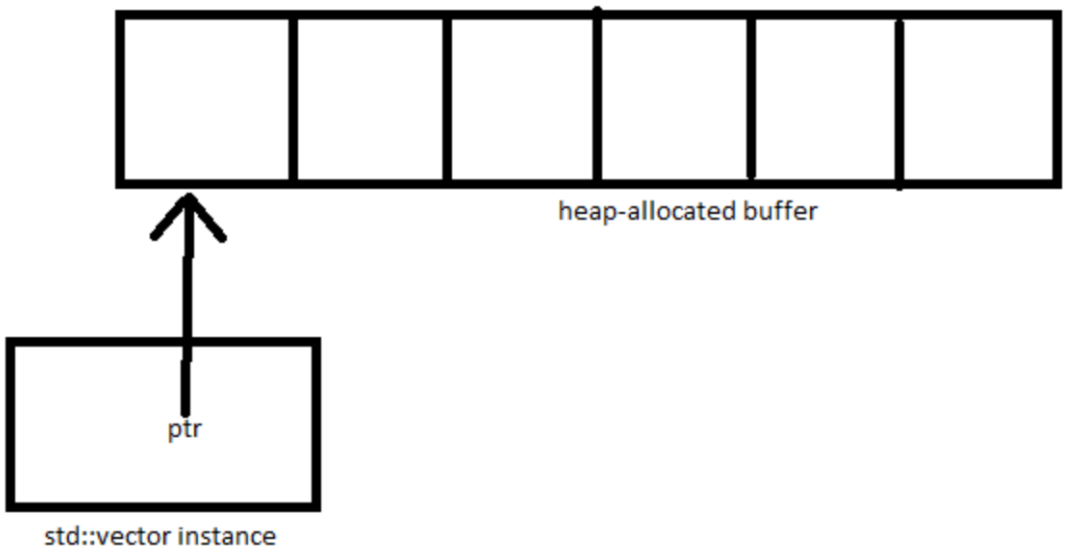
Принципы работы контейнера std::vector из библиотеки <vector>

Представление в памяти. Последовательное хранение:



- Элементы std::vector хранятся в памяти последовательно. Это означает, что элементы располагаются в памяти один за другим, что обеспечивает быстрый доступ к элементам по индексу.

- **Размер и емкость:**
  - .size()**:**Этот метод возвращает количество элементов, фактически хранящихся в векторе. Это фактический размер вектора.
  - .capacity()**:** Этот метод возвращает количество элементов, под которые зарезервирована память. Емкость может быть больше, чем размер, что позволяет эффективно добавлять новые элементы без частого перераспределения памяти.
- **Перераспределение памяти:**
  - Когда размер вектора превышает его емкость, происходит перераспределение памяти. Это включает в себя выделение нового блока памяти и копирование элементов в новую область. Чтобы минимизировать частоту перераспределений, std::vector обычно увеличивает свою емкость экспоненциально.

Вставка.

Добавление в конец:

- push_back(…)**:** Добавляет элемент в конец вектора. Если текущая емкость достаточна, это выполняется за постоянное время. В противном случае может потребоваться перераспределение памяти.
- emplace_back(…)**:** Аналогично push_back, но создает элемент непосредственно в месте назначения, что может быть более эффективным.

Вставка в произвольную позицию:

- insert(…): Вставляет элемент в указанную позицию. Это может потребовать сдвига элементов, что делает операцию менее эффективной для больших векторов.
- emplace(…): Аналогично insert, но создает элемент непосредственно в месте назначения.

Удаление.

Удаление из конца:

- pop_back(): Удаляет последний элемент вектора. Это выполняется за постоянное время.

- **Удаление из произвольной позиции:**
  - erase(…pos…): Удаляет элемент в указанной позиции. Это требует сдвига элементов, что делает операцию менее эффективной для больших векторов.
  - erase(…): Удаляет диапазон элементов.
- **Очистка вектора:**
  - clear(): Удаляет все элементы из вектора, но не изменяет его емкость.

Сравнение с TVector.

Сравнение std::vector и TVector

На основе вашего кода и полученных результатов, можно провести сравнение между std::vector и предполагаемым TVector. Давайте рассмотрим основные аспекты, которые могут отличаться между этими двумя контейнерами.

