Содержание

1 Move-семантика.

Мотивация. Пусть у нас есть

```
#include <string>
#include <vector>

int main()
{
    std::string string = "Some-very-long-string";
    std::vector<std::string> strings;

    strings.push_back(string);
}
```

Как мы знаем, в последней строке происходит копирование. А если строка очень длинная, то это долго. И не всегда нам это надо, иногда исходная строка нам не нужна, хочется просто указатели поперекидывать и всё.
Хуже ситуация обстоит с переаллокациями, потому что там происходят копирования (очень много копирований). *Вообще в C++03 компилятор мог знать, что* `std::vector<std::string>` *можно как-то оптимизировать*, но если тип ваш, то нет.
А есть ещё другая проблема: есть некопируемые типы. Правда ли, что мы не можем сделать `std::vector<std::fstream>`? Традиционное решение — `std::vector<std::fstream*>`. Но это, понятно, не очень (это indirection и не exception-safe). Вариант, с которым можно жить: `std::vector<boost::shared_ptr<std::fstream>>` (до C++11 не было `std::shared_ptr`), но это лишние расходы под счётчик ссылок (который всегда 1 или 2).

Избегание копирования. Rvalue-ссылки — по сути метод оптимизации копирований. Поэтому давайте сначала поймём, где копирование есть, а где — нет.

```
#include <string>

void f(std::string);
std::string g();

int main()
{
    f(g());
}
```

В функции `main` осуществляется ноль копирований. Почему?

- Сначала посмотрим на `f`. Что происходит, когда мы передаём в функцию какой-то большой тип? Например, GCC делает две вещи: копирует объект и передаёт его по ссылке. Но тут возникает вопрос: а где происходит копирование, внутри `f` или вне? Все компиляторы C++ делают вне. Потому что иначе вы никак не сможете этой копии избежать. А иногда избежать хочется, потому что не всё имеет смысл копировать. А если точнее, **по стандарту при передаче в функцию копируются только lvalue**.

- Теперь давайте разберёмся с возвращаемыми значениями. Что происходит в компиляторе, когда вы возвращаете что-то большое? Оттранслировать возврат большой структуры можно в передачу дополнительного указателя в функцию, по которому пишется ответ. Во что оттранслируется, например, вот это?

```
my_type g()
{
    return my_type();
}
```

Вот что-то примерно такое:

```
void g(my_type* uninitialized_result)
{
    new (uninitialized_result) mytype();
}
```

То есть мы просто на нашей неинициализированной памяти вызываем конструктор. Иногда, правда, это конструктор копирования от чего-то. От чего это зависит? Вот тут уже не так просто. По сути ситуация следующая. Если вы возвращаете prvalue (это особый вид rvalue, смотри ниже), то копирования не происходит (это называется RVO — return value optimization). Но иногда его также не происходит, если вы возвращаете lvalue. Например, вот когда:

```
std::array<int, 100> h()
{
    std::array<int, 100> arr;

    arr[2] = 42;
    arr[3] = 43;

    return arr;
}
```

Тут никто не мешает вам транслировать это так:

```
void h(std::array<int, 100>* ununitialized_result)
{
    new (ununitialized_result) std::array<int, 100>();

    (*ununitialized_result)[2] = 42;
    (*ununitialized_result)[3] = 43;
}
```

Это называется NRVO (named return value optimization).

К move-семантике. Давайте посмотрим на то, что можно было бы сделать вместо копирования. Ну, теоретически, мы могли бы сделать что-то, что похоже на копирование, но такое, которое может менять объект. Типа такого:

```
string(string& other) // Типа перемещение.
    : data(other.data)
    , size(other.size)
    , capacity(other.capacity)
{
    other.size = other.capacity = 0;
    other.data = nullptr;
}
```

Таким образом можно жить и с контейнерами (они обычно работают с указателями), и с потоками данных (скопировать все данные одного в другой, а другой сделать как будто закрытым).
Хорошо, вот выдумали мы такую операцию. Как бы она работала для строк, например? В каком состоянии она бы оставляла строку, которую вы перемещаете? А вот unspecified, потому что у вас может быть small-object оптимизация, и маленькую строку проще скопировать, чем скопировать и очистить.
Хорошо. А какими бы ещё свойствами обладала бы наша выдуманная операция? А нам надо было бы к многим методам (например, `push_back`) присобачить вариант, который такой же, но для перемещения. Это неудобно. Чисто теоретически можно было бы пытаться реализовать то же самое на C++03, но была бы куча проблем, поэтому язык расширили и...

Синтаксис rvalue-ссылок. ...и поэтому придумали новый вид ссылок.

```
int a;

int&  l1 = a;  // Успех.
int&  l2 = 42; // Ошибка.

int&& r1 = a;  // Ошибка.
int&& r2 = 42; // Успех.
```

Теперь мы можем написать вот это:

```
template <class T>
class vector
{
    // ...

    void push_back(const T& element) { /* Для lvalue. */ }
    void push_back(T&&)              { /* Для rvalue. */ }
};
```

Первый копирует, второй перемещает.

Специальные функции-члены класса. Понятно, что теперь мы можем и наши объекты перемещать друг в друга. А значит помимо копирующего конструктора и копирующего оператора присваивания, теперь есть и перемещающая пара.

```
class my_type
{
    my_type(my_type&&);
    my_type& operator=(my_type&&) &;
};
```

std::move. Всё это, конечно, прелестно, но давайте научимся перемещать существующий объект. Ну, изи вообще.

```
std::string string;
std::vector<std::string> strings;

strings.push_back(static_cast<std::string&&>(string));
```

Правда, это уже есть, и называется `std::move`.

Некоторые интересные проблемы.

```
class person
{
private:
    std::string name;

public:
    person(const std::string& name)
        : name(name)
    {}

    person(std::string&& name)
        : name(name)
```

```
    {}
};
```

Корректен ли этот код?
Увы, нет, потому что во втором конструкторе `name` — это lvalue (переменная же), поэтому правильно так:

```
class person
{
private:
    std::string name;

public:
    person(const std::string& name)
        : name(name)
    {}

    person(std::string&& name)
        : name(std::move(name))
    {}
};
```

Best practices. А давайте возьмём предыдущий пример, но добавим ещё `surname`. Нам что, 4 конструктора писать? Не хочется, знаете ли. А смотрите, как можно:

```
class person
{
private:
    std::string name, surname;

public:
    person(std::string name, std::string surname)
        : name(std::move(name)), surname(std::move(surname))
    {}
};
```

Теперь смотрите. Тот параметр, который надо скопировать, скопируют. Который не надо — не скопируют. А потом эти копии мы просто переместим в наши переменные, и всё хорошо, и слава тебе, Буреотец.
Понятно, что тут вы лишний раз перемещаете то, что скопировали, и если вы предполагаете, что ваш тип долго перемещать, то так не надо. Однако это ещё недо поискать такой тип, что его перемещение будет дорогим.

Worst practices.

```
std::string foo()
{
    std::string res;
    res += "Hello, ";
    res += "world!";
    return res;
}
```

```
std::string&& foo()
{
    std::string res;
    res += "Hello, ";
    res += "world!";
    return std::move(res);
    // Без move не компилируется.
}
```

```
std::string foo()
{
    std::string res;
    res += "Hello, ";
    res += "world!";
    return std::move(res);
}
```

Правильный вариант здесь первый. Второй в принципе не работает (вы возвращаете ссылку на локальную переменную из функции), а третий предотвращает NRVO. Если у вас работает в первом

случае NRVO, оно и так работает, без перемещения. Если не сработает, то там и так будет вызываться конструктор перемещения.

xvalue.

```
struct mytype {};

mytype rvalue();
mytype& lvalue();

void foo(const mytype&);
void foo(mytype&&);

mytype test()
{
    foo(rvalue());   // rvalue.
    foo(lvalue());   // lvalue.

    return rvalue(); // rvalue, RVO.
    return lvalue(); // lvalue, нет RVO.
}
```

А теперь давайте создадим вот такую штуку:

```
struct mytype {};

mytype&& unknown();

void foo(const mytype&);
void foo(mytype&&);

mytype test()
{
    foo(unknown());   // rvalue.

    return unknown(); // Не можем RVO, нам же ссылку дали, значит lvalue.
}
```

Так вот такое промежуточное — это xvalue. Он считается подвидом rvalue, а те rvalue, которые не xvalue, называются prvalue.

