Лекция 10. `std::vector` и итераторы

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

Содержание лекции

1. Проблема динамических массивов. `std::vector` как решение.
2. Создание `vector`, доступ к элементам.
3. Размер и ёмкость: `size`, `capacity`, `reserve`, `shrink_to_fit`, `resize`, `clear`.
4. Добавление и удаление: `push_back`, `pop_back`, `insert`, `erase`, `emplace`, `emplace_back`.
5. Итераторы: `begin`, `end`, `rbegin`, `rend`, range-based `for`.
6. Недействительные итераторы.
7. Присваивание: `assign` и `assign_range` (C++23).
8. `std::erase` и `std::erase_if` (C++20).
9. `std::span` (повторение).
10. Итоги.

Проблема динамических массивов

- Ручное управление памятью с `new[]` и `delete[]`:
  - Утечки памяти (забыли `delete[]`).
  - Висячие указатели (использование после удаления).
  - Сложность изменения размера (создание нового массива, копирование, удаление старого).

Пример: изменение размера

```
int* resize(int* old, int oldSize, int newSize) {
    int* newArr = new int[newSize]{};
    int copySize = (oldSize < newSize) ? oldSize : newSize;

    for (int i = 0; i < copySize; ++i)
        newArr[i] = old[i];

    delete[] old;
    return newArr;
}
```

- Код легко усложняется, появляются ошибки.

std::vector как решение

- `std::vector` — шаблонный класс из стандартной библиотеки (подключается через `<vector>`).
- Представляет динамический массив, который автоматически управляет памятью.
- Элементы хранятся непрерывно в памяти (как обычный массив).
- Предоставляет удобные методы для работы с размером, вставкой, удалением.
- Знает свой размер и ёмкость.
- Является последовательным контейнером.

Подключение и создание vector

Подключение

```
#include <vector>
using namespace std; // или std::vector
```

Создание пустого вектора

```
vector<int> v;        // пустой вектор типа int
vector<double> w;     // пустой вектор типа double
```

- `vector` — **шаблон**, но мы пока используем его как готовый тип с указанием типа элементов.
- Создаётся пустой вектор, элементов пока нет, `size() == 0`.

Создание vector с размером и инициализацией

```
vector<int> v1(10);       // 10 элементов, инициализированы нулями (int())
vector<int> v2(10, 5);    // 10 элементов, все равны 5
vector<int> v3 = {1,2,3,4,5}; // инициализация списком (C++11)
vector<int> v4(v3);       // копирование (копия вектора v3)
vector<int> v5(v3.begin(), v3.end()); // из диапазона (итераторы)
```

- В первых двух случаях выделяется память и инициализируются элементы.
- Список инициализации удобен для небольших фиксированных наборов.

Доступ к элементам: operator[]

Синтаксис

```
vector<int> v = {10, 20, 30, 40};
cout << v[0]; // 10
v[1] = 100;   // изменение
```

- operator[] — быстрый доступ, но **без проверки границ**.
- Выход за пределы — неопределённое поведение (UB).
- Аналогичен доступу к элементам обычного массива.

Важно

Используйте operator[], когда уверены, что индекс корректен.

Безопасный доступ: at()

Синтаксис

```
vector<int> v = {10, 20, 30, 40};
try {
    cout << v.at(5); // бросает std::out_of_range
}
catch (const std::out_of_range& e) {
    cout << "Error: " << e.what();
}
```

- at() проверяет границы и бросает исключение при выходе.
- Проверка границ может добавлять накладные расходы.
- Рекомендуется использовать в отладочных целях или когда границы могут быть нарушены.

Доступ к концам: front() и back()

Синтаксис

```
vector<int> v = {10, 20, 30, 40};
cout << v.front(); // 10 (первый элемент)
cout << v.back();  // 40 (последний элемент)
v.front() = 100;   // изменение первого элемента
v.back() = 400;    // изменение последнего
```

- front() возвращает ссылку на первый элемент (если вектор не пуст).
- back() возвращает ссылку на последний элемент (если вектор не пуст).
- Удобны для быстрого доступа к концам.

size()

Синтаксис

```
vector<int> v = {1, 2, 3};
cout << v.size(); // 3
```

- size() возвращает количество элементов в векторе (тип size_t).
- Если вектор пуст, size() возвращает 0.

capacity()