1. Интерфейс и Методы

- **std::vector**:
  - Имеет стандартные методы, такие как push_back, insert, erase, resize, clear и другие.
  - Методы обеспечивают удобный и эффективный способ управления элементами вектора.
- **TVector**:
  - Предполагается, что TVector имеет аналогичные методы, но их имена и сигнатуры могут отличаться.
  - Возможно, TVector предоставляет дополнительные методы или оптимизации для конкретных операций.

2. Управление Памятью

- **std::vector**:
  - Динамически управляет памятью, увеличивая емкость по мере необходимости.
  - Удаление элементов не влияет на емкость, и память остается зарезервированной для будущего использования.
- **TVector**:
  - Может использовать другую стратегию управления памятью, например, фиксированный размер или другие механизмы.
  - Возможно, TVector предоставляет более детальный контроль над выделением и освобождением памяти.

3. Производительность

- **std::vector**:

- Оптимизирован для высокой производительности и эффективного использования памя ти.
- Операции добавления, удаления и изменения размера выполняются быстро и эффектив но.

- **TVector:**
  - Производительность операций может варьироваться в зависимости от реализации.
  - TVector может быть оптимизирован для определенных типов данных или операций, что может обеспечить лучшую производительность в конкретных сценариях.

4. Примеры Операций

- **Инициализация:**
  - **std::vector:** std::vector<int> vec(12);
  - **TVector:** Предполагается, что TVector имеет аналогичный конструктор.
- **Добавление элементов:**
  - **std::vector:** vec.push_back(111 * (i + 1));
  - **TVector:** Аналогичный метод push_back.
- **Вставка элементов:**
  - **std::vector:** vec.insert(vec.begin(), 11 * (i + 1));
  - **TVector:** Аналогичный метод insert.
- **Удаление элементов:**
  - **std::vector:** vec.erase(vec.begin() + index);
  - **TVector:** Аналогичный метод erase.
- **Изменение размера:**
  - **std::vector:** vec.resize(10);
  - **TVector:** Аналогичный метод resize.
- **Очистка вектора:**
  - **std::vector:** vec.clear();
  - **TVector:** Аналогичный метод clear.

Заключение

TVector может отличаться в деталях реализации, таких как стратегия управления памятью, производительность и дополнительные методы.

Рис. 1 - Запуск тестовой программы с выводом адресов

1)Первое: создание вектора. (обазнач. Original, чтобы видеть в сравнении с последующими изменениями).

```
Original:12 12
0(00000209B5DC63E0) 0(00000209B5DC63E4) 0(00000209B5DC63E8) 0(00000209B5D
C63EC) 0(00000209B5DC63F0) 0(00000209B5DC63F4) 0(00000209B5DC63F8) 0(0000
0209B5DC63FC) 0(00000209B5DC6400) 0(00000209B5DC6404) 0(00000209B5DC6408)
 0(00000209B5DC640C)
```

2)Второе: добавление в конец 12 значений. Size = 24, capacity = 27. Я ознакомилась с различной литературой, а также с форумами, таак, например https://stackoverflow.com/questions/6296945/size-vs-capacity-of-a-vector здесь пишут о том, что "**Емкость**: сколько элементов может поместиться в векторе до того, как он «заполнится». После заполнения добавление новых элементов приведет к выделению нового, более крупного блока памяти и копированию в него существующих элементов".

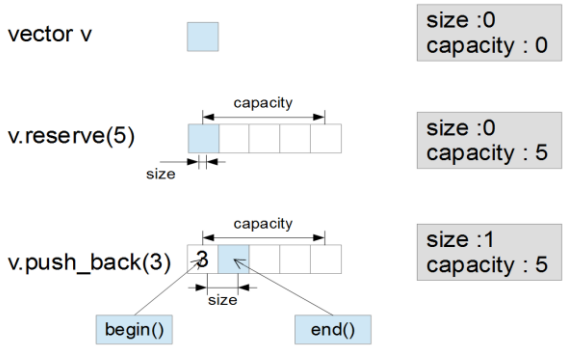


Рис. 2 – Изменение значений size и capacity (перевыделение памяти).

