицию в векторе.
- `.emplace()` - вставляет элементы в заданную позицию в векторе, создавая их на месте внутри вектора.

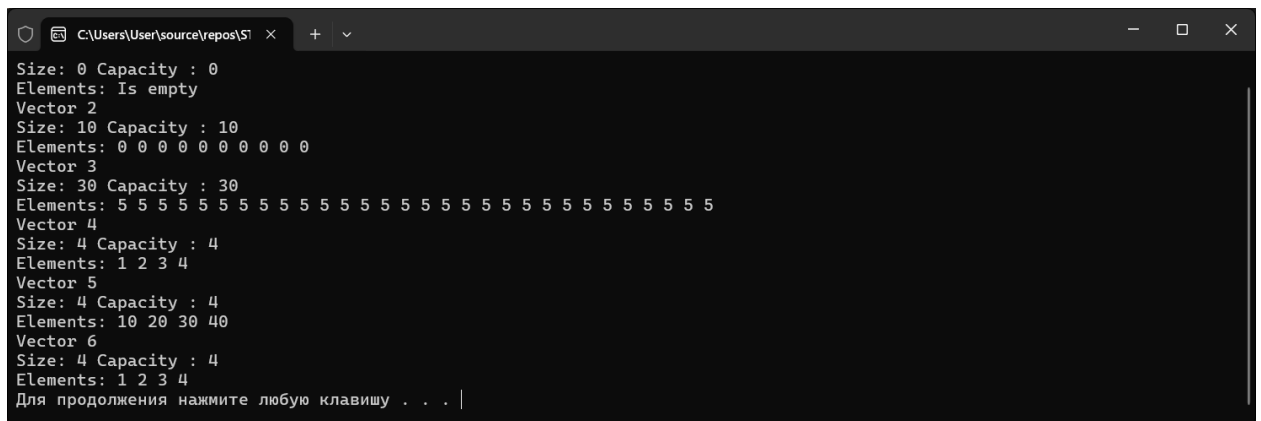
Удаление

Удаление элементов не приводит к перераспределению памяти. Удаляемый объект будет уничтожен, но память по-прежнему будет принадлежать вектору (так как `capacity` изменяться не будет, а `size` будет). Поэтому временная сложность удаления одного элемента с конца - это $O(1)$ (при других способах удаления учитывается сдвиг элементов).

Основные методы удаления в векторе:

- `.clear()` - удаляет все элементы из вектора, эффективно опустошая его.
- `.pop_back()` - удаляет последний элемент вектора.
- `.erase()` - удаляет элемент или диапазон элементов в векторе из заданных позиций.

Проверка информации из документации



```
C:\Users\User\source\repos\SI x + v
Size: 0 Capacity : 0
Elements: Is empty
Vector 2
Size: 10 Capacity : 10
Elements: 0 0 0 0 0 0 0 0 0 0
Vector 3
Size: 30 Capacity : 30
Elements: 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
Vector 4
Size: 4 Capacity : 4
Elements: 1 2 3 4
Vector 5
Size: 4 Capacity : 4
Elements: 10 20 30 40
Vector 6
Size: 4 Capacity : 4
Elements: 1 2 3 4
Для продолжения нажмите любую клавишу . . . |
```

Тест 1 - Инициализация

Приложение А: инициализация вектора

```
#include <iostream>
#include <vector>

void print_vector_info(int num, std::vector<int> &vec) {
    std::cout << "Vector " << num << std::endl;
    std::cout << "Size: " << vec.size() << " " << "Capacity : " << vec.capacity()
<< std::endl;
    std::cout << "Elements: ";
    if (vec.size() == 0) {
        std::cout << "Is empty";
    }

    for (int i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << ' ';
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> v1;
    std::vector<int> v2(10);
    std::vector<int> v3(30, 5);
    std::vector<int> v4 = { 1, 2, 3, 4 };
    int arr[] = { 10, 20, 30, 40 };
    std::vector<int> v5(arr, arr + 4);
    std::vector<int> v6(v4);

    print_vector_info(1,v1);
    print_vector_info(2,v2);
    print_vector_info(3,v3);
```

```

    print_vector_info(4, v4);
    print_vector_info(5, v5);
    print_vector_info(6, v6);
    system("pause");
    return 0;
}