Синтаксис

```
vector<int> v = {1, 2, 3};
cout << v.capacity(); // 3
```

- capacity() возвращает объём выделенной памяти (в элементах).
- Всегда capacity >= size.
- При добавлении элементов перевыделение происходит, если новый размер превышает текущую ёмкость.

reserve()

Синтаксис

```
vector<int> v;
v.reserve(100); // резервируем память на 100 элементов
cout << v.size() << " " << v.capacity(); // 0 100
```

- `reserve(n)` — запрос на выделение памяти под `n` элементов.
- `reserve()` не создаёт элементы и не изменяет `size()`.
- Если `n > capacity`, выделяется новая память.
- Если `n <= capacity`, ничего не происходит.
- Позволяет избежать множественных перевыделений при последовательном добавлении.

shrink_to_fit()

Синтаксис

```
vector<int> v = {1,2,3,4,5}; // size = 5
v.reserve(100);     // capacity = 100
v.shrink_to_fit();  // просим уменьшить capacity до size
cout << v.capacity(); // станет 5 (не гарантируется)
```

- `shrink_to_fit()` — запрос на уменьшение `capacity` до `size`.
- Это не обязательная операция — реализация стандартной библиотеки может проигнорировать.
- Полезна для освобождения лишней памяти после удаления элементов.

resize()

Синтаксис

```
vector<int> v = {1, 2, 3};
v.resize(5);    // добавляет два нуля
v.resize(2);    // удаляет последние три элемента
```

- `resize(n)` — изменяет размер вектора до `n`.
- Если `n > size`, добавляются элементы со значением по умолчанию (для `int` — 0).
- Если `n < size`, лишние элементы удаляются.
- Можно указать значение для новых элементов: `resize(n, value)`.

clear()

Синтаксис

```
vector<int> v = {1, 2, 3};
v.clear();
cout << v.size() << " " << v.capacity(); // 0 3 (capacity не изменилась)
```

- `clear()` удаляет все элементы, устанавливая `size = 0`.
- Память (capacity) НЕ освобождается.
- Это полезно, если вы планируете повторно использовать вектор для новых данных.

Вопрос: что выведет этот код?

```
vector<int> v;
v.reserve(10);
v.resize(5);
v.push_back(42);
cout << v.size() << " " << v.capacity();
```

Вопрос: что выведет этот код?

```
vector<int> v;
v.reserve(10);
v.resize(5);
v.push_back(42);
cout << v.size() << " " << v.capacity();
```

Ответ: 6 10. `reserve(10)` задал ёмкость 10, `resize(5)` установил размер 5, `push_back` добавил элемент, размер стал 6, ёмкость осталась 10.

Вопрос: А какие значения будут выведены, если пройти циклом по этому вектору?

```
for (auto x : v)
    cout << x << ' ';
```

push_back()

Синтаксис

```
vector<int> v;
v.push_back(10);
v.push_back(20);
// v = {10, 20}
```

- `push_back(value)` — добавляет элемент в конец вектора.
- Амортизированная сложность O(1).
- Если `capacity` недостаточна, происходит перевыделение (обычно удвоение).

pop_back()

Синтаксис

```
vector<int> v = {1, 2, 3};
v.pop_back(); // удаляет последний элемент
// v = {1, 2}
```

- pop_back() — удаляет последний элемент.
- Сложность O(1).
- Не возвращает значение (в отличие от некоторых языков).
- Вызов на пустом векторе — неопределённое поведение!

emplace_back()

Синтаксис

```
vector<int> v;
v.emplace_back(5); // аналогично push_back(5)
// для простых типов нет разницы
```

- `emplace_back(args)` — конструирует элемент на месте из переданных аргументов.
- В данном примере результат совпадает с `push_back`.
- Для сложных типов позволяет избежать временного объекта (будет позже).
- Пока просто знайте, что это есть.

insert() — вставка в середину

Синтаксис

```
vector<int> v = {1, 2, 3};
auto it = v.begin() + 1; // итератор на второй элемент
v.insert(it, 99);        // вставка перед it
// v = {1, 99, 2, 3}
v.insert(v.end(), 100);  // вставка в конец (как push_back)
```

- `insert(pos, value)` — вставляет значение перед позицией, на которую указывает итератор.
- Может вставлять несколько элементов или диапазон.
- Сложность O(n) (сдвиг элементов).
- Все итераторы после вставки могут стать недействительными.