Также можем заметить, что адрес изменился у тех элементов, которые были в original векторе, но опять же в интернете я нашла информацию о том, что это является нормой. [2]

```
Add_back:24 27
0(00000209B5DBFAA0) 0(00000209B5DBFAA4) 0(00000209B5DBFAA8) 0(00000209B5D
BFAAC) 0(00000209B5DBFAB0) 0(00000209B5DBFAB4) 0(00000209B5DBFAB8) 0(0000
0209B5DBFABC) 0(00000209B5DBFAC0) 0(00000209B5DBFAC4) 0(00000209B5DBFAC8)
 0(00000209B5DBFACC) 1(00000209B5DBFAD0) 2(00000209B5DBFAD4) 3(00000209B5
DBFAD8) 4(00000209B5DBFADC) 5(00000209B5DBFAE0) 6(00000209B5DBFAE4) 7(000
00209B5DBFAE8) 8(00000209B5DBFAEC) 9(00000209B5DBFAF0) 10(00000209B5DBFAF
4) 11(00000209B5DBFAF8) 12(00000209B5DBFAFC)
```

3)Третье: обратимся к рисунку 2, на основании его можно сказать, что capacity вновь перевыделил память для последующих изменений, если они последуют. [2]

```
Add_middle:35 40
44(00000209B5DE4540) 33(00000209B5DE4544) 22(00000209B5DE4548) 11(0000020
9B5DE454C) 0(00000209B5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(
00000209B5DE455C) 0(00000209B5DE4560) 0(00000209B5DE4564) 0(00000209B5DE4
568) 0(00000209B5DE456C) 0(00000209B5DE4570) 0(00000209B5DE4574) 0(000002
09B5DE4578) 0(00000209B5DE457C) 1(00000209B5DE4580) 2(00000209B5DE4584) 3
(00000209B5DE4588) 4(00000209B5DE458C) 5(00000209B5DE4590) 6(00000209B5DE
4594) 7(00000209B5DE4598) 8(00000209B5DE459C) 9(00000209B5DE45A0) 10(0000
0209B5DE45A4) 11(00000209B5DE45A8) 12(00000209B5DE45AC) 111(00000209B5DE4
5B0) 222(00000209B5DE45B4) 333(00000209B5DE45B8) 444(00000209B5DE45BC) 55
5(00000209B5DE45C0) 666(00000209B5DE45C4) 777(00000209B5DE45C8)
```

Можно объединить полученные выводы (пункты 1-3) и сказать, что при выполнении таких функций, как добавление (add), происходит изменение адресов, перевыделение памяти емкости (capacity).

4-6) При удалении не происходит перевыделения памяти, так как это не требуется (мы не выходим за границы capacity), значение сдвигается на нужную ячейку, в соответствии с заданной функцией.

```
Erase_front:34 40
33(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209B
5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(000
00209B5DE4564) 0(00000209B5DE4568) 0(00000209B5DE456C) 0(00000209B5DE4570) 0(00000209B5DE4574)
 0(00000209B5DE4578) 1(00000209B5DE457C) 2(00000209B5DE4580) 3(00000209B5DE4584) 4(00000209B5D
E4588) 5(00000209B5DE458C) 6(00000209B5DE4590) 7(00000209B5DE4594) 8(00000209B5DE4598) 9(00000
209B5DE459C) 10(00000209B5DE45A0) 11(00000209B5DE45A4) 12(00000209B5DE45A8) 111(00000209B5DE45
AC) 222(00000209B5DE45B0) 333(00000209B5DE45B4) 444(00000209B5DE45B8) 555(00000209B5DE45BC) 66
6(00000209B5DE45C0) 777(00000209B5DE45C4)

Erase_middle:33 40
33(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209B
5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(000
00209B5DE4564) 0(00000209B5DE4568) 0(00000209B5DE456C) 0(00000209B5DE4570) 0(00000209B5DE4574)
 1(00000209B5DE4578) 2(00000209B5DE457C) 3(00000209B5DE4580) 4(00000209B5DE4584) 5(00000209B5D
E4588) 6(00000209B5DE458C) 7(00000209B5DE4590) 8(00000209B5DE4594) 9(00000209B5DE4598) 10(0000
0209B5DE459C) 11(00000209B5DE45A0) 12(00000209B5DE45A4) 111(00000209B5DE45A8) 222(00000209B5DE
45AC) 333(00000209B5DE45B0) 444(00000209B5DE45B4) 555(00000209B5DE45B8) 666(00000209B5DE45BC)
777(00000209B5DE45C0)

Erase_back:32 40
33(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209B
5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(000
00209B5DE4564) 0(00000209B5DE4568) 0(00000209B5DE456C) 0(00000209B5DE4570) 0(00000209B5DE4574)
 1(00000209B5DE4578) 2(00000209B5DE457C) 3(00000209B5DE4580) 4(00000209B5DE4584) 5(00000209B5D
E4588) 6(00000209B5DE458C) 7(00000209B5DE4590) 8(00000209B5DE4594) 9(00000209B5DE4598) 10(0000
0209B5DE459C) 11(00000209B5DE45A0) 12(00000209B5DE45A4) 111(00000209B5DE45A8) 222(00000209B5DE
45AC) 333(00000209B5DE45B0) 444(00000209B5DE45B4) 555(00000209B5DE45B8) 666(00000209B5DE45BC)
```

