Лекция 3. Структуры данных. Перегрузка функций. Вектор

Структура `struct`

Структура представляет собой пользовательский тип данных, позволяющий объединять несколько переменных разных типов под одним именем.

Синтаксис определения: `struct <name> { <variables>... };`

- `<name>` - имя структуры, является именем нового пользовательского типа данных
- `<variables>...` - ноль и более переменных

Синтаксис объявления: `struct <name>;`

- может пригодиться, когда необходимо использовать структуру раньше, чем она определена.

Переменная, объявленная внутри структуры, называется **полем** структуры.

Гарантируется, что поля структуры располагаются в памяти в порядке объявления полей внутри структуры.

Размер структуры

Размер структуры зависит от размера входящих в неё полей и выравнивания (*alighnment*).

```
struct Point { int x; int y; };
struct A { int x; int y; double d; };
struct B { int x; double d; int y;};
struct C { int x; double d; int y; bool b};

std::cout << sizeof(Point) << std::endl; // 8
std::cout << sizeof(A) << std::endl; // 16
std::cout << sizeof(B) << std::endl; // 24
std::cout << sizeof(C) << std::endl; // 32
```

Создание объекта

Имя структуры используется в качестве имени типа при создании объекта.

Синтаксис: `<struct_name> <var_name>;`

Рекомендуется сразу инициализировать поля объекта, сделать это можно, используя фигурные скобки `{}` и перечисляя значение полей в том порядке, в каком они определены в структуре. Также, можно использовать `()` непосредственно после имени объекта.

```
struct Product {
    int count;
    double price;
};

Product obj;
Product obj_empty = {};
Product obj_init = {10, 300.0};
Product obj_part = {10};        // partitial
Product obj_agr{10, 300.0};     // uniform
```

Имя типа структуры может совпадать с каким-либо объектом, тогда при объявлении можно разрешить конфликт имен добавив `struct` перед именем типа структуры (как правило, лучше **ИЗБЕЖАТЬ** совпадающих названий):

Синтаксис: `struct <struct_name> <var_name>;`

```
struct Product {
    int count;
    double price;
};

std::string Product = "some_product_name";
// Product p = {10, 300.0};     // compile error
struct Product p = {10, 300.0}; // OK
```

Можно объявить структуру сразу с созданием объектов данного типа, **НО** лучше так **НЕ** делать, особенно в глобальной области видимости.

Синтаксис: `struct <struct_name> { <variables>... } <var_name1>, <var_name2>...;`

```
struct Point {
    int x;
    int y;
} p1, p2;
```

Инициализация полей значениями по умолчанию

Начиная с *C++11* можно инициализировать поля значениями по умолчанию через `=` при определении структуры. Тогда при создании объекта данного типа поля будут инициализированы значениями по умолчанию.

```
struct Product {
    int count = 0;
    double price = 100.0;
};

Product one;
```

```
Product two = {};
Product three{};
```

Доступ к полям структуры. Оператор `.`

Доступ к полям структуры через объект типа структуры осуществляется через оператор `.` точка. После имени объекта ставится `.` и затем имя поля:

```
struct Product {
    int count;
    double price;
};

Product obj;
obj.count = 10;
obj.price = 300.0;
```

Доступ к полям структуры. Оператор `->`

Доступ к полям структуры через указатель на объект типа структуры осуществляется через оператор `->` стрелочка. После указателя на объект ставится `->` и затем имя поля:

```
struct Product {
    int count;
    double price;
};

Product obj;
Product* ptr = &obj;
ptr->count = 10;
ptr->price = 300.0;
```

Объединение `union`

Объединение представляет собой пользовательский тип данных, который позволяет хранить разные типы данных в одной области памяти. В один момент может храниться только один тип данных. Размер объединения соответствует наибольшему по размеру типу, входящего в объединение.

