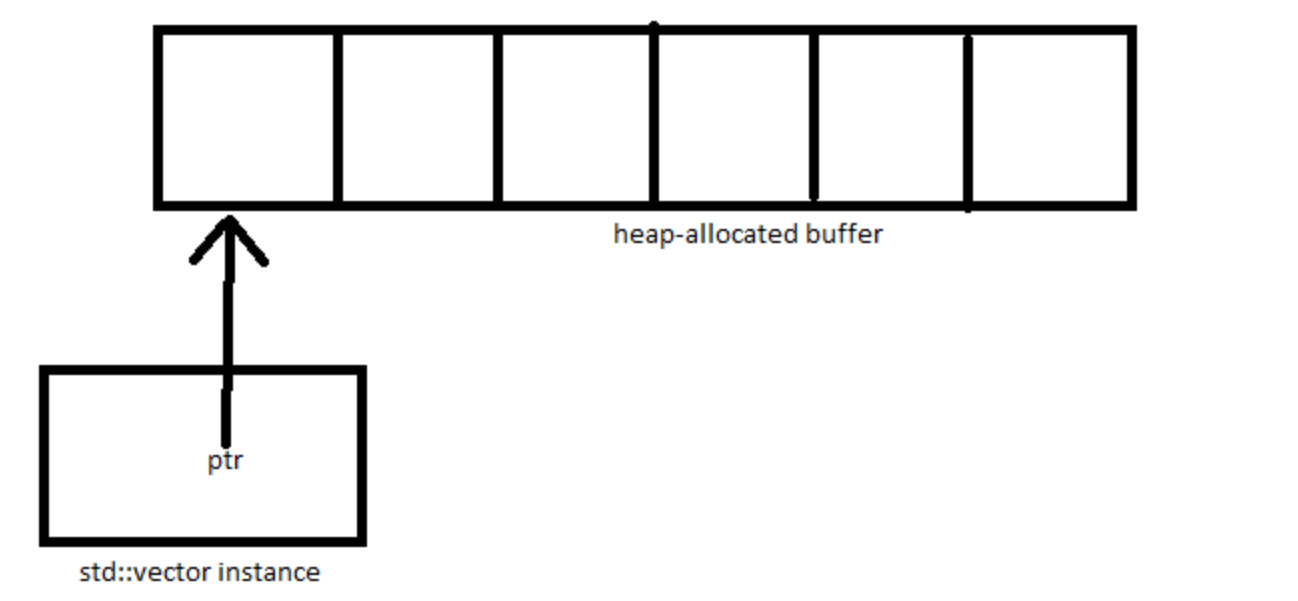
**Принципы работы контейнера std::vector из библиотеки <vector>**

**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 памяти.
  + Операции добавления, удаления и изменения размера выполняются быстро и эффективно.
* **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.