7) Удаление заданного диапазона значений из середины, сдвиг значений влево, уменьшение size до нужного размера.

```
After erase elements from index 3 to 5:
29 40
33(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209B
5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(000
00209B5DE4564) 0(00000209B5DE4568) 1(00000209B5DE456C) 2(00000209B5DE4570) 3(00000209B5DE4574)
 4(00000209B5DE4578) 5(00000209B5DE457C) 6(00000209B5DE4580) 7(00000209B5DE4584) 8(00000209B5D
E4588) 9(00000209B5DE458C) 10(00000209B5DE4590) 11(00000209B5DE4594) 12(00000209B5DE4598) 111(
00000209B5DE459C) 222(00000209B5DE45A0) 333(00000209B5DE45A4) 444(00000209B5DE45A8) 555(000002
09B5DE45AC) 666(00000209B5DE45B0)
```

8-11) Замена значений, size и capacity остаются прежними.

```
Replacement_front:29 40
999(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209
B5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(00
000209B5DE4564) 0(00000209B5DE4568) 1(00000209B5DE456C) 2(00000209B5DE4570) 3(00000209B5DE4574
) 4(00000209B5DE4578) 5(00000209B5DE457C) 6(00000209B5DE4580) 7(00000209B5DE4584) 8(0000209B5
DE4588) 9(00000209B5DE458C) 10(00000209B5DE4590) 11(00000209B5DE4594) 12(00000209B5DE4598) 111
(00000209B5DE459C) 222(00000209B5DE45A0) 333(00000209B5DE45A4) 444(00000209B5DE45A8) 555(00000
209B5DE45AC) 666(00000209B5DE45B0)

Replacement_middle:29 40
999(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209
B5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(00
000209B5DE4564) 0(00000209B5DE4568) 1(00000209B5DE456C) 2(00000209B5DE4570) 3(00000209B5DE4574
) 888(00000209B5DE4578) 5(00000209B5DE457C) 6(00000209B5DE4580) 7(00000209B5DE4584) 8(00000209
B5DE4588) 9(00000209B5DE458C) 10(00000209B5DE4590) 11(00000209B5DE4594) 12(00000209B5DE4598) 1
11(00000209B5DE459C) 222(00000209B5DE45A0) 333(00000209B5DE45A4) 444(00000209B5DE45A8) 555(000
00209B5DE45AC) 666(00000209B5DE45B0)


Replacement_back:29 40
999(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209
B5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(00
000209B5DE4564) 0(00000209B5DE4568) 1(00000209B5DE456C) 2(00000209B5DE4570) 3(00000209B5DE4574
) 888(00000209B5DE4578) 5(00000209B5DE457C) 6(00000209B5DE4580) 7(00000209B5DE4584) 8(00000209
B5DE4588) 9(00000209B5DE458C) 10(00000209B5DE4590) 11(00000209B5DE4594) 12(00000209B5DE4598) 1
11(00000209B5DE459C) 222(00000209B5DE45A0) 333(00000209B5DE45A4) 444(00000209B5DE45A8) 555(000
00209B5DE45AC) 777(00000209B5DE45B0)
```