```

Запуск приложения А показывает, что вектор содержит в себе базовые способы инициализации: пустая (v1), значением по умолчанию (v2, по умолчанию – ноль), заданным значением (v3), списком (v4), из массива (v5), копированием (v6). Также важно заметить, что при инициализации size всегда равен capacity, что доказывают строки с маленьким и большим количеством элементов. Причиной этого является то, что вектор стремится к оптимальному использованию памяти.

```

Initial: size = 0, capacity = 0
After push_back(0): size = 1, capacity = 1
After push_back(1): size = 2, capacity = 2
After push_back(2): size = 3, capacity = 3
After push_back(3): size = 4, capacity = 4
After push_back(4): size = 5, capacity = 6
After push_back(5): size = 6, capacity = 6
After push_back(6): size = 7, capacity = 9
After push_back(7): size = 8, capacity = 9
After push_back(8): size = 9, capacity = 9
After push_back(9): size = 10, capacity = 13
After push_back(10): size = 11, capacity = 13
After push_back(11): size = 12, capacity = 13
After push_back(12): size = 13, capacity = 13
After push_back(13): size = 14, capacity = 19
After push_back(14): size = 15, capacity = 19
After push_back(15): size = 16, capacity = 19
After push_back(16): size = 17, capacity = 19
After push_back(17): size = 18, capacity = 19
After push_back(18): size = 19, capacity = 19
After push_back(19): size = 20, capacity = 28
After push_back(20): size = 21, capacity = 28
After push_back(21): size = 22, capacity = 28
After push_back(22): size = 23, capacity = 28
After push_back(23): size = 24, capacity = 28
After push_back(24): size = 25, capacity = 28
After push_back(25): size = 26, capacity = 28

```

Тест 2 - Увеличение capacity, начало

```

After push_back(73): size = 74, capacity = 94
After push_back(74): size = 75, capacity = 94
After push_back(75): size = 76, capacity = 94
After push_back(76): size = 77, capacity = 94
After push_back(77): size = 78, capacity = 94
After push_back(78): size = 79, capacity = 94
After push_back(79): size = 80, capacity = 94
After push_back(80): size = 81, capacity = 94
After push_back(81): size = 82, capacity = 94
After push_back(82): size = 83, capacity = 94
After push_back(83): size = 84, capacity = 94
After push_back(84): size = 85, capacity = 94
After push_back(85): size = 86, capacity = 94
After push_back(86): size = 87, capacity = 94
After push_back(87): size = 88, capacity = 94
After push_back(88): size = 89, capacity = 94
After push_back(89): size = 90, capacity = 94
After push_back(90): size = 91, capacity = 94
After push_back(91): size = 92, capacity = 94
After push_back(92): size = 93, capacity = 94
After push_back(93): size = 94, capacity = 94
After push_back(94): size = 95, capacity = 141
After push_back(95): size = 96, capacity = 141
After push_back(96): size = 97, capacity = 141
After push_back(97): size = 98, capacity = 141
After push_back(98): size = 99, capacity = 141
After push_back(99): size = 100, capacity = 141

```

Тест 2 - Увеличение capacity, конец

Приложение В: увеличение capacity

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec;
    std::cout << "Initial: size = " << vec.size()
              << ", capacity = " << vec.capacity() << "\n";

    for (int i = 0; i < 100; ++i) {
        vec.push_back(i);
        std::cout << "After push_back(" << i << "): "
                  << "size = " << vec.size()
                  << ", capacity = " << vec.capacity() << "\n";
    }

    return 0;
}
```

Запуск приложения В показывает, что переменная capacity у вектора считается таким образом: $\text{old_capacity} * 1,5$ и округлить вниз. Исключением является только $\text{old_capacity} = 0$ (при этом $\text{size} = \text{capacity}$) и $\text{old_capacity} = 1$ (при этом идет округление не вниз, а вверх). Именно это дает нам понимание, что capacity работает нетривиально и не всегда будет больше size при перераспределении памяти. Это поведение важно запомнить, для дальнейшего понимания логики работы вектора. Приведу пример.