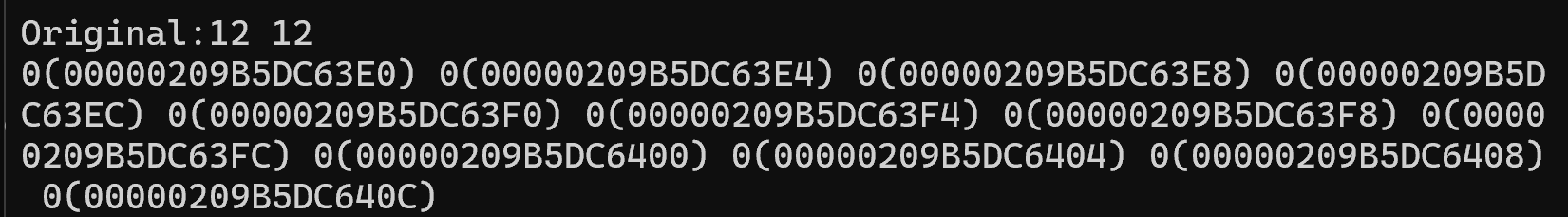
**Заключение**

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

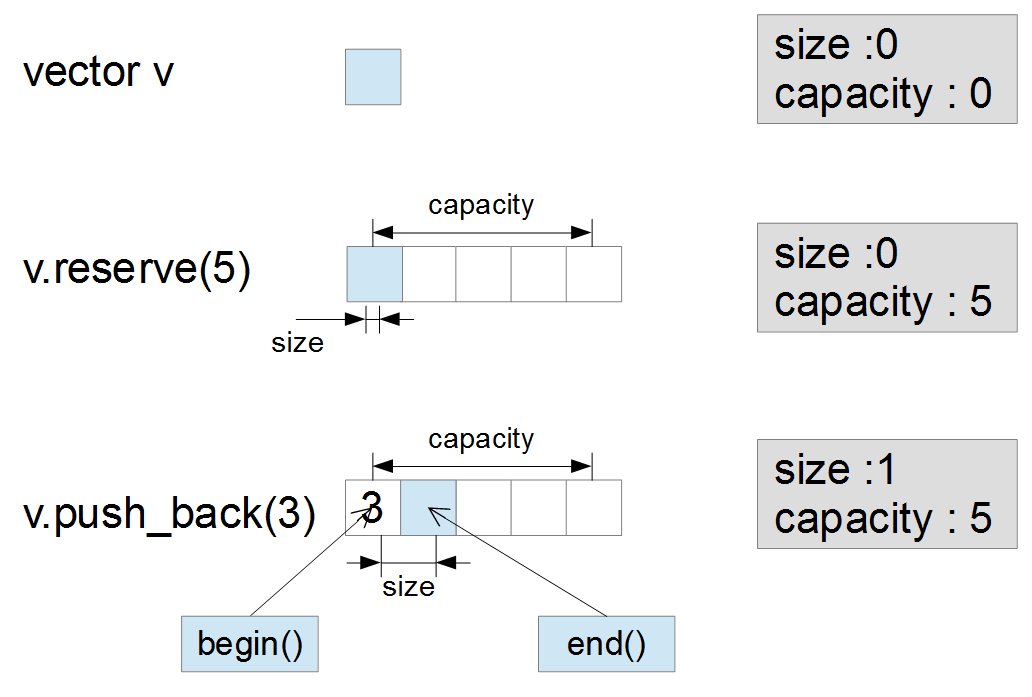


*Рис. 1 - Запуск тестовой программы с выводом адресов*

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

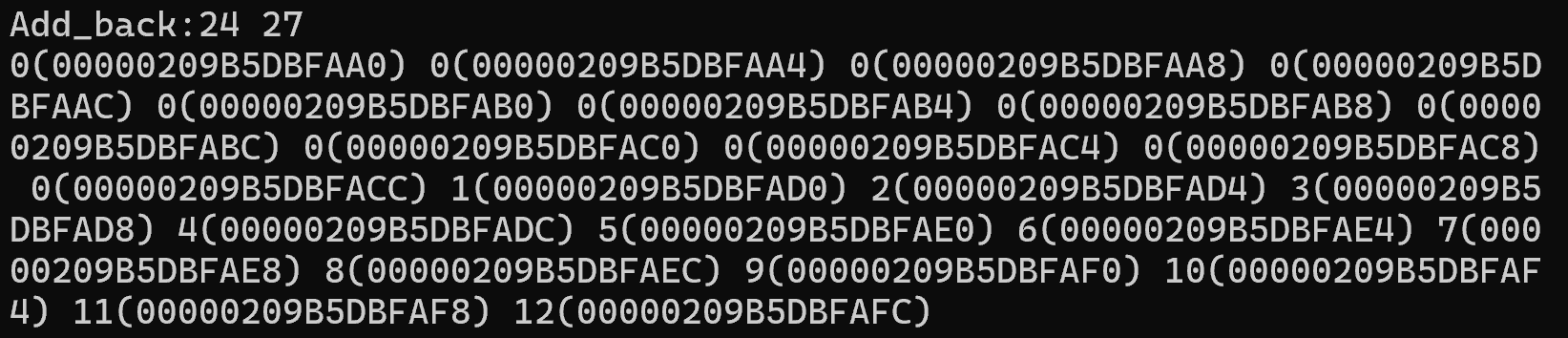


2)Второе: добавление в конец 12 значений. Size = 24, capacity = 27. Я ознакомилась с различной литературой, а также с форумами, таак, например <https://stackoverflow.com/questions/6296945/size-vs-capacity-of-a-vector> здесь пишут о том, что “**Емкость**: сколько элементов может поместиться в векторе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