Увеличение времени жизни *(since C++03)*. Давайте посмотрим на вот это:

```
const my_type& ref = my_type();
```

Казалось бы, `ref` ссылается на некорректный объект, потому что он же временный, его удалят в этой же строке. Но это не практично, как оказалось, поэтому есть правило продление жизни. **Когда вы вешаете на временный объект (prvalue) ссылку (`const&` или `&&`), время жизни временного объекта длится до окончания жизни ссылки.**

2 Intrusive контейнеры.

Наблюдение касательно двусвязных списков. Насколько мы знаем, двусвязные списки, например, используются очень редко. Да, они поддерживают быстрое удаление середины, но эту середину обычно надо найти, поэтому в среднем не очень хорошо получаются. Да, они поддерживают splic'ы,

но это бывает нужно очень редко.
Но есть же те 0.1% програм, где двусвязные списки используются (то же ядро Linux, где их тьма). Они любо тупые, либо что-то знают.

Пример. Давайте вообразим себе то, что у нас есть StarCraft. Там все юниты хранятся в двусвязном списке. А мы хотим хранить какое-то их подмножество. Тривиальное решение: `std::unordered_set<unit*>`. Почему `unordered_map`? Ну, потому что мы хотим делать юнит не-выбранным, без `unordered_set` сложно.
О'кей, как это чудо выглядит в памяти? Ну, есть хэш-таблица, в ней по каждому ключу список, в каждом элементе которого указатель на другой список. Кажется, что список внутри таблицы избыточен. Можно сделать таблицу с открытой адресацией, но давайте рассмотрим другое решение:

```
struct unit
{
    unit* next;
    unit* prev;
    // ...
    unit* selected_next;
    unit* selected_prev;
};
```

То есть мы берём наш основной список, и мы «провязываем» сквозь него выбранные узлы. По факту в StarCraft выбранные юниты хранятся не так, но много чего другого — так. И у этого куча преимуществ:

- Нет аллокаций при вставке/удалении
- Двусвязный список содержит на константу операций меньше, чем таблица.
- Лучшая локальность ссылок (не заводятся узлы для таблицы).

А ещё, кстати, к этой задаче можно приспособить и `std::list`. Взять две штуки, первый хранит все элементы, а второй — указатели на выбранные из первого (в то время как в первом есть указатели обратно). Выглядит это как-то так:

```
struct unit
{
    // ...
    std::list<unit*>::iterator backlink;
    // ...
};

std::list<unit*> selected;
```

В случае предыдущего способа (с двумя указателями внутри) проблема в том, что мы храним два указателя даже для тех, кому не надо, а в случае с этим способом — расходы имеются в виде лишнего указателя. Что из этого выгоднее — зависит от того, насколько много у вас выделенных юнитов.

Ещё пример. А теперь вот что смотрите: совершенно в любую структуру мы можем провязать внутрь дерево поиска, чтобы там что-то искать. Или тот же список, чтобы хранить подмножество. Да что там, хоть хэш-таблицу со списками. Используя это чудо, можно создать, например, такую структуру данных как LRU-кэш (least recently used):
У нас есть функция, которая долго считается. И мы хотим кэшировать её результаты. Нам поможет `std::map`, но тогда память будет бесконечно расти, поэтому мы хотим выкидывать то, к чему не обращались дольше всего. Как это сделать? Ну, провязать списком, у которого в хвосте самые давно используемые элементы. Произошло обращение — внутри списка перекидываем элемент в начало. Добавляют новый — удаляем старый из хвоста.

```
struct node
{
    Key key;
    Value value;

    // Ordered by key:
    node* parent;
    node* left;
    node* right;

    // Ordered by access time:
    node* next;
    node* prev;
};
```

Вынесение общего кода. Хм-м, кажется, мы умеем встраивать любую структуру данных в любую. Хочется как-то промышленно это делать, а не руками вписывать два поля в каждую структуру.

На C. На C мы бы сделали так:

```
struct list_node
{
    list_node next;
    list_node prev;
};

struct unit
{
    // ...
    list_node everything_list;
    list_node selected_list;
};
```

Тут есть существенная проблема: наш список ссылается на `list_node`, а не на `unit`. Делается вот такое чудо:

```
#define container_of(ptr, type, member) \
    ((type*)((char*)(ptr) - offsetof(type, member)))
```

Это минималистичный вариант, который не проверяет типы, например, но суть ясна: по полю структуры мы достаём саму структуру. Подобная штука используется в C в ядре Linux везде вообще, а в C++ с этим есть сложности. Во-первых, это большой UB, который, конечно, работает, но давайте так не надо. Во-вторых, `offsetof` применяется не ко всем штукам. Например, если есть виртуальна функция, туда по стандарту нельзя.

На C++. А давайте найдём аналогичную ситуацию в C++ (когда у нас указатель надо двигать походим образом). А это же множественное наследование! Мы это проговаривали в подобном разделе, вот такой пример там был:

```
struct base1
{
    int x;
};
struct base2
{
    int y;
```

```
};
struct derived : base1, base2 {};

base2& to_base2(derived& d)
{
    return static_cast<base2&>(d); // Тут происходит смещение указателя.
}
```

Следующая проблема: нам надо дважды отнаследоваться от одного и того же. Это по канону пишется при помощи тегов:

```
template <class Tag> // Не используется внутри
struct list_element
{
    list_element* next;
    list_element* prev;
};

struct everything_tag;
struct selected_tag;

struct unit : list_element<everything_tag>, list_element<selected_tag>
{
    // ...
};

intrusive_list<unit, everything_tag> all_units;
intrusive_list<unit, selected_tag> selected_units;
```

Или, даже чуть проще,

```
template <class Tag> // Не используется внутри
struct list_element
{
    list_element* next;
    list_element* prev;
};

struct unit : list_element<struct everything_tag>, list_element<struct selected_tag>
{
    // ...
};

intrusive_list<unit, everything_tag> all_units;
intrusive_list<unit, selected_tag> selected_units;
```

Реализация. Когда-то давно появился proposal в стандарт, чтобы добавить туда intrusive контейнеры, но он как-то забросился. Зато контейнеры есть в boost'е, причём там они умеют работать и через наследование, и через `offsetof`. Более того, там есть возможность 3 различными способами взаимодействовать с контейнерами (ничего особенного не делать, автоматически отвязываться в деструкторе, проверять, что вы не вставляетесь в контейнер дважды).

Сравнение с обычными контейнерами.

	Intrusive	Не intrusive
Менеджмент памяти	Внешнее	Внутреннее
Время на вставку/удаление	Быстрее	Медленнее
Локальность памяти	Лучше	Хуже
Можно вставить один объект в два контейнера	Нет	Да
Гарантия исключений	Nothrow	Как попало
Получить итератор из значения	Тривиально	Очень сложно
Использование памяти	Минимально	Не настолько минимально
Вставить объект, сохраняя полиморфизм	Можно	Нельзя из-за slicing'а
Пользователь должен изменить определение объекта	Да	Нет
Возможность копировать контейнер	Нельзя	Можно
Время жизни определяется...	Пользователем	Контейнером
Возможность сломать инварианты контейнера	На изи вообще	Очень сложно
Аналитика потокобезопасности	Сложная	Простая

Мультииндекс. Есть такая весёлая штука, называется boost `multi_index_container`. Это штука, в которую можно запихнуть, что вы хотите иметь внутри заданные intrusive-контейнеры, но чтобы они владелись. И именно этим и занимается мультииндекс. Например, написав вот такую штуку, мы получим LRU-кэш:

```
boost::multi_index::indexed_by<
    boost::multi_index::sequenced<>,
    boost::multi_index::ordered_unique<
        boost::multi_index::member<value_type, const key_type, &value_type::first>>>
```

Потому что тут последовательность и упорядоченность по полю `first` типа `key_type` в структуре типа `value_type`.

3 shared_ptr.