erase() — удаление из середины

Синтаксис

```
vector<int> v = {1, 2, 3, 4, 5};
auto it = v.begin() + 2; // итератор на третий элемент
v.erase(it);             // удаляет элемент по итератору
// v = {1, 2, 4, 5}
v.erase(v.begin(), v.begin() + 2); // удаление диапазона
// v = {4, 5}
```

- `erase(pos)` — удаляет элемент по итератору.
- `erase(first, last)` — удаляет диапазон `[first, last)`.
- Сложность O(n) (сдвиг элементов).
- Возвращает итератор на элемент, следующий за удалённым, либо `end()`, если следующего элемента нет.

emplace() — вставка с конструированием на месте

Синтаксис

```
vector<int> v = {1, 2, 3};
auto it = v.begin() + 1;
v.emplace(it, 99); // аналогично insert, но конструирует на месте
// v = {1, 99, 2, 3}
```

- `emplace(pos, args)` — вставляет элемент, конструируя его на месте из аргументов.
- В данном примере результат совпадает с `insert`.
- Сложность O(n).

Сравнение сложности операций

Операция	**Сложность**
Доступ по индексу (`operator[]`, `at`)	O(1)
`push_back`, `emplace_back`	Амортизированно O(1), отдельная операция — O(n)
`pop_back`	O(1)
`insert` в конец	Амортизированно O(1), при перевыделении — O(n)
`erase` последнего элемента	O(1)
`insert` / `erase` в середине	O(n)

- Вставка/удаление в конце — быстрые.
- Вставка/удаление в середине — медленные (сдвиг элементов).
- Доступ по индексу имеет постоянную сложность O(1).

Вопрос: что выведет этот код?

```
vector<int> v = {1, 2, 3};
v.push_back(4);
v.insert(v.begin(), 0);
v.pop_back();
for (int x : v) cout << x << " ";
```

Вопрос: что выведет этот код?

```
vector<int> v = {1, 2, 3};
v.push_back(4);
v.insert(v.begin(), 0);
v.pop_back();
for (int x : v) cout << x << " ";
```

Ответ: 0 1 2 3. Сначала добавили 4 → 1,2,3,4, вставили 0 в начало → 0,1,2,3,4, удалили последний → 0,1,2,3.

Что такое итератор?

Итератор — обобщённый указатель

- Итератор позволяет перебирать элементы контейнера, не зная его внутренней структуры.
- Он ведёт себя как указатель: поддерживает операции ++ (переход к следующему), * (доступ к значению), == и !=.
- В `vector` итератор — это, по сути, указатель на элемент (но в общем случае это может быть и более сложный объект).

Аналогия

- Для массива итератор — это указатель на элемент.
- `begin()` — указатель на первый элемент.
- `end()` — указатель за последним элементом.

begin() и end()

Синтаксис

```
vector<int> v = {10, 20, 30};
auto it = v.begin(); // итератор на первый элемент (10)
cout << *it;         // 10
it++;                // переход к следующему
cout << *it;         // 20
auto end = v.end();  // итератор за последним элементом
// *end -- нельзя разыменовывать!
```

- `begin()` возвращает итератор на первый элемент.
- `end()` возвращает итератор за последним элементом (не разыменовывается).
- Диапазон `[begin, end)` полуоткрытый.

Обход через итераторы

Цикл с итераторами

```
vector<int> v = {1, 2, 3, 4, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << " ";
}
// Вывод: 1 2 3 4 5
```

- Используем auto для удобства. За ним скрывается vector<int>::iterator.
- Условие останова: it != v.end().
- ++it переводит итератор на следующий элемент.
- Аналогия с указателями: for (int* p = arr; p != arr+n; ++p).

Range-based for

Синтаксис

```
vector<int> v = {1, 2, 3, 4, 5};
for (int x : v) {
    cout << x << " "; // копия элемента
}
for (int& x : v) {
    x *= 2; // изменение через ссылку
}
```

- Range-based for — это сокращённая запись обхода через итераторы.
- Компилятор автоматически вызывает begin() и end().
- Можно использовать const auto& для чтения без копирования.

Обратные итераторы: rbegin() и rend()

Синтаксис

```
vector<int> v = {1, 2, 3, 4, 5};
for (auto it = v.rbegin(); it != v.rend(); ++it) {
    cout << *it << " ";
}
// Вывод: 5 4 3 2 1
```

- rbegin() — итератор на последний элемент.
- rend() — итератор перед первым элементом (граница обратного обхода, не разыменовывается).
- При ++ обход идёт в обратном порядке.
- Очень удобно для вывода в обратном порядке.