Синтаксис: `union <union_name> { <variables>... };`

```
union Union {
    char c;
    int i;
    double d;
};
```

Все входящие в объединение типы переменных расположены по одному адресу.

Доступ к переменным аналогичен структурам.

Использование неактивной переменной

Активная переменная определяется последним присваиванием, при этом время жизни предыдущей активной переменной заканчивается. Использование неактивной переменной это **UB**. Однако, современные компиляторы, как правило, представляют расширение языка и позволяют читать неактивные переменные. Таким образом, можно получить доступ к отдельным байтам.

```
union Color {
    uint32_t value;
    struct {
        uint8_t a
        uint8_t b;
        uint8_t g;
        uint8_t r;
    } components;
};

Color c;
c.value = 0x00FF9955;

std::cout << "Red: " << c.components.r << std::endl;     // 0
std::cout << "Green: " << c.components.g << std::endl;   //
std::cout << "Blue: " << c.components.b << std::endl;    // 255
std::cout << "Alpha: " << c.components.a << std::endl;   // 0
```

- но это по-прежнему **UB** и платформо-зависимый код, на *little-endian* и *big-endian* различный вывод.

Аналогично можно работать отдельно с битами:

```
union HardwareRegister {
    uint32_t raw;
    struct {
        uint32_t enable : 1;
        uint32_t mode : 3;
        uint32_t clock_divider : 4;
        uint32_t reserved : 24;
```

```
    } bits; // 1 + 3 + 4 + 24 = 32
};

HardwareRegister reg;
reg.raw = 0;
reg.bits.enable = 1;
reg.bits.mode = 5;
reg.bits.clock_divider = 8;
```

Анонимные объединения

Можно создавать анонимные объединения, тогда все внутренние имена переменных попадают во внешнюю область видимости объединения и не должно быть конфликта имен.

```
struct Variant {
    enum Type { INT, FLOAT, STRING } type;

    union {
        int intValue;
        float floatValue;
        char stringValue[50];
    };
};
```

Пара `std::pair`

Пара `std::pair` - это шаблонная структура, позволяющая хранить переменные двух типов в полях `first` и `second`.

Заголовочный файл: `<utility>`

Объявление: `std::pair<T1, T2> <name>;`

- `T1` - тип первой переменной
- `T2` - тип второй переменной

Методы:

- `swap` - для обмена значений с другой парой

Доступны операторы сравнения, сравнение производится лексикографически.

Пара используется в стандартных ассоциативных контейнерах, при возврате ключа и значения контейнера. А также, может быть использована при возврате двух значений из функции. Как правило, это менее информативно, чем структура с понятными именами полей.

Инициализация и доступ к полям

Доступ к соответствующим переменным в паре, осуществляется через имена полей `first` и `second`.

Инициализировать пару:

```
std::pair<char, int> p;                 // uninitialized
p.first = 'c';
p.second = static_cast<int>(p.first);

std::pair<bool, std::string> p1 = {true, "Book"};
std::pair<int, double> p2(10, 3.14);    // ctor
std::pair<bool, double> p3{true, 2.73}; // uniform C++11
```

Доступ к полям с помощью `std::get` (*C++11*)

В заголовочном файле `<utility>` есть шаблонная функция `std::get`, которая на вход принимает пару и возвращает ссылку на поле пары.

Может работать по индексу:

- `0` - `first`
- `1` - `second`
- другие значения приведут к ошибке компиляции

Может работать по типу переменной, в случае одинаковых типов будет ошибка компиляции.

```
std::pair<bool, int> p = {true, 10};
std::cout << '(' << std::get<0>(p) << ", " << std::get<1>(p) << ')' <<
std::endl;
std::cout << '(' << std::get<bool>(p) << ", " << std::get<int>(p) << ')' <<
std::endl;
std::get<bool>(p) = false;
std::get<1>(p) = 0;
```