Как мы знаем, умный указатель — это то, что позволяет ссылаться на объект, и в деструкторе в том или ином случае удаляет его. С `unique_ptr` мы уже знакомы, например.
`shared_ptr` же — это штука, которую можно скопировать, и вы будете просто копировать указатели до тех пор, пока все указатели не умрут. Когда умрёт последний, объект удалят. Чтобы понять, что последний умер, используется просто счётчик.

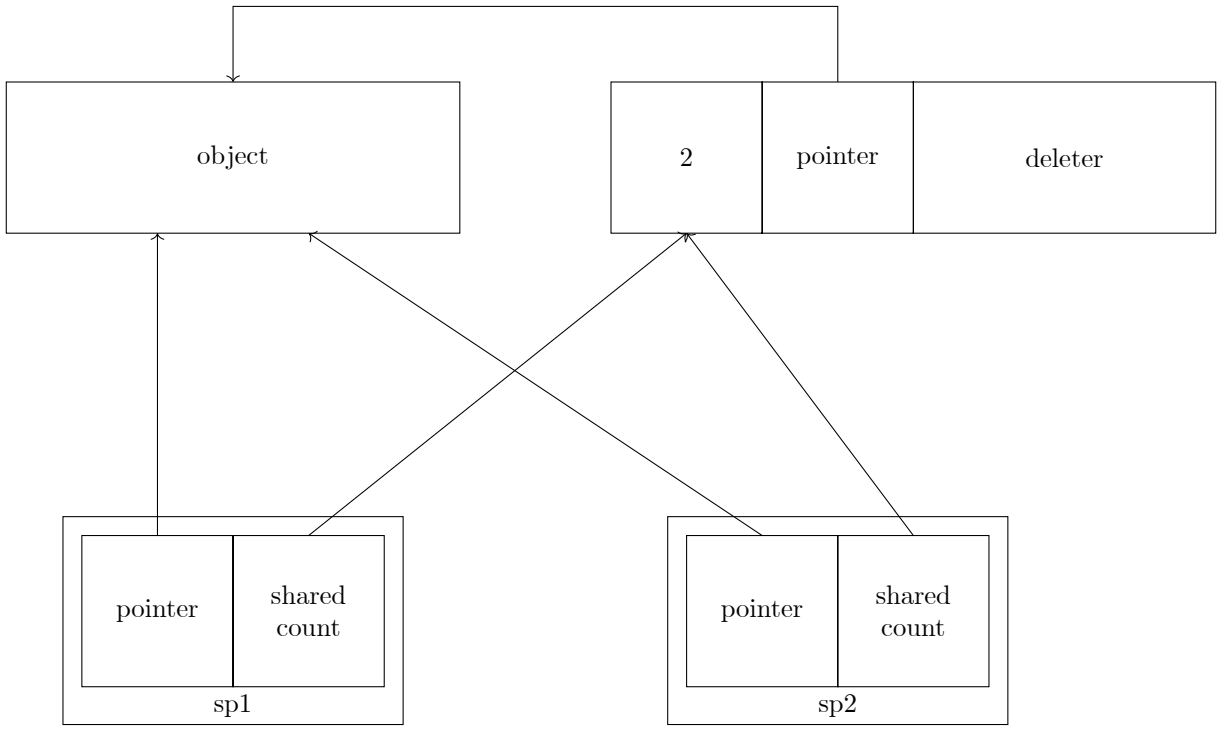
Bad practices. Не надо абьюзить `shared_ptr` (как, например мы уже обсуждали пример в секции про move). Используйте `shared_ptr` тогда, когда вы уверены в том, что именно как указатель вы хотите его использовать.

Фичи этого чуда. Но вообще там же, где число, можно хранить информацию о том, как мы создали объект. А точнее, информацию, как его удалять. Это может быть `free`, `delete`, какой-нибудь возврат объекта в наш пул, что-то ещё.
Давайте вот на что посмотрим. У нас есть машина. И у неё есть 4 колеса. И нам хочется, чтобы кто-то мог отдельно ссылаться на колёса, а не на саму машину. Более того, вам может захотеться, чтобы если кто-то ссылался на колёса, то сама машина не удалялась. Это делается так:

```
std::shared_ptr<vehicle> v(new vehicle());
std::shared_ptr<wheel> v(v, &v->wheel[2]);
```

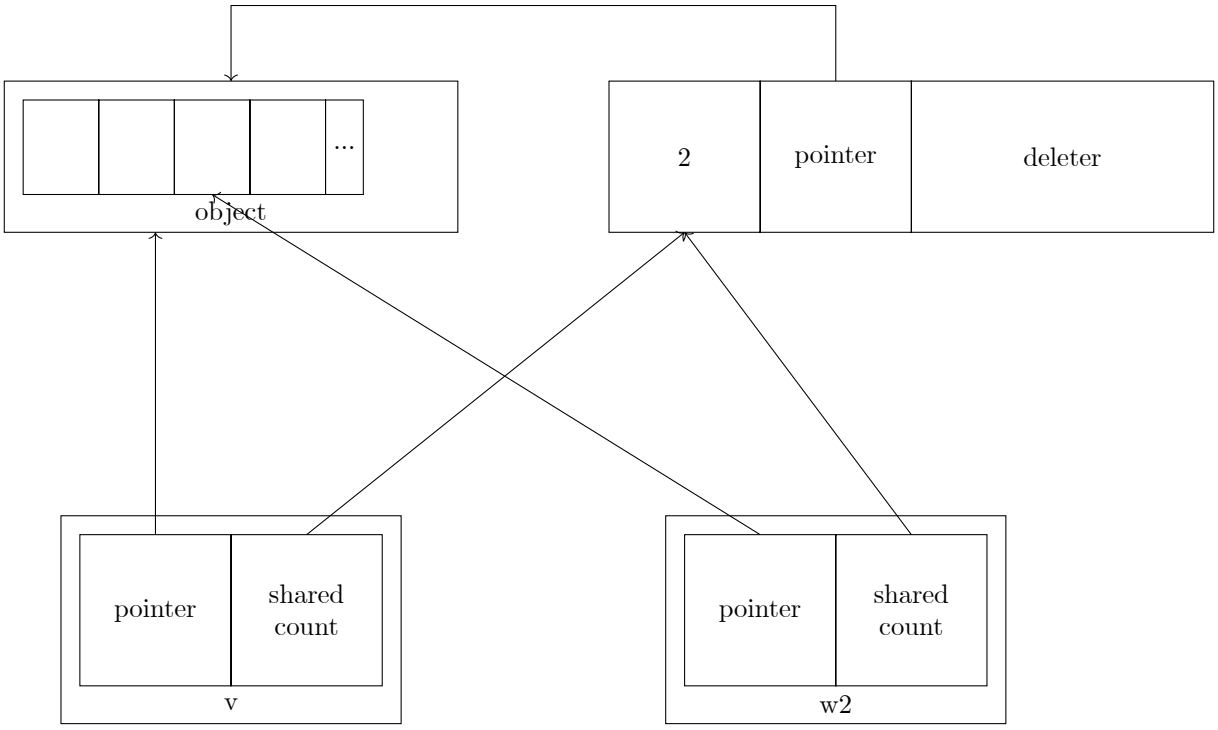
Это называется aliasing constructor, и он делится владением с оригинальным конструктором, но содержит внутри себя другой указатель. И тут уже ваша ответственность следит за тем, что колесо не умрёт раньше автомобиля. В данном случае это так, но в более сложном случае это может быть не так.
То же самое также типично используется, если вам нужно хранить `shared_ptr` с базовым классом. Через это чудо работают pointer cast'ы.

Реализация. Необходимо где-то считать ссылки. И делается это, разумеется, в каком-то месте, общем для всех `shared_ptr`'ов. То есть создаётся отдельный кусочек памяти:




```
std::shared_ptr<type> sp1(new type());
std::shared_ptr<type> sp2 = sp1;
```

В Windows этот кусочек справа сверху называется control block. Зачем там хранится ещё указатель? А для того, чтобы работал aliasing constructor. Как выглядит в таком случае `shared_ptr`? Ну, вот так:



Понятно, что имея только `w2`, очень сложно откуда-то взять, что это за колесо, и как из него получить оригинальной объект. Поэтому указатель на оригинальный объект надо бы похранить.

