

std::vector

Denis Bakin

Зачем нужны контейнеры

- В задачах часто требуется сохранять последовательность данных
- Количество элементов заранее может быть неизвестно
- Пример: получение цифр числа и их вывод в правильном порядке
- Решение: контейнер, способный хранить объекты произвольного типа и изменяемого размера
- В C++ стандартный контейнер: `std::vector`

Основные сведения

- Динамический массив, умеет автоматически расширяться
- Быстро добавляет элементы в конец (push_back)
- Хранит последовательность объектов одного типа

Примеры объявления:

```
std::vector<int> v1;           // пустой
std::vector<int> v2(10);       // размер 10
std::vector<int> v3(10, -1);   // 10 элементов со значением -1
std::vector<int> v4 = {1, 2, 3, 4}; // список инициализации
```

Итерация по вектору

Range-based for

```
std::vector<std::string> data = {"Just", "some", "random", "words"};
for (std::string &word : data) {
    std::cout << word << ' ';
}
```

- Нужно подключить заголовок: `#include <vector>`
- В `<>` указывается тип элементов (`int`, `std::string`, ...)
- Range-based for удобен для чтения

Итерация по индексам

```
for (size_t i = 0; i < data.size(); ++i) {  
    std::cout << data[i] << ' '  
}
```

- Тип индекса — `size_t`
- Это беззнаковое целое (нельзя хранить отрицательные числа)
- Размер зависит от архитектуры: 4 байта (32-бит), 8 байт (64-бит)

Основные методы std::vector

```
data.size();           // количество элементов
data.empty();          // пуст ли вектор
data.front();          // первый элемент
data.back();           // последний элемент

data.push_back(x);     // добавление в конец
data.erase(it);        // удаление по итератору
```

Пример: вывод числа в произвольной системе

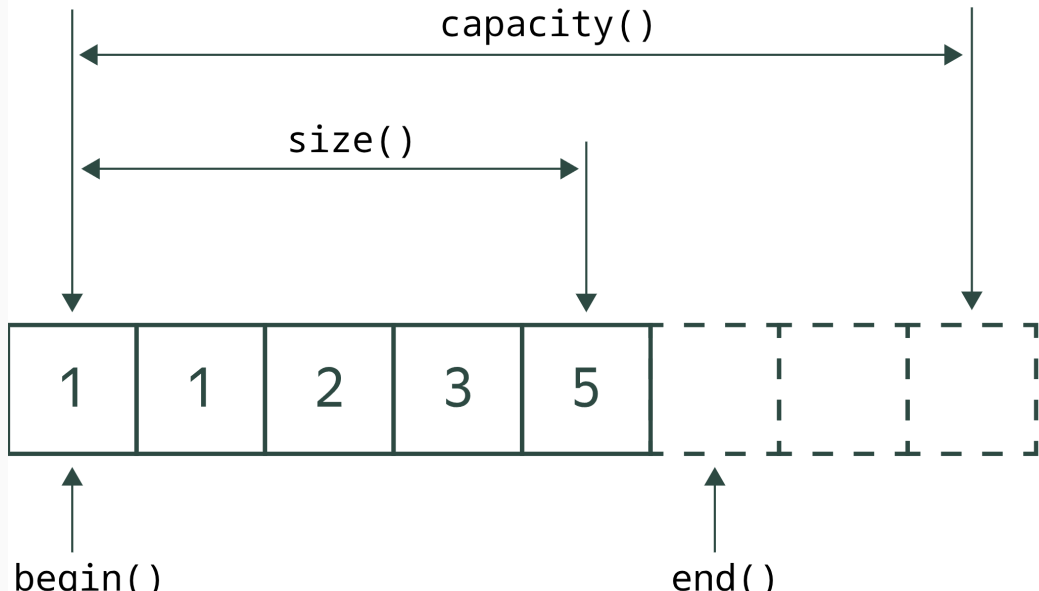
Пример: вывод числа в произвольной системе

```
std::vector<int> digits;
while (num) {
    digits.push_back(num % d);
    num /= d;
}

for (size_t i = digits.size(); i > 0; --i) {
    std::cout << digits[i - 1];
}
```

- Элементы сохраняются при делении
- Выводим с конца к началу — получаем правильный порядок

size vs capacity



size vs capacity

- **size** — текущее число элементов
- **capacity** — выделенная память (вместимость)
- При нехватке памяти вектор удваивает capacity
- `push_back`:
 - если места хватает $\rightarrow O(1)$
 - если места нет $\rightarrow O(n)$, но амортизированно всё равно $O(1)$

Демонстрация роста capacity

```
std::vector<int> data = {1, 2};  
std::cout << data.size() << " " << data.capacity() << "\n"; // 2 2  
  
data.push_back(3);  
std::cout << data.size() << " " << data.capacity() << "\n"; // 3 4  
  
data.push_back(4);  
std::cout << data.size() << " " << data.capacity() << "\n"; // 4 4
```

Вывод: вместимость растёт скачками (обычно в 2 раза).

Сравнение подходов

```
std::vector<int> data(length); // память выделена сразу
```

или

```
std::vector<int> data;  
data.reserve(length);          // зарезервировать память
```

- В обоих случаях все `push_back` выполняются за $O(1)$
- Полезно при известной длине последовательности

- `std::vector` — универсальный контейнер для динамических массивов
- Поддерживает работу с элементами по индексу и через итераторы
- Важные понятия: **size**, **capacity**, амортизированная сложность `push_back`
- Основной контейнер, который чаще всего применяют в C++