```
Vector 1
Size: 10 Capacity : 10
Elements: 0 0 0 0 0 0 0 0 0 0
Vector 1
Size: 15 Capacity : 15
Elements: 0 0 0 0 0 0 0 0 0 1 2 3 4 5
Для продолжения нажмите любую клавишу . . . |
```

Тест 2.1 - Пример

Приложение В.1: пример

```
#include <iostream>
#include <vector>

void print_vector_info(int num, std::vector<int> &vec) {
    std::cout << "Vector " << num << std::endl;
    std::cout << "Size: " << vec.size() << " " << "Capacity : " << vec.capacity()
              << std::endl;
    std::cout << "Elements: ";
    if (vec.size() == 0) {
        std::cout << "Is empty";
    }

    for (int i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << ' ';
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> v1(10);
    print_vector_info(1, v1);
    for (int i = 0; i < 5; i++) { v1.push_back(1 * (i + 1)); }
    print_vector_info(1, v1);
    system("pause");
    return 0;
}
```

Запуск приложения В1, показывает, что при добавлении 5 элементов в вектор $size = capacity = 15$, кажется, что вектор сработал неправильно, но на самом деле все сделано так, как нужно, ведь наш $capacity$ зависит не от нового $size$, а от так называемого $old_capacity$, то есть $capacity = 10 * 1,5 = 15$.

```

Size: 12 Capacity : 12
Elements: 0 (0000020834644250) 0 (0000020834644254) 0 (0000020834644258) 0 (000002083464425C) 0 (0000020834644260) 0 (0000020834644264) 0 (0000020834644268) 0 (000002083464426C) 0 (0000020834644270) 0 (0000020834644274) 0 (0000020834644278) 0 (000002083464427C)
Vector 1
Size: 19 Capacity : 27
Elements: 0 (000002083463C380) 0 (000002083463C384) 0 (000002083463C388) 0 (000002083463C38C) 0 (000002083463C390) 0 (000002083463C394) 0 (000002083463C398) 0 (000002083463C39C) 0 (000002083463C3A0) 0 (000002083463C3A4) 0 (000002083463C3A8) 0 (000002083463C3AC) 1 (000002083463C3B0) 2 (000002083463C3B4) 3 (000002083463C3B8) 4 (000002083463C3BC) 5 (000002083463C3C0) 6 (000002083463C3C4) 7 (000002083463C3C8)
Vector 1
Size: 24 Capacity : 27
Elements: 0 (000002083463C380) 0 (000002083463C384) 0 (000002083463C388) 0 (000002083463C38C) 0 (000002083463C390) 0 (000002083463C394) 0 (000002083463C398) 0 (000002083463C39C) 0 (000002083463C3A0) 0 (000002083463C3A4) 0 (000002083463C3A8) 0 (000002083463C3AC) 1 (000002083463C3B0) 2 (000002083463C3B4) 3 (000002083463C3B8) 4 (000002083463C3BC) 5 (000002083463C3C0) 6 (000002083463C3C4) 7 (000002083463C3C8) 8 (000002083463C3CC) 9 (000002083463C3D0) 10 (000002083463C3D4) 11 (000002083463C3D8) 12 (000002083463C3DC)
Vector 1
Size: 22 Capacity : 27
Elements: 0 (000002083463C380) 0 (000002083463C384) 0 (000002083463C388) 0 (000002083463C38C) 0 (000002083463C390) 0 (000002083463C394) 0 (000002083463C398) 0 (000002083463C39C) 0 (000002083463C3A0) 0 (000002083463C3A4) 0 (000002083463C3A8) 0 (000002083463C3AC) 1 (000002083463C3B0) 2 (000002083463C3B4) 3 (000002083463C3B8) 4 (000002083463C3BC) 5 (000002083463C3C0) 6 (000002083463C3C4) 7 (000002083463C3C8) 8 (000002083463C3CC) 9 (000002083463C3D0) 10 (000002083463C3D4)
Для продолжения нажмите любую клавишу . . .

```

Тест 3 - Вставка и удаление

Приложение С: вставка и удаление

