4.24 Контейнер vector: внутреннее устройство с алгоритмической точки зрения, методы reserve, capacity, shrink_to fit, их действие. Проблема реализации выделения памяти в методе reserve. Реализация методов reserve, resize и push_back с использованием аллокатора (здесь можно без поддержки exception safety). Правила инвалидации итераторов в vector.

Устройство вектора:

```
template <typename T, typename Alloc = std::allocator<T>>
class Vector {
private:
    T* arr;
    size_t sz;
    size_t capacity;
    Alloc alloc;

    using Alloctraits = std::alloctator_traits<Alloc>;

public:
    Vector(size_t n, const T& value = T(), const Alloc& alloc = Alloc());

    T& operator[](size_t i) {
        return arr[i];
    } //also this method but for const Vector

    T& at(size_t i) {
        if (i >= sz) throw std::out_of_range("...");
        return arr[i];
    } //also this method but for const Vector

    size_t size() const {
        return sz;
    }

    size_t capacity() const {
        return cap;
    }

    void resize(size_t n, const T& value = T());
    void reserve(size_t n);
};
```

Замечание: При использовании в векторе типа без конструктора по умолчанию, мы обязаны при инициализации указывать тогда каким значением проинициализировать ячейки

В чем отличие resize от reserve?

- Resize - выделяет памяти столько, чтобы ее хватило на n элементов, т.е меняет размер.
- Reserve - меняет capacity. Обычно вектор не уменьшает capacity, чтобы потом не перевыделять память снова. Но если хотим уменьшить capacity до текущего размера, то можно вернуть системе с помощью вызова `shrink_to_fit()`

Рассмотрим реализацию метода reserve() поэтапно:

Этап 1: В коде ниже представлена плохая реализация reserve(). Почему она плохая? У нас фактически reserve() работает как resize(), что плохо, а мы просто должны выделять память на n объектов, а не заполнять их значениями по умолчанию (просто потому, что конструктора по умолчанию типа T может просто не быть).

```
void reserve(size_t n) {
    if (n <= cap)
        return;

    T* newarr = new T[n];
    for (size_t i = 0; i < sz; ++i) {
        newarr[i] = arr[i];
    }

    delete[] arr;
    arr = newarr;
}
```

Этап 2: Более приемлемая реализация уже выделяет необходимое количество байт для хранения. Но мы не можем делать присваивание к newarr[i], так как в реальности под newarr[i] лежит сырая память ⇒ будет SegFault. Таким образом, нам нужно вызвать конструктор T по данному адресу от данного объекта.
Для этого существует специальный синтаксис: **placement-new**. (смотри строку 7)
Но проблема не устранена, так как delete[] тоже будет SegFault, так как в arr в реальности лежит sz объектов, а не cap, то есть часть объектов – это сырая память. (Решим эту проблему на следующем этапе с помощью вызова деструктора вручную для каждого объекта и потом возврата системе сырой памяти)

```
void reserve(size_t n) {
    if (n <= cap)
        return;

    T* newarr = reinterpret_cast<T*>(new int8_t[n * sizeof(T)]);
    for (size_t i = 0; i < sz; ++i) {
        new(newarr + i) T(arr[i]);
    }

    delete[] arr;
    arr = newarr;
}
```

Этап 3: Корректная реализация с помощью *uninitialized_copy*, которая безопасна относительно исключений.

```
void reserve(size_t n) {
    if (n <= cap) return;

    T* newarr = reinterpret_cast<T*>(new int8_t[n * sizeof(T)]);
    std::uninitialized_copy(arr, arr + sz, newarr);

    for (size_t i = 0; i < sz; ++i) {
        (arr + i)->~T();
    }
    delete[] reinterpret_cast<int8_t*>(arr);
    arr = newarr;
}
```

При этом сам *uninitialized_copy* реализован так:

```
////// uninitialized_copy realization
size_t i = 0;
try {
    for (; i < sz; ++i) {
        new(newarr + i) T(arr[i]);
    }
} catch(...) {
    for (size_t j = 0; j < i; ++j) {
        (newarr + j)->~T();
    }
    delete[] reinterpret_cast<int8_t*>(newarr);
} //////
```

Этап 4: Копирование – это плохо и неэффективно. Можно сделать умнее! Самая хорошая реализация с помощью Allocator & std::move

```
void reserve(size_t n) {
    if (n <= cap) return;

    T* newarr = AllocTraits::allocate(Alloc, n);

    size_t i = 0;
    try {
        for (; i < sz; ++i) {
            AllocTraits::construct(alloc, newarr + i, std::move(arr[i]));
        }
    } catch(...) {
        for (size_t j = 0; j < i; ++j){
            AllocTraits::destroy(alloc, newarr + j);
        }
        AllocTraits::deallocate(newarr, n);
        throw;
    }

    for (size_t i = 0; i < sz; ++i) {
        AllocTraits::destroy(alloc, arr + i);
    }
    AllocTraits::deallocate(arr, n);
    arr = newarr;
}
```

Важно понимать, что **memcpy использовать нельзя**, так как у нас производный тип, а копирование может быть не тривиальным, например, если объект хранит ссылки, то при копировании ссылки могут начать указывать не туда.

Реализация остальных методов vector'a:

```
void push_back(const T& value) {
    if (sz == cap)
        reserve(2 * cap);

    //new(arr + sz) T(value);
    AllocTraits::construct(alloc, arr + sz, value);
    ++sz;
}

void pop_back(const T& value) {
    //(arr + sz - 1)-> T();
    AllocTraits::destroy(alloc, arr + sz - 1);
    --sz;
}

void resize(size_t n, const T& value = T()) {
    if (n < cap) reserve(cap);
    /*...*/
}
```

Инвалидация итераторов

Допустим у нас есть вектор и свободное пространство рядом с ним. Если cap закончилось, то вектор реаллоцирует свой storage: перекладываем все элементы и уничтожаем старый. Что если у нас был итератор на старый вектор??? После перекладывания наш итератор инвалидировался. Если теперь обратиться по итератору, то это UB. Так же push_back инвалириует обычные указатели и ссылки на элементы вектора.

```
vector<int> v;
v.push_back(1); // sz=1, cap=1
vector<int>::iterator it = v.begin();
int* p = &v.front();
int& r = v.front();
v.push_back(2); // sz=2, cap=2
//storage reallocated - so we face invalidation
```