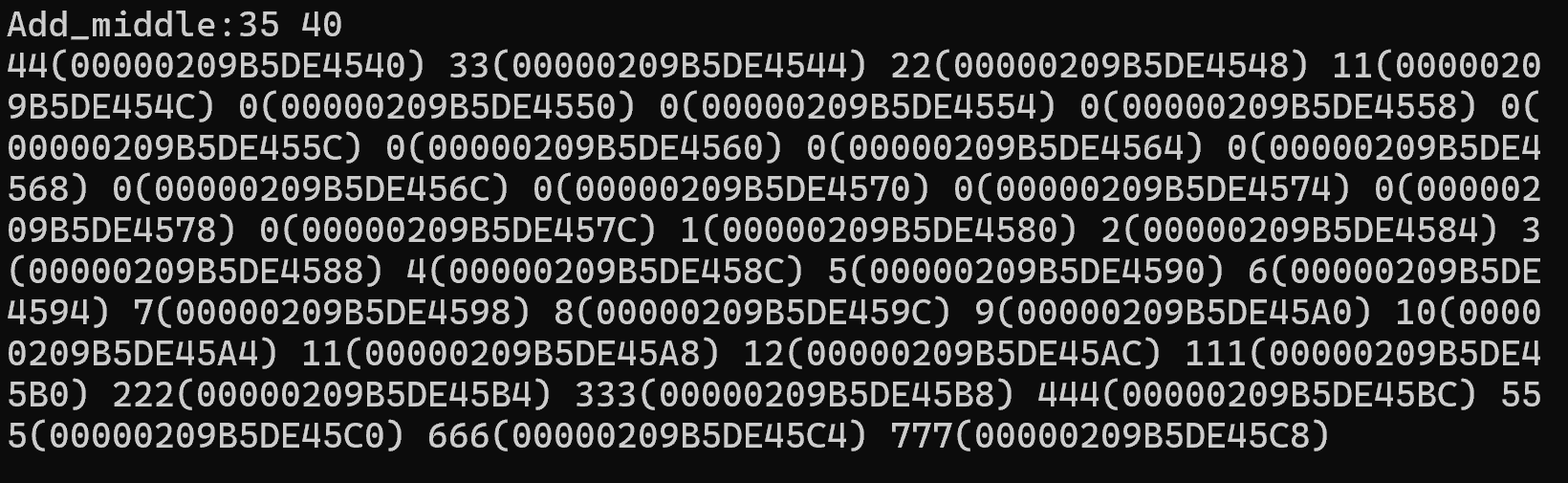


*Рис. 2 – Изменение значений size и capacity (перевыделение памяти).*

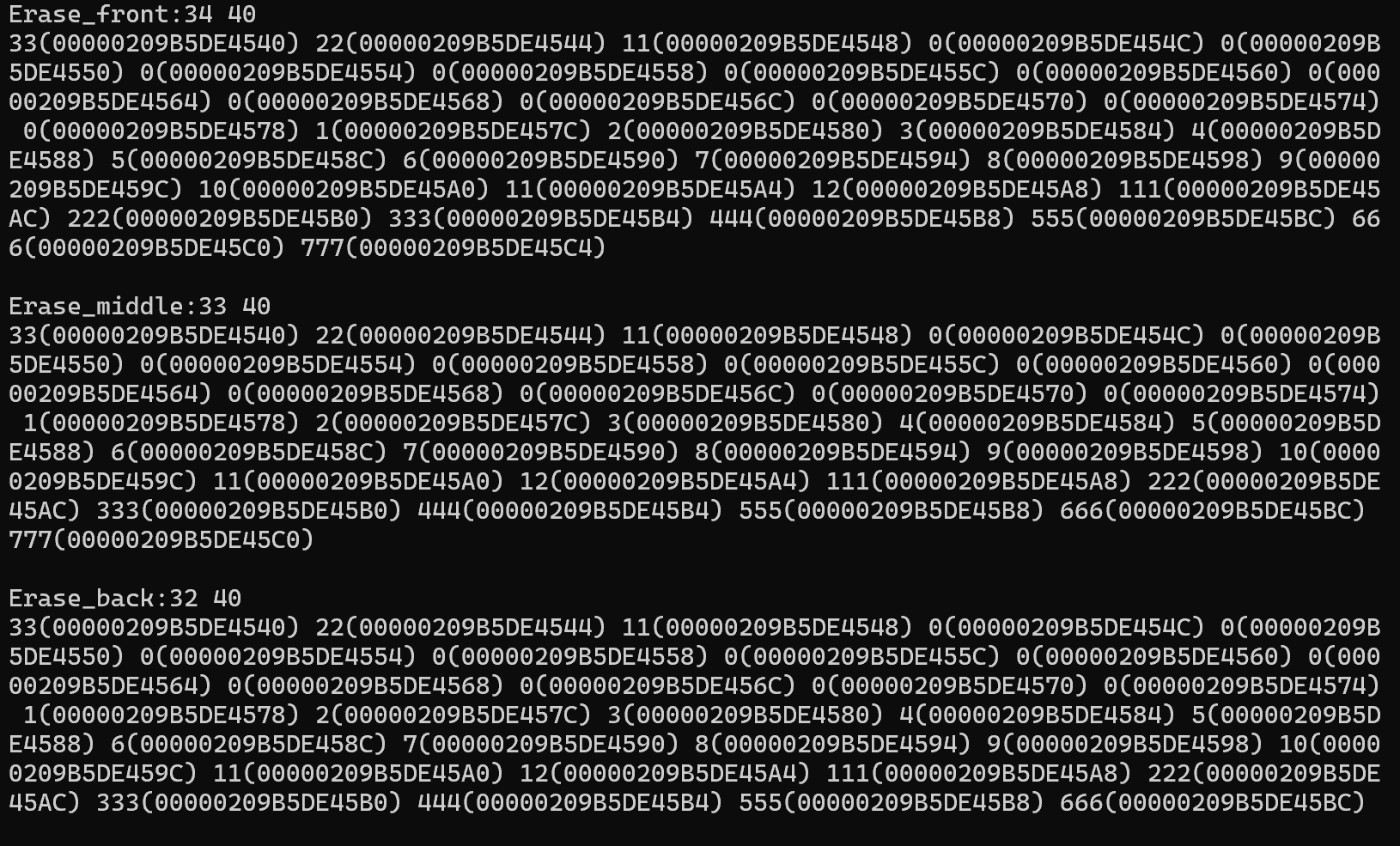
Также можем заметить, что адрес изменился у тех элементов, которые были в original векторе, но опять же в интернете я нашла информацию о том, что это является нормой. [2]