12-11)Изменение size и capacity, память которой зарезервирована. Очищение памяти.

```
After resize to size 10:
10 40
999(00000209B5DE4540) 22(00000209B5DE4544) 11(00000209B5DE4548) 0(00000209B5DE454C) 0(00000209
B5DE4550) 0(00000209B5DE4554) 0(00000209B5DE4558) 0(00000209B5DE455C) 0(00000209B5DE4560) 0(00
000209B5DE4564)

After clear():0 40
```

Приложение А: проведение эксперимента

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

void print_vector_info(const std::vector<int>& vec) {
    std::cout << vec.size() << " " << vec.capacity() << std::endl;
    for (size_t i = 0; i < vec.size(); ++i) {
        std::cout << vec[i] << "(" << &vec[i] << ") ";
    }
    std::cout << std::endl;
    if (vec.empty()) {
        std::cout << "Вектор пуст." << std::endl;
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    std::vector<int> vec(12);
    std::cout << "\nOriginal:";
    print_vector_info(vec);
    for (int i = 0; i < 12; i++) { vec.push_back(i + 1); }
    std::cout << "\nAdd_back:";
    print_vector_info(vec);
    for (int i = 0; i < 4; i++) { vec.insert(vec.begin(), 11 * (i + 1)); }
    std::cout << "\nAdd_front:";
    print_vector_info(vec);

    for (int i = 0; i < 7; i++) {
        vec.push_back(111 * (i + 1));
```

```
    }
    std::cout << "\nAdd_middle:";
    print_vector_info(vec);
    vec.erase(vec.begin() + 0);
    std::cout << "\nErase_front:";
    print_vector_info(vec);

    vec.erase(vec.begin() + 4);
    std::cout << "\nErase_middle:";
    print_vector_info(vec);

    vec.erase(vec.begin() + vec.size() - 1);
    std::cout << "\nErase_back:";
    print_vector_info(vec);

    if (vec.size() >= 6) {
        vec.erase(vec.begin() + 3, vec.begin() + 6);
        std::cout << "\nAfter erase elements from index 3 to 5:" << std::endl;
        print_vector_info(vec);
    }
    else {
        std::cout << "\nNot enough elements to erase." << std::endl;
    }

    if (!vec.empty()) { vec[0] = 999; }
    std::cout << "\nReplacement_front:";
    print_vector_info(vec);

    if (!vec.empty()) {
        size_t middle_index = vec.size() / 2;
        if (middle_index < vec.size()) { vec[middle_index] = 888; }
    }
    std::cout << "\nReplacement_middle:";
    print_vector_info(vec);

    if (!vec.empty()) {
        vec[vec.size() - 1] = 777;
    }
    std::cout << "\nReplacement_back:";
    print_vector_info(vec);

    vec.resize(10);
    std::cout << "\nAfter resize to size 10:" << std::endl;
    print_vector_info(vec);

    vec.clear();
    std::cout << "\nAfter clear():";
    print_vector_info(vec);
    system("pause");
    return 0;
}
```

Вывод:
код демонстрирует основные операции с std::vector, такие как добавление, вставка, удален
ие и изменение размера. Основные выводы следующие:

- **Динамическое управление памятью:** std::vector автоматически увеличивает емкость по мере необходимости, чтобы вместить новые элементы.
- **Неизменяемая емкость при удалении:** Удаление элементов не влияет на емкость вектора, и память остается зарезервированной.
- **Изменение размера:** Изменение размера вектора не влияет на емкость, если новый размер меньше текущего.
- **Очистка вектора:** Очистка вектора не влияет на емкость, и память остается зарезервированной.

Эти принципы демонстрируют эффективность и гибкость std::vector как контейнера для динамических массивов в C++.

Литература:

1) https://stackoverflow.com/questions/6296945/size-vs-capacity-of-a-vector – форум.
2) https://ru.wikipedia.org/wiki/Vector_%28C%2B%2B%29 – Wikipedia