`make_shared`. Хочется не иметь две аллокации памяти, а иметь одну. И control block можно запихать внутрь объекта. Это делается при помощи `std::make_shared`, которому в качестве параметров передаются параметры конструктора вашего объекта:

```
struct my_type
{
    my_type(int, int, int);
};

std::shared_ptr<my_type> pointer = std::make_shared(1, 2, 3);
```

Притом тогда можно даже выпилить из этого блока deleter и ptr. Потому что зачем они.

`weak_ptr`. Это штука, которая не мешает объекту удалиться, но позволяет смотреть за ним. И проверять, удалился он или нет. Чтобы это реализовать, вам надо как-то продлевать время жизни control block'а, и обычно это делается при помощи добавления в control block ещё одного числа. Оно равно количеству слабых ссылок, если сильных ссылок нет, либо тому же, но плюс один, если есть. Потому что так получается меньше `if`'ов.

Пример. Представим себе, что у нас есть функция, которая грузит виджет. При этом разные куски могут просить одно и то же. Хочется им одно и то же и давать. Тогда мы обернём виджеты в `shared_ptr`.

```
std::shared_ptr<widget> get_widget(int id)
{
    static std::map<int, std::weak_ptr<widget>> cache;
    auto sp = cahce[id].lock();
    if (!sp)
```

```
            cache[id] = sp = load_widget(id);
        return sp;
    }
```

Но тут есть существенная проблема: если мы грузом виджеты по порядку (в процессе их удаляя), мы получаем в итоге очень большой `std::map`, в котором виджетов, конечно, нет, то там куча пустых `weak_ptr`'ов. Их надо как-то убрать. Как это править?
При помощи deleter'а, очевидно. Мы подсунем каждому объекту deleter, который знает, по какому итератору хранится данный объект. Это, кстати, причина, почему в примере `std::map`, а не `std::unordered_map`.
Но тут имеется проблема: у нас три объекта в памяти (данные, control block и узел дерева). Давайте уменьшим их количество. Во-первых, зачем на deleter. Он нужен, только если мы не знаем, как удалять объект. А мы делаем что-то вместе с удалением, давайте это «что-то» в деструктор запихнём. Теперь мы умеем делать `make_shared`, потому что нет необходимости в деструкторе. А дальше заменяем `map` на его интрузивный вариант, и получаем всего один объект.

`std::enable_shared_from_this`. Пусть мы делаем вот такой стандартный код:

```
std::shared_ptr<int> sp1(new int(42));
std::shared_ptr<int> sp2 = sp1;
```

Тут всё хорошо, мы в первой строке забираем владение, генерируем control block, а потом просто меняем данные в нём. А можно ещё ошибиться и случайно написать так:

```
int* p = new int(42);

std::shared_ptr<int> sp1(p);
std::shared_ptr<int> sp2(p);
```

Тут понятно, какая ошибка, double free. Но всё же. Бывают такие случаи, когда мы сделали `shared_ptr`, а потом достали из него указатель и куда-то передали. Например, чтобы вызвать на нём метод. А потом в методе захотели что-то сделать с `shared_ptr`'ом. Но мы не можем новый сделать. И старый получить не можем. Поэтому есть такая штука как `std::enable_shared_from_this` — отнаследовавшись от неё, вы сможете пользоваться методом `shared_from_this`, который даёт вам тот самый `shared_ptr`. Работает она тривиально, просто внутри себя храня `weak_ptr` на текущий объект.

4 Идеальная передача и шаблоны с переменным количеством аргументов.

Мотивация. Есть у нас `std::make_shared` и `std::make_unique`. Они принимают на вход параметры конструктора. Вопрос: это что за магия такая?
Что мы глобально хотим? У нас есть функция, она имеет несколько перегрузок:

```
void f(int&);
void f(char);
void f(std::string*);
```

А нам хочется написать функцию `g`, которая принимает на вход всё, что угодно, и передаёт это на вход функции `f`. Ну так идейно понятно, давайте сделаем наш параметр шаблонным:

```
template <class T>
void g(T x)
{
    f(x);
}
```

Но тут есть пара проблем:

1. Если `f` принимает, скажем, `std::vector<int>&&`, то у нас при передаче из `g` в `f` передастся `std::vector<int>&`, потому что в `f` мы будем передавать `x`, то есть переменную, то есть lvalue, и будет `f(std::vector<int>&)`.
2. Если у нас есть переменная типа `int`, то `g` внутри делает копию, и передача в `f(int&)` бесполезна. Более того, если мы передадим в `g` временный объект типа `int`, то мы вполне сможем его передать (`f` вызовется с ссылкой на перемещённое значение), а хочется, чтобы это даже не компилировалось.

В итоге хочется решение, которое, во-первых, сохраняет `const`'ы, во-вторых, запоминает value category.

Специальные правила для rvalue-ссылок в C++11.

Правило сворачивания ссылок. В C++ вы не можете создать ссылку на ссылку. Но тем не менее, никто же не мешает вам написать что-то такое:

```
using type = int&;
tyoe& x;
```

Или такое:

```
template <class T> void foo(T& x);

foo<int&>(x);
```

У Вас технически возникает ссылка на ссылку. В результате все компиляторы в C++03 считали, что ссылка на ссылку — это просто ссылка. А в C++11 Вам надо определить, что происходит, если Вы сворачиваете разные ссылки. Так вот, когда вы сворачиваете две ссылки, одна из которых lvalue, получается lvalue-ссылка. Если вы сворачиваете две rvalue-ссылки, получаете rvalue-ссылку. Почему это так определено? Чтобы работал perfect forwarding.

Правило вывода ссылок. Вот есть у вас

```
template <class T>
void foo(T x);
```

Как на основе вызова определить, какой тип у нас у `x`? У компилятора есть какие-то правила для разных вещей. Для вот такого есть какие-то правила, например:

```
template <class T>
void foo(std::pair<T, T*> x);
```

А в C++11 надо ввести правило, как выводился тип у такого

```
template <class T>
void foo(T&& x);
```

А вот выводится оно так:

```
template <class T>
void foo(T&& x);

int main()
{
    foo(42); // T -> int.
    foo(static_cast<int const&&>(42)); // T -> int const.

    int u;
    int const v;
    foo(u); // T -> int&.
    foo(v); // T -> int const&
}
```

Непосредственно, perfect forwarding. Теперь мы можем беспрепятственно написать идеальную передачу для примера выше:

```
template <class T>
void g(T&& x)
{
    f(static_cast<T&&>(x));
}
```

Почему это работает? Если в g приходит 42, то T выводится в int, и наша функция превращается в

```
void g(int&& x)
{
    f(static_cast<int&&>(x));
}
```

Это же ровно то, что нам нужно.
Если Вы передадите переменную типа int, T выведется в T&, а значит, с учётом сворачивания ссылок, g превратится в

```
void g(int& x)
{
    f(static_cast<int&>(x));
}
```

С const'ами тоже всё будет хорошо, можете проверить сами.
То, что мы только что сделали, называется perfect forwarding. Кстати, обычно он делается не при помощи static_cast'а, а специальной функцией: std::forward, которая, по сути, static_cast и делает. Давайте попробуем сами её написать:

```
template <typename T>
T&& forward(T& x)
{
    return static_cast<T&&>(x);
}
```

Но тут есть существенная проблема: если мы напишем forward(x) вместо forward<T>(x), у нас будет тупо std::move. Поэтому в стандартной библиотеке решили подавить автоматический вывод ссылок (чтобы всегда явно передавали). Самый простой сделать это вот так:

```
template <class T>
struct identity
{
    using type = T;
};

template <typename T>
T&& forward(typename identity<T>::type& x)
{
    return static_cast<T&&>(x);
}
```

Когда у вас в аргументах подобный dependent-тип, его в общем случае очень сложно вывести Но в случае со стандартным std::forward решили не заводить отдельную структуру, а использовать std::remove_reference_t, который, вообще говоря, полностью эквивалентен identity, потому что в identity у нас всё равно ссылки коллапсируют в lvalue-ссылку.