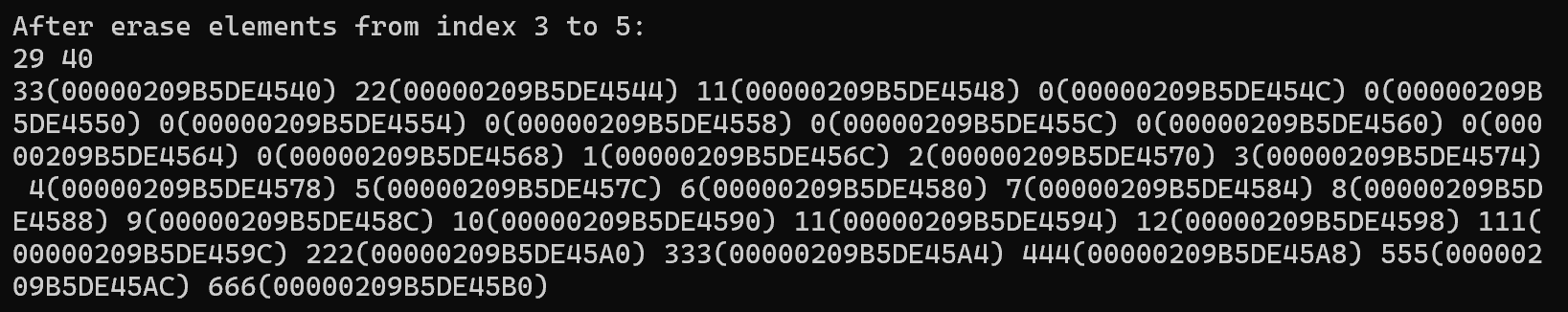


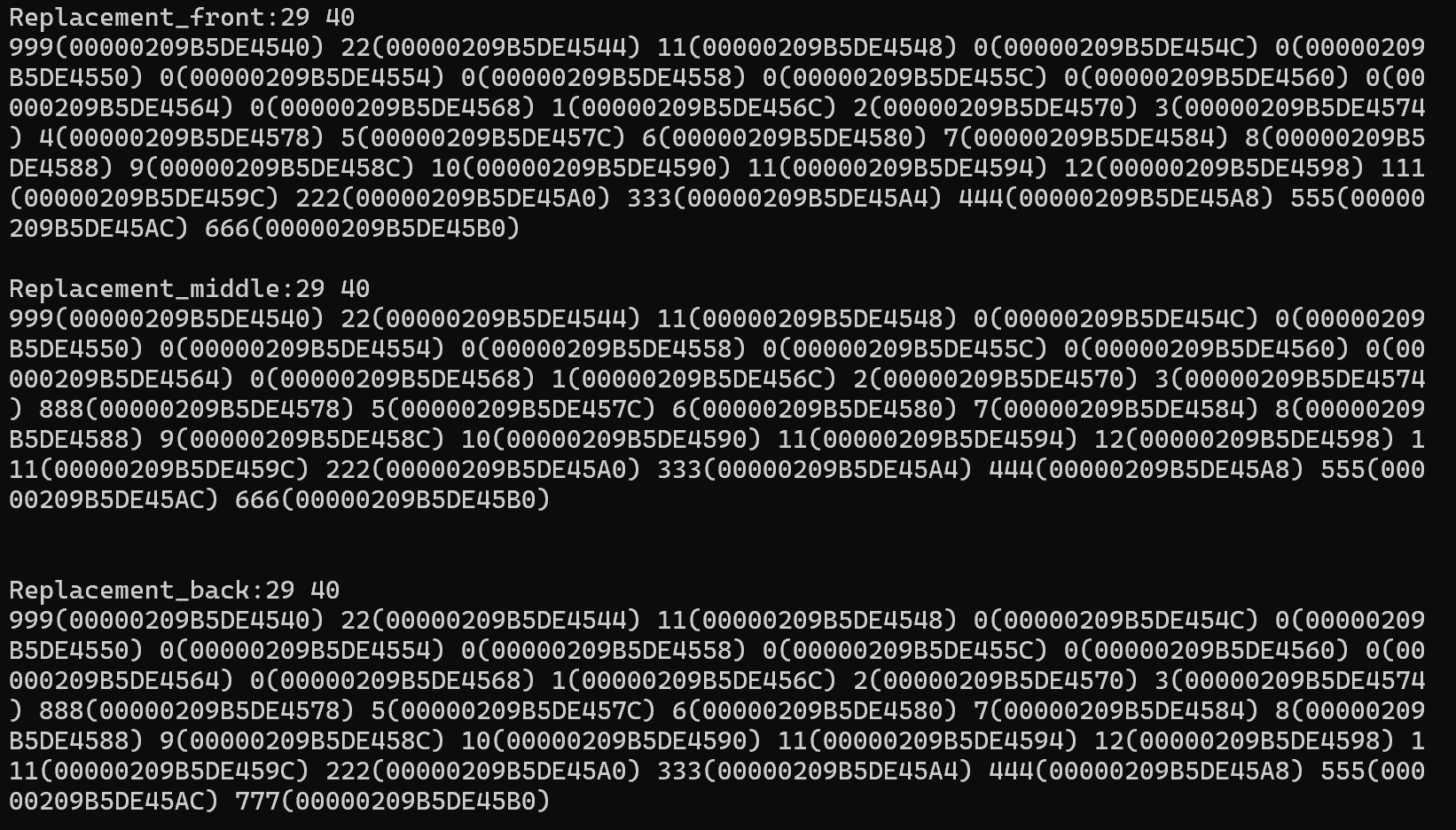
3)Третье: обратимся к рисунку 2, на основании его можно сказать, что capacity вновь перевыделил память для последующих изменений, если они последуют. [2]



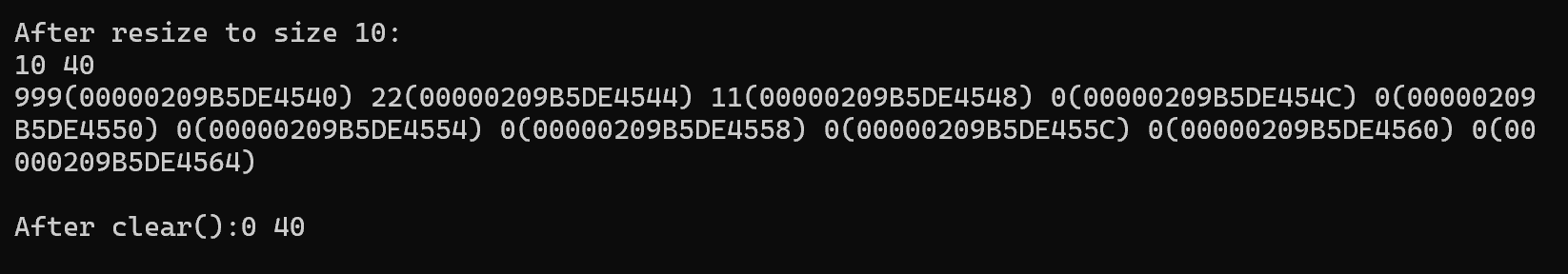
Можно объединить полученные выводы (пункты 1-3) и сказать, что при выполнении таких функций, как добавление (add), происходит изменение адресов, перевыделение памяти емкости (capacity).

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.

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

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

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



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, удаление 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