```

#include <iostream>
#include <vector>

void print_vector_info(int num, std::vector<int> &vec) {
    std::cout << "Vector " << num << std::endl;
    std::cout << "Size: " << vec.size() << " " << "Capacity : " << vec.capacity()
<< std::endl;
    std::cout << "Elements: ";
    if (vec.size() == 0) {
        std::cout << "Is empty";
    }

    for (int i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << ' ' << "(" << &vec[i] << ") ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> v1(12);
    print_vector_info(1, v1);
    for (int i = 0; i < 7; i++) { v1.push_back(1 * (i + 1)); }
    print_vector_info(1, v1);
    for (int i = 0; i < 5; i++) { v1.push_back(1 * (i + 1) + 7); }
    print_vector_info(1, v1);
    for (int i = 0; i < 2; i++) { v1.pop_back(); }
    print_vector_info(1, v1);
    system("pause");
    return 0;
}

```

Запуск приложения С показывает, что в начале (при инициализации) $size = capacity$, поэтому при дальнейшей вставке произошло перераспределение, и адреса у элементов вектора изменились, это можно увидеть по первому же элементу (подчеркнут красным на картинке). То, что при любой вставке не происходит перераспределение доказывает вторая вставка: адреса у элементов не изменились из-за того, что $size$ не превзошло $capacity$.

(одинаковые адреса подчеркнуты синим у одного и того же элемента). При удалении наше `capacity` не изменилось, последние два элемента удалились и перераспределение не произошло (адрес также подчеркнут синим), что соответствует логике работы вектора.

```
Vector 1
Size: 5 Capacity : 5
Elements: 2 (000001D24B085C20) 2 (000001D24B085C24) 2 (000001D24B085C28) 2 (000001D24B085C2C) 2 (000001D24B085C30)
Vector 1
Size: 6 Capacity : 7
Elements: 2 (000001D24B0835B0) 2 (000001D24B0835B4) 2 (000001D24B0835B8) 2 (000001D24B0835BC) 2 (000001D24B0835C0) 1 (000001D24B0835C4)
Vector 2
Size: 6 Capacity : 6
Elements: 2 (000001D24B083370) 2 (000001D24B083374) 2 (000001D24B083378) 2 (000001D24B08337C) 2 (000001D24B083380) 1 (000001D24B083384)
Для продолжения нажмите любую клавишу . . .
```

Тест 4 - Конструктор копирования

Приложение D: конструктор копирования

```
#include <iostream>
#include <vector>

void print_vector_info(int num, std::vector<int> &vec) {
    std::cout << "Vector " << num << std::endl;
    std::cout << "Size: " << vec.size() << " " << "Capacity : " << vec.capacity()
<< std::endl;
    std::cout << "Elements: ";
    if (vec.size() == 0) {
        std::cout << "Is empty";
    }

    for (int i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << ' ' << "(" << &vec[i] << ") ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> v1(5, 2);
    print_vector_info(1, v1);
    for (int i = 0; i < 1; i++) { v1.push_back(1 * (i + 1)); }
    print_vector_info(1, v1);
    std::vector<int> v2(v1);
    print_vector_info(2, v2);
    system("pause");
    return 0;
}
```

Запуск приложения D показывает, что при использовании конструктора копирования значения элементов нового вектора соответствуют значениям старого, меняются адреса (выделено синим), так как выделяется другой блок памяти, но наше новое `capacity` не равно старому (обведено красным), почему так происходит? На это есть достаточно логичный ответ, использование конструктора копирования является по своей сути инициализацией, а как мы знаем при инициализации вектор старается оптимально использовать память, поэтому и не выделяет лишнего места для хранения элементов.