Создание с помощью `std::make_pair`

В заголовочном файле `<utility>` есть шаблонная функция `std::make_pair` принимающая два аргумента и возвращающая пару из двух аргументов. Удобство заключается в возможности использования вывода типа переменной, если написать ключевое слово `auto`.

```cpp
auto p1 = std::make_pair(15, 'c');    // std::pair<int, char>
auto p2 = std::make_pair(true, 3.14); // std::pair<bool, double>

int value = 3;
int array[] = {1, 2, 3};
auto p3 = std::make_pair(std::ref(value), array); // std::pair<int&, int*>
```

Кортеж `std::tuple` (*C++11*)

Кортеж `std::tuple` - шаблонный класс, позволяющий хранить произвольное число аргументов различных типов.

Заголовочный файл: `<tuple>`

Объявление: `std::tuple<T...> <name>;`

- `T...` - произвольное количество различных типов

Методы:

- `swap` - для обмена значений с другим аналогичным по типам кортежем

Для кортежа определены операторы сравнения.

Инициализация

Инициализировать кортеж можно привычными способами:

```cpp
std::tuple<char, int> t;                                    // uninitialized
std::tuple<bool, std::string, int> t1 = {true, "Book", 10}; // copy
std::tuple<int, double, int> t2(10, 3.14, -1);              // ctor
std::tuple<bool, double, int, char> p3{true, 2.73, 0, '0'}; // uniform C++11
```

Доступ к полям с помощью `std::get`

Поскольку кортеж не имеет прямого доступа к полям, то получить значения полей кортежа или изменить их можно с помощью шаблонной функции `std::get`.

Функция принимает кортеж, а в качестве параметров шаблона внутри `<>` указывается либо порядковый номер начиная с `0` и меньше количества переменных, либо тип данных, если он уникален для кортежа

```cpp
std::tuple<float, char, int> tpl(10., 'a', 10);
std::get<0>(tpl) += 1;
std::get<1>(tpl) += 1;
```

```
std::get<2>(tpl) += 1;
// tpl = {11, 'b', 11};
std::get<float>(tpl) += 1;
std::get<char>(tpl) += 1;
std::get<int>(tpl) += 1;
// tpl = {12, 'c', 12};
```

Создание с помощью `std::make_tuple`

В заголовочном файле `<tuple>` есть шаблонная функция `std::make_tuple` принимающая произвольное число аргументов различных типов и возвращающая кортеж из переданных аргументов.

Удобство заключается в возможности использования вывода типа переменной, если написать ключевое слово `auto`.

```
auto t1 = std::make_tuple(15, 'c', 3.14); // std::tuple<int, char, double>

struct Product { int count; double price; };
auto t2 = std::make_pair(true, Product{10, 3.14}); // std::tuple<bool, Product>
std::cout << std::get<1>(t2).count << " " << std::get<1>(t2).price;
```

Функция связывания `std::tie`

Шаблонная функция позволяющая создать кортеж ссылок на переданные аргументы или ссылку на специальный объект `std::ignore`

- применяется для лексикографического сравнения структур:

```
struct Pair { int value; std::string name; } p1, p2;
std::tie(p1.value, p1.name) < std::tie(p2.value, p2.name);
```

- для распаковки кортежа или пары (так как пара конвертируется в кортеж):

```
int i;
double d;
std::tie(i, std::ignore, d) = std::tuple(1, "Test", 2.0);
```

Структурное связывание (*structered bindings*) *C++17*

Structered bindings - синтаксическая конструкция, введенная в *C++17*, которая позволяет распаковывать составляющие составных типов данных в отдельные переменные.

