1. Стандартная библиотека C++. Назначение, структура и основные принципы организации.

Назначение: Стандартная библиотека C++ предоставляет набор готовых компонентов для работы с данными, ввода-вывода, алгоритмами, контейнерами и другими часто используемыми функциями. Она упрощает разработку, повышает надежность и переносимость кода.

Структура стандартной библиотеки C++

1. Контейнеры (Containers): Хранение и управление коллекциями данных. Примеры: Последовательные: vector, list, deque, array**.** Ассоциативные: map, set, unordered_map

2. Алгоритмы (Algorithms): Стандартные операции над данными: сортировка, поиск, модификация. sort(), find()

3. Итераторы (Iterators): Унифицированный доступ к элементам контейнеров. Типы: input_iterator, forward_iterator, random_access_iterator и др.

4. Строки (Strings): Класс std::string для работы с текстом.

5. Потоки ввода-вывода (I/O Streams): iostream, fstream

6. Многопоточность (Threading): thread, mutex, atomic,futur

Основные принципы организации

1.Обобщенное программирование (Generics): Шаблоны (templates) позволяют создавать универсальные компоненты (нап: , конт_нры vector<T> для любого типа T).

2. Итераторы как абстракция доступа

Единый интерфейс для работы с разными контейнерами.

3. Алгоритмы, независимые от контейнеров: Алгоритмы работают через итераторы, а не напрямую с контейнерами.

2. Контейнер vector. Назначение и основные принципы устройства контейнера.

Назначение: std::vector — это динамический массив, который автоматически управляет памятью, позволяя хранить элементы одного типа в непрерывной области памяти. Он обеспечивает: 1. Быстрый доступ к элементам по индексу (за O(1)). 2.Динамическое изменение размера (в отличие от статических массивов). 3.Эффективные операции в конце (push_back, pop_back).

Основные принципы устройства

1.Динамический массив: Элементы хранятся в непрерывном блоке памяти, как в обычном массиве.

2.Автоматическое управление памятью

3. Три ключевых параметра

4.Итераторы и совместимость с алгоритмами STL

3. Контейнер vector. Типы данных, итераторы, доступ к элементам, конструирование. Размер и емкость.

1. Типы данных (Data Types)

```
#include <vector> std::vector<int> vec1; std::vector<std::string> vec2;  std::vector<double> vec3;
```

2. Итераторы:

```
std::vector<int> vec = {10, 20, 30};
for (auto it = vec.begin(); it != vec.end(); ++it) {
   std::cout << *it << " "; }
std::sort(vec.begin(), vec.end());
```

3. Доступ к элементам (Element Access)

Вектор предоставляет несколько способов доступа к элементам:

```
std::vector<int> vec = {1, 2, 3}; std::cout << vec[0];
std::cout << vec.at(1);  // 2 (проверка границ)
std::cout << vec.back(); // 3
```

4. Конструирование (Constructors): Вектор можно создать разными способами:

```
std::vector<int> empty;              // {}
std::vector<int> sized(3);           // {0, 0, 0}
std::vector<int> filled(3, 42);      // {42, 42, 42}
std::vector<int> copied(filled);     // {42, 42, 42}
std::vector<int> ranged(filled.begin(), filled.end()); // {42, 42,42}
std::vector<int> init_list = {1, 2, 3}; // {1, 2, 3}
```

5. Размер и ёмкость (Size & Capacity)

Вектор управляет двумя параметрами: 1. size() — текущее количество элементов. 2. capacity() — размер выделенной памяти (≥ size()).

```
std::vector<int> vec;
vec.push_back(1); vec.push_back(2); vec.push_back(3);
std::cout << vec.size();     // 3
std::cout << vec.capacity();  // 4 (зависит от реализации)
vec.reserve(100);
std::cout << vec.capacity();  // 100
```

4. Контейнер vector. Стековые и списочные операции.

1. Стековые операции (Stack-like Operations)

Вектор может использоваться как стек (LIFO — Last In, First Out), так как операции с концом (push_back, pop_back) выполняются за O(1) (амортизированное время).

```
int main() {    std::vector<int> stack;
   stack.push_back(1);  // [1]    stack.push_back(2);  // [1, 2]
   stack.push_back(3);  // [1, 2, 3]
   std::cout << stack.back() << "\n";  // 3
   stack.pop_back(); // [1, 2]     std::cout << stack.back() << "\n";  // 2 }
```

2. Списочные операции (List-like Operations)

Вектор не является оптимальным для частых вставок/удалений в середину, так как требует O(n) времени из-за сдвига элементов.

```
int main() {    std::vector<int> vec = {10, 20, 30};
   // Вставка в середину
   vec.insert(vec.begin() + 1, 99);  // [10, 99, 20, 30]
   // Удаление из середины
   vec.erase(vec.begin() + 2);      // [10, 99, 30]
   for (int x : vec) {
      std::cout << x << " ";  // 10 99 30    } }
```

5. Специализация vector.

Стандартная библиотека C++ позволяет специализировать std::vector для: 1. Пользовательских типов данных (классы, структуры). 2.Оптимизации под конкретные сценарии

(например, vector<bool>). 3. Использования кастомных аллокаторов.

1. Специализация для пользовательских типов vector может хранить любые типы, включая:

1.Объекты классов. 2. Указатели. 3.std::pair, std::tuple.

```
struct Person {    std::string name;    int age; };
int main() { std::vector<Person> people;
people.push_back({"Alice", 30}); people.push_back({"Bob", 25});
(const auto& p : people) {
      std::cout << p.name << " (" << p.age << ")\n";     }}
```

2. Специализация vector<bool>

std::vector<bool> — это частичная специализация, которая хранит bool в упакованном виде (1 бит на элемент), что экономит память, но имеет особенности:

```
int main() {   std::vector<bool> bits = {true, false, true};
  bits[1] = true;  // Изменение второго бита
  for (bool b : bits) {   std::cout << b << " ";  // 1 1 1    } }
```

3. Специализация для указателей и умных указателей

vector может хранить: 1. Обычные указатели (int*).

2. Умные указатели (std::unique_ptr, std::shared_ptr).

```
int main() {    std::vector<std::unique_ptr<int>> ptrs;
   ptrs.push_back(std::make_unique<int>(10));
   ptrs.push_back(std::make_unique<int>(20));
   for (const auto& ptr : ptrs) {
      std::cout << *ptr << " ";  // 10 20    } }
```

6. Контейнеры стандартной библиотеки. Классификация и перечень.

Контейнеры STL— это шаблонные классы, предназначенные для хранения и управления коллекциями данных. Они делятся на последовательные, ассоциативные, неупорч-енные (хеш-таблицы) и адаптеры.

Классификация контейнеров

1.Последовательные контейнеры (Sequential): Хранят элементы в линейном порядке, но без гарантии сортировки: vector, deque, list, forward_list,Односвязный, список, array

2. Ассоциативные контейнеры (Associative): Хранят элементы в отсортированном порядке (по умолчанию <). Основаны на бинарных деревьях. Set, multiset, map, multimap

3. Неупорядоченные контейнеры (Unordered , Hash Tables): Хранят элементы в хеш-таблицах.: unordered_set, unordered_multiset, unordered_map , unordered_multimap

4. Адаптеры контейнеров (Adapters): Ограниченные интерфейсы поверх других контейнеров.stack, queue

7. Контейнеры стандартной библиотеки. Требования к элементам контейнеров.

Типы данных, которые хранятся в контейнерах стандартной библиотеки, должны удовлетворять определённым требованиям. Эти требования зависят от типа контейнера и операций, которые над ними выполняются.

Общие требования для всех контейнеров:

- Конструктор по умолчанию (для некоторых операций, например, resize()).
- Конструктор копирования и оператор копирующего присваивания.
- Деструктор.

Для последовательных контейнеров (vector, deque, list, forward_list):

for
- Тип элемента должен быть копируемым или перемещаемым.
- Для некоторых операций — наличие конструктора по умолчанию.
- Для алгоритмов поиска и сортировки — наличие операторов сравнения (например, ==, <).

8. Контейнер list. Назначение, принципы устройства, основные операции.

Назначение: std::list — это двусвязный список, который предоставляет эффективные операции вставки и удаления элементов в любом месте за время O(1).

1. Принципы устройства

Двусвязная структура: Каждый элемент (node) содержит: 1.Данные. 2. Указатель на следующий элемент.3. Указатель на предыдущий элемент (prev).

Непрерывность памяти отсутствует: Элементы разбросаны в памяти, нет реаллокаций.

Итераторы: Подд_вают двунаправленный обход (++, --)

2. Основные операции:

Общие: size(), empty(), max_size() — работа с размером.

Итераторы: begin(), end() — прямые итераторы; rbegin(), rend() — обратные итераторы.

Стековые операции: push_front(), pop_front(push_back(), pop_back(emplace_front(), emplace_back(

Списочные операции: insert(),emplace(),erase(),clear()

Дополнительные специфические операции: splice(), merge(), unique(),remove(), remove_if(), reverse()

9. Контейнер deque. Назначение, принципы устройства, основные операции.

Контейнер deque (double-ended queue): это дина-ческий контейнер C++, сочетающий свойства vector и list.

Назначение: 1. эффективное добавление и удаление элементов как в начало, так и в конец; 2. доступ к эле-там по индексу с константной сложностью;

3. применим, когда нужны частые вставки в обе стороны и быстрый доступ по индексу.

Устройство: 1. хранит элементы в виде набора блоков фиксированного размера (обычно 512 или 1024 элемента);

2. дополнительно используется структура map, которая хранит указатели на блоки;

3.новые блоки добавляются по мере необходимости в начало или конец;

Доступ к элементам: 1.поддерживается случайный доступ

через operator[] и at(), обе — O(1);
2. итераторы случайного доступа.

Основные операции: `push_front()`, `push_back()`. `pop_front()`, `pop_back()` — удаление с начала и конца `insert()`, `erase()` — вставка и удаление внутри контей `clear()` — очистка (O(n));

Алгоритмическая сложность:

1.доступ по индексу — O(1); **2.** вставка и удаление в середине — O(n); **3.** вставка и удаление в начале — O(1).

10. Адаптеры стандартных контейнеров. Стек.

stack — это адаптер стандартных контейнеров C++, реализующий структуру данных "стек" (LIFO)

Устройство и назначение: 1. stack не являетсясамостоя-ьным контейнером — он использует другой
контейнер внутри (по умолчанию — deque);
2. меняет интерфейс базового контейнера: оставляет только операции для работы как со стеком.

Основные операции:

push(value) — добавить элемент на вершину;
pop() — удалить верхний элемент; top(),empty()size()

Контейнер-основа: по умолчанию: deque<T>, но можно использовать vector<T> или list<T>, если они поддерживают push_back(), pop_back() и back().

Сложность операций: все операции — O(1)

11. Адаптеры стандартных контейнеров. Очередь.

queue — адаптер стандартных контейнеров, реализующий структуру данных "очередь" (FIFO) **Устройство и назначение:** 1.queue также использует другой контейнер (по умолчанию — deque); 2. предоставляет интерфейс для работы строго как с очередью. **Основные операции:**

push(value) — добавить элемент в конец;
pop() — удалить элемент из начала;
front() — получить доступ к первому элементу;
back() — доступ к последнему элементу;
empty(), size() — проверка и размер.

Контейнер-основа: 1.по умолчанию: deque<T>, может использовать list<T>, если поддерживаются push_back(), pop_front(), front() и back().

Сложность операций: все операции — O(1).

12. Адаптеры стандартных кон-ров. Очередь с приоритетом.

priority_queue — адаптер стандартных контейнеров, реализующий приоритетную очередь (обычно max-heap).

Устройство и назначение: 1.по умолчанию реализуется на базе vector<T>, с поддержкой операций кучи;

2. в priority_queue всегда

доступен элемент с наивысшим приоритетом (по умолчанию максимальный элемент).**Основные операции:** `push(value)`, `pop()`, `top()`, `empty()`, `size()`. **Контейнер-основа:** по умолчанию: `vector<T>`, допускаются и `deque<T>`, если поддерживаются итераторы случайного доступа.

Сравнение: используется компаратор (по умолчанию `std::less<T>`, что даёт max-heap);
2.можно передавать пользовательский компаратор для изменения порядка приоритетов.

13. Ассоциативный контейнер map. Назначение, принципы устройства, основные операции.

map — это ассоциативный контейнер C++, хранящий пары ключ-значение (std::pair<const Key, T>) с уникальными ключами и упорядочиванием по ключу.

Назначение: 1. хранение данных в виде пар (ключ, значение); 2. быстрый поиск, вставка и удаление по ключу с логарифмической сложностью.

Устройство: 1.основан на самобалансирующихся деревьях поиска (обычно красно-черное дерево);
2. обеспечивает строгое упорядочивание элементов по ключу при каждой вставке;
3. итераторы — двунаправленные.

Основные операции: 1. *Доступ к элементам:*
1.1.operator[] — возвращает ссылку на значение по ключу; если ключ отсутствует, создаёт новый элемент с этим ключом и значением по умолчанию.
1.2 at() — аналогично, но выбрасывает исключение out_of_range при отсутствии ключа.
Итераторы: 1. begin(), end() — прямой обход по возрастанию ключей.
2. rbegin(), rend() — обратный обход.

***Вставка**:*
1.insert() — вставка пары ключ-значение; возвращает пару (итератор, bool), где bool показывает успешность вставки.
2.insert(hint, value) — вставка с подсказкой (оптимизация для уже отсортированных данных).
3.insert(first, last) — вставка диапазона элементов.

***Удаление**:*
1.erase(pos) — удаляет элемент по итератору.
2.erase(key) — удаляет элемент по ключу.
3. erase(first, last) — удаляет диапазон.

Поиск:
1.find(key) — возвращает итератор на элемент с ключом.
2.count(key) — возвращает количество элементов с данным ключом (у map — всегда 0 или 1).
3.lower_bound(key), upper_bound(key), equal_range(key) — поиск диапазона по ключу.

Сложность операций:

1. все основные операции (поиск, вставка, удаление) — O(log n) за счёт структуры дерева;
2. случайный доступ operator[] — тоже O(log n).

14. Ассоциативный контейнер set. Назначение, принципы устройства, основные операции.

set — это ассоциативный контейнер C++, хранящий уникальные элементы, которые сами являются ключами.

Назначение: 1. хранение множества уникальных значений с автоматическим поддержанием порядка;

2. быстрый поиск, вставка и удаление элементов.

Устройство: 1. реализуется на основе самобалансирующегося дерева поиска (обычно красно-черное дерево, как и map);

2.каждый элемент одновременно является ключом и значением; 3. итераторы — двунаправленные.

Основные операции:

Доступ к элементам:

*1.*прямого доступа по ин-ксу нет (отсутствуют operator[] и at());

*2.*доступ осуществляется только через итераторы.

Итераторы: begin(), end(), rbegin(), rend() — стандартные итераторы по отсортированному набору элементов.

Вставка: **insert(value)** — добавляет элемент; если такой уже есть, вставка игнорируется;

insert(hint, value) — вставка с подсказкой (оптимизация для отсортированных данных); **insert(first, last)** — вст-а диапазона.

Удаление: **erase(pos)** — удаление по итератору;

erase(value) — удаление по значению;

erase(first, last) — удаление диапазона.

Поиск: **find(value)** — поиск элемента;

count(value) — вернёт 0 или 1 (так как все эле-ты уникальны);

lower_bound(value), upper_bound(value), equal_range(value) — стандартный поиск диапазонов.

Сложность операций: все вставки, уда-ния, поиска — O(log n).

15. Ассоциативные контейнеры multimap и multiset. Назначение, принципы устройства и основные отличия от контейнеров map и set.

multimap и **multiset** — это ассоциативные контейнеры, в которых допускаются дубликаты ключей.

multimap:Назначение: хранение пар (ключ, значение) с возможностью иметь несколько элементов с одинаковым ключом. **Устройство:** **1.**реализуется на основе самобалансирующегося дерева поиска (обычно красно-черное дерево); 2. элементы хранятся как pair<const Key, T>.

Основные отличия от map:

1.допускает повторяющиеся ключи;

2.отсутствует operator[] и at();

3. доступ ко всем элементам по ключу осуществляется через equal_range(), lower_bound() и upper_bound().

Сложность операций: все вставки, удаления, поиска — O(log n).

multiset: Назначение: хранение набора значений, допускающего дубликаты.

Устройство: аналогично set, но допускает несколько одинаковых элементов.

Основные отличия от set:

1.допускает несколько одинаковых значений;

2.отсутствует operator[] и at() (как и у set);

3.доступ к элементам — только через итераторы.

Сложность операций: все вставки, удаления, поиска — O(log n).

Таким образом: map — уникальные ключи, пары (ключ, значение); multimap — ключи могут повторяться;

- set — уникальные значения = ключи;
- multiset — значения (ключи) могут повторяться.

16. Понятие итератора. Основные принципы.

Итератор в стандартной библиотеке C++ — это абстракция указателя на элемент последовательности. Итераторы обеспечивают универсальный доступ к элементам контейнеров, независимо от их внутренней структуры. **Ключевые принципы:**

1. Разыменование элемента: *iterator — получение значения; iterator->member — доступ к члену.

2. Переход по элементам: инкремент ++iterator (движение вперёд). 3. Проверка на равенство: iterator1 == iterator2. 4. Итераторы можно копировать.

Отличие от указателя:

1.Итератор не имеет значения "null" (может быть end()).

2.Нельзя разыменовывать недействительные итераторы (например, после удаления элемента, выхода за границы и т.д.).

3.Любой корректный итератор указывает на существующий элемент контейнера или на конец (end()).

Примеры итераторов:

- int* — итератор для массива int[];
- list<int>::iterator — итератор для list;
- vector<string>::const_iterator — константный итератор для vector.

17. Категории итераторов.

Итераторы в стандартной библиотеке C++ классифицируются по набору поддерживаемых операций:

Итераторы ввода (InputIterator):

1.Однократное чтение элементов. 2. Операции: разыменование (*it), переход (++it), сравнение (==, !=).

3. Используются, например, при чтении из потока (istream_iterator).

Итераторы вывода (OutputIterator):

1.Однократная запись. 2. Операции: разыменование для записи (*it = value), переход (++it). 3.Используются, например, при записи в поток (ostream_iterator).

Прямые итераторы (ForwardIterator):

1.Повторный проход вперёд. 2.Поддерживают все операции ввода и вывода, копирование итераторов.

Двунаправленные итераторы (BidirectionalIterator):

1.Поддерживают перемещение в обе стороны (++it, --it).

2.Используются, например, в list, set, map.

Итераторы случайного доступа (RandomAccessIterator):
1.Поддерживают арифметику (it + n, it - n, it1 < it2 и др.).
2.Обеспечивают быстрый доступ по индексу: it[n].
3.Используются в vector, deque, массивах.
Расстояние между итераторами:
1.Для измерения количества элементов между двумя итераторами используется функция distance().
2. Перемещение на n позиций выполняется через advance().

18. Обратные итераторы.
Обратный итератор (reverse_iterator) — это обёртка над обычным итератором, которая позволяет просматривать элементы контейнера в обратном порядке.
Назначение: 1.Позволяет использовать существующие алгоритмы STL, но идти в обратном направлении.
2.Не требует ручной реализации прохода назад.
Особенности: 1.Разыменование *rit даёт тот элемент, который расположен перед текущим положением базового итератора.
2.Инкремент ++rit фактически декрементирует базовый итератор. 3. Декремент --rit — наоборот: увеличивает базовый итератор. **Создание:** 1.Метод rbegin() возвращает обратный итератор на последний элемент.
2.Метод rend() возвращает обратный итератор на позицию "до первого элемента".**Пример:**
```
std::vector<int> v = {1, 2, 3, 4, 5};
for (auto rit = v.rbegin(); rit != v.rend(); ++rit)
   std::cout << *rit << " "; // Выведет: 5 4 3 2 1
```
Обратные итераторы поддерживают:
1.все те же категории, что и их базовые итераторы (например, reverse_iterator<RandomAccessIterator>);
2.безопасно работают с алгоритмами STL (например, std::copy(rbegin(), rend(), ...)).

19. Потоковые итераторы (ввод и вывод).
Потоковые итераторы позволяют связать стандартные потоки ввода и вывода с алгоритмами STL. **Назначение:**
Организовать удобный ввод/вывод данных через стандартные алгоритмы, не писать явных циклов.
Типы потоковых итераторов:
1.istream_iterator<T>
>Предоставляет итератор для чтения из потока (std::cin, ifstream и др.). **>**При каждом разыменовании считывает следующий элемент из потока.**Пример:**
```
std::istream_iterator<int> in(std::cin);
std::istream_iterator<int> end; std::vector<int> v(in, end);
```
2.ostream_iterator<T>
>Предоставляет итератор для записи в поток (std::cout, ofstream и др.).
>Каждое присваивание записывает значение в поток.
>Можно задать разделитель (например, пробел).**Пример:**
```
std::ostream_iterator<int> out(std::cout, " ");
std::copy(v.begin(), v.end(), out);
```
Особенности: 1.Работают как OutputIterator (для вывода) или
InputIterator (для ввода).
2.Позволяют легко связывать потоки с алгоритмами STL.

20. Понятие аллокатора. Основные принципы.
Аллокатор (allocator) — это объект, который управляет выделением и освобождением памяти для контейнеров STL. Контейнеры используют аллокаторы для изоляции от низкоуровневых деталей работы с памятью.
Назначение аллокатора: 1.обеспечивает унифицированный интерфейс выделения и освобождения памяти; 2.позволяет заменить стандартную стратегию управления памятью (например, использовать пул памяти, файл или специально оптимизированные схемы);
3.обеспечивает переносимость и гибкость контейнеров.
Обязательные члены аллокатора:
value_type — тип хранимых объектов;
size_type, difference_type — типы размера и разности;
pointer, const_pointer — указатели на элементы;
reference, const_reference — ссылки на элементы;
allocate(n) — выделяет память для n объектов;
deallocate(p, n) — освобо-ает ранее выделенную память;
construct(p, args...) — вызывает конструктор объекта
destroy(p) — вызывает деструктор объекта.
Стандартный аллокатор:
>В стандартной библиотеке определён std::allocator<T>, который работает поверх new и delete, вызывая их для выделения и освобождения памяти.
>Исп-ется по умолчанию во всех стан-тных контейнерах.
Пример производительности:
>пул памяти работает быстрее стандартного аллокатора;
>аллокатор с файловой системой намного медленнее из-за обращения к диску.
Таким образом: аллокаторы обеспечивают полную гибкость в управлении памятью контейнеров, позволяя подстраивать стратегию выделения памяти под конкретные задачи.

21. Алгоритмы стандартной библиотеки. Основные принципы и классификация.
Алгоритмы стандартной библиотеки C++ — это обобщённые функции, которые работают с контейнерами через итераторы, не завися от типа конкретного контейнера.
Принципы: 1. Все алгоритмы реализованы как шаблонные функции. 2.Работают с итераторами любых категорий. 3. Не имеют доступа к внутреннему устройству контейнеров. 4. Не проверяют границы контейнеров.

5. Работают как с контейнерами STL, так и с обычными массивами C++ (указатели работают как итераторы).

Классификация алгоритмов:

1.Немодифицирующие (анализируют, но не изменяют контейнер): find, for_each, count, equal, mismatch и др.

2. Модифицирующие (изменяют содержимое): copy, remove, replace, transform, fill, generate и др. 3.Алгоритмы сортировки: sort, stable_sort, partial_sort, nth_element. 4.Алгоритмы поиска: binary_search, lower_bound, upper_bound. 5. Алгоритмы слияния и разбиения: merge, partition. 6. Алгоритмы над множествами: set_union, set_intersection, set_difference и др.

22. Функциональные объекты (функторы). Предикаты.

Функциональный объект (функтор) — это объект класса, который реализует оператор operator(). Благодаря этому объект можно вызывать как функцию. Функторы гибко настраиваются и позволяют хранить состояние между вызовами. **Пример:** struct Sum { int total = 0;
void operator()(int x) { total += x; } };

Такие объекты часто передаются в алгоритмы для задания логики обработки элементов.

Предикаты — это частный случай функтора или функции, возвращающей bool. Они определяют критерий выбора элементов:

>Унарный предикат: принимает 1 аргумент, возвращает bool. Например: [](int x) { return x % 2 == 0; } (проверка на чётность).

>Бинарный предикат: при-мает 2 аргумента, возвращает bool.

>Например: [](int a, int b) { return a < b; } —
может использоваться в сортировке или сравнении.

Зачем нужны предикаты и функторы в алгоритмах:

>позволяют управлять логикой поиска, подсчёта, сортировки и фильтрации;

>обеспечивают обобщённость алгоритмов;

23. Немодифицирующие алгоритмы (for_each, count, equal, mismatch).

Немодифицирующие алгоритмы читают и анализируют содержимое контейнеров, но не изменяют его.

for_each(first, last, func)

Применяет фу-ию func ко всем эле-там диапазона [first, last). Можно использовать для подсчёта, вывода, накопления и др.

Пр-ер: std::for_each(v.begin(), v.end(), [](int x){ std::cout << x; });

count(first, last, value) Считает кол-ство эле-тов, равных value.

count_if(first, last, pred)

Считает ко-тво элементов, удовлетворяющих предикату pred.

Пример: int n = std::count_if(v.begin(), v.end(), [](int x){ return x > 0; });

equal(first1, last1, first2) Сравнивает две последовательности попарно на равенство.

mismatch(first1, last1, first2)

Находит первую позицию, где элементы двух последо-ностей различаются.Возвращает пару итераторов на эти элементы.

24. Немодифицирующие поисковые алгоритмы.

Эти алгоритмы выполняют поиск элементов или подпоследовательностей, не изменяя содержимого.

find(first, last, value):Ищет первое вхождение элемента value. Если не найден — возвращает last.

find_if(first, last, pred): Ищет первый элемент, удовлетв-ющий предикату pred.

find_if_not(first, last, pred):Ищет первый элемент, НЕ удовлетворяющий предикату.

find_first_of(first1, last1, first2, last2): Ищет первый элемент из первой последовательности, который есть во второй. Можно передать бинарный предикат для настройки критерия совпадения.

find_end(first1, last1, first2, last2)

То же самое, что search, но воз-щает посл-нее вхож-ние.

search_n(first, last, count, value): Ищет первую группу подряд идущих одинаковых эле-ов value длиной count.

25. Модифицирующие алгоритмы (копирующие алгоритмы,transform, unique).

Модифицирующие алгоритмы изменяют содержимое контейнеров или создают новые контейнеры на основе преобразования существующих данных.

Копирующие алгоритмы: 1.copy(first, last, res) — копирует все элементы из диапазона в новый.

2.copy_n(first, count, res) — копирует count элементов.

3.copy_if(first, last, res, pred) — копирует только те элементы, которые удовлетворяют предикату pred.

4.copy_backward(first, last, res) — копирует элементы в обратном порядке. **Алгоритм transform:**

1.transform(first, last, res, op) — применяет операцию op к каждому элементу и записывает результат в res.

2.transform(first1, last1, first2, res, binary_op) — применяет бинарную операцию к парам элементов из двух последовательностей. >Часто используется для математических преобразований, масштабирования данных, вычисления новых значений.

Алгоритм unique: 1.unique(first, last) — удаляет подряд идущие одинаковые элементы (оставляет по одному).

2.unique(first, last, pred) — аналогично, но с бинарным предикатом. 3. unique не физически удаляет элементы из контейнера: он сдвигает элементы влево, возвращая итератор на новый конец. Окончательное удаление происходит через erase.

Алгоритм unique_copy: 1. Копирует элементы, удаляя подряд идущие дубликаты, в новую последова-льность.

26. Модифицирующие алгоритмы (замена и удаление элементов).

Алгоритмы замены:

1.replace(first, last, old_value, new_value) — заменяет все вхождения old_value на new_value.

2.replace_if(first, last, pred, new_value) — заменяет элементы, удовлетворяющие pred, на new_value.

Алгоритмы удаления:

1. remove(first, last, value) — уда-ет все вхождения value.

2.remove_if(first, last, pred) — удаляет элементы, удовлетворяющие предикату pred.

Как и в unique, фактическое удаление происходит через erase(remove(...), end()).

27. Модифицирующие алгоритмы (fill, generate, reverse, rotate).

Алгоритмы генерации: 1.fill(first, last, value) — заполняет весь диапазон заданным значением.

2.fill_n(first, count, value) — заполняет первые count элем-тов.

3.generate(first, last, generator) — заполняет значениями, возвращаемыми функцией generator.

4.generate_n(first, count, generator) — аналогично для первых count элементов.

Алгоритмы изменения порядка:

1.reverse(first, last) — разворачивает порядок элементов.

2.reverse_copy(first, last, res) — копирует элементы в обратном

3.rotate(first, middle, last) — циклически сдвигает диапазон так, что middle становится началом.

4.rotate_copy(first, middle, last, res) — выполняет сдвиг и копирует в новую последовательность.

Алгоритмы обмена:

1.swap(a, b) — обменивает значения двух переменных.

2.swap_ranges(first1, last1, first2) — обменивает элементы двух диапазонов (размер должен совпадать).

rotate часто используется при реализациях перестановок, сдвигов, циклических вращений.

28. Алгоритмы сортировки. Бинарный поиск.

Сортировка:

1.sort(first, last). 2. sort(first, last, comp): Позволяет задать собственный бинарный предикат компаратора для сортировки по своим правилам.

3.stable_sort(first, last): Гарантирует сохра-ие относительного порядка равных элементов (стабильная сортировка).

4.partial_sort(first, middle, last): Сортирует только middle - first первых элементов, остальные остаются произвольными.

Бинарный поиск:

Алгоритмы бинарного поиска применяются к отсортированным диапазонам: 1.binary_search(first, last, value): Проверяет наличие элемента value.

2.lower_bound(first, last, value)

Возвращает итератор на первый элемент, не меньший value.

3. upper_bound(first, last, value): Возвращает итератор на первый элемент, больший value.

4.equal_range(first, last, value): Возвращает пару итераторов — границы диапазона элементов, равных value.

29. Алгоритмы слияния и разбиения (merge, partition).

Слияние: 1. merge(first1, last1, first2, last2, res) Объединяет два отсортированных диапазона в один отсортированный. Требует предварительной сортировки обоих диапазонов. **2.** inplace_merge(first, middle, last) Выполняет слияние двух отсортированных частей одного диапазона прямо на месте, без выделения дополнительной памяти.

Разбиение (partition): 1. partition(first, last, pred) Переставляет элементы так, что сначала идут все, удовлетворяющие предикату pred, затем — остальные. Порядок внутри групп не сохраняется.

2. stable_partition(first, last, pred): То же самое, но сохраняет относительный порядок элементов внутри каждой из групп.

Пример использования partition:

std::partition(v.begin(), v.end(), [](int x){ return x % 2 == 0; });
Разделит элементы на чётные и нечётные.

30. Алгоритмы, реализующие операции над множествами (includes, set_union, set_intersection, set_difference, set_symmetric_difference).

Работают с отсортированными диапазонами.

- includes(first1, last1, first2, last2) Проверяет, содержатся ли все элементы второго множества в первом.
- set_union(first1, last1, first2, last2, res) Записывает объединение двух множеств.
- set_intersection(first1, last1, first2, last2, res) Записывает пересечение множеств.
- set_difference(first1, last1, first2, last2, res) Записывает разность множеств (элементы первого, отсутствующие во втором).
- set_symmetric_difference(first1, last1, first2, last2, res) Записывает симметрическую разность (элементы, входящие только в одно из двух множеств).

1. Стандартная библиотека C++. Назначение, структура и основные принципы организации.
2. Контейнер vector. Назначение и основные принципы устройства контейнера.
3. Контейнер vector. Типы данных, итераторы, доступ к элементам, конструирование. Размер и емкость.
4. Контейнер vector. Стековые и списочные операции.
5. Специализация vector.
6. Контейнеры стандартной библиотеки. Классификация и перечень.
7. Контейнеры стандартной библиотеки. Требования к элементам контейнеров.
8. Контейнер list. Назначение, принципы устройства, основные операции.
9. Контейнер deque. Назначение, принципы устройства, основные операции.
10. Адаптеры стандартных контейнеров. Стек.
11. Адаптеры стандартных контейнеров. Очередь.
12. Адаптеры стандартных контейнеров. Очередь с приоритетом.
13. Ассоциативный контейнер map. Назначение, принципы устройства, основные операции.
14. Ассоциативный контейнер set. Назначение, принципы устройства, основные операции.
15. Ассоциативные контейнеры multimap и multiset. Назначение, принципы устройства и основные отличия от контейнеров map и set.
16. Понятие итератора. Основные принципы.
17. Категории итераторов.
18. Обратные итераторы.
19. Потоковые итераторы (ввод и вывод).
20. Понятие аллокатора. Основные принципы.
21. Алгоритмы стандартной библиотеки. Основные принципы и классификация.
22. Функциональные объекты (функторы). Предикаты.
23. Немодифицирующие алгоритмы (for_each, count, equal, mismatch).
24. Немодифицирующие поисковые алгоритмы.
25. Модифицирующие алгоритмы (копирующие алгоритмы, transform, unique).
26. Модифицирующие алгоритмы (замена и удаление элементов).
27. Модифицирующие алгоритмы (fill, generate, reverse, rotate).
28. Алгоритмы сортировки. Бинарный поиск.
29. Алгоритмы слияния и разбиения (merge, partition).
30. Алгоритмы, реализующие операции над множествами (includes, set_union, set_intersection, set_difference, set_symmetric_difference).

Контейнер	Назначение	Доступ	Вставка	Удаление	Поиск
vector	Динамический массив	O(1)	O(1) в конец, O(n) в середину	O(n)	O(n)
deque	Двусторонняя очередь	O(1)	O(1) в начало и конец, O(n) в середину	O(1)/O(n)	O(n)
list	Двунаправленный список	O(n)	O(1) (по итератору)	O(1) (по итератору)	O(n)
forward_list	Односвязный список	O(n)	O(1) (после текущей позиции)	O(1) (после текущей позиции)	O(n)
set	Множество	O(log n)	O(log n)	O(log n)	O(log n)
multiset	Множество с дубликатами	O(log n)	O(log n)	O(log n)	O(log n)
map	Ассоциативный массив	O(log n)	O(log n)	O(log n)	O(log n)
multimap	Ассоц. массив с дубликатами	O(log n)	O(log n)	O(log n)	O(log n)
unordered_set	Хеш-множество	O(1)	O(1)	O(1)	O(1)
unordered_multiset	Хеш-множество с дубликатами	O(1)	O(1)	O(1)	O(1)
unordered_map	Хеш-ассоц. массив	O(1)	O(1)	O(1)	O(1)
unordered_multimap	Хеш-ассоц. массив с дубликатами	O(1)	O(1)	O(1)	O(1)
stack	Стек (LIFO)	—	O(1)	O(1)	—
queue	Очередь (FIFO)	—	O(1)	O(1)	—
priority_queue	Очередь с приоритетом	—	O(log n)	O(log n)	—