Константные итераторы

Синтаксис

```
vector<int> v = {1, 2, 3};
auto it = v.cbegin(); // константный итератор (только чтение)
// *it = 10; // ОШИБКА!
for (auto it = v.cbegin(); it != v.cend(); ++it) {
    cout << *it << " "; // OK
}
```

- `cbegin()` и `cend()` возвращают итераторы, которые не позволяют изменять элементы.
- Полезны, когда нужно гарантировать неизменность данных.
- Для обычного `begin()` можно получить константный итератор, если вектор константный.

Что такое недействительный итератор?

Определение

Инвалидация — потеря действительности ранее полученного итератора вследствие операции над контейнером. После этого итератор нельзя использовать для доступа к элементам или продолжения обхода.

- Некоторые операции с вектором делают ранее полученные итераторы недействительными.
- Разыменование недействительного итератора — неопределённое поведение (UB).
- Такому итератору можно присвоить новое, действительное значение.

Не путайте с `end()`

Актуальный `end()` — действительный итератор, обозначающий позицию за последним элементом. Его нельзя разыменовывать, но можно использовать для сравнения при обходе.

Когда итераторы становятся недействительными?

- **Перевыделение памяти:** `push_back`, `insert`, `resize` (если `capacity` недостаточна) — все прежние итераторы, указатели и ссылки на элементы становятся недействительными.
- **Вставка без перевыделения** — недействительны итераторы в позиции вставки и после неё, включая прежний `end()`.
- **Удаление элементов:** `erase` делает недействительными все итераторы, начиная с удалённого и до конца, включая прежний `end()`.
- `clear()`: все прежние итераторы становятся недействительными.

Пример

```
vector<int> v = {1, 2, 3};
v.resize(v.capacity()); // теперь size() == capacity()
auto it = v.begin();

v.push_back(4); // необходимо перевыделение памяти
cout << *it;    // UB: старый итератор недействителен
```

Правило: После операции, которая могла инвалидировать итератор, получайте его заново.

assign(n, value)

Синтаксис

```
vector<int> v = {1, 2, 3};
v.assign(5, 10); // заменяет содержимое на 5 элементов, равных 10
// v = {10, 10, 10, 10, 10}
```

- `assign(n, value)` — заменяет всё содержимое вектора на `n` копий значения `value`.
- Размер становится `n`.
- Старые элементы удаляются.

assign(begin, end)

Синтаксис

```
vector<int> v = {1, 2, 3};
int arr[] = {5, 6, 7, 8};
v.assign(arr, arr + 4); // замена содержимым из массива
// v = {5, 6, 7, 8}
```

- `assign(first, last)` — заменяет содержимое элементами из диапазона `[first, last)`.
- Диапазон может быть из другого контейнера или массива.
- Размер становится равным длине диапазона.

assign(initializer_list)

Синтаксис

```
vector<int> v = {1, 2, 3};
v.assign({10, 20, 30}); // замена из списка
// v = {10, 20, 30}
```

- `assign(initializer_list)` — заменяет содержимое на элементы из списка.
- Удобно для быстрой замены на фиксированный набор.

assign_range (C++23)

Синтаксис (пример)

```
#include <vector>
#include <list>
vector<int> v = {1, 2, 3};
list<int> l = {10, 20, 30, 40};
v.assign_range(l); // замена содержимым из диапазона (list)
// v = {10, 20, 30, 40}
```

- `assign_range(range)` — заменяет содержимое элементами из любого диапазона (например, другого контейнера).
- Более универсален, чем старый `assign` с итераторами.

insert_range (C++23)

Синтаксис (пример)

```
#include <vector>
#include <list>
vector<int> v = {1, 2, 3};
list<int> l = {10, 20};
auto pos = v.begin() + 1;
v.insert_range(pos, l); // вставка элементов из list перед позицией pos
// v = {1, 10, 20, 2, 3}
```

- `insert_range(pos, range)` — вставляет все элементы из диапазона перед позицией `pos`.

append_range (C++23)

Синтаксис (пример)

```
#include <vector>
#include <list>
vector<int> v = {1, 2, 3};
list<int> l = {10, 20};
v.append_range(l); // добавление элементов из range в конец
// v = {1, 2, 3, 10, 20}
```

- append_range(range) — добавляет все элементы из диапазона в конец вектора.