Синтаксис: `<cv> auto <ref> [<name>...] = <var>;`

Работает с массивами, структурами, парами, кортежами.

```
int array[] = {1, 2};

auto& [first, second] = array;
first = 3;
second = 4; // array = {3, 4};
```

```
struct Product { int count; double price; };

Product prod{10, 100.0};
auto [count, price] = prod;
auto& [ref_count, ref_price] = prod;
const auto& [cref_count, cref_price] = prod;
```

```
std::pair<bool, int> p(true, 10);
const auto& [b, i] = p;
```

```
std::tuple<float, char, int> tpl(10., 'a', 10);
auto [f, c, i] = tpl;
```

Перегрузка функций

Перегрузка функций позволяет иметь функции с одинаковыми именами для различных входных аргументов.

Перегрузить функцию можно только по аргументам, изменение возвращаемого типа, без изменения аргументов **НЕ** является перегрузкой.

CV-квалификаторы изменяют тип, поэтому добавление константности к аргументам является перегрузкой.

```
void PrintValues(const int* begin, const int* end) {}
void PrintValues(int* begin, int* end) {}
void PrintValues(const int* array, size_t size) {}
void PrintValues(const Point& point) {}
// Point PrintValues(const Point& point) {} // compile error
```

Функция, имеющая значения по умолчанию, имеет перегрузки функции, без аргументов по умолчанию, причем количество перегрузок соответствует количеству аргументов по умолчанию.

```
void PrintValues(int x, float f = 0.0, char c = 'a') {
    std::cout << "x = " << x << ", f = " << f << ", c = " << c << std::endl;
```

```
}

PrintValues(x, f, c);
PrintValues(x, f);
PrintValues(x);
```

Перегрузка операторов

Поскольку операторы являются функциями, то операторы можно перегрузить.
Что позволяет использовать операторы для своих типов данных.

Синтаксис: `<return_type> operator<op_symbol>(<arguments>...);`

- `<op_symbol>` - символьное обозначение оператора (`+` , `<<` , `==` , ...)
- `<arguments>...` - один или два аргумента (для унарного и бинарного операторов)
- `<return_type>` - тип возвращаемого значения

НЕЛЬЗЯ перегрузить операторы: `::` , `?:` , `.` , `.*` , `typeid` , `sizeof` , `alignof`

Перегрузка оператора `<<` для вывода в поток:

```
struct Pair {
    int value;
    std::string name;
};

std::ostream& operator<<(std::ostream& os, const Pair& pair) {
    os << '(' << value << ", " << name << ')';
    return os;
}
```

Перегрузка операторов сравнения:

```
bool operator==(const Pair& lhs, const Pair& rhs) {
    return lhs.value == rhs.value && lhs.name == rhs.name;
}

bool operator<(const Pair& lhs, const Pair& rhs) {
    if (lhs.value == rhs.value) {
        return lhs.name < rhs.name;
    }
    return lhs.value < rhs.value;
//    return (lhs.value == rhs.value) ? lhs.name < rhs.name : lhs.value <
rhs.value;
//    return std::tie(lhs.value, lhs.name) < std::tie(rhs.value, rhs.name);
}

bool operator!=(const Pair& lhs, const Pair& rhs) {
    return !(lhs == rhs);
```

```
}

bool operator>(const Pair& lhs, const Pair& rhs) {
    return !(lhs < rhs || lhs == rhs);
}

bool operator>=(const Pair& lhs, const Pair& rhs) {
    return !(lhs < rhs);
}

bool operator<=(const Pair& lhs, const Pair& rhs) {
    return lhs < rhs || lhs == rhs;
}
```

Асимптотическая сложность

Асимптотическая сложность — это математический способ оценки, как время выполнения или объём потребляемой памяти алгоритма растут с увеличением размера входных данных (`n`), когда `n` стремится к бесконечности.

При оценке игнорируются константы и младшие члены в формуле, что позволяет сосредоточиться на основном факторе масштабирования. Оценка позволяет сравнивать эффективность алгоритмов независимо от аппаратного обеспечения.

Обычно используется O-нотация («O большое») для определения верхней границы (наихудшего случая).

