Программирование на C++

Содержание

Идиома RAII

RAII (Resource Acquisition Is Initialization) — идиома, при которой ресурс (память, файл, сокет, мьютекс) захватывается в конструкторе и освобождается в деструкторе. Это гарантирует, что ресурс будет освобождён при любом выходе из области видимости — нормальном завершении, `return` посередине функции, или раскрутке стека при исключении.

Проблема без RAII: код может закончить работу до того, как вызовется `delete` — например, из-за раннего `return` или исключения. Память утечёт.

```
// Плохо: если return сработает раньше delete — утечка
void func() {
    int* ptri = new int(5);
    return;
    /* ... */
    delete ptri;  // никогда не выполнится
}
```

Решение — обернуть указатель в класс, который сам управляет временем жизни ресурса:

```
// Используя идиому RAII
class AutoPtr {
```

```
public:
    AutoPtr(int value)
        : value_(new int(value))  // в списке инициализации можно вызывать new
    {}

    ~AutoPtr() {
        delete value_;
    }

    int operator*() {
        return *value_;
    }
private:
    int* value_;
};

void func() {
    AutoPtr ptr(5);
    return;  // деструктор сам освободит память
}
```

Конструктор копирования и оператор присваивания

- Чтобы поддерживать цепочки присваиваний (`a = b = c`), оператор = должен возвращать `*this` по ссылке.
- Дефолтный конструктор копирования просто скопирует указатель — а потом деструктор каждого из двух объектов попытается освободить **одну и ту же память**. Это **double-free** → undefined behavior.
- Поэтому конструктор копирования должен **сам выделять новую память** и копировать в неё значение.

```
AutoPtr(const AutoPtr& other)
    : value_(new T(*(other.value_)))
{}

AutoPtr& operator=(const AutoPtr& other) {
    if (&other == this) {        // защита от самоприсваивания
```

```
        return *this;
    }
    delete value_;                // освобождаем старое
    value_ = new T(*(other.value_));  // копируем новое
    return *this;
}
```

Если же указатель не сам создаёт объект, а ему **передают владение** уже существующим указателем — есть три стратегии копирования.

Стратегия 1. «Передаю владение ресурсом» — `std::auto_ptr` (устарел)

Передаём значение, а у источника обнуляем — теперь владелец только один. Это стандарт C++98, `std::auto_ptr`. **Был удалён в C++17** из-за коварства: владение могло «незаметно» утечь, например при копировании внутрь контейнера.

```
AutoPtr(AutoPtr& other) {
    value_ = other.value_;
    other.value_ = nullptr;   // отбираем владение
}

AutoPtr& operator=(AutoPtr& other) {
    if (&other == this) {
        return *this;
    }
    value_ = other.value_;
    other.value_ = nullptr;
    return *this;
}
```

Пример коварства:

```
int main() {
    auto_ptr<Boo> b{new Boo()};
    std::vector<auto_ptr<Boo>> boos(1);

    boos[0] = b;          // владение незаметно ушло из b в boos[0]
```

```
    boos[0]->func();
    auto_ptr<Boo> a = boos[0];   // и теперь — из boos[0] в a
    a->func();

    // b->func();         // Segmentation fault — b больше ничем не владеет
    // boos[0]->func();   // Segmentation fault — boos[0] тоже пуст

    return 0;
}
```

Стратегия 2. «Ресурс мой, никому не отдам» — `std::unique_ptr`

Просто запрещаем копирование. Один объект — один владелец. Передача владения возможна только явно через `std::move` (об этом позже).

```
AutoPtr(const AutoPtr& other) = delete;
AutoPtr& operator=(const AutoPtr& other) = delete;
```

У `std::unique_ptr` есть второй шаблонный параметр — **`Deleter`** (функтор). По умолчанию это `delete`, но можно передать свой, чтобы освобождать не только память. Например, файл закрывается через `fclose`:

```
template<typename T, typename DeletePtr>
class AutoPtr {
public:
    AutoPtr(T* value) : value_(value) {}
    ~AutoPtr() {
        if (value_) {
            DeletePtr deleter;
            deleter(value_);
        }
    }
    /* ... */
private:
    T* value_;
};
```

```cpp
struct FileDeleter {
    void operator()(FILE* f) const { fclose(f); }
};

int main() {
    FILE* file = fopen("in.txt", "r");
    AutoPtr<FILE, FileDeleter> fPtr(file);
    // fclose будет вызван автоматически в деструкторе
}
```

Также у `unique_ptr` есть метод `release()` — отдаёт сырой указатель и снимает с себя ответственность за его освобождение.

Стратегия 3. «Делим ресурс» — `std::shared_ptr`

Несколько объектов могут совместно владеть одним ресурсом. Ресурс освобождается, **когда уничтожается последний владелец**.

Как работает: хранится указатель на счётчик владельцев (reference count). Копирование увеличивает счётчик, деструктор уменьшает. При счётчике 0 — ресурс освобождается.

> **Совет:** сначала тянись к `unique_ptr`, и только если действительно нужно разделять владение — к `shared_ptr`. Он немного дороже (атомарный счётчик, дополнительная аллокация).

Проблема циклических ссылок

```cpp
struct A;
struct B {
    B()  { std::cout << "B\n";  }
    ~B() { std::cout << "~B\n"; }
    std::shared_ptr<A> ptr;
};
struct A {
    A()  { std::cout << "A\n";  }
    ~A() { std::cout << "~A\n"; }
```

```
    std::shared_ptr<B> ptr;
};

void func() {
    std::shared_ptr<A> a{new A()};
    std::shared_ptr<B> b{new B()};
    a->ptr = b;
    b->ptr = a;

    std::cout << a.use_count() << " " << a->ptr.use_count() << std::endl;
    std::cout << b.use_count() << " " << b->ptr.use_count() << std::endl;
    // выйдя из функции, локальные a и b уйдут — но ресурсы НЕ освободятся:
    // a держит b, b держит a → счётчики так и останутся равными 1
}

int main() {
    func();
}
```

Решение: `std::weak_ptr`

`weak_ptr` **не владеет** объектом напрямую (не увеличивает счётчик), но знает о его существовании. Чтобы получить доступ — вызывают `lock()`: если объект ещё жив, вернётся `shared_ptr`; если умер — пустой `shared_ptr`.

```
struct A;
struct B {
    B()  { std::cout << "B\n";  }
    ~B() { std::cout << "~B\n"; }
    std::weak_ptr<A> ptr;    // заменили shared_ptr на weak_ptr
};
struct A {
    A()  { std::cout << "A\n";  }
    ~A() { std::cout << "~A\n"; }
    std::weak_ptr<B> ptr;    // и здесь тоже
};
```

```cpp
void func() {
    std::shared_ptr<A> a{new A()};
    std::shared_ptr<B> b{new B()};
    a->ptr = b;
    b->ptr = a;
    // теперь циклической shared-зависимости нет — деструкторы вызовутся
}
```

STL — Standard Template Library

Библиотека обобщённых компонентов: - Контейнеры - Обобщённые алгоритмы - Итераторы - Функциональные объекты - Адаптеры - Аллокаторы - Вспомогательные функции

Гарантии производительности: везде, где можно задать вопрос про асимптотику, она зафиксирована стандартом.

Контейнеры

Последовательные (sequence): - `vector<T>` - `deque<T>` - `list<T>` - `array<T>` - `forward_list<T>`

Ассоциативные (ключ → значение): - `set<T>` - `map<Key, T>`

Неупорядоченные ассоциативные (на основе хеш-таблицы): - `unordered_set<T>` - `unordered_map<Key, T>`

Контейнеры заменяют массив с его недостатками (фиксированная длина, выделение на стеке). Ассоциативные хранят пары ключ–значение, **значения может и не быть** (в `set`). Неупорядоченные ассоциативные — это пары ключ–значение поверх хеш-таблицы.

Обобщённые алгоритмы

Работают на любых контейнерах, поддерживающих нужную категорию итератора: - `find`, `max`, `merge`, `replace`, `sort`, ...

Итераторы

Итераторы — это: - Указателеобразные объекты: похожи на указатели, но при этом полноценные объекты со своим интерфейсом. - Связующее звено между алгоритмами и контейнерами. - Работают с **диапазонами** `[first, last)` — корректный диапазон полуоткрытый: `first` входит, `last` — нет.

Категории (от слабых к сильным)

- входные (Input)
- выходные (Output)
- однонаправленные (Forward)
- двунаправленные (Bidirectional)

- произвольного доступа (Random Access)
- непрерывные (Contiguous)

Входной итератор — `find` (O(n))

```
template<typename InputIterator, typename T>
InputIterator find(
    InputIterator first,
    InputIterator last,
    const T& value
) {
    while (first != last && *first != value)
        ++first;
    return first;
}
```

Требования к входному итератору: - `operator==`, `operator!=` - `++iterator` и `iterator++` - `value = *iterator` (разыменование для чтения) - Все операции — `O(1)`

Выходной итератор — `copy` (O(n))

Через `*iterator` теперь не читают, а **пишут**.

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copy(
    InputIterator first,
    InputIterator last,
    OutputIterator result
) {
    while (first != last) {
        *result = *first;
        ++first;
        ++result;
    }
    return result;
}
```

Требования к выходному итератору: - `*iterator = value` - `++iterator` и

`iterator++` - O(1)

> Обычный указатель `T*` одновременно удовлетворяет требованиям и входного, и выходного итератора.

Однонаправленный итератор — `replace` (O(n))

Всё то же, что у входного/выходного, но итератор можно **сохранить и переиспользовать** — он не «портится» от прохода.

```
template<typename ForwardIterator, typename T>
void replace(
    ForwardIterator first,
    ForwardIterator last,
    const T& x,
    const T& y
) {
    while (first != last) {
        if (*first == x)
            *first = y;
        ++first;
    }
}
```

Двунаправленный итератор

Однонаправленный + умеет идти назад.

Требования: - `++iterator`, `iterator++` - `--iterator`, `iterator--` - чтение и запись через `*iterator`

Пример алгоритма: `std::reverse`.

Итератор с произвольным доступом

Двунаправленный + умеет прыгать на произвольное расстояние за O(1).

Для итераторов `r`, `s` и целого `n`: - `r + n`, `n + r`, `r - n` - `r[n] == *(r + n)` - `r += n`, `r -= n` - `r - s` → целое число - `r < s`, `r > s`, `r <= s`, `r >= s`

При этом итераторы **не обязаны** лежать в памяти подряд (могут быть «разрывы» — как в `deque`).

Пример: `std::binary_search`.

Непрерывный итератор (Contiguous)

Произвольного доступа + гарантия, что элементы **физически лежат подряд в памяти**. Можно получать сырой адрес: `&*(it + n) == &*(it) + n`.

Обычный указатель `T*` — это непрерывный итератор.

```
graph TD
    Input[Input Iterators] --> Forward[Forward Iterators]
    Output[Output Iterators] --> Forward
    Forward --> Bidirectional[Bidirectional Iterators]
    Bidirectional --> RandomAccess[Random Access Iterators]
    RandomAccess --> Contiguous[Contiguous Iterators]
```

Связь итераторов с алгоритмами и контейнерами

- Каждый контейнер описывает, какие итераторы он предоставляет.
- Каждый алгоритм описывает, какая категория итератора ему нужна.
- Интерфейсы STL спроектированы так, чтобы **поддерживать эффективные комбинации** и не давать неэффективные. Например, `binary_search` требует random-access — поэтому он гарантированно работает за O(log n).
- Если алгоритм требует **входной** итератор, то с ним можно использовать любой более сильный (forward, bidirectional, random-access, contiguous).

`iterator` vs `const_iterator` — для константных контейнеров используются константные итераторы.

Container	Iterator Type	Category
`T a[n]`	`T*`	mutable, contiguous
`T a[n]`	`const T*`	const, contiguous
`vector<T>`	`vector<T>::iterator`	mutable, contiguous
`vector<T>`	`vector<T>::const_iterator`	const, contiguous
`deque<T>`	`deque<T>::iterator`	mutable, random access
`deque<T>`	`deque<T>::const_iterator`	const, random access
`list<T>`	`list<T>::iterator`	mutable, bidirectional
`list<T>`	`list<T>::const_iterator`	const, bidirectional

Container	Iterator Type	Category
`set<T>`	`set<T>::iterator`	const, bidirectional
`set<T>`	`set<T>::const_iterator`	const, bidirectional
`multiset<T>`	`multiset<T>::iterator`	const, bidirectional
`map<Key, T>`	`map<Key, T>::iterator`	mutable, bidirectional
`map<Key, T>`	`map<Key, T>::const_iterator`	const, bidirectional
`multimap<Key, T>`	`multimap<Key, T>::iterator`	mutable, bidirectional
`multimap<Key, T>`	`multimap<Key, T>::const_iterator`	const, bidirectional

Container	Iterator Type	Category
`unordered_set<T>`	`unordered_set<T>::iterator`	mutable, forward
`unordered_set<T>`	`unordered_set<T>::const_iterator`	const, forward
`unordered_map<Key, T>`	`unordered_map<Key, T>::iterator`	mutable, forward
`unordered_map<Key, T>`	`unordered_map<Key, T>::const_iterator`	const, forward
`unordered_multiset<T>`	`unordered_multiset<T>::iterator`	mutable, forward
`unordered_multiset<T>`	`unordered_multiset<T>::const_iterator`	const, forward
`unordered_multimap<Key, T>`	`unordered_multimap<Key, T>::iterator`	mutable, forward
`unordered_multimap<Key, T>`	`unordered_multimap<Key, T>::const_iterator`	const, forward

Классификация алгоритмов

- **Неизменяющие** — только читают: `find`, `find_if`, `adjacent_find`, `count`, `for_each`, `mismatch`, `equal`, `search`.
  - `find` / `find_if` — находят первое вхождение; разница в том, что `_if` принимает предикат.
  - `count` — сколько раз встречается значение, O(n).
- **Изменяющие** — модифицируют последовательность: `copy`, `copy_backward`, `fill`, `generate`, `partition`, `random_shuffle`, `remove`, `replace`, `rotate`, `swap`, `swap_ranges`, `transform`, `unique`.
  - `fill` / `fill_n` — заполняет диапазон значением; `_n` — заполняет ровно `n` элементов.

  - `generate` — заполняет результатами вызова функции, O(n).
- **С предикатами** — принимают функцию/функтор как аргумент. В `std::sort`, например, можно передать кастомный компаратор.

Erase-remove idiom

`remove()` ничего не удаляет физически — он **перемещает «выживших» в начало** и возвращает итератор на новый конец. Размер контейнера не меняется, в хвосте — мусор.

```
// Идея remove (упрощённо):
template<typename Iterator, typename T>
Iterator remove(Iterator first, Iterator last, const T& value) {
    Iterator cur = first;
    while (first != last) {
        if (*first != value) {
            *cur = *first;
            ++cur;
        }
        ++first;
    }
    return cur;
}
```

Чтобы реально стереть «отрезанный» хвост, используют `erase()` контейнера. **Идиома:**

```
v.erase(std::remove(v.begin(), v.end(), value), v.end());
```

Теоретико-множественные операции

Смотрят на отсортированные диапазоны как на множества: - `includes` — содержится ли один в другом - `set_union` — объединение - `set_intersection` — пересечение - `set_difference` — разность - `set_symmetric_difference` — симметрическая разность

Обобщённые числовые алгоритмы

- `accumulate` — свёртка (по умолчанию сумма, можно передать бинарный оператор)

- `partial_sum` — частичные суммы
- `adjacent_difference` — разности соседних элементов
- `inner_product` — скалярное произведение двух диапазонов

  STL = контейнеры + алгоритмы, связанные через итераторы. Каждый алгоритм требует определённую категорию итератора, разные контейнеры предоставляют разные категории — это и даёт гарантии асимптотики.

```
std::vector<int>::iterator myremove(
    std::vector<int>::iterator begin,
    std::vector<int>::iterator end,
    int num
) {
    std::vector<int>::iterator cur_place = begin;
    while (begin != end) {
        if (*begin != num) {
            *cur_place = *begin;
            ++cur_place;
        }
        ++begin;
    }
    return cur_place;
}
```

Hand-made `remove`, но работает только на `std::vector<int>`. Шагами обобщения: 1. Шаблоны по типу элемента — но всё ещё привязка к `std::vector`. 2. Шаблонный `IterType` вместо итератора вектора — теперь работает с любым контейнером… но **не каждый** итератор подойдёт. Нужны требования (forward-итератор).

Функциональные объекты (функторы)

Функтор — объект класса с перегруженным `operator()`. Параметризует алгоритм: алгоритму даём «как сравнивать» / «как складывать» / «по какому критерию фильтровать».

Стандартные функторы из `<functional>`: - арифметические: `plus`, `minus`, `multiplies`, `divides`, `modulus`, `negate` - сравнения: `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal` - логические: `logical_and`, `logical_or`, `logical_not` - битовые: `bit_and`, `bit_or`, `bit_xor`

Named Requirements

Формальное описание того, чего стандарт ожидает от типа. Описывает: - какие вложенные типы должны быть определены; - какие выражения должны быть валидны; - какой набор методов должен присутствовать.

- Container

- ReversibleContainer
- AllocatorAwareContainer
- SequenceContainer
- ContiguousContainer
- AssociativeContainer
- UnorderedAssociativeContainer

Последовательные контейнеры

`array`, `vector`, `deque`, `forward_list`, `list` — решают проблему массива (фиксированный размер) и предоставляют разные компромиссы между скоростью доступа и скоростью вставки.

`std::array`

Удовлетворяет: `Container`, `ReversibleContainer`, `SequenceContainer`, `ContiguousContainer`.

Реализация интерфейса `Container`:

```
iterator begin()             { return iterator(data()); }
const_iterator begin() const { return const_iterator(data()); }
iterator end()               { return iterator(data() + _Size); }
const_iterator end() const   { return const_iterator(data() + _Size); }

size_type size() const     { return _Size; }
size_type max_size() const { return _Size; }
bool empty() const         { return _Size == 0; }
```

`ReversibleContainer`:

```
reverse_iterator rbegin()             { return reverse_iterator(end()); }
const_reverse_iterator rbegin() const { return const_reverse_iterator(end()); }
reverse_iterator rend()               { return reverse_iterator(begin()); }
const_reverse_iterator rend() const   { return const_reverse_iterator(begin()); }
```

- `iterator` для `array` — это по сути просто указатель (поэтому `end() = data() + size`).
- `reverse_iterator` смотрит на контейнер «с конца к началу»: `rbegin()` возвращает «обёртку» над `end()`, которая при разыменовании даёт последний элемент.

- Классический фиксированный массив.

`std::vector`

- Один большой непрерывный кусок памяти. Выделяется с **запасом** (`capacity` ≥ `size`).
- При вставке: если есть свободное место — добавляем, иначе делаем **реаллокацию** (обычно выделяется в ~2 раза больше памяти, всё копируется/перемещается). Реаллокация случается редко, поэтому амортизированная сложность вставки в конец — O(1).
- Предоставляет самый сильный итератор (contiguous) — работает со всеми STL-алгоритмами.
- Быстрый доступ по индексу за счёт непрерывности.
- Вставка/удаление в середину — O(n).
- Может не найтись такого большого непрерывного куска памяти.

`std::deque`

- Похож на вектор, но память состоит из **нескольких непрерывных блоков** (массив массивов). Если не хватает — добавляется ещё один блок.
- Компромисс между вектором и списком.
- Индексация чуть медленнее (нужно понять, в каком блоке элемент).
- Вставка/удаление в середину — всё ещё неэффективна.
- Зато эффективна вставка в начало и в конец — O(1).

`std::list`

- Двусвязный список (циклический — у некоторых реализаций).
- Эффективная вставка/удаление в любую позицию — O(1) (если есть итератор).
- Долгий доступ по индексу — O(n).
- Перерасход памяти: каждая нода хранит ещё два указателя.
- Не поддерживает все алгоритмы (только bidirectional итератор).

`std::forward_list`

- Однонаправленный список.
- Меньше памяти, чем у `list` (один указатель на ноду вместо двух).
- Forward-итератор — самый слабый из «полноценных».

Цитата (китайская мудрость, 17 век до н.э.): Если не знаешь, что выбрать — выбирай вектор. Только он не должен быть большим.

- Если нужно много вставок/удалений в середине → список.

Ассоциативные контейнеры

Обычно реализованы как красно-чёрное дерево (но в стандарте конкретная реализация не зафиксирована — гарантируется только асимптотика).

- **`set`** — множество уникальных ключей. Требует, чтобы у типа был определён `operator<` (или передан кастомный компаратор). `operator<` определять опасно — он распространяется на весь код; иногда лучше написать функтор-компаратор.
- **`map`** — множество уникальных пар «ключ → значение». Двунаправленный итератор. Доступ по ключу, вставка, удаление — O(log n).
- **`multiset`** / **`multimap`** — то же, но допускают дубликаты ключей.

Неупорядоченные ассоциативные контейнеры

Под капотом — хеш-таблица.

- **`unordered_map`**, **`unordered_set`** — Forward-итератор; доступ амортизированно O(1), но в худшем случае (много коллизий) — O(n). Требует хеш-функции для ключа. Памяти ест больше, чем `map`.
- **`unordered_multiset`**, **`unordered_multimap`** — с дубликатами.

Iterator invalidation

Классическая ловушка многих контейнеров — итераторы могут стать невалидными после модификации контейнера: - **Вектор:** реаллокация (при `push_back`, `insert`, `resize`, `reserve`) инвалидирует **все** итераторы. Иначе — только те, что были после точки вставки/удаления. - **`deque`:** `insert`/`erase` инвалидирует все итераторы. - **`list`, `forward_list`:** валидны почти всегда, кроме итераторов на удалённые элементы. - **`unordered_*`:** rehash инвалидирует все итераторы.

Allocator (введение)

Аллокатор — абстракция, которая отвечает за выделение и освобождение памяти от имени контейнера. Контейнер не знает деталей malloc/new — он просит у

аллокатора: - `allocate(n)` — выдай память под `n` элементов; - `deallocate(p, n)` — освободи; - `construct` / `destroy` — построй/разрушь объект в этой памяти.

Самый простой аллокатор оборачивает `malloc` и `free`:

```
template <typename T>
class CSimpleAllocator {
public:
    pointer allocate(size_type size) {
        pointer result = static_cast<pointer>(malloc(size * sizeof(T)));
        if (result == nullptr) {
            // error
        }
        std::cout << size << " " << result << std::endl;
        return result;
    }
    void deallocate(pointer p, size_type n) {
        std::cout << p << std::endl;
        free(p);
    }
};
```

Пример использования:

```
struct SPoint {
    int x;
    int y;
};

int main() {
    std::allocator_traits<CSimpleAllocator<int>> at;
    std::vector<SPoint, CSimpleAllocator<SPoint>> data;

    data.push_back({10, 20});
    data.pop_back();

    return 0;
}
```

Зачем вообще всё это? — https://habr.com/ru/articles/274827/ **Доп. инфа** — https://habr.com/ru/articles/505632/

Allocator

— это **класс, который абстрагирует выделение и освобождение памяти** для различных объектов. Предоставляет механизм для выделения и конструирования объектов, а также их освобождения и уничтожения.

Два ключевых метода аллокатора: - выделить память под `n` элементов типа `T` - освободить память по указателю, который был ранее выдан

Зачем писать свой аллокатор? Например, для `std::list` хочется выделять память сразу под несколько нод: - Если на каждую вставку вызывать `malloc` — это неоптимально: обращение к ОС, маппинг страниц на виртуальную память. - Долго и неудобно «таскать» с собой кэш процессора между разными участками памяти (плохая локальность).

Выводы: - Аллоцировать не каждый раз при вставке, а **сразу большим куском**, а потом раздавать его «по чуть-чуть». - Если выделить память на стеке — обращение к ней быстрее (горячий кэш, нет системного вызова). - Простейшая реализация: выделить массив; возвращать первый свободный элемент, сдвигая указатель; при выходе за пределы массива — ошибка.

```
template <typename T>
class CSimpleAllocator {
public:
    pointer allocate(size_type size) {
        pointer result = static_cast<pointer>(malloc(size * sizeof(T)));
        if (result == nullptr) {
            // error
        }
        std::cout << "Allocate count: " << size << " elements. Pointer: " << result <<
        return result;
    }
    void deallocate(pointer p, size_type n) {
        std::cout << "Deallocate pointer: " << p << std::endl;
        free(p);
    }
};
```

rebind

Объясняет компилятору, как, имея аллокатор для типа `T`, получить аллокатор для типа `U`. Это нужно, потому что **внутри реализации контейнера часто требуется аллокатор для другого типа**, чем тот, который виден снаружи.

Классический пример: `std::list<int>` снаружи кажется, что хранит `int`, но внутри он хранит **ноды** — структуры вида `{ T value; Node* prev; Node* next; }`. Поэтому контейнер через `rebind` запрашивает у переданного `Allocator<int>` соответствующий `Allocator<Node<int>>`.

StackAllocator

Простая идея: вместо `malloc` использовать заранее выделенный массив; раздавать память «накатом», сдвигая внутренний указатель. `deallocate` ничего не делает — освобождать что-то посередине нельзя, вся память живёт до уничтожения самого аллокатора.

```
#include <iostream>
#include <vector>

template<typename T, size_t SIZE>
class CStackAllocator {
public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;
    typedef T         value_type;

    template<typename U>
    struct rebind {
        typedef CStackAllocator<U, SIZE> other;
        // Важно: rebind должен давать аллокатор для типа U, а не T.
        // Например, std::list<int> внутри хранит не int, а Node<int>
        // (значение + два указателя), поэтому контейнер через rebind
        // запрашивает аллокатор именно под Node<int>.
```

```
    };

    pointer allocate(size_type n) {
        pointer result = buffer_ + size_;
        std::cout << "Allocate " << result << "  " << n << std::endl;
        size_ += n;
        return result;
    }

    void deallocate(pointer p, size_type n) {
        std::cout << "Deallocate " << p << " " << n << std::endl;
        // Реального освобождения нет — память стековая, живёт до уничтожения аллокатор
    }

private:
    T      buffer_[SIZE];
    size_t size_ = 0;
};

int main() {
    CStackAllocator<int, 100> al;
    std::vector<int, CStackAllocator<int, 100>> v;
    for (int i = 0; i < 10; ++i)
        v.push_back(i);
}
```

> **Грубая идея:** «У меня есть массив на `SIZE` элементов типа `T` — выдавай мне куски из него».

По сути, мы передаём ответственность за управление памятью **другой сущности** (аллокатору). Контейнер не должен знать, откуда берётся память.

Так, например, работает `std::list`: при вставке он понимает, что нужна нода (`value + prev + next`), и просит у аллокатора память под неё. Сам список не аллоцирует.

Аллокатор по умолчанию (`std::allocator`) для произвольного `T` делает `::operator new` (по сути `malloc`) при выделении и `::operator delete` (по сути `free`) при освобождении — как `CSimpleAllocator` выше.

`std::vector` использует аллокатор для своего внутреннего буфера, где живут `size` элементов в пределах `capacity`.

Адаптеры

- **Адаптеры контейнеров** — оборачивают существующие контейнеры в другой интерфейс:
  - `std::stack`
  - `std::queue`
  - `std::priority_queue`
- **Адаптеры итераторов** — отдельный вид итераторов со специальным поведением. Упрощают работу с контейнерами в стандартных алгоритмах (доп. информация):
  - `back_insert_iterator` — output-итератор; вставляет в конец
  - `front_insert_iterator` — вставляет в начало
  - `insert_iterator` — вставляет в заданную позицию
- **Потоковые итераторы** — позволяют работать с потоком (`std::cin`, `std::cout` и т.д.) как с контейнером и применять к нему стандартные алгоритмы.

Tag Dispatch Idiom

Техника, при которой создаются **пустые теги-типы**, чтобы компилятор выбирал нужную перегрузку функции по переданному тегу. Это позволяет специализировать поведение алгоритма под конкретную категорию итератора (или другую характеристику) **без `if`-веток в рантайме**.

```
template<typename T>
void func_dispatch(const T& value, const tag_1&) {
    std::cout << "tag1\n";
}

template<typename T>
void func_dispatch(const T& value, const tag_2&) {
    std::cout << "tag2\n";
}

template<typename T>
```

```
void evaluate(const T& value) {
    func_dispatch(value, typename my_traits<T>::tag());
}
```

```
struct tag_1 {};
struct tag_2 {};
struct tag_3 : public tag_2 {};

struct TypeA {};
struct TypeB {};
struct TypeC {};

template<typename T>
struct my_traits {
    typedef tag_1 tag;
};

template<> struct my_traits<TypeB> {
    typedef tag_2 tag;
};

template<> struct my_traits<TypeC> {
    typedef tag_3 tag;
};
```

iterator_traits

Позволяет алгоритмам узнать свойства переданного итератора (категорию, тип элемента, тип разности) и выбрать наиболее эффективную реализацию для каждой категории. Например, `std::distance` для random-access — `O(1)` (вычитание), а для остальных — `O(n)` (счёт в цикле).

Лекция 5. Обработка ошибок

Какие ошибки могут возникать (делаем что-то некорректное — не потому что мы криворукие, а потому что внешний мир мог не дать): - выход за границу массива; - деление на ноль; - невозможность выделить память; - отсутствие прав на открытие файла; - недоступность внешнего сервера; - ...

Варианты обработки

Обычный assert (runtime)

Если проверка ложна — программа просто падает.

```
#include <cassert>

int main() {
    assert(2 + 2 == 4);
    assert(2 + 2 == 5);
    return 0;
}
// Assertion '2+2==5' failed.
```

static_assert (compile-time)

Помогает выяснить разные факты про типы прямо при компиляции.

```
// Пример: есть ли у произвольного типа T дефолтный конструктор
static_assert(sizeof(int) == 4, "int must be 4 bytes");

template <typename T>
struct data_structure {
    static_assert(
        std::is_default_constructible<T>::value,
        "Data Structure requires default-constructible elements"
    );
};

struct no_default {
    no_default() = delete;
};
```

```
int main() {
    data_structure<no_default> ds_error;   // ошибка компиляции
    return 0;
}
```

Код возврата

Получаем число/значение ошибки, потом перебираем варианты:

- **Просто возвращаемое значение.** Например, количество успешно записанных байт:

  ```
  size_t fwrite(const void* buffer, size_t size, size_t count, FILE* stream);
  ```

- **errno** — глобальный макрос, в который записывается код ошибки, который потом можно перевести в текст.

  - Ошибок очень много, тяжело учесть все.
  - Это **глобальная переменная** — следующая ошибка затрёт предыдущую.

  ```
  FILE* fopen(const char* filename, const char* mode);
  ```

  Возвращает указатель на `FILE` или `nullptr` (нет прав / файл не существует / занят другой программой); подробности через `errno`.

- **Ошибка как код возврата (`errno_t`).** Само значение возвращается через out-параметр:

  ```
  errno_t fopen_s(
      FILE* restrict* restrict streamptr,
      const char* restrict filename,
      const char* restrict mode
  );
  ```

Обработка в месте возврата

Куча `if`-ов на возврат значений — неудобно. На просмотр всех этих `if`-ов уходит больше сил, чем на сам код. Бизнес-логика тонет в обработке ошибок.

Exceptions: `throw` / `try` / `catch`

```
int foo() {
    throw std::runtime_error("error");
}

void boo() {
    throw 2;
}

void coo() {
    throw std::string("Hello world");
}

int main(int, char**) {
    try {
        foo();
    }
    catch(...) {
        // do something...
    }
}
```

- `throw` — сообщает об исключительной ситуации.
- `try` — в коде внутри блока пытаемся найти исключения.
- `catch` — обрабатываем поднятые в `try` исключения.
- После `throw` запускается механизм **stack unwinding** («раскрутка стека»).

Stack unwinding (по шагам)

1. Сконструированный в `throw` объект-исключение пробрасывается обратно по стеку.
2. Стек раскручивается до первого подходящего `catch` (если такого нет — аварийное завершение через `std::terminate`).
3. По пути уничтожаются все объекты с automatic storage duration (вызываются их деструкторы) — это то, ради чего нужен RAII.
4. Если в процессе раскрутки возникает **ещё одно** исключение — вызывается `std::terminate` (принудительное завершение программы).

5. Поэтому **деструкторы по умолчанию `noexcept`** — нельзя кидать из деструктора при раскрутке.
6. Сам объект-исключение хранится в специальной (неопределённой реализацией) области памяти.

```
struct Foo {
    Foo()  { std::cout << "Foo()\n";  }
    ~Foo() { std::cout << "~Foo()\n"; }
};

void internalFunc() {
    Foo f;
    throw std::runtime_error("Some error");
}

void externalFunc() {
    try { internalFunc(); }
    catch (const std::exception& e) { std::cout << e.what() << std::endl; }
}
```

Вывод:
Foo()
~Foo()
Some error

Объяснение: - `main` → `externalFunc` → `internalFunc` → бросили `throw`. - Раскручиваем стек, по пути вызывается деструктор `f`. - Находим блок `try/catch` в `externalFunc` и обрабатываем. - Можно обрабатывать **не в месте возникновения**, а где удобно — главное где-то выше по стеку. Это и есть главное преимущество исключений: бизнес-логика не загромождается проверками.

> **Очень важно:** именно поэтому нужен RAII. Если мы вручную делали `new`, то после `throw` мы пролетим мимо `delete` — получим утечку.
>
> ```
> void internalFunc() {
>     Foo* f = new Foo;
>     throw std::runtime_error("Some error");  // утечка!
>     delete f;                                // никогда не выполнится
> }
> ```

```cpp
int main() {
    try {
        foo();
    } catch (const std::overflow_error& e) {
        // do something
    } catch (const std::runtime_error& e) {
        // do something
    } catch (const std::exception& e) {
        // do something
    } catch (...) {
        // do something
    }
}
```

Несколько `catch`-блоков

- Ищем первый подходящий `catch`-блок (тип брошенного исключения должен приводиться к типу пойманного — в т.ч. через наследование).
- Гарантированно попадаем **только в один** `catch`-блок.
- Если ни один не подошёл — `terminating due to uncaught exception`.
- Если подходят несколько — берётся **первый** в порядке записи.
- **Порядок важен:** от самого конкретного (производного класса) к самому общему (базового). Иначе более общий перехватит всё первым.
- `catch(...)` ловит **что угодно**, в т.ч. не-исключения (например, `throw 2`). Полезен как «последний шанс», но плох тем, что не знаем, что именно случилось.
- Передача в `catch` — как в функцию. Стандарт: **по `const&`** — чтобы не было лишних копий.

Гарантии безопасности исключений

- **No guarantee** — никаких гарантий, объект может остаться в произвольном состоянии.
- **Basic guarantee** — инвариант сохраняется, утечек нет, но точное состояние может быть произвольным.
- **Strong guarantee** — либо операция полностью успешна, либо состояние возвращается к тому, что было **до** вызова (как будто операции не было). Никаких

утечек.

- **Nothrow guarantee** — функция не бросает исключений вообще.

Пример: в наивном `operator=` после исключения в конструкторе копирования объект остаётся в полусобранном состоянии — это basic-гарантия (`nullptr` — нормальный инвариант), но не strong. Для strong-гарантии используется **Copy-and-swap idiom**: 1. Делаем копию принимаемого объекта (если конструктор копирования бросит — старый объект цел). 2. Свапаем поля копии и текущего объекта (своп — `noexcept`). 3. Копия с «нашими старыми данными» уничтожается на выходе из функции.

`noexcept`

Ключевое слово — гарантия, что функция **не бросает исключений**. - Если она всё-таки бросит — `std::terminate`. - Не нужна разворотная инфраструктура — компилятор лучше оптимизирует код. - **Деструкторы — `noexcept` по умолчанию.** - В обмен на это компилятор не генерирует машинный код для раскрутки стека из этой функции. - Все STL-контейнеры дают гарантии на свои методы (часто — strong).

> **P.S.** Когда объявление сопровождается инициализацией — вызывается **конструктор копирования** (`Boo b1 = b;`), а когда переменная уже существовала — **оператор присваивания** (`b1 = b;`).

std::exception

Базовый класс для исключений стандартной библиотеки. Кидать просто строки или числа — малоинформативно: тип исключения сам по себе — полезная информация, а уж дополнение текстовым сообщением — вообще хорошо.

```
class exception {
public:
    exception() noexcept;
    exception(const exception&) noexcept;
    exception& operator=(const exception&) noexcept;
    virtual ~exception();
    virtual const char* what() const noexcept;
};
```

Свои исключения

Наследуем от `std::exception` (или от более конкретного, например `std::runtime_error`):

```
class my_exception : public std::exception { // derived from std::exception
public:
    my_exception(const std::string& what)
        : what_(what){
    }
    const char* what() const noexcept override {
        return what_.c_str();
    }
private:
    std::string what_;
};
```

> **WARNING: Важно!** Не нужно бояться писать свои исключения! Они могут быть даже очень тривиальными и это не страшно.

Example: (очень похож на runtime error):

```
int foo() {
    throw my_exception("error"); // by rvalue
}
```

```
int main(int, char**) {
    try{
        foo();
    }
    catch(const my_exception& e) { // by const reference
        std::cerr << e.what();
        std::runtime_error
    }
}
```

Правила работы с исключениями

- Исключения — **исключительно для обработки ошибок**, не для бизнес-логики (механизм броска стоит дорого).
- Работа с исключениями строится вокруг **инварианта объекта** — после возможного исключения объект должен оставаться в осмысленном состоянии.
- **Кидаем по значению, ловим по ссылке** (`const&`):
  - Исключения хранятся в особом месте памяти.
  - Ловить по ссылке — чтобы избежать лишнего копирования.

Цена исключений

Бенчмарк на разном числе потоков и разном количестве ошибок: - **Рост по столбцам** — раскрутка стека не бесплатная: чем больше ошибок, тем больше времени тратится на unwinding. - **Рост по строкам** — два потока **не могут обрабатывать исключения одновременно**, компилятор/рантайм сериализует обработку.

`std::expected` (C++23)

Идея: объединить плюсы кодов возврата и исключений. - Коды возврата легковесные. - Исключения удобные, не требуют обработки в месте получения.

Шаблонный тип с двумя параметрами: тип результата при успехе и тип ошибки.

- Возвращает либо ожидаемое значение, либо ошибку.
- Накладные расходы сравнимы с кодом возврата.
- Передаёт ответственность за обработку вызывающему коду.

Три ключевых класса: 1. `std::expected<T, E>` — главный класс. 2. `std::unexpected<E>` — обёртка, чтобы вернуть ошибку. 3. `std::bad_expected_access` — бросается, если попытались обратиться к значению, а там лежит ошибка.

```
enum class EDivError {
    DivisionByZero = 0,
};

std::expected<int, EDivError> my_div(int a, int b) {
    if (b == 0)
        return std::unexpected{EDivError::DivisionByZero};

    return a / b;
}

int main() {
    auto r = my_div(8, 0);

    // 1 способ — как с кодом возврата
    if (r)
        std::cout << *r << std::endl;

    // 2 способ — ловим исключение
    try {
        std::cout << r.value() << std::endl;
    } catch (std::bad_expected_access& err) {
        std::cout << err.what() << std::endl;
    }

    return 0;
}
```

Как обычно, **идеального решения нет** — приходится выбирать: 1. **Исключения** — единообразная обработка, но не в месте возникновения. 2. **Коды возврата** — обработка сразу в месте возникновения, но неединообразна. 3. **`std::expected`** — комбинированный подход.

Пример: пишем `to_uint`

```
uint32_t to_uint(std::string_view str) {
    uint32_t result = 0;
    for (char c : str) {
        result *= 10;
        result += c - '0';
    }
    return result;
}

int main(int, char**) {
    std::cout << to_uint("100500") << std::endl;
}
```

Тут проблема: если в строке что-то кроме цифр — поведение странное. Варианты обработки (см. презу):

(а) Просто проверить, цифра ли символ; иначе вернуть то, что вышло до этого момента: странный результат — обработано не всё, и непонятно, что вернётся.

(б) Возвращаем `bool`, а значение — через ссылку: не отвечает на вопрос «что случилось».

(в) Через `errno` (C-стиль):

```
uint32_t to_uint(std::string_view str) {
    uint32_t result = 0;
    if (str.empty()) {
        errno = EINVAL;
        return result;
    }

    for (char c : str) {
        if (c < '0' || c > '9') {
            errno = EDOM;
            return result;
        }
        result *= 10;
```

```
        result += c - '0';
    }

    return result;
}
```

(г) Проброс исключения:

```
uint32_t to_uint(std::string_view str) {
    uint32_t result = 0;
    if (str.empty())
        throw std::invalid_argument("String is empty");

    for (char c : str) {
        if (c < '0' || c > '9')
            throw std::invalid_argument(std::format("Argument {} is not a number", str)

        result *= 10;
        result += c - '0';
    }

    return result;
}
```

(д) `std::optional` — похоже на (б): хранит либо значение, либо «ничего» (`std::nullopt`):

```
std::optional<uint32_t> to_uint(std::string_view str) {
    if (str.empty())
        return {};

    uint32_t result = 0;
    for (char c : str) {
        if (c < '0' || c > '9')
            return {};
        result *= 10;
        result += c - '0';
    }
```

```
    return result;
}
```

(e) `std::expected`:

```
std::expected<uint32_t, std::invalid_argument> to_uint(std::string_view str) {
    if (str.empty())
        return std::unexpected{std::invalid_argument("String is empty")};

    uint32_t result = 0;
    for (char c : str) {
        if (c < '0' || c > '9')
            return std::unexpected{
                std::invalid_argument{std::format("Argument {} is not a number", str)}
            };
        result *= 10;
        result += c - '0';
    }
    return result;
}
```

Вывод

В зависимости от того, как удобнее обрабатывать ошибки — есть три (плюс один) варианта: - коды возврата - исключения - `std::expected` - (либо `std::optional`)

Указатели на функцию

Работают для глобальных функций и статических методов классов:

```
return_type (*pointer_name)(arg_type1, arg_type2, ... arg_typen)
```

Пример:

```
int* findMax(int* array, size_t size, bool(*compare)(int, int)) {
    int* result = array;
    for (int i = 1; i < size; ++i) {
        if (!compare(*result, *(array + i)))
            result = array + i;
    }
    return result;
}

bool greater(int a, int b) {
    return a > b;
}

int main() {
    int array[] = {1, 4, 5, 3, 10, 9};
    std::cout << *findMax(array, sizeof(array) / sizeof(int), greater);
    return 0;
}
```

С `using` — чище:

```
using TComparer = bool(*)(int, int);

int* findMax(int* array, size_t size, TComparer comparer) {
    int* result = array;
    for (int i = 1; i < size; ++i) {
        if (!comparer(*result, *(array + i)))
            result = array + i;
    }
    return result;
}
```

Через шаблон — ещё чище (и можно передавать функторы, не только функции):

```
template<typename TCompare>
int* findMax(int* array, size_t size, TCompare comparer) {
    int* result = array;
    for (int i = 1; i < size; ++i) {
        if (!comparer(*result, *(array + i)))
            result = array + i;
    }
    return result;
}

int main() {
    int array[] = {1, 4, 5, 3, 10, 9};
    std::cout << *findMax(array, sizeof(array) / sizeof(int), std::greater<int>());
    return 0;
}
```

Функторы

Функтор, в отличие от функции, **может хранить состояние**.

Примеры стандартных функторов из `<functional>`: `std::less`, `std::equal_to`, `std::plus`, `std::logical_and`, и т.д.

```
class GreaterThen {
public:
    GreaterThen(int limit)
        : limit_(limit)
    {}

    bool operator()(int value) const {
        return value > limit_;
    }

private:
    int limit_;
};
```

```
int main() {
    std::vector v = {1, 2, 3, 4, 5, 6, 7};

    auto it = std::find_if(
        v.begin(), v.end(),
        GreaterThen{4}
    );

    if (it != v.end())
        std::cout << *it;

    return 0;
}
```

`mutable` — поле, которое может быть изменено из константных методов.

Проблемы такого функтора: - Не хочется ради такой мелочи выделять целый класс. - Есть почти то же самое — `std::greater`.

`std::bind` помогает «зафиксировать» часть аргументов существующего функтора:

```
int main() {
    std::vector v = {1, 2, 3, 4, 5, 6, 7};

    auto it = std::find_if(
        v.begin(), v.end(),
        std::bind(std::greater<int>{}, std::placeholders::_1, 4)
    );

    if (it != v.end())
        std::cout << *it;

    return 0;
}
```

Промежуточный итог по функторам

- Позволяют параметризовать алгоритмы.

- Отделены от вызывающего кода.
- Использовать стандартные функторы в нестандартных ситуациях затруднительно.

Lambda

Замыкание — позволяет создавать неименованные функторы с захватом переменных из текущей области видимости.

Синтаксис:

`[capture] (params) attrs -> return { body }`

- `(params)` — optional;
- `attrs` — optional (например, `mutable`, `noexcept`);
- `-> return` — optional (тип возвращаемого значения; помогает компилятору и читающему код).

```cpp
int main() {
    std::vector v = {1, 2, 3, 4, 5, 6, 7};

    auto it = std::find_if(
        v.begin(), v.end(),
        [](int value) { return value > 4; }
    );
    if (it != v.end())
        std::cout << *it;

    return 0;
}
```

Более читаемо — синтаксический сахар над функтором.

```cpp
// всё это корректно
int x = 1;
[]{};
[](int i) { return i + 1; };
[](int i) -> float { return i + 1; };
[x](int i) { return x + i; };
[](int i) noexcept { return i + 1; };
[&x](int i) mutable { ++x; return i + x; };
```

Компилятор за нас создаёт функциональный объект — у каждой лямбды свой уникальный тип:

Если написать **две идентичные** лямбды — это всё равно будут два **разных типа**:

Проблемы функторов

- Реализация далеко от места вызова.
- Много текста.
- Часто функтор используется один раз *(самостоятельно писать редко приходится — обычно хватает стандартных)*.

  Начиная с C++11 вместо собственных функторов используют **лямбда-функции**.

Преимущества лямбд

- Не нужно отдельно писать функционал — прямо в месте использования.
- Не нужно искать реализацию.
- Более элегантный синтаксис, чем у функтора.
- **Захват переменных** из локальной области видимости.

Захват переменных (capture)

- `[x, y]` — by value
- `[=]` — все используемые в теле — by value, у которых automatic storage duration
- `[&x, &y]` — by reference
- `[&]` — все используемые — by reference (с automatic storage)
- `[this]` — захват `this`-указателя (доступ к полям без копии — `field` обращается к полю объекта)
- `[*this]` — захват **копии текущего объекта** (C++17)

Example 1: Capture by Value and by Reference

```
int main() {
    int x = 1;
    int y = 2;
```

```
    auto f = [x, &y](int v) { return v + x + y; };
}
```

Example 2: Capture *this by Value (C++17)

```
struct Foo {
    int field = 0;
};

int func(int i) {
    auto f = [*this](int value) { return field + value; };
    return f(i);
}
```

Example 3: Capture this by Value (Pointer)

```
struct Foo {
    int field = 0;
};

int func(int i) {
    auto f = [this](int value) { return field + value; };
    return f(i);
}
```

`mutable` лямбда

- Явно разрешает менять переменную, **захваченную по значению**.
- Захваченная **по ссылке** — меняется и снаружи (как и в обычной функции).
- Захваченная **по значению с `mutable`** — внутри лямбды состояние сохраняется между вызовами, но снаружи переменная не меняется.

  Если написать «сырую» лямбду и сразу её вызвать `[]{ ... }();` — она вызовется один раз на месте.

Условный выбор объекта-функции

Если нужно в зависимости от условия использовать разные функциональные объекты — это раньше было больно (вызов дефолтного конструктора, потом присваивание; вопросы константности; дефолтного конструктора может вообще не быть).

Варианты: - **Указатель на функцию** — но тогда сами объекты живут на куче, а не на стеке. - **Лямбда** — инициализируем переменную лямбдой, возвращающей нужный объект.

Изначально лямбды задумывались как упрощённый способ объявления функторов для передачи; сейчас их используют **просто на месте** для красоты.

> Чтобы отказаться от вызова через `()`, можно использовать `std::invoke(f, args...)` — чисто ради эстетики.

Лямбды и наследование

Если сделать структуру, наследующую от двух функторов, и подключить их `operator()` через `using` — компилятор сможет различать вызовы по типу аргумента (или ругаться при конфликте). По сути функциональный объект = двум функциональным объектам сразу.

Как это упростить: 1. Сделать отдельную фабричную функцию для красивого создания такого объекта. 2. Передавать в эту функцию лямбды вместо функторов.

Раньше похожее делали через `std::bind`, но с появлением лямбд он по большей части не нужен — в лямбду можно вложить другую лямбду.

Generic lambda (C++14)

Вместо типов аргументов можно ставить `auto` — при инстанциации компилятор сам подберёт тип и создаст шаблонный функтор:

```
auto f = [](auto x, auto y) { return x + y; };
```

Рекурсивная лямбда

Рекурсия прямо в месте использования. Поскольку лямбда не знает своего имени, есть два классических способа:

```
// (а) Через std::function — лямбда знает себя по имени переменной
std::function<int(int)> fact = [&](int n) {
    return n <= 1 ? 1 : n * fact(n - 1);
};

// (б) Через "Y-комбинатор" — передаём саму себя как аргумент
auto fact = [](auto self, int n) -> int {
```

```
    return n <= 1 ? 1 : n * self(self, n - 1);
};
fact(fact, 5);
```

Function pointer ↔ Lambda

Лямбды **без захвата** обратно совместимы с указателями на функцию — они конвертируются в обычный указатель на функцию. Лямбды с захватом — нет (у них есть состояние).

Пишем свой `MyFunction`

Шаг 1. Шаблонный по типу указателя на функцию

```
template<typename T>
class MyFunction;
```

Шаг 2. Конструктор от функтора/функции

Чтобы извлечь сигнатуру вызываемого объекта, делаем **специализацию шаблона по сигнатуре**:

```
template<typename R, typename Arg>
class MyFunction<R(Arg)> {
    // ...
};
```

> **Почему нельзя сразу так?** Потому что специализируется то, чего ещё не существует. Шаблон с конкретным типом в скобках — это специализация исходного шаблона.

Шаг 3. Оператор вызова

```
R operator()(Arg arg) {
    return R{};
}
```

Шаг 4. Конструктор от типов

```
private:
    // ...

public:
    MyFunction(TFuncPtr ptr) {}

    template<typename T>
    MyFunction(T func) {}
```

Как положить и лямбду, и функтор в указатель на функцию? — спрятать

тип за полиморфизмом.

Type Erasure

Как спрятать `TFunc` (его нельзя выразить через `R` и `Arg`): - Делаем **чисто виртуальную базовую структуру** с чисто виртуальным `operator()`. - Указатель на эту базу храним в поле `MyFunction`. - Внутри объявляем **шаблонный класс-наследник**, который хранит реальный объект (лямбду/функтор) и пробрасывает вызов.

Прикол: - Принимаем в конструктор объекты разного типа, у которых одинаковая семантика: они принимают такие-то аргументы и имеют `operator()`. - Прячем тип создаваемого объекта. - Внутри создаём «обёртку», семантически работающую так же.

Две идеи: 1. **Полная специализация шаблона**, которая тоже является шаблонным классом. 2. **Спрятали реальный тип за базовым полиморфным интерфейсом** (type erasure).

Это и есть `std::function`.

Касты — обзор

- **Implicit** (неявное)
- **Explicit** (явное):
  - `const_cast`
  - `static_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
  - C-style cast

Касты числовых типов

- Преобразование от меньшего ранга к большему безопасно (`short → int` норм).
- При одинаковом ранге, но разной знаковости — могут быть приколы со знаком.

Указатели

- Любой указатель можно кастовать к `void*` и обратно.

- Указатели одного размера можно кастовать друг в друга (но это уже опасно).

Неявные преобразования

- `0` неявно приводится к `false`.
- В `for (int i = 0; i < v.size(); ++i) int` сравнивается с `unsigned long long` → знаковый кастится к беззнаковому, может быть переполнение.

Явный C-style cast

```
int i = 0;
std::cout << *(double*)&i;  // считаем биты int как double — мусор
```

```
int i = 0;
int* p = &i;
double* pd = (double*)p;
double d = *pd;
```

- `int` занимает 4 байта, `double` — 8. Кастуя `int*` к `double*`, мы говорим: «читай 8 байт и интерпретируй как double» — это вылет за границы корректной памяти.
- Представление чисел тоже разное (целые двоичные vs IEEE 754 для double).
- В функцию можно «подсунуть» указатель на другой класс — поле, которого нет, будет интерпретировано как мусор.

C-style cast — опасная штука, никак не проверяет, можем ли мы это делать.

`const_cast`

- Убирает `const` или `volatile` с переменной.
- Может работать с указателями и ссылками на одинаковые типы.

  EXAMPLE: Типичные опасности - Если объект изначально константный, изменение через `const_cast` — **UB**. Забота о корректности — на программисте. - Изменение полей в `const`-методе работает, только если сам объект изначально неконстантный. - Совместимость с C-кодом, где нет `const`.

`static_cast`

- Пытается преобразовать через конструкторы и операторы приведения.
- Работает в **compile-time**.
- Подходит для стандартных типов.
- Умеет приводить указатели внутри одной иерархии классов.
- Может кастовать из `void*`.

Более явный и безопасный, чем C-style cast. Если каст невозможен — ошибка компиляции.

> **EXAMPLE: Типичный пример** Преобразование классов в одной иерархии. Если приводим `Base*` к `Derived*`, а на самом деле там `Base` — поля производного класса не проинициализированы, чтение даст мусор.

```cpp
int main(int, char**) {
    double d = -12.3456789;
    std::cout << d  << std::endl;

    float f = d;
    std::cout << f  << std::endl;

    int i = d;
    std::cout << i  << std::endl;

    uint32_t ui = d;
    std::cout << ui << std::endl;

    char ch = d;
    std::cout << ch << std::endl;

    return 0;
}
```

Какие касты существуют

- **Implicit** (неявное преобразование)
- **Explicit** (явные):
  - `const_cast`
  - `static_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
  - C-style cast

Implicit cast

Происходит автоматически, когда компилятор умеет преобразовать один тип в другой без явного указания. Пример выше: `double` → `float` → `int` → `uint32_t` → `char` — всё это неявные преобразования, **каждое из которых может потерять данные**.

C-style cast

Синтаксис: `(type)expression`

```
int main(int, char**) {
    int i = 1;
    std::cout << *(double*)&i << std::endl;
}
```

Опасен тем, что **компилятор не проверяет корректность** — он просто делает то, что сказано. По сути это «умная» обёртка, которая последовательно пробует `const_cast` → `static_cast` → `reinterpret_cast`. **Лучше использовать явные C++ касты.**

`const_cast`

- Убирает `const` или `volatile` с переменной.
- Преобразует указатели и ссылки на одинаковые типы данных.

  Использовать только если есть **полная уверенность**, что объект изначально не был константным — иначе **undefined behavior**. Обращаться очень аккуратно.

Типичные сценарии: - В константном методе модифицировать что-то (когда сам объект изначально не константный). - Совместимость с C-кодом, где API не принимает `const`-указатели, но не модифицирует данные.

`static_cast`

- Работает в **compile-time** — быстро, без накладных расходов в рантайме.
- Пытается выполнить преобразование через конструкторы и операторы приведения типов.
- Подходит для стандартных типов (предпочтительная замена C-style касту).
- Умеет приводить указатели внутри одной иерархии классов (вверх и вниз).
- Умеет кастовать из `void*` (обратно к конкретному типу).

Гарантия компилятора: если цепочка преобразований невозможна — код **просто не скомпилируется**.

Осторожно с кастом вниз по иерархии (`Base*` → `Derived*`): - Если объект изначально и был `Derived` — всё работает корректно. - Если объект изначально был `Base` — формально получим «полноценный» `Derived`, но новые поля производного

класса никак не проинициализированы → **undefined behavior** (могут лежать какие угодно байты — мусор). - `static_cast` **не проверяет** в рантайме, корректен ли каст вниз. Для этого есть `dynamic_cast`.

`dynamic_cast`

- Преобразует указатели и ссылки **вверх и вниз** по иерархии.
- Работает в **runtime** через **RTTI** (Runtime Type Identification).
- Использует таблицу виртуальных функций для проверки реального типа объекта.

Поведение при неудачном касте: - Если кастуем **указатель** — вернёт `nullptr`. - Если кастуем **ссылку** — бросит исключение `std::bad_cast`.

Цена: - Дополнительное время в рантайме на проверку. - Класс **обязан** иметь хотя бы одну виртуальную функцию (иначе нет vtable → класс неполиморфный → ошибка компиляции).

`reinterpret_cast`

Страшно, бойтесь, противозаконно...

- Кастует **несовместимые типы** — просто переинтерпретирует побитовое представление памяти, **никаких преобразований не делает**.
- Компилятор **ничего не проверяет** и не предупреждает.

Нужно **точно знать**, что лежит в этом блоке памяти. Типичный легитимный сценарий — **сериализация**: кастуем указатель на структуру к `char*` и пишем байты в файл; при чтении кастуем обратно, зная размер структуры.

Подводные камни: - **Выравнивание (alignment)** — у разных типов разные требования. - Размер одних и тех же типов **может различаться** на разных платформах (`long`, `size_t`). - **Порядок байт** — big-endian vs little-endian. - Представление `float`/`double` тоже может различаться.

Если запись и чтение происходят **на одной и той же машине** — об этих проблемах можно не беспокоиться.

Практическое применение — быстрая запись в бинарный файл: кастуем указатель на начало вектора к `char*` и пишем всё одним системным вызовом `write`. Это быстрее (один syscall вместо многих) и компактнее, чем текстовый вывод.

Rvalue reference

- && — rvalue-ссылка (& — lvalue-ссылка).
- Позволяет передавать в функцию `rvalue`.
- Продлевает жизнь временным объектам (так же, как обычная `const&`).
- Move constructor.
- Move assignment operator (`operator=(T&&)`).
- Reference collapsing.

```
int&& func(int&& i) {
    // тут типа работаем с i как с lvalue
    return i;
}

int main() {
    int&& i = 1;
    const int&& j = 2;
    std::cout << func(1);

    int x = 2;
    int&& rx = x;          // ERROR: x — lvalue, нельзя привязать к rvalue&&
    const int&& crx = x;   // ERROR: то же самое

    return 0;
}
```

Важная тонкость: хотя `i` объявлен как `int&&`, **внутри функции** он используется как `lvalue` — у него есть имя, есть адрес. Категория «rvalue» относится к моменту **передачи** аргумента, а не к самому объекту в теле функции.

Правила выбора перегрузки: - Если можно передать по обычной ссылке (`T&`) — передаётся по ней (неконстантный аргумент к неконстантному параметру). - Если есть `rvalue`-перегрузка — `rvalue` передастся через неё. - Иначе `rvalue` передаётся по константной ссылке (`const T&`).

Move-конструктор и move-присваивание

- Принимают на вход `rvalue-reference`.
- Знаем, что источник нам больше не нужен — значит, можно «украсть» его внутренности.

- Например, для динамического массива — просто **передать указатели** на буфер, не копируя содержимое.
- Экономим на пересоздании объекта после знания, что оригинал больше не понадобится.

Что делает move: - Передаёт значения полей в текущий объект. - Оставляет источник в **корректном, но неопределённом состоянии** (так требует стандарт — над объектом ещё можно вызывать деструктор, `operator=` и т.д., но нельзя предполагать, что в нём какие-то конкретные данные). - Очищает ресурсы текущего объекта.

`default` / `delete`, **правило 5** (если определили один из специальных методов — обычно нужно определить все пять: деструктор, copy ctor, copy assign, move ctor, move assign) и **правило 0** (если значения по умолчанию устраивают — не определяй ничего).

```
int main() {
    CArray arr1{5};
    CArray arr2{};
    arr2 = arr1;                  // lvalue → copy assignment
    arr2 = createArray();          // prvalue → move assignment
    arr2 = std::move(arr1);        // xvalue → move assignment
    return 0;
}
```

`std::move`

- Делает из `lvalue` — `xvalue` (eXpiring); при передаче в функцию это `rvalue`.
- **Сам по себе ничего не «двигает»** — просто кастует через `static_cast` к `rvalue`-ссылке. Реальное «движение» делает уже move-конструктор/оператор.

```
template <class _Tp>
typename remove_reference<_Tp>::type&&
move(_Tp&& __t) _NOEXCEPT {
    typedef typename remove_reference<_Tp>::type _Up;
    return static_cast<_Up&&>(__t);
}
```

- После `std::move(obj)` сам `obj` **всё ещё жив и валиден**, но в неопределённом состоянии. Его можно дальше использовать (присваивать, разрушать),

но нельзя предполагать, какие в нём данные.

Copy-and-swap для move

В операторе присваивания до этого делали копию для безопасности. С появлением move можно: - Сделать **один шаблон**: `operator=(CArray other)` — принимает по копии. Если передан lvalue — будет copy ctor; если rvalue — move ctor. Внутри просто свап. - Минус: даже при rvalue будет один лишний move-конструктор. - Если эта цена устраивает — **один** оператор покрывает оба случая (copy-and-swap idiom).

Эффективный `swap`

Раньше — три копирования. С move — три перемещения:

```
template<typename T>
void std::swap(T& x, T& y) {
    T tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
}
```

Forwarding reference (универсальная ссылка)

```
template<typename T>
void function(T&& value) {
    // ...
}

int main(int, char**) {
    Foo* foo = new Foo{};
    auto&& value = foo;
}
```

- Это **не rvalue-ссылка**, а **forwarding reference** (универсальная). Срабатывает только когда `T&&` стоит при **шаблонном** параметре, тип которого выводится; либо в `auto&&`.
- Если передан `lvalue` — `T` выводится как `Foo&`, и `T&&` после reference collapsing становится `Foo&`.

- Если передан `rvalue` — `T` выводится как `Foo`, и `T&&` остаётся `Foo&&`.

Reference collapsing: компилятор «схлопывает» ссылки по правилам: - `Foo& &` → `Foo&` - `Foo&& &` → `Foo&` - `Foo& &&` → `Foo&` - `Foo&& &&` → `Foo&&`

Проблема: внутри функции `value` — это lvalue (у него есть имя). Но мы хотим **сохранить категорию** (lvalue/rvalue) при передаче дальше: - `std::move` — безусловный каст к rvalue (плох: lvalue превратится в rvalue, его «украдут»). - **`std::forward<T>(value)`** — условный каст: - если `T` — lvalue-ссылка, оставляет lvalue; - если `T` — обычный тип (бывший rvalue), кастует к rvalue.

И `std::move`, и `std::forward` используют `static_cast` — это делается **на этапе компиляции**, без рантайм-затрат.

Copy elision

Механика, при которой компилятор **избавляется от лишних копирований**. - **RVO (Return Value Optimization):** если функция возвращает значение, а вызывающий им инициализирует переменную, компилятор может конструировать объект **сразу в памяти переменной** — без вызова copy/move конструктора. - **NRVO (Named RVO):** работает, даже если внутри функции у возвращаемого объекта было имя: `cpp Foo f() { Foo result; // ... return result; // NRVO — result строится сразу в памяти получателя } ` - Там, где **неоднозначно** (например, тернарный оператор возвращает разные именованные объекты) — адрес объекта подставить нельзя, оптимизации не будет.

Резюме

- **`std::move`** — снимает с типа ссылку и возвращает rvalue. Безусловный каст.
- **`std::forward`** — сохраняет исходную категорию аргумента (lvalue или rvalue).

Вариативные шаблоны

В C++11 — рекурсивный вызов функции с уменьшающимся количеством аргументов. На примере функции `to_strings`:

```
template<typename T, typename... Args>
void to_strings(T value, Args... args) {
    // махинации с value
    to_strings(args...);   // вызов функции от всех аргументов, кроме T value
}
```

По факту мы каждый раз преобразовываем только `value`, а потом запускаем эту же функцию от всех аргументов, кроме `value`. Тогда первый аргумент из `args` становится `value`. И так далее. Нужна **базовая** функция от 0 аргументов (или специализация на одном аргументе) — это не очень удобно.

Parameter pack: возможности

- Сам `...` — оператор «развёртки» пакета.
- Модификаторы `const`/`volatile` работают с пакетом обычно.
- `sizeof...(Args)` — **количество** элементов в пакете (а не их суммарный размер).
- `fold-expression` — компактная свёртка пакета через бинарный оператор.

Где может использоваться parameter pack: - Параметр шаблона — 0..n шаблонных аргументов. - Аргументы функции — 0..n аргументов. - В `if constexpr`. - В `sizeof...` — количество элементов в пакете. - При вызове функции через `f(args...)` — она получит **все** аргументы пакета.

Fold-expression (C++17)

Применяет бинарный оператор ко всем элементам пакета **одним выражением**, без рекурсии.

```
template<typename... Args>
auto sum(Args... args) {
    return (args + ...);     // унарная fold справа
}
```

Виды fold:

Запись	Раскрытие
`(args op ...)`	`args0 op (args1 op (args2 op args3))` — правая
`(... op args)`	`((args0 op args1) op args2) op args3` — левая
`(args op ... op init)`	`args0 op (args1 op (args2 op init))` — правая с инитом
`(init op ... op args)`	`((init op args0) op args1) op args2` — левая с инитом

Применимо и для **унарных**, и для **бинарных** операторов.

Разная расстановка скобок важна для **неассоциативных** операций: - Правая: `(args / ...)` → `8 / (4 / 2) = 4` - Левая: `(... / args)` → `(8 / 4) / 2 = 1`

Comma fold pattern

`(func(args), ...)` — последовательность вызовов: `func(arg0), func(arg1), func(arg2), …`

Удобно, когда хочется выполнить действие для каждого элемента пакета.

```
template<typename... Args>
void print_all(Args... args) {
    ((std::cout << args << ' '), ...);
}
```

Класс с переменным числом параметров — `std::tuple`

Главный пример вариативного шаблонного класса. Наивная реализация — через рекурсию-наследование:

```
// Базовый случай — 0 аргументов
template<typename... Ts>
struct tuple {};

// Специализация: голова + хвост
template<typename Head, typename... Tail>
struct tuple<Head, Tail...> : tuple<Tail...> {
```

```
    Head value;

    tuple(Head h, Tail... t)
        : tuple<Tail...>(t...), value(h)
    {}
};
```

- Класс наследуется от `tuple` **с меньшим числом аргументов**.
- В конструкторе инициализируем своё поле и базовый класс.
- Чтобы получить более глубокие поля, можно `static_cast` структуры к её базовому классу.
- Альтернатива — хранить поле `base` (или ссылку на него) и получать значения как `t.base.base.value`.
- Для геттера по индексу делаем шаблонный аргумент-индекс и `using` по нему; нужна также специализация для `Index == 0`.

(Полная реализация `std::tuple` — на следующей лекции.)

Дописываем `tuple` — Getter

Делаем шаблонную рекурсию, чтобы получить тип N-го элемента:

```
template<size_t N, typename Head, typename... Tail>
struct Getter<N, Head, Tail...> {
    using value_type = typename Getter<N - 1, Tail...>::value_type;
};

template<typename Head, typename... Tail>
struct Getter<0, Head, Tail...> {
    using value_type = Head;
};
```

- В шаблоне лежит индекс элемента из `tuple`.
- На каждом шаге специализация откусывает один элемент.
- Отдельная специализация для индекса 0 — конец рекурсии: возвращаем тип `Head`.

Получение значения: когда доходим до 0 — возвращаем `value` текущего уровня.

TAD vs CTAD

Перечислять все типы `tuple` при вызове геттера — громоздко. Хочется, чтобы компилятор сам вывел типы. - **CTAD** (Class Template Argument Deduction) не подходит, потому что для него нужен конструктор. - **TAD** (Template Argument Deduction) — работает для функций:

```
template<size_t N, typename... Ts>
auto& get(tuple<Ts...>& t) {
    return Getter<N, Ts...>::get(t);
}
```

Здесь компилятор сам выведет `Ts...` из аргумента-тюпла.

`make_tuple`

Функция, которая сама выводит все типы:

```
template<typename... Ts>
auto make_tuple(Ts... ts) {
```

```
    return tuple<Ts...>(ts...);
}
```

Подсказки для вывода типов конструктора

Например, `std::vector` можно конструировать из пары итераторов. Чтобы CTAD заработал, в стандартной библиотеке прописаны **deduction guides** — подсказки компилятору о том, какие шаблонные аргументы вывести при конструировании.

Overload pattern

Класс, наследующийся от нескольких лямбд, и подтягивающий их `operator()` через `using`. За счёт того, что лямбды принимают разные типы — выбирается нужная перегрузка.

```
template<class... Ts>
struct overloaded : Ts... { using Ts::operator()...; };
template<class... Ts>
overloaded(Ts...) -> overloaded<Ts...>;   // deduction guide
```

Используется со `std::variant` / `std::visit`.

`std::variant`

- **Строго типизированный union** (хранит одно из значений перечисленных типов).
- Решает главную проблему `union` — не нужно вручную помнить, какой тип сейчас лежит.

Типичное использование: - Конструктор сам определяет, какой тип кладётся. - Типы не приводятся друг к другу. - В `std::get<T>(v)` указываем тип, который мы ожидаем достать; если там лежит другой тип — `std::bad_variant_access`. - Если в варианте лежит `int(11)`, достать как `long` не выйдет (никакого автоматического каста).

`std::visit`

Мёрджим overload pattern и `std::variant`. Вызываем `std::visit(overloaded{...}, variant)` — будет вызвана та функция (лямбда), которая принимает текущий хранимый тип:

```cpp
std::variant<int, std::string, double> v = 42;

std::visit(overloaded{
    [](int i)              { std::cout << "int: " << i; },
    [](const std::string& s){ std::cout << "string: " << s; },
    [](double d)           { std::cout << "double: " << d; }
}, v);
```

Variadic CRTP

Обычный CRTP — derived наследует от `Base<Derived>`. Variadic CRTP — наследование сразу от нескольких базовых шаблонов:

```cpp
template<typename Derived, template<typename> class... Mixins>
struct CRTPBase : Mixins<Derived>... {
    // ...
};
```

Из-за того, что базы тоже шаблонные, при перечислении их в списке нужно указывать, что аргументы шаблонные (через `template<typename> class`).

Метапрограммирование (введение)

Compile-time evaluation: - Шаблон работает как «приём аргументов» на этапе компиляции. - Экономим время в рантайме. - Все данные для вычисления должны быть известны до компиляции.

`constexpr`

- Позволяет функции или переменной быть вычисленной в **compile-time**.
- Если можно посчитать в compile-time — посчитается там; иначе — в рантайме.
- **`constexpr` переменная:**
  - Литеральный тип.
  - Инициализация через константное выражение (явно, `constexpr`-функция и т.д.).
- **`constexpr` функция:**
  - Возвращает литеральный тип.
  - Содержит переменные литеральных типов.

- Не виртуальная.
- Без исключений (в C++11; позже ограничения смягчили).

Теперь достаточно просто написать `constexpr`, а не городить трюки с `enum` (как делали раньше для compile-time констант в шаблонах).

Compile-time evaluation

- Шаблон работает как приём аргументов.
- Экономим на времени в рантайме.
- Все данные для вычисления должны быть известны до компиляции.

Template Specialization

В зависимости от **порядка функций**, специализация шаблона может относиться к разным функциям. Например: - Если у нас сначала шаблон по `T*`, а потом специализация для `int*` — выбирается `int*` (более специфичная). - Если же сначала специализировать `T` как `int*`, а потом написать функцию для `T*` — выбирается более общий вариант, как более «удачный».

Пишем свой `is_same`

```
template<typename T, typename U>
struct is_same {
    static constexpr bool value = false;
};

template<typename T>
struct is_same<T, T> {
    static constexpr bool value = true;
};
```

По дефолту — `false`. Специализация для одинаковых типов — `true`.

Пишем `identity`

Фокус: в шаблоне принимаем тип `T`, а потом значение `T value` — внутри структуры лежит это же значение. Можно сделать инкремент — внутри хранить `T value + 1`.

Если положить в шаблон сразу `auto` — не надо указывать тип:

```
template<auto Value>
struct identity {
    static constexpr auto value = Value;
};
```

А если сделать снаружи `constexpr` переменную, которая держит `value` от структуры — не нужно писать `::value`.

Метафункции, возвращающие типы

- Просто пишем `using` внутри структуры.
- Возвращается тип — а значит, его можно положить в другой шаблон.

```
template<typename T>
struct type_identity {
    using type = T;
};
```

is_same (наивно)

```
template<typename T, typename U>
struct is_same {
    static constexpr bool value = false;
};

template<typename T>
struct is_same<T, T> {
    static constexpr bool value = true;
};

int main() {
    static_assert(is_same<int, int>::value);
    static_assert(!is_same<int, float>::value);
    static_assert(!is_same<int, int&>::value);
    static_assert(!is_same<const int, int>::value);
}
```

Отличает одинаковые T: имеет специализацию для одинаковых типов (`true`), во всех других случаях — `false`.

> Если убрать `::value` и забыть про треугольные скобочки — шаблон почти превращается в обычную функцию.
>
> По сути мы пишем структуру, которая создаёт **метафункцию**. Любую функцию можно попробовать переписать на метафункцию.

identity

В прошлый раз писали `identity` — обычная функция:

```
template<typename T>
T&& identity(T&& value) {
    return std::forward<T>(value);
}

int main() {
    int x = identity(239);
}
```

Переписываем на метафункцию — теперь вычисляется в compile-time:

```
template<typename T, T Value>
struct value_identity {
    static constexpr T value = Value;
};

int main() {
    int x = value_identity<int, 239>::value;
}
```

Ограничения: - Нужно знать данные на момент компиляции. - Только литеральные типы.

Грустно, что нужно указывать `int`. Используем `auto`:

```
template<auto Value>
struct value_identity {
    static constexpr auto value = Value;
};

int main() {
    static_assert(value_identity<239>::value == 239);
}
```

Метафункции прекрасно работают с вариативными шаблонами:

```
template<auto... Value>
struct sum {
    static constexpr auto value = (Value + ...);
};

int main() {
    static_assert(sum<1, 2, 3, 4, 5>::value == 15);
}
```

Обычная функция возвращает конкретное значение. **Метафункция** может вернуть и значение, и даже сам тип.

Два типа метафункций: 1. Возвращающие типы. 2. Возвращающие значения.

```cpp
struct Boo {};

int main() {
    static_assert(
        std::is_same<
            std::type_identity<Boo>::type,
            Boo
        >::value
    );
}
```

Хотим избавиться от `::`. Договорённость такая: если есть метафункция, для неё пишут алиас `что-то_t` (если возвращает тип) или `что-то_v` (если значение):

```cpp
template<class T, class U>
constexpr bool is_same_v = is_same<T, U>::value;

template<typename T>
using type_identity_t = typename std::type_identity<T>::type;

int main() {
    static_assert(
        std::is_same_v<std::type_identity_t<Boo>, Boo>
    );
}
```

integral_constant

Принимает `T` и значение. Объединяет обе идеи: возвращает и тип, и значение.

```cpp
template<typename T, T Value>
struct integral_constant {
    static constexpr T value = Value;
    using value_type = T;
    using type = integral_constant;
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; }
};
```

```cpp
template<bool B>
using bool_constant = integral_constant<bool, B>;

using true_type  = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;
```

> **Интегральный тип** — это целочисленный тип (`bool`, `char`, `int`, `long`, ...). Поэтому константа называется *integral*.

Эта гениальная штука позволяет писать метафункции лаконичнее:

```cpp
template<class T, class U>
struct is_same : std::false_type {};

template<class T>
struct is_same<T, T> : std::true_type {};
```

<type_traits>

Заголовок <type_traits> содержит набор метафункций для работы с типами: - Primary type categories (`is_integral`, `is_pointer`, ...) - Composite type categories (`is_arithmetic`, `is_object`, ...) - Type properties (`is_const`, `is_signed`, ...) - Supported operations (`is_constructible`, `is_assignable`, ...) - Type relationships (`is_same`, `is_base_of`, ...) - Const-volatility specifiers (`remove_const`, `add_volatile`, ...) - и др.

Свой is_pointer

```cpp
template<class T> struct is_pointer                    : std::false_type {};
template<class T> struct is_pointer<T*>                : std::true_type  {};
template<class T> struct is_pointer<T* const>          : std::true_type  {};
template<class T> struct is_pointer<T* volatile>       : std::true_type  {};
template<class T> struct is_pointer<T* const volatile> : std::true_type  {};

template<typename T>
inline constexpr bool is_pointer_v = is_pointer<T>::value;
```

Проблема: `const`, `volatile` мешают обычному TAD по `T*` — нужны отдельные специализации. На двух квалификаторах уже четыре комбинации, на трёх было бы

восемь — комбинаторика.

Хотим **избавиться от обилия перегрузок**. Напишем метафункцию, снимающую const:

```
template<typename T>
struct remove_const {
    using type = T;
};
template<typename T>
struct remove_const<const T> {
    using type = T;
};
```

Аналогично для volatile:

```
template<typename T>
struct remove_volatile {
    using type = T;
};
template<typename T>
struct remove_volatile<volatile T> {
    using type = T;
};
```

Объединяем:

```
template<typename T>
using remove_cv_t = typename remove_volatile<typename remove_const<T>::type>::type;
```

Переписываем is_pointer через remove_cv:

```
template<typename T>
inline constexpr bool is_pointer_v = is_pointer<remove_cv_t<T>>::value;
```

Александр Павлович сказал: *not so good* — ведь нельзя пользоваться просто is_pointer<T> (без _v). Перепишем полностью:

```
template<typename T>
struct is_pointer_inner : std::false_type {};

template<typename T>
```

```cpp
struct is_pointer_inner<T*> : std::true_type {};

template<typename T>
struct is_pointer : is_pointer_inner<remove_cv_t<T>> {};
```

SFINAE

- **«Substitution Failure Is Not An Error»** (неудавшаяся подстановка — не ошибка).
- Если для перегрузки функции невозможно вывести шаблонные параметры (type deduction) и инстанцировать функцию, **это не ошибка компиляции**. Такая перегрузка просто опускается (ill-formed) — а компилятор пробует остальные.
- SFINAE работает только с перегрузками функций.
- SFINAE смотрит только на **заголовок** функции (сигнатуру), а не на тело.
- SFINAE отбрасывает только **шаблонные** функции.
- За счёт SFINAE можно создавать условия, при которых перегрузка отбрасывается, оставляя только подходящие (well-formed).

Полезная конструкция: `T::*` (указатель на член класса)

Хотим функцию, по-разному работающую для пользовательских типов и всех остальных. - Шаблонная функция с параметром `int T::*` — будет валидна только для классов/структур (для `int` или `double` такого члена быть не может). - Шаблонная перегрузка с `...` — для всех остальных типов.

Ошибки компиляции не будет — выберется подходящая.

> Если функцию не вызывать — её можно только объявить, не определяя. `decltype` не вызывает функцию — смотрит только на сигнатуру.

```cpp
void print(...) {
    std::cout << "No implementation\n";
}

void print(int i) {
    std::cout << "int value " << i << std::endl;
}
```

```
int main() {
    print(1);
    print("Hello world");
    print(1, 1);
}
```

Усложняем:

```
struct Boo {};

int main() {
    using IntBooMemberPtr = int Boo::*;      // OK
    using IntIntMemberPtr = int int::*;       // ERROR — int не класс
}
```

`int Boo::*` — указатель на член класса `Boo` типа `int`.

Делаем void-функции:

```
template<typename T>
void foo(int T::*) {
    std::cout << "foo(int T::*)\n";
}

template<typename T>
void foo(...) {
    std::cout << "foo(...)\n";
}
```

Используем для определения, является ли тип классом:

```
template<typename T>
std::true_type  can_have_member_ptr(int T::*);

template<typename T>
std::false_type can_have_member_ptr(...);

int main() {
    static_assert(decltype(can_have_member_ptr<Boo>(nullptr)){});
    static_assert(!decltype(can_have_member_ptr<int>(nullptr)){});
```

```
}
```

Метафункция `is_class`:

```
template<typename T>
std::true_type  check_class(int T::*);

template<typename T>
std::false_type check_class(...);

template<typename T>
struct is_class : decltype(check_class<T>(nullptr)) {};

template<typename T>
constexpr bool is_class_v = is_class<T>::value;

int main() {
    static_assert(is_class_v<Boo>);
    static_assert(!is_class_v<int>);
}
```

std::enable_if

- Если в шаблон передать `false` — внутри нет `::type`, попытка использовать `enable_if_t<false>` приведёт к **ошибке подстановки** → перегрузка отбрасывается (SFINAE), компилируется другая.
- Если `true` — `::type` есть, всё хорошо.

В чём фокус: теперь можно класть в `enable_if` метафункцию (например, `is_pointer_v<T>`) и явно показывать компилятору, какие перегрузки разрешены, а какие — нет.

```
template<bool, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> {
    using type = T;
};
```

```cpp
template<bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

```cpp
template<typename T>
void print(const T& value,
           std::enable_if_t<std::is_pointer_v<T>, void*> = nullptr) {
    std::cout << *value << std::endl;
}

template<typename T>
void print(const T& value,
           std::enable_if_t<!std::is_pointer_v<T>, void*> = nullptr) {
    std::cout << value << std::endl;
}

int main() {
    int i = 1;
    print(i);
    print(&i);
}
```

> То же самое можно сделать через `if constexpr` (внутри одной функции). Что лучше — SFINAE или `if constexpr` — дело вкуса; `if constexpr` обычно читабельнее, но не позволяет различать **перегрузки**, только ветви внутри одной функции.

Metaprogramming + variadic

С вариативными шаблонами можно, например, проверить, что **все** типы интегральные:

```cpp
template<typename... Ts>
constexpr bool all_integral = (std::is_integral_v<Ts> && ...);
```

> Домашнее задание — написать свою `conjunction`.

> До концептов ограничения на шаблонные параметры выражались через `enable_if`, `conjunction`, `is_integral`, `void_t` и подобные SFINAE-трюки. Concepts — это **официальный, читаемый способ** делать то же самое.

Зачем нужны Concepts

Concepts позволяют задавать **ограничения на шаблонные параметры** — указывать, каким требованиям должен удовлетворять тип, чтобы шаблон инстанцировался.

Преимущества перед альтернативами: - **Проще, чем `enable_if`** — не нужно городить SFINAE. - **Лучше, чем `if constexpr`** — `if constexpr` работает внутри тела функции и не влияет на разрешение перегрузок; концепты участвуют в overload resolution. - **Понятные ошибки компиляции** — компилятор сразу говорит, какое требование не выполнено. - **Неявный SFINAE** — если концепт не выполнен, перегрузка просто не рассматривается.

Синтаксис `requires`

Ключевое слово `requires` пишется перед телом функции и задаёт условие для выбора перегрузки:

```
// Перегрузка для указателей
template<typename T>
requires std::is_pointer_v<T>
void print(const T& value) {
    std::cout << *value << std::endl;
}

// Перегрузка для всех остальных типов
template<typename T>
void print(const T& value) {
    std::cout << value << std::endl;
}
```

Объявление концепта

`concept` — это **именованное требование** к типам, которое можно переиспользовать:

```cpp
template<typename T, typename U>
concept Addable = requires(T a, U b) {
    a + b;   // тип должен поддерживать operator+
};

template<typename T, typename U>
requires Addable<T, U>
auto add(const T& a, const U& b) {
    return a + b;
}

int main() {
    add(1, 2);          // OK
    add(Foo{}, Foo{});  // OK, если Foo поддерживает operator+
}
```

Способы использования концепта

```cpp
// 1. В requires-clause перед телом функции
template<typename T, typename U>
requires Addable<T, U>
auto add(const T& a, const U& b);

// 2. Прямо в списке шаблонных параметров
template<typename T, Addable<T> U>
auto add(const T& a, const U& b);

// 3. Сокращённый синтаксис с auto (без явного шаблона)
auto add(const Addable auto& a, const auto& b);
```

Виды требований внутри `requires`-выражения

1. Simple requirement (простое требование)

Просто выражение, которое должно быть синтаксически корректным. Возвращаемое значение не проверяется.

```
template<typename... Args>
concept Addable = requires(Args... args) {
    (args + ...);   // simple requirement: operator+ должен существовать
};
```

2. Nested requirement (вложенное требование)

requires-выражение внутри requires-блока — позволяет добавить булево условие:

```
template<class T, typename... TArgs>
constexpr bool are_all_same = std::conjunction_v<std::is_same<T, TArgs>...>;

template<typename... Args>
concept Addable = requires(Args... args) {
    (args + ...);                       // simple
    requires sizeof...(Args) > 1;       // nested: аргументов должно быть больше одного
    requires are_all_same<Args...>;     // nested: все типы должны совпадать
};
```

3. Compound requirement (составное требование)

Проверяет не только корректность выражения, но и **тип результата** и/или noexcept:

```
// Вспомогательный алиас для получения типа первого аргумента пакета
template<typename First, typename...>
struct first_arg { using type = First; };

template<typename... Ts>
using first_arg_t = typename first_arg<Ts...>::type;

template<typename... Args>
concept Addable = requires(Args... args) {
    (args + ...);                       // simple
    requires sizeof...(Args) > 1;       // nested
    requires are_all_same<Args...>;     // nested
    // compound: выражение не бросает исключений,
```

```
    // и результат имеет тип первого аргумента
    { (args + ...) } noexcept -> std::same_as<first_arg_t<Args...>>;
};
```

4. Type requirement (требование к типу)

Проверяет, что вложенный тип существует:

```
template<typename T>
concept HasValueType = requires {
    typename T::value_type;   // у T должен быть вложенный тип value_type
};
```

> В примере с `Addable` выше type requirement явно не используется, но синтаксис именно такой: `typename SomeType;` внутри `requires`-блока.

Полный пример с вариативным шаблоном

```
template<typename... Args>
requires Addable<Args...>
auto add(Args&&... args) {
    return (args + ...);
}

int main() {
    add(1, 2, 3, 4);      // OK
    add(Foo{}, Foo{});    // OK, если Foo удовлетворяет Addable
}
```

Область применения

- Выбор перегрузки функции в зависимости от свойств типа.
- Ограничение шаблонных классов/функций с **читаемыми** ошибками.
- Замена громоздких SFINAE-конструкций на выразительные именованные требования.

Закон Мура и почему появилась многопоточка

Закон Мура: число транзисторов на микрочипах удваивается каждые ~2 года. Это значило, что одна и та же программа со временем работала ~в 2 раза быстрее (ну, не совсем, но как-то так).

В какой-то момент уперлись в физический барьер — уменьшать размер транзисторов дальше тяжело. Придумали: вместо повышения производительности **одного ядра** — делать **многоядерные** процессоры.

То есть появилась физическая возможность производить несколько вычислений за один такт. Однопоточные программы это **не используют**.

Однопоточка:

```
graph LR
    A[Действие1] --> B[Действие2]
    B --> C[Действие3]
    C --> D[Действие4]
    D --> E[Действие5]
    E --> F[Действие6]
```

Многопоточка (две независимые цепочки):

```
flowchart LR
    subgraph Left
        direction TB
        A["Действие1"] --> B["Действие2"]
        B --> C["Действие3"]
    end

    subgraph Right
        direction TB
        D["Действие4"] --> E["Действие5"]
        E --> F["Действие6"]
    end

    Left ~~~ Right
```

В реальных программах — обычно граф зависимостей:

```
graph LR
    A[Действие1] --> B[Действие2]
    A --> C[Действие3]
    C --> D[Действие4]
    B --> E[Действие5]
    D --> E
    E --> F[Действие6]
```

(Лаба про расписания be like.)

Concurrency vs Parallelism

- **Parallelism** — физическое выполнение нескольких действий **одновременно** (требует нескольких ядер).
- **Concurrency** — выполнение двух или более задач **одновременно с точки зрения логики**: они могут чередоваться на одном ядре, но программа структурирована так, чтобы они «шли параллельно».

В курсе фокус на **concurrency** — нас не очень волнует количество ядер, важно научиться запускать независимые потоки выполнения.

С чего начнём? Конечно же, с Hello World!

```cpp
#include <thread>
#include <iostream>

int main(int argc, char** argv) {
    std::thread tr([]() { std::cout << "Hello World" << std::endl; });
    tr.join();
    return 0;
}
```

`std::thread`

Прикольная библиотека для многопоточки. Основное API: - Конструктор принимает функцию/лямбду и её аргументы — запускает новый поток. - `join()` — заблокировать текущий поток, пока запущенный не завершится. - `detach()` — отвязать поток (он продолжит работать самостоятельно, мы про него «забываем»).

Processes vs Threads

Раньше: каждая программа = отдельный **процесс** (отдельный поток выполнения), процессы изолированы — не могут читать память друг друга.

Сейчас: программа может запускать несколько **потоков** внутри одного процесса (потоки делят память).

- Каждый процесс содержит хотя бы один поток.
- Потоки **шарят общие ресурсы** процесса (память, файловые дескрипторы и т.д.).
- У потоков **общее виртуальное адресное пространство**, но у каждого свой стек.

```cpp
#include <iostream>
#include <thread>

int main(int argc, char* argv[]) {
    for (int i = 0; i < 4; ++i) {
        std::thread tr{
            []() {
                int i = 0;
                std::cout << &i << "\n";
            }
        };
        tr.detach();
    }

    std::getchar();
}
```

Вывод (адреса локальной переменной в каждом потоке):

```
0x16d28af34
0x16d3a2f34
0x16d316f34
0x16d1fef34
```

Видно, что **у каждого потока — свой стек**, поэтому одинаковая локальная переменная лежит по разным адресам.

Sequential vs Parallel

Заполняем 1 миллиард элементов в векторе через `rand`: - sequential: ~20 секунд - parallel: ~14 секунд

Вынесли аллокацию вектора из обеих веток (раньше аллокация была и в sequential, и в parallel; теперь она в `main`, и оба варианта работают с одним `values`): - sequential: ~8 секунд - parallel: ~2 секунды

> **Мораль:** memory allocation — дорого; всегда стоит учитывать её при сравнении.

Закон Амдала

Не все алгоритмы хорошо параллелятся. Например, **bubble sort** — параллелится плохо: на каждом шаге зависимость от предыдущего. Чем больше доля **последовательного** кода — тем меньше выигрыш от параллелизма (что и формализует закон Амдала).

Проблемы многопоточки

Race Condition (гонка)

Ошибка проектирования многопоточной системы, при которой результат зависит от того, **в каком порядке выполняются части кода**.

Классика: 4 потока делают `result += data[i]`. Под капотом каждый поток: 1. Читает `result` в регистр. 2. Читает `data[i]` в другой регистр. 3. Складывает. 4. Записывает обратно в `result`.

Если несколько потоков делают это одновременно, чтения «перекрываются» — потерянные обновления, сумма получается неправильной.

Решения: - **`mutex`** — позволяет защитить часть данных от одновременного доступа. Поток захватил мьютекс — остальные ждут. После разблокировки — следующий. - Локальные `result` в каждом потоке, потом одно сложение под мьютексом — гораздо эффективнее, чем мьютекс на каждом инкременте. - **Condition variables** — позволяют потоку ожидать наступления условия. - **Semaphores** — счётный мьютекс (разрешает доступ N потокам одновременно). - **`std::atomic`**: - Низкоуровневые инструкции процессора. - Хорошо подходит для простых операций (`add`, `store`, `exchange`). - Не подходит для сложных синхронизаций. - Имеет полные и частичные специализации.

Deadlock

Ситуация в многозадачной среде, когда несколько процессов **бесконечно ждут ресурсов**, которые занимают сами эти процессы (поток A держит ресурс X и ждёт Y, поток B держит Y и ждёт X — оба замерли навсегда).

Livelock

Ситуация, в которой система **не застревает** (как при дедлоке), а **занимается бесполезной работой** — состояние меняется, но полезного прогресса нет. Классическая аналогия: два человека в коридоре пытаются разойтись, шагают одновременно в одну и ту же сторону, потом одновременно в другую — двигаются, но не расходятся.

Thread Pool

Механизм, который **эффективно управляет потоками и переиспользует их**. Предоставляет ограниченное число заранее созданных потоков, готовых выполнять задачи из общей очереди. Это решает две проблемы: - Не платим за создание/уничтожение потока на каждую задачу. - Не создаём бесконтрольно тысячи потоков — пул ограничен.

(По коду из презентации — см. слайды.)