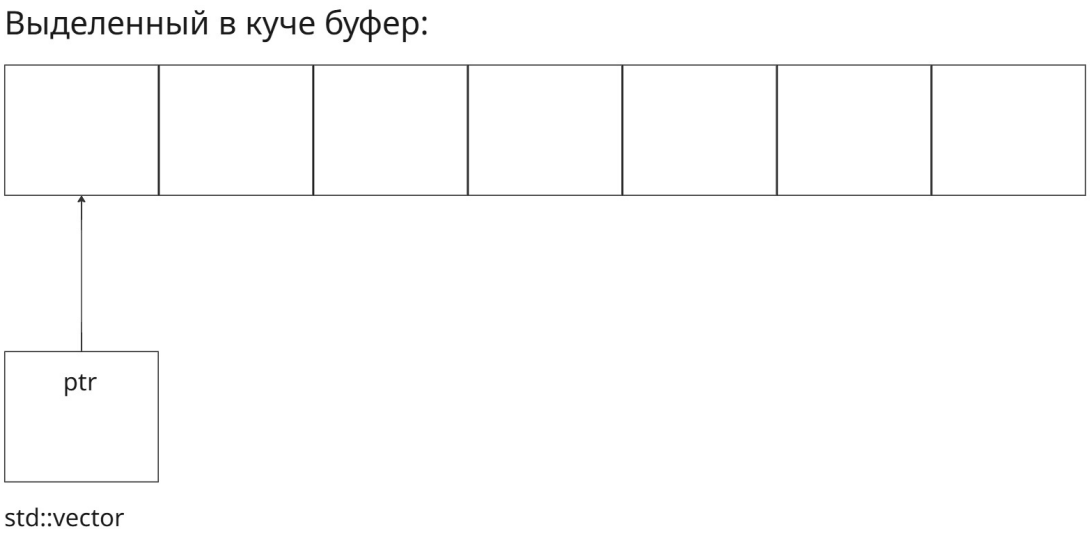
Принципы работы контейнера std::vector из библиотеки <vector>

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:



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 равен 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

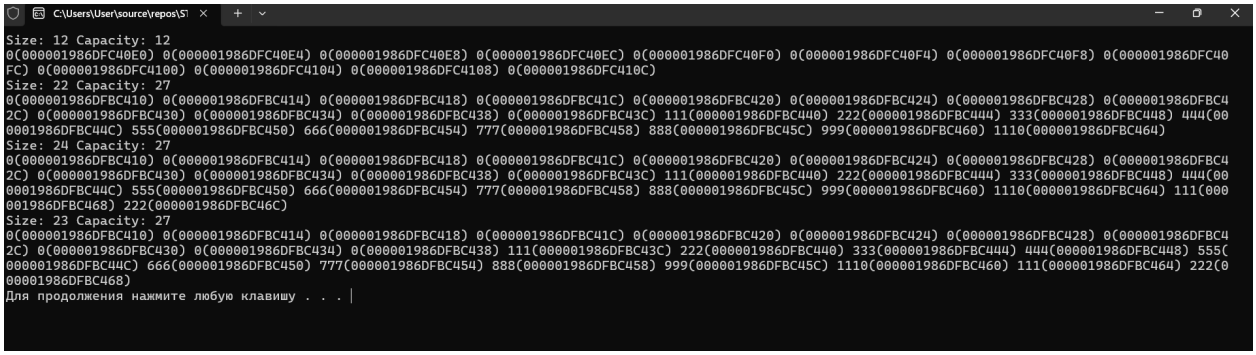


Рис.1 - Запуск Тестовой Программы

Приложение А: проведение эксперимента

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

void print_vector_info(std::vector<int> &vec) {
    std::cout << "Size: " << vec.size() << " " << "Capacity: " << vec.capacity() << 
std::endl;

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

}

int main() {
    std::vector<int> vec(12);
    print_vector_info(vec);
    for (int i = 0; i < 10; i++) { vec.push_back(111 * (i + 1)); }
    print_vector_info(vec);
    for (int i = 0; i < 2; i++) { vec.push_back(111 * (i + 1)); }
    print_vector_info(vec);
    vec.erase(vec.begin() + 4);
    print_vector_info(vec);
    system("pause");
    return 0;
}
```

Результаты проверки

Запуск тестовой программы показал нам ожидаемые результаты. Важно заметить, что в начале (при инициализации) size = capacity, поэтому при любой дальнейшей вставке произойдет перераспределение, и адреса у элементов вектора изменятся. То, что при любой вставке не происходит перераспределение доказывает вторая вставка (адреса у элементов не изменились из-за маленького количества вставляемых). Удаление также сработало правильно, удалив элемент и сдвинув все остальные, что можно посмотреть по адресам. Также интересно изменение capacity с увеличением количества элементов: до 5 элементов size = capacity, далее оно всегда превосходит size в 1,5 – 2 раза. Вывод: реализованные функции и работа std::vector из библиотеки <vector> соответствуют документации.

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 удаленных элементов).

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 из презентации).

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.