Распространенная асимптотическая сложность:

1. — Константная сложность
   - Время выполнения не зависит от размера входных данных.
2. — Логарифмическая сложность
   - Медленный рост. Удвоение `n` увеличивает время работы лишь на константу.
3. — Линейная сложность
   - Время растет прямо пропорционально n.
4. — Линейно-логарифмическая сложность
   - Золотой стандарт для эффективных алгоритмов сортировки общего назначения.
5. — Степенная сложность
   - Время растет пропорционально степени размера данных. При больших `n` становится очень медленно.
6. — Экспоненциальная сложность
   - Чрезвычайно быстрый рост. Практически неприменимо для задач с `n` > 30-40.
7. — Факториальная сложность
   - Рост быстрее экспоненциальной. Характерна для задач полного перебора всех перестановок.

Последовательные контейнерыv

Стандартная библиотека *STL* содержит множество шаблонных классов, которые представляют различные структуры данных. Как правило, их называют контейнерами. Контейнеры способны хранить произвольный тип данных. Контейнеры представляют собой коллекцию элементов определенного типа. Контейнеры предоставляют удобный способ взаимодействовать с элементами

В качестве параметров шаблона внутри треугольных скобок `<>` обычно необходимо указать тип данных, который использует контейнер.

Для последовательных контейнеров определен оператор доступа по индексу `[]`

Контейнер `std::vector`

Контейнер `std::vector` предназначен для хранения массива данных определенного типа непрерывно в динамической памяти.

Контейнер обеспечивает быстрое добавления элементов в конец и увеличение размера блока памяти под элементы при необходимости.

Для гарантии непрерывности памяти, при добавлении элемента в заполненный контейнер, происходит **реалокация** всех элементов контейнера.

Заголовочный файл: `<vector>`

Синтаксис: `std::vector<T> <name>;`

- `T` - произвольный тип данных
- `<name>` - имя переменной

Инициализация `std::vector`

Доступны множество способов инициализации вектора:

```
std::vector<int> v1;                // empty
std::vector<int> v2(v1);            // copy ctor from other vector
std::vector<int> v3 = v1;           // copy ctor from other vector
// ctor
std::vector<int> v4(5);             // 5 elemets with zero value
std::vector<int> v5(5, 2);          // 5 elements equal 2
// initializer list
std::vector<int> v6{1, 2, 4, 5};    // with elements 1, 2, 4, 5
std::vector<int> v7 = {1, 2, 3, 5}; // with elements 1, 2, 3, 5
// iterators
std::vector<int> v8(v7.begin(), v7.end()); // same as v7
```

Устройство `std::vector`

Контейнер должен иметь **указатель на данные** (адрес начала памяти в куче, выделенной под хранение элементов вектора), **вместимость выделенной памяти** (сколько элементов данного типа можно положить), **текущее количество элементов** (сколько элементов данного типа уже положено в контейнер).

Условно можно представить себе, что хранится 3 поля: указатель на данные, размер данных *size*, размер выделенной памяти *capacity*

Контейнер имеет размер 24 байта.

Фактически, стандартный контейнер устроен несколько иначе, внутри хранится 3 указателя: на начало данных, за конец выделенной памяти, за конец элементов

Элементы контейнера располагаются не на **стеке**, а в области памяти, называемой куча.

Обзор методов `std::vector`

Контейнер имеет встроенные функции для работы с контейнером, такие функции называются **методы**.

Доступ к элементам:

- `at(size_t pos)` - метод, принимающий индекс элемента и возвращающий ссылку
- `[size_t pos]` - оператор `[]`, принимающий индекс и возвращающий ссылку
- `font()` - метод, возвращающий ссылку на первый элемент
- `back()` - метод, возвращающий ссылку на последний элемент
- `data()` - метод, возвращающий указатель на начало данных

Вместимость контейнера:

- `empty()` - возвращает `true` для пустого контейнера
- `size()` - возвращает количество элементов
- `capacity()` - возвращает вместимость контейнера
- `reserve(size_t count)` - выделяет память под `count` элементов
- `shrink_to_fit()` - уменьшает вместимость вектора в соответствии с `size()`