Проблема удаления элементов

Стандартный способ

```
vector<int> v = {1, 2, 3, 2, 4, 2};
auto it = v.begin();
while (it != v.end()) {
    if (*it == 2) it = v.erase(it);
    else ++it;
}
```

- Ручной цикл с `erase` — громоздкий и подвержен ошибкам.
- Нужно правильно обновлять итератор.
- В C++20 появились `std::erase` и `std::erase_if`, которые упрощают удаление.

std::erase (C++20)

Синтаксис

```
#include <vector>
std::vector<int> v = {1, 2, 3, 2, 4, 2};
std::erase(v, 2); // удаляет все элементы, равные 2
// v = {1, 3, 4}
```

- `std::erase(container, value)` — удаляет все элементы, равные `value`.
- Просто и понятно.
- Для `std::vector` удаляет все элементы, равные заданному значению.

std::erase_if (C++20)

Синтаксис

```
#include <vector>
std::vector<int> v = {1, 2, 3, 4, 5, 6};
std::erase_if(v, [](int x){ return x % 2 == 0; }); // удаление всех чётных
// v = {1, 3, 5}
```

- `std::erase_if(container, pred)` — удаляет все элементы, удовлетворяющие предикату.
- Предикат — функция или лямбда, возвращающая `bool`.
- Очень мощный и удобный инструмент.

Сравнение подходов к удалению

Подход	Характеристика
Ручной цикл с `erase`	Громоздкий, легко ошибиться.
`std::erase`	Простое удаление по значению.
`std::erase_if`	Гибкое удаление по условию (предикат).

- `std::erase` и `std::erase_if` — рекомендуемый способ в C++20.
- Они делают код чище и безопаснее.

std::span — невладеющее представление

Что такое std::span?

- std::span (C++20) — это невладеющее представление непрерывной последовательности данных.
- Это как «окно» в массив или вектор без копирования данных.
- Позволяет передавать часть контейнера в функцию без копирования.

Создание span из vector

Синтаксис

```
#include <span>
#include <vector>
vector<int> v = {10, 20, 30, 40, 50};
span<int> sp(v); // span на весь вектор
span<int> sp2(v.data(), v.size()); // аналогично
for (int x : sp) cout << x << " "; // 10 20 30 40 50
```

- span можно создать из vector, массива, std::array.
- Не копирует данные, только хранит указатель и размер.

Когда использовать std::span?

- **Передача части массива в функцию** без копирования.
- **Единый интерфейс** для работы с C-массивами, `std::array`, `std::vector`.
- **Удобная работа с подмассивами**: указатель и размер передаются вместе.
- Пример: функция, принимающая `span<int>`, может работать с различными непрерывными последовательностями элементов типа `int`.

Важно

`span` не владеет данными. После уничтожения элементов или перевыделения памяти вектора прежнее представление может стать недействительным.

Пример работы с vector

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v;
    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.insert(v.begin() + 1, 15); // {10, 15, 20, 30}
    for (int x : v) cout << x << " "; // 10 15 20 30
    cout << endl;
    v.pop_back(); // {10, 15, 20}
    cout << "size: " << v.size() << ", capacity: " << v.capacity() << endl;
    return 0;
}
```

Вопрос: что выведет этот код?

```
vector<int> v = {1, 2, 3, 4, 5};
std::erase_if(v, [](int x){ return x > 3; });
for (int x : v) cout << x << " ";
```

Вопрос: что выведет этот код?

```
vector<int> v = {1, 2, 3, 4, 5};
std::erase_if(v, [](int x){ return x > 3; });
for (int x : v) cout << x << " ";
```

Ответ: 1 2 3. `erase_if` удалил все элементы, большие 3 (4 и 5).

Итоги лекции

- `std::vector` — динамический массив с автоматическим управлением памятью.
- Методы для работы с размером и ёмкостью: `size`, `capacity`, `reserve`, `resize`, `clear`.
- Добавление/удаление: `push_back`, `pop_back`, `insert`, `erase`, `emplace`.
- Итераторы — обобщённые указатели для обхода контейнера.
- Обратные итераторы `rbegin`/`rend`.
- `std::erase` и `std::erase_if` упрощают удаление.
- `std::span` — невладеющее представление для непрерывных данных.

Что дальше?

- Следующая лекция: строки (C-строки и `std::string`).
- Научимся работать с текстовыми данными, сравнивать, искать, заменять.
- На практике: использование `vector` для решения задач.

Вопросы?