```
C:\Users\User\source\repos\SI x + v
[Before insert]
Vector 1
Size: 16 Capacity : 22
Elements: 1 (00000292BA7AC410) 2 (00000292BA7AC414) 3 (00000292BA7AC418) 4 (00000292BA7AC41C) 5 (00000292BA7AC420) 6 (00000292BA7AC424) 7 (00000292BA7AC428) 8 (00000292BA7AC42C) 9 (00000292BA7AC430) 10 (00000292BA7AC434) 11 (00000292BA7AC438) 12 (00000292BA7AC43C) 13 (00000292BA7AC440) 14 (00000292BA7AC444) 15 (00000292BA7AC448) 16 (00000292BA7AC44C)
[After insert at position 2]
Vector 1
Size: 17 Capacity : 22
Elements: 1 (00000292BA7AC410) 2 (00000292BA7AC414) 111 (00000292BA7AC418) 3 (00000292BA7AC41C) 4 (00000292BA7AC420) 5 (00000292BA7AC424) 6 (00000292BA7AC428) 7 (00000292BA7AC42C) 8 (00000292BA7AC430) 9 (00000292BA7AC434) 10 (00000292BA7AC438) 11 (00000292BA7AC43C) 12 (00000292BA7AC440) 13 (00000292BA7AC444) 14 (00000292BA7AC448) 15 (00000292BA7AC44C) 16 (00000292BA7AC450)

[Before erase]
Vector 2
Size: 16 Capacity : 22
Elements: 1 (00000292BA7A95F0) 2 (00000292BA7A95F4) 3 (00000292BA7A95F8) 4 (00000292BA7A95FC) 5 (00000292BA7A9600) 6 (00000292BA7A9604) 7 (00000292BA7A9608) 8 (00000292BA7A960C) 9 (00000292BA7A9610) 10 (00000292BA7A9614) 11 (00000292BA7A9618) 12 (00000292BA7A961C) 13 (00000292BA7A9620) 14 (00000292BA7A9624) 15 (00000292BA7A9628) 16 (00000292BA7A962C)
[After erase at position 1]
Vector 2
Size: 15 Capacity : 22
Elements: 1 (00000292BA7A95F0) 3 (00000292BA7A95F4) 4 (00000292BA7A95F8) 5 (00000292BA7A95FC) 6 (00000292BA7A9600) 7 (00000292BA7A9604) 8 (00000292BA7A9608) 9 (00000292BA7A960C) 10 (00000292BA7A9610) 11 (00000292BA7A9614) 12 (00000292BA7A9618) 13 (00000292BA7A961C) 14 (00000292BA7A9620) 15 (00000292BA7A9624) 16 (00000292BA7A9628)
Для продолжения нажмите любую клавишу . . . |
```

Рис. 5 - Методы insert() и erase()

Приложение Е: методы insert() и erase()

```
#include <iostream>
#include <vector>

void print_vector_info(int num, std::vector<int> &vec) {
    std::cout << "Vector " << num << std::endl;
    std::cout << "Size: " << vec.size() << " " << "Capacity : " << vec.capacity()
    << std::endl;
    std::cout << "Elements: ";
    if (vec.size() == 0) {
        std::cout << "Is empty";
    }

    for (int i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << ' ' << "(" << &vec[i] << ") ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> v1 = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 };
    std::vector<int> v2 = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 };
    for (int i = 0; i < 1; i++) { v1.push_back((i + 1)+15); }
    for (int i = 0; i < 1; i++) { v2.push_back((i + 1) + 15); }
    std::cout << "[Before insert]\n";
    print_vector_info(1, v1);
    v1.insert(v1.begin() + 2, 111);
    std::cout << "[After insert at position 2]\n";
    print_vector_info(1, v1);
    std::cout << std::endl;
    std::cout << "[Before erase]\n";
    print_vector_info(2, v2);
    v2.erase(v2.begin() + 1);
    std::cout << "[After erase at position 1]\n";
    print_vector_info(2, v2);
    system("pause");
    return 0;
}
```

Запуск приложения Е, показывает, что при использовании метода insert элемент (подчеркнут зеленым на картинке) вставляется на указанную позицию, а все остальные,

которые выше его по индексу, сдвигаются вправо на 1, это мы можем заметить по адресам элементов (подчеркнуты красным и желтым). С удалением работает почти также: элемент (выделен синим) удаляется с заданной позиции, а все остальные, которые выше его по индексу сдвигаются влево на 1, смотрим на адреса элементов (подчеркнуто фиолетовым). При большом количестве элементов, insert и erase будут работать аналогично, вставка или удаления с конца этими методами не будут передвигать элементы.

Сравнение с TVector

1. Управление памятью и удаление элементов

TVector:

- Использует систему пометок состояний элементов (empty, busy, deleted).
- При удалении элемента он помечается как deleted, но физически не удаляется из памяти до перераспределения.
- Перераспределение памяти происходит только при превышении порога удалённых элементов (например, 15%).

std::vector:

- Удаляет элементы физически (сразу же после удаления), сдвигая оставшиеся элементы.
- Нет понятия "пометок" или перераспределения после достигнутого числа удаленных элементов.

Вывод: наш класс TVector имеет преимущество в производительности, так как очистит массив от удаленных элементов и сдвинет остальные всего 1 раз, при достигнутом лимите удаленных элементов, когда std::vector сделает это n раз (n – количество проведенных удалений методами).

2. Дополнительные функции

TVector:

- Поддерживает функции для работы с состояниями вектора (is_full(), is_empty()).
- Реализация многих методов зависит от "пометок" (empty, busy, deleted).
- Реализует алгоритмы Фишера-Йейтса (перемешивание) и быстрой сортировки (сортировка Хоара).

std::vector:

- Не включает встроенные алгоритмы сортировки или перемешивания (эти операции выполняются через <algorithm>):.

Вывод: у TVector чуть больше функций чем у стандартного вектора, логикой он почти не отличается, но реализацию имеет совсем иную, что доказывают так называемые "пометки".

3. Совместимость с STL

TVector:

- Несовместим с алгоритмами STL (например, `std::sort`, `std::find`).
- Требует написания собственных аналогов (например, сортировки Хоара).

std::vector:

- Полностью интегрирован с STL: работает с `std::sort`, `std::copy`, `std::accumulate` и другими алгоритмами.

4. Сложность операций

TVector:

- Вставка в начало или середину: $O(n)$ (из-за сдвигов).
- Вставка в конец: $O(1)$.
- Вставка с перевыделением памяти (из-за вместимости): $O(n)$
- Удаление: $O(1)$ или $O(n)$ с перевыделением памяти (из-за превышения количества удаленных).

std::vector:

- Вставка в начало или середину: $O(n)$ (из-за сдвигов).
- Вставка в конец: $O(1)$.
- Вставка с перевыделением памяти (из-за вместимости): $O(n)$.
- Удаление: $O(1)$, если в конец и $O(n)$ всегда, если в другие места (учет сдвига).

Вывод по сравнению

Хотя `TVector` и `std::vector` решают схожие задачи (динамический массив с автоматическим управлением памятью), их реализация и назначение существенно различаются. `TVector` — это специализированный учебный/экспериментальный контейнер, который демонстрирует альтернативный подход к управлению памятью. Он уступает `std::vector` в универсальности, но полезен для глубокого понимания работы динамических массивов и нестандартных оптимизаций. Именно поэтому в индивидуальном проекте будем использовать `Tvector`, для точного отслеживания состояний элементов и лучшего понимания действий происходящих с памятью.