Модифицирующие методы:

- `assign()` - заменить текущее содержимое на новое
- `clear()` - очистить содержимое (без фактического очищения)
- `push_back()` - добавить элемент в конец контейнера
- `pop_back()` - удалить последний элемент из контейнера
- `insert()` - добавить элементы в определенную позицию
- `erase()` - удалить элементы с определенной позиции

- `resize()` - изменить количество элементов контейнера
- `swap()` - обменять содержимое контейнеров.

Вместимость контейнера `std::vector`

По умолчанию, неинициализированный, пустой контейнер **НЕ** выделяет память под элементы.

Инициализированный контейнер выделяет память ровно под количество элементов.

Методы, проверяющие вместимость и размер:

- `empty()` — возвращает true, если контейнер пуст
- `size()` — возвращает текущий размер, количество элементов в контейнере
- `capacity()` — возвращает вместимость контейнера, под какое количество элементов выделена память

Добавление в конец `std::vector`

Контейнер оптимизирован под добавление элементов в конец.

Метод, добавляющий элемент в конец:

- `push_back()` - добавить элемент, принимаемый в аргументах, в конец контейнера

Заполнение контейнера в цикле (не оптимально). Несколько раз в ходе заполнения контейнера произойдет реалокация, то есть выделится новый блок памяти в куче, достаточный для элементов контейнера, и все элементы будут скопированы по новому адресу.

Как правило, при добавлении элемента в полностью заполненный контейнер, вместимость контейнера будет **увеличиваться в 2 раза**

Сложность: **амортизированная O(1)**

```
std::vector<int> v;
size_t capacity = v.capacity();

std::cout << v.data() << std::endl; // int* (address)
std::cout << v.size() << ' ' << v.capacity() << std::endl;
for (int i = 0; i < 100; ++i) {
    v.push_back(i);
    if (capacity < v.capacity()) {
        capacity = v.capacity();

        std::cout << v.data() << std::endl; // int* (address)
        std::cout << v.size() << ' ' << v.capacity() << std::endl;
```

```
    }
}
```

Доступ к элементам `std::vector`

Доступ следует осуществлять только к валидным элементам.

Доступ к элементам осуществляется с помощью оператора `[]`, принимающего индекс, и с помощью методов:

- `[size_t pos]` — принимает индекс и возвращает ссылку на элемент
- `at(size_t pos)` — принимает индекс и возвращает ссылку на элемент, если переданный индекс (позиция) невалиден, то выбрасывается исключение `std::out_of_range`
- `fornt()` — возвращает ссылку на первый элемент
- `back()` — возвращает ссылку на последний элемент
- `data()` — возвращает сырой указатель на начало данных

Доступ к элементам вне диапазона контейнера, получение ссылок на элементы для пустого контейнера (`v.empty()` возвращает `true`) это **UB**

Итераторы

Итераторы - фундаментальная концепция языка C++, выступающая в качестве обобщения указателей. Итераторы обеспечивают удобный **способ доступа** и **перемещения** по элементам различных контейнеров.

На данном этапе будем относиться к итераторам, как к удобной обертке над указателем.

Итераторы имитируют поведение **указателей**: разыменование, арифметику, сравнение

Существуют разные категории итераторов, поэтому в зависимости от контейнера поддерживаются не все операции с итераторами

Каждый контейнер имеет **свой тип итератора**.

Например, для вектора целочисленных значений тип итератора:
`std::vector<int>::iterator`

Константная версия итератора имеет тип:
`std::vector<int>::const_iterator`

Используются для реализации **range-based for** и в алгоритмах стандартной библиотеки

Как правило, можно получить итератор на начало (первый элемент) **begin** и на конец **end** (за последний элемент). А также, их константные версии **cbegin** и **cend**

Итераторы `std::vector`

Итератор можно получить воспользовавшись методами контейнера:

- `begin()` — возвращает итератор на начало
- `end()` — возвращает итератор на конец
- `rbegin()` — возвращает обратный итератор на начало
- `rend()` — возвращает обратный итератор на конец

Также присутствуют методы с приставкой с возвращающие их константные версии.