Variadic templates. Кто такой эти ваши variadic-шаблоны? Да всё элементарно, вы просто можете взять и добавить в шапку шаблона произвольное количество параметров:

```
template <class... T>
class tuple
{
    // ?????.
}

template <class... T>
void foo(T... args)
{
    tuple<T...>(args) t;
    // ...
}
```

Возникает вопрос, а почему `tuple<T...>`, а не просто `tuple<T>`? Да понятно, на самом деле, вот есть у нас `tuple<tuple<T>>`, и непонятно, что это должно быть, `tuple<tuple<T...>>` или `tuple<tuple<T>...>`. С `tuple` немного дурацкий пример, но если вот с какими-нибудь функциями это вполне адекватно:

```
template <class... T>
void foo(T... args)
{
    f(g(args...)); // f(g(arg0, arg1, arg2, ...))
    f(g(args)...); // f(g(arg0), g(arg1), g(arg2), ...)
}
```

Правда, вообще многоточие может раскрыть очень много всего. Мы можем раскрыть такое:

```
f(g(args, args...)...);
```

И это будет два разных раскрытия. Сначала у нас раскроется внутренний `args`, а потом внешний:

```
f(g(arg0, arg0, arg1, arg2, ...),
  g(arg1, arg0, arg1, arg2, ...),
  g(arg2, arg0, arg1, arg2, ...)
  ...);
```

Можем раскрыть такое:

```
f(g(args, args)...)
```

И это раскроется в

```
f(g(arg0, arg0),
  g(arg1, arg1),
  g(arg2, arg2),
  ...)
```

При этом pack'и даже разными могут быть (в последнем примере, если они отличаются по длине, ошибка компиляции).

Примеры. Давайте попытаемся сделать `std::make_unique`, пользуясь то, что мы только что изучили. Как это сделать, если у нас один аргумент? Ну, вот так:

```
template <class T, class Arg0>
std::unique_ptr<T> make_shared(Arg0&& arg0)
{
    return std::unique_ptr<T>(new T(std::forward<Arg0>(arg0)));
}
```

А как это обобщить на несколько параметров? Ну, всего лишь понять, где мы хотим раскрыть наши аргументы:

```
template <class T, class... Args>
std::unique_ptr<T> make_shared(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

И тут у нас по сути две вещи, которые надо распаковывать (`Args` и `args`), и многоточие распаковывает их одновременно парами, как было в примере с `f(g(args, args)...)`.

«Perfect backwarding».

`decltype`, `declval`. . Теперь представьте, что мы хотим не только forward'ить аргументы, но ещё и возвращать значение.

```
template <typename T>
??? g(T&& x)
{
    return f(std::forward<T>(x));
}
```

Для этого есть ключевое слово `decltype`, которое по значению даёт его тип по сути:

```
decltype(2 + 2) x = 7;
```

При этом `decltype` сохраняет все ссылки и все `const`'ы:

```
int foo();
int& bar();
int&& baz();

int main()
{
    decltype(foo()) x; // prvalue  int x;
    decltype(bar()) x; // lvalue   int& x;
    decltype(baz()) x; // xvalue   int&& x;
}
```

По-научному `decltype` даёт вам тот или иной вид ссылок в зависимости от value category.
Но на самом деле существует по сути два `decltype`: для выражений и для имён. То есть мы можем сделать что-то такое:

```
struct mytype
{
    int nested;
};

decltype(mytype::nested) a;
```

То есть `mytype::nested` — некорректное выражение, но брать от него `decltype` можно.
Давайте больше мемов:

```
int a;
decltype(a) b;   // int b, так как от переменной.
decltype((a)) c; // int& c, так как от выражения, а это lvalue.
```

О'кей, а теперь как написать нашу функцию?

```
template <typename T>
decltype(f(std::forward<T>(x))) g(T&& x)
{
    return f(std::forward<T>(x));
}
```

Но это не компилируется, потому что компилятор на этапе разбора возвращаемого значения не знает об x. Ну так и пофиг, давайте так сделаем:

```
template <typename T>
decltype(f(T())) g(T&& x)
{
    return f(std::forward<T>(x));
}
```

Это не норм, потому что конструктор по-умолчанию может отсутствовать, да и как сконструировать ссылку от ничего, не ясно. Поэтому, для порождения значений переменной из ниоткуда, в стандартной библиотеке есть функция std::declval, которая возвращает значение типа T&&:

```
template <typename T>
decltype(f(std::declval<T>())) g(T&& x)
{
    return f(std::forward<T>(x));
}
```

При этом сама функция обычно не имеет тела, чтобы никакой дурак не вздумал её вызвать, вы можете её использовать только там, где не происходит вычисление. Это называется unevaluated context (другими его примерами, помимо **decltype**, являются **sizeof** или **alignof**).

Trailing return type. . На самом деле вам может всё же захотеться сослаться на x.

```
struct foobar
{
    using type = int;

    type f();
    void g(type);
};
```

Что будет, если мы напишем реализацию этих функций вне класса? Ну, всё будет хорошо

```
void foobar::g(type x) {...}
```

Но при этом мы не можем также написать f:

```
type foobar::f() {...} // ошибка компиляции
foobar::type foobar::f() {...} // норм
```

Фундаментальная проблема здесь в одном: мы пишем возвращаемый тип до параметров, а не после. Поэтому есть другой синтаксис:

```
auto foobar::f() -> type {...}
```

И тут нам полностью всё равно, квалификаторы у нас, не квалификаторы, нужны параметры, не нужны, всё хорошо работает:

```
template <class T>
auto f(T&& x) -> decltype(f(std::forward<T>(x)))
{
    return f(std::forward<T>(x));
}
```

auto. А что будет, если мы просто напишем

```
template <class T>
auto f(T&& x)
{
    return f(std::forward<T>(x));
}
```

А будет, на самом деле, не то, что хочется. Но для этого обсудим **auto**.
Что происходит, если мы напишем **auto** x = expr? А тип будет ровно тот же самый, что выведется в примере:

```
template <class T>
void test(T x);
```

Это, вообще, почти всегда правда, но мы пока не проходили конструкций, которые приводят к случаю, когда это не так. Так вот, если у нас написано **auto&&** x или **auto const&** x, то просто test становится test(T&& x) или test(T const& x).
Начиная с C++14 вы можете писать **auto** как возвращаемое значение функции, и тогда оно определится на основе **return**. В частности, в нашем примере

```
template <class T>
auto f(T&& x)
{
    return f(std::forward<T>(x));
}
```

У нас всегда **return** — это rvalue, и мы будем получать не то, что хотим. Ровно как мы и не можем написать

```
template <class T>
auto& f(T&& x)
{
    return f(std::forward<T>(x));
}
```

или

```
template <class T>
auto&& f(T&& x)
{
    return f(std::forward<T>(x));
}
```

decltype(auto). Боже мой, что это за ужас? Ну, а это то, что нам нужно, оно берёт возвращаемое выражение, делает ему **decltype** и это тип возвращаемого значения. И это же можно написать в любом месте.

nullptr. До C++11 у вас не было **nullptr**, а вам приходилось руками делать **int*** p = 0; (выражение, на этапе компиляции имеющее значение 0, можно привести в указатель). Это создавало не так уж и много проблем, но потом появился perfect forwarding, и std::forward<T>(x) — это ни в каком месте не compile-time константа, равная нулю. Поэтому и ввели **nullptr** — константу, имеющую тип std::nullptr_t. Этот тип тупо неявно приводится к любому указателю.

5 Анонимные функции и их друзья.

Статический и динамический полиморфизм. Мотивация анонимных функций. Давайте сравним два способа вызвать сортировку: когда мы передаём `less<int>()` и `&int_less`, где

```
template <class T>
struct less
{
    bool operator()(const T& a, const T& b)
    {
        return a < b;
    }
};

bool int_less(int a, int b)
{
    return a < b;
}
```

Оказывается, вариант `less<int>` будет быстрее процентов на 30. Более того, если массив уже отсортирован, то `less<int>` будет быстрее более чем вдвое, (а в этом случае у нас сортировка сама по себе делает меньше работы и больше работы делает компаратор).
В чём дело? Давайте посмотрим, что у нас произойдёт, если мы скомпилируем что-то такое:

```
void foo(bool (*cmp)(int, int))
{
    cmp(1, 2);
}
```

У нас по сути нет варианта, кроме как взять и вызвать функцию, которую мы передали. С другой стороны, если у нас что такое:

```
template <typename F>
void bar(F f)
{
    f(1, 2);
}
```

Тут при подстановке шаблона мы чётко знаем, что такое `F`, какие у него методы, а значит мы можем спокойно их подставлять.
То есть идейно в `foo` у нас код генерируется однократно, и мы можем туда подставлять что угодно, а для `bar` создаётся код для каждого типа отдельно.
То, что мы сейчас обсуждаем, — это на самом деле виды полиморфизма. Когда мы на этапе компиляции знаем, что мы хотим делать — это статический полиморфизм, а когда у нас указатель на функцию — это динамический (ведь знаем его мы только на этапе исполнения).

Преимущества статического полиморфизма.	Недостатки статического полиморфизма
Возможность оптимизации на этапе компиляции.	Возможность получить большое количество копий одинаковых сущностей.
	Невозможность выбирать сущность на этапе исполнения.

Понятно, что у компиляторов есть способ бороться с недостатками динамического полиморфизма. Например, GCC под LTO может анализировать, какие аргументы приходят к Вам в сортировку, и если они всегда одни и те же, то inline'ить. Ещё GCC на *`-O3`* умеет в ещё один трюк: *`-fipa-cp-clone`*. Это вот что. GCC видит, что вашу функцию для какого-то конкретного параметра хорошо оптимизируется. Он может её взять и склонировать с заданным параметром. А ещё бывает вот какая оптимизация:

```
// ...
if (comp == &int_less)
    int_less(a, b);
else if (comp == &int_greater)
    int_greater(a, b);
else
    comp(a, b);
// ...
```

Так, а с чего это вдруг оптимизация? А потому что эти `if`'ы выносятся за пределы кода, и тогда у нас получается почти такой же эффект, как и *`-fipa-cp-clone`*. И в GCC тоже такое есть, оно называется *`-fdevirtualize-speculatively`*.
Кстати. Дело тут не только в inline'инге, потому что вообще GCC может не inline'ить, но всё равно вытаскивать из функции какую-то информацию о ней (типа, может она не пишет в память, значит нам не надо лишний раз читать память).

Безымянные функции (a.k.a. лямбда-функции). Представьте, что мы хотим со специфическим компаратором отсортировать массив. Традиционно до C++11 вам приходилось писать честные структуры, что не очень удобно, да и раскидано определение компаратора от использования. Поэтому теперь Вы можете написать вот такую чудесную конструкцию:

```
[](int a, int b) { return a % 10 < b % 10; };
```

Во что это преобразуется компилятором? Ну, в тривиальную вещь, в локальный класс, который вы создали прямо внутри функции и у него есть `operator()`. А вообще безымянная функция — это некоторый тип, который имеет конструктор копирования, присваивание, ещё немного штук, но не имеет имени.

Списки захвата. Теперь поговорим немного за синтаксис. Что происходит в круглых и фигурный скобках, я думаю, понятно, параметры и тело `operator()`. Но вот что такое квадратные? А представьте, что мы хотим сравнивать не по модулю `10`, а по модулю `n`. Тогда локальный класс можно было бы написать так:

```
class mod_n_less
{
private:
    int n;

public:
    bool operator()(int a, int b) const
    {
        return a % n < b % n;
    }

    mod_n_less(int n) : n(n) {}
};
```

И то же самое Вы можете сделать и с лямбдой:

```
[n](int a, int b) { return a % n < b % n; };
```

Но есть вопрос: а Вам надо ссылку на `n` или копию? Тут копия, а можно ссылку: `[&n]`. И ещё есть короткие записи: `[=]` и `[&]`. Это способ передать по значению или ссылке соотвественно все переменные, на которые лямбда ссылается. И ещё можно написать что-то такое: `[=, &a]` — передать всё по значению, а `a` — по ссылке. Квадратные скобки называются список захвата (capture).
Вопрос на 3000000 рублей: а можете ли вы захватить глобальную переменную? Нет, зачем вам, Вы же

можете и так к глобальной переменной обратиться без проблем. Теперь вопрос на 5000000 рублей: а поле класса можно? Так вот нет. **Мы умеем захватывать только локальные переменные и параметры функции**. Поэтому Вам нужно захватить `this`. Или `[*this]` (второе по значению). После этого Вы можете использовать все поля текущего класса. Кстати, важно знать, что `[=]` захватывает `[this]` (не `[*this]`). Поэтому, по причине ассиметрии, это на данный момент deprecated, и лучше Вам явно писать `[=, this]` или `[=, *this]`.
Всё это очень хорошо, но вдруг нам надо, например, переместить объект внутрь списка захвата или ещё что сложное сделать. Так вот, в C++14 на здоровье:

```
void foo(std::string str)
{
    [s = std::move(str)]() { /* ... */ };
}
```

Возвращаемое значение? Кстати, а что у лямбды с возвращаемым значением? А вот то, что мы не написали, это `auto`. А можно явно указать тип:

```
[n](int a, int b) -> bool { return a % 10 < b % 10; };
```

`mutable`. А вот ещё. Представьте вот такую лямбду: `[n]() { return n++; }`. То есть по сути лямбда, которая считает, сколько её вызвали. Так просто у Вас не получится, потому что сгенерируется код

```
class
{
private:
    int n;

public:
    bool operator()(int a, int b) const
    {
        return n++; // Error: const this.
    }
};
```

Тогда можно указать `[n]() mutable { return n++; }`.

Шаблонные лямбды. Теперь представьте себе разницу

```
template <typename T>
class less1
{
    bool operator()(const T&, const T&) { /* ... */ }
};

class less2
{
    template <typename T>
    bool operator()(const T&, const T&) { /* ... */ }
};
```

Если бы мы могли сделать шаблонную лямбду, что нам из этого надо? Ну, объективно, простое, потому что лямбду вы моментально создаёте, зачем вам шаблонный класс. И как шаблонную лямбду сделать? Есть общий способ (C++20):

```
[]<typename T>(const T& a, const T& b){ /* ... */ }
```

А ещё есть способ из C++17:

```
[](auto a, auto b){ /* ... */ }
```

Означает это, что каждое `auto` — отдельный шаблонный параметр-тип. Кстати, с C++20, синтаксис с `auto` в параметрах перешёл и на обычные функции (с такой же семантикой: каждое `auto` в аргументах функции — отдельный шаблонный параметр типа). Причём буквально:

```
void foo(auto x)
{
    // ...
}

foo<int>(23);
```

Прочее. Круто, а какими свойствами обладает лямбда? В смысле, какие у неё есть методы в классе? Во-первых, они умеют копироваться и перемещаться (до тех пор, пока захваченные сущности это умеют). Во-вторых, у них есть конструктор по-умолчанию, если они ничего не захватывают. И, в-третьих, они не умеют присваиваться (ни одним из двух способов). А ещё лямбды с пустым списком захвата можно конвертировать в указатель на функцию. Шаблонные можно только если их присваивать как-то так:

```
void (*p)(int) = []<typename T>(T x) { /* ... */ }; // deduced lambda<int>.
```

Как с обычными шаблонными функциями, короче.

Type erasure. `std::function`. Давайте попытаемся передать лямбду в зависимости от флага куда-то. При этом у нас нет проблемы с производительностью, потому что лямбда вызывается редко, а ещё лямбда создаётся где-то далеко от вызова. То есть хочется

```
std::vector<int> v;

/* unknown-type */ cmp = cond ? less<int>() : greater<int>();
std::ranges::sort(v, cmp);
```

Да, это будет динамический полиморфизм, но всё же, если нам надо, то надо. Так вот, в C++11 есть специальный тип `std::function`. Так вот нужный нам тип — `std::function<bool (int, int)>`. Тернарный оператор всё также нам не поможет, но присвоить мы можем без проблем:

```
std::vector<int> v;

std::function<bool (int, int)> cmp;
if (cond)
    cmp = less<int>()
else
    cmp = greater<int>();
std::ranges::sort(v, cmp);
```

Как эта чёрная магия устроена? Начнём с простого: что такое `bool (int, int)`? Это тип. Функция. Как указатель на функцию, только сама функция. И дальше понятно, как это в шаблон засунуть:

```
template <class T>
struct function;

template <class R, class... Args>
struct function<R (Args...)
{
    /* ... */
};
```

Теперь ближе к делу. Что наш `function` должен уметь? Он должен уметь создаваться от любого функционального объекта (будь это лямбда, `std::less` или ещё что). Ну, тут мы можем сделать только одну штуку: шаблонный конструктор. Но непонятно пока, как это сохранить. Ну, как бы, `void*`.