Чтобы не писать длинный тип используется ключевое слово **auto**

Итерирование `std::vector`

Контейнер поддерживает различные способы итерирования, **range-based for**:

```
std::vector<int> v = {1, 2, 3, 5};
for (auto vi : v) {
    std::cout << vi << std::endl;
}
```

Итерирование по индексу (порядковыому номеру):

```
std::vector<int> v = {1, 2, 3, 5};
for (size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << std::endl;
}
```

Использование прямых итераторов:

```
std::vector<int> v = {1, 2, 3, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Использование обратных итераторов, для обхода контейнера в обратном порядке:

```
std::vector<int> v = {1, 2, 3, 5};
for (auto it = v.rbegin(); it != v.rend(); ++it) {
```

```
    std::cout << *it << std::endl;
}
```

Метод `resize()`

Метод `resize()` изменяет размер контейнера, причем в случае увеличения создаются значения по умолчанию, в случае уменьшения вместимость контейнера, как правило, **НЕ** уменьшается

- `resize(size_t count)` — принимает требуемое количество элементов
- `resize(size_t count, const T& value)` — принимает требуемое количество элементов и значение элемента, которым необходимо заполнить контейнер в случае увеличения размера

Сложность: **O(n)** или **O(1)** при уменьшении.

Тратятся ресурсы на заполнение **значениями по умолчанию**

Изменяет *size* контейнера, а также *capacity* при необходимости **увеличения**

Метод `reserve()`

Метод `reserve()` выделяет нужное количество памяти, не заполняя элементы значением по умолчанию, при этом *size* контейнера не изменяется, а *capacity* может только увеличиваться в соответствии с заданным значением

- `reserve(size_t new_capacity)` — принимает новое значение вместимости контейнера

Сложность: **O(n)**

Если известна требуемая вместимость заранее, наиболее оптимальный способ создания:

```
std::vector<int> v;
v.reserve(100);
for (size_t i = 0; i < 100; ++i) {
    v.push_back(i);
}
```

Метод `pop_back()`

Метод, удаляющий последний элемент:

- `pop_back()` — удаляет элемент из конца контейнера

Как правило, просто уменьшает размер контейнера, или перемещает указатель на конец элементов, **НЕ** вызывает реалокацию памяти

Сложность: **O(1)**

`pop_back()` от пустого контейнера (`v.empty()` возвращает `true`) это **UB**

Метод `clear()`

Метод `clear()` очищает контейнер, *size* = 0, но *capacity* **НЕ** изменяется

Как правило, просто уменьшает размер контейнера, перемещает указатель на конец элементов на начало. **НЕ** освобождает память

Сложность: **O(1)** как правило просто переносится указатель конца элементов на начало

Остальные методы `std::vector`

Основные модифицирующие методы контейнера:

- `shrink_to_fit()` — уменьшает *capacity* в соответствии с текущим *size*, реалокация, **O(n)**
- `insert()` — вставляет элемент или диапазон в произвольное место в контейнер, **O(n)**, вставка в конец **O(1)**. Возвращает итератор на начало вставки.
- `erase()` — удаляет элемент или диапазон из произвольного места в контейнер, **O(n)**, поскольку необходимо передвинуть все элементы контейнера, удалить из конца **O(1)**. Возвращает итератор после последнего удаленного элемента.
- `swap()` — обменивает содержимое двух контейнеров, **O(1)** поскольку осуществляется обмен указателей
- `assign()` — заменяет содержимое контейнера на указанное значение или на диапазон, **O(n)**

и другие ...