```
template <class T>
struct function;

template <class R, class... Args>
struct function<R (Args...)
{
private:
    void* func;

public:
    template <class F>
    function(F func)
        : func(new F(std::move(func)))
    {}
};
```

Но тогда у нас проблема с деструктором. Нам нужно как-то освободить память, при этом вызвав деструктор. А вот что можно:

```
template <class T>
void deleter_func(void* ptr)
{
    delete static_cast<T*>(ptr);
}

template <class T>
struct function;

template <class R, class... Args>
struct function<R (Args...)
{
private:
    void* func;
    void (*deleter)(void*);

public:
    template <class F>
    function(F func)
        : func(new F(std::move(func)))
        , deleter(deleter_func<F>)
    {}

    ~function()
    {
        deleter(p);
    }
};
```

И ровно то же самое можно сделать для `operator()`:

```
template <class T>
void deleter_func(void* ptr)
```

```
{
    delete static_cast<T*>(ptr);
}
template <class F, class R, class... Args>
R invoker_func(void* ptr, Args args...)
{
    return (static_cast<T*>(ptr))(args...);
}

template <class T>
struct function;

template <class R, class... Args>
struct function<R (Args...)
{
private:
    void* func;
    void (*deleter)(void*);
    R (*invoker)(void*, Args...);

public:
    template <class F>
    function(F func)
        : func(new F(std::move(func)))
        , deleter(deleter_func<F>)
        , invoker(invoker_func<F, R, Args...>)
    {}

    ~function()
    {
        deleter(p);
    }

    R operator()(Args... args)
    {
        return invoker(p, args...);
    }
};
```

Дальше бывает нужно учиться делать копию, перемещать, и всё прочее, но это уже понятно.
Кроме `function` люди пытались писать похожие типы (`std::any`, например). Или в разных библиотеках есть `any_iterator`. Правда, в жизни `any_iterator` очень сильно медленные. Тот паттерн, что мы только что написали — type erasure, знакомьтесь. А если Вы хотите почитать что-то ещё по теме, то вот.

6 Сигналы. Реентерабельность.

Давайте представим ситуацию, что у нас есть несколько компонент программы, и когда что-то происходит в одном месте, должно что-то происходить совсем в другом. Например, когда мы в тестовом редакторе тыкаем правую кнопку мыши, у нас появляется контекстное меню (которое к редактору отношения не имеет совершенно). Как бы мы могли это реализовать? Ну, мы могли бы хранить у себя какой-нибудь `std::function`, и при происхождении какого-то события его вызывать. Обычно такого достаточно, но нем всегда. Например, вашим событием интересуется несколько сущностей. С нажатием ПКМ это вряд ли, но вот если у вас текст в редакторе поменялся, вы можете хотеть сделать

несколько действий. Поэтому вам нужно будет сделать `std::vector<std::function</* ??? */>>`. В таком случае событие называется **сингалом**, обработчики называются **слотами**, добавление нового обработчика — **connect**'ом, а вызов всех обработчиков — **emit**'ом.
Давайте это писать. Кажется, что это просто вектор функций, но на самом деле есть детали. У нас будет сигнал `() -> void`, но вообще можно и параметры передавать, и даже иногда (в `boost`, например) возвращать значение. Но это не суть сейчас.

```
struct signal
{
    using slot_t = std::functional<void ()>;

    signal() = default;

    void connect(slot_t slot)
    {
        slots.push_back(std::move(slot));
    }

    void operator()() const
    {
        for (slot_t const& slot : slots)
            slot();
    }

private:
    std::vector<slot_t> slots;
};
```

Здесь не хватает `disconnect`. Причём непонятно, как его реализовывать, `std::function` не умеют сравниваться, чтобы их в `vector`'е искать, да и обязывать пользователя сохранять к себе каждый `std::function`, который он добавляет — такое себе. Поэтому можно возвращать из `connect` итератор вектора, а в `disconnect` его принимать. Разве что давать любому лоху сам итератор, во-первых, небезопасно (нельзя вызывать на нём `++` и `--`), а, во-вторых, то что у нас вектор — деталь реализации. Поэтому создадим обёртку.

```
struct signal
{
    using slot_t = std::functional<void ()>;

    signal() = default;

    struct connection
    {
        signal* sig;
        std::vector<slot_t>::iterator it;

        connection(signal* sig, std::vector<slot_t>::iterator it)
            : sig(sig), it(it) {}

        void disconnect()
        {
            sig->slots.erase(it);
            sig = nullptr;
        }
    };
```

```
    connection connect(slot_t slot)
    {
        slots.push_back(std::move(slot));
        return connection(this, std::prev(slots.end()));
    }

    void operator()() const
    {
        for (slot_t const& slot : slots)
            slot();
    }

    signal(signal&&) = delete;
    signal(signal const&) = delete;

    std::vector<slot_t> slots;
};
```

Можно было сделать `disconnect` и в `signal`, но в boost он внутри `connection`, но это не так важно. А вот что важно, так это то, что `vector` инвалидирует итераторы много когда. И у нас при `disconnect`'е или при добавлении. Это можно поправить через `std::list` или придумать соединением идентификаторы, которые хранить в хэш-таблице. Но это нам не от всего поможет. Есть вот такой случай, когда ничего не работает:

```
struct text_editor
{
    signal text_changed;
};

struct my_component
{
private:
    text_editor& editor;
    connection text_changed_connection;

    my_component()
    {
        text_changed_connection =
                editor.signal.connect([this] { text_changed_handler(); });
    }

    void text_changed_handler()
    {
        /* ... */
        text_changed_connection.disconnect();
    }
};
```

Тут у нас на середине emit'а мы будем удалять соединение. То есть в середине collection for'а мы удаляем текущий итератор. Можно пытаться это поправить, заменив collection for на обычный, и делать `*(it++)`. Это поможет от удаления самого себя, но тогда мы не можем отписать другой компонент. Необходимость делать так довольно редка, но всё же встречается (типа вы при сигнале удаляете какой-то компонент, а он подписан на то же событие, и отпишется от него).
Как править будем? Самый просто вариант — превентивно скопируем список и будем итерироваться

по нему. Тут у нас вот в чём может быть проблема. Представьте, что у вас есть компонент, который при рождении подписывается, в при смерти — отписывается от сигнала. Тогда если в вызове сигнала ваш объект кто-то удалит, он не удалится из копии списка, и вам придёт оповещение после смерти.
Поэтому правильный подход другой. Мы можем как-то помечать наши connection'ы как удалённые. Например, мы можем очищать `std::function` при удалении, а из списка удалять соединения при emit'е. Это работает, но если у нас emit происходит редко, а disconnect — часто, то мы проиграли. Поэтому давайте в сигнале хранить флаг, находится ли он сейчас в emit'е.

```
struct signal
{
    using slot_t = std::functional<void ()>;

    signal() = default;

    struct connection
    {
        signal* sig;
        std::vector<slot_t>::iterator it;

        connection(signal* sig, std::vector<slot_t>::iterator it)
                : sig(sig), it(it) {}

        void disconnect()
        {
            if (sig->inside_emit)
                *it = slot_t();
            else
                sig->slots.erase(it);
            sig = nullptr;
        }
    };

    connection connect(slot_t slot)
    {
        slots.push_back(std::move(slot));
        return connection(this, std::prev(slots.end()));
    }

    void leave_emit() const noexcept
    {
        inside_emit = false;

        for (auto it = slots.begin(); it != slot.end();)
            if (*it)
                ++it;
            else
                it = slots.erase(it);
    }

    void operator()() const
    {
        inside_emit = true;
        try
        {
            for (slot_t const& slot : slots)
```

```
                slot();
            }
            catch (...)
            {
                leave_emit();
                throw;
            }
            leave_emit();
        }

        signal(signal&&) = delete;
        signal(signal const&) = delete;

        mutable std::list<slot_t> slots;
        mutable bool inside_emit = false;
    };
```

У этого всё ещё есть проблемы, но в очень специфичном случае: если у нас сигнал подписан на сам себя (возможно, косвенно). Поэтому `inside_emit` делают счётчиком вложенности.
Есть ещё более интересная ситуация. Когда мы при исполнении слота убиваем сигнал целиком. Обычно это происходит, когда мы убиваем не сигнал, а класс, в котором он лежит. И тут можно либо обмазаться `shared_ptr`, либо сделать иначе: можно добавить в сигнал `bool* destroyed`, и в emit'е делать вот что:

```
    void operator()() const
    {
        bool is_destroyed = false;

        bool* old_destroyed = destroyed;
        destroyed = &is_destroyed;

        ++inside_emit;
        try
        {
            for (slot_t const& slot : slots)
            {
                if (slot)
                {
                    slot();
                    if (is_destroyed)
                    {
                        *old_destroyed = true;
                        return;
                    }
                }
            }
        }
        catch (...)
        {
            destroyed = old_destroyed;
            leave_emit();
            throw;
        }

        destroyed = old_destroyed;
        leave_emit();
```

```
}
```

Мораль от всего отсюда:

1. **Никогда не пишите сигналы руками**. Не надо думать: «А, я сделаю `std::vector<std::function>` и всё будет норм». Нет, не будет.
2. Когда вы используете сигналы из библиотеки, ознакомьтесь с её документацией. Она может давать не все гарантии. Библиотека может делать копию слотов при emit'е, например.
3. Если у вас простой случай, и вы хотите написать простой `signal` чисто под этот случай, то напишите, но поставьте `assert`'ы.

Вообще у нас тут происходит вот что: мы думаем о том, можем ли мы, находясь в глубине стека, делать что-то. Это имеет научное название: реентерабельность. И мы тут изобретали такой сигнал, чтобы все его методы были реентерабельными. Причём тут вопрос ещё более грустный. Мы справились с `signal`'ом. Но он же у нас где-то лежит, и не будет ли проблем с реентерабельностью методов того класса...
Кстати, обычно, реентерабельность проявляется много где. Вот считаете вы хэш-код. И вызываете функцию. Может ли она взять и в эту таблицу полезть? Да ну бред какой-то, зачем вам это. Но вот она: реентерабельность во всей красе. Только тут вы изначально делаете что-то странное. И в большинстве случаев про реентерабельность просто не говорят.

7 Обработка ошибок, ассёрты, контракты.

В этой теме есть фундаментальная проблема. В её обсуждении люди расходятся во мнениях. И неправда, что кто-то из них глуп или не обладает информацией или ещё чего. Тут разница во мнениях, вероятно, вызвана индивидуальными предпочтениями.
Но всё же наличие спорных моментов не значит, что про эту тему не надо говорить. Во многом мнения людей сходятся, да и для обсуждения разных мнений нам нужна терминология.

`assert`. Несмотря на то, что эта тема уже была, давайте вкратце её повторим.
`assert` — это макрос, который, в зависимости от макроса `NDEBUG` может значит либо ничего, либо проверка условия и `std::abort()`.

```
#ifdef NDEBUG
#  define assert(condition) ((void)0)
#else
#  define assert(condition) /*implementation defined*/
// В жизни бывает что-то такое:
// if (!(condition))
//     std::abort();
#endif
```

Когда этой штукой надо пользоваться? Ну, например, когда мы пишем структуру данных и хотим проверить, что у нас всегда выполняется инвариант. Или мы пишем свою функцию сортировки, и после этого хотим проверить, что массив у нас отсортирован: `assert(std::is_sorted(vec.begin(), vec.end()))`.
И зачем это идейно всё нужно? Чтобы мы не могли получить некорректные данные, а потом их передать. Получив некорректные данные, мы сразу узнаем их некорректность, тем самым сэкономив себе время на отладку.
Когда `assert` не надо использовать? Ну, например тут не надо:

```
FILE* f = fopen("filename.txt", "r");
assert(f);
```

Когда файл не открылся, это штатная ситуация. Более того, если мы пишем, что наша программа не может работать без файла, логично сообщить об этом пользователю и завершиться напрямую. Ведь `assert`'ы можно выключит и остаться без этой проверки. Или ещё `assert` не надо использовать так:

```
assert(fclose(f) == 0);
```

Тут, если отключить вам `assert`'ы, вы никогда не закроете файл.
Это всё очень хорошо, но зачем тогда `assert`'ы отключать? Ну, потому что когда вы написали достаточно `assert`'ов, ваша программа замедляется вдвое. Вы пишете двоичный поиск и ставите `assert` на то, что массив отсортирован. И асимптотика с $O(\log n)$ возрастает до $O(n)$.
Кстати. Пишите `assert`'ы, пожалуйста, попроще. Если у вас сложные `assert`'ы, вы могли набагать в самих условиях, которые вы пишете. И тут у вас совсем всё грустно получается.
Также помимо `assert`'ов у нас есть санитайзеры, которые работают сходим образом и дебажная версия стандартной библиотеки. Второе нужно, чтобы мы не делали `++` past-to-end итератору списка (`std::list` обычно циклический, сделав `++` итератору `end`, получим `begin`, и никакие address sanitizer'ы нас не спасут).
А чем отличаются все вот эти друзья:

```
std::abort();
std::terminate();
std::exit(EXIT_FAILURE);
std::quick_exit(EXIT_FAILURE);
std::_Exit(EXIT_FAILURE);
__debugbreak();   // MSVC only.
__builtin_debugtrap(); // Clang only.
raise(SIGTRAP);   // POSIX only.
__builtin_trap(); // GCC only.
```

А вот:

```
/* Функция из C, аварийно завершающая программу.
 * Для операционной системы это то же, что и деление на ноль или ещё что страшное.
 */
std::abort();
/* До C++03 вызывается из-за проблемы с исключениями.
 * Двойное исключение, или исключение из noexcept-функции, или ещё чего такое.
 * Но в C++11 появились и другие причины.
 * Поэтому сейчас это скорее плюсовый аналог abort.
 * Ещё terminate можно перекрыть обработчиком,
 *     но вообще это то же самое аварийное завершение.
 */
std::terminate();

/* Более штатное завершение программы. Для операционной системы всё хорошо.
 * Всем троим можно повесить обработчики, но у них есть проблемы.
 * Ни exit, ни quick_exit не вызовут деструкторы локальных переменных.
 * Exit вообще ничей деструктор не вызовет, а просто завершит программу.
 * В случае с exit и quick_exit всё особенно плохо с многопоточным кодом:
 *     вы начнёте удалять глобальные переменные, а у вас ещё потоки работают.
 * Лучше просто выходите из main'а по return.
 */
std::exit(EXIT_FAILURE);
std::quick_exit(EXIT_FAILURE);
std::_Exit(EXIT_FAILURE);

/* Функции, приводящая к специальной инструкции.
```

```
 * Инструкция называется int 3 --- используется отладчиками для break point'ов.
 * Если у вас нет отладчика, программа завершается.
 * Кстати, в MSVC abort является похож __debugbreak.
 */
__debugbreak();   // MSVC only.
__builtin_debugtrap(); // Clang only.
raise(SIGTRAP);   // POSIX only.

// Целенаправленно некорректная инструкция (ud2).
__builtin_trap(); // GCC only.
```

Мораль: если вы хотите сообщить о критической ошибке — используйте `std::abort`/`std::terminate`. Эти двое могут дать управление отладчику, чтобы всё у вас было хорошо.
Всё, что мы обсуждали, все наши ошибки, можно разделить на два класса: internal consistency error и uncommon case. Первое — это всё, когда у нас проблема в Вашем коде, второе — просто странные ситуации, а которых мы обычно не думаем (файл не открылся, например). Второе нужно обрабатывать честно в зависимости от контекста. Не открылся файл — сообщите об этом вызывающей функции, чтобы она с этим разобралась. Или если у вас есть контекст, сами уведомите об этом пользователя. А вот что делать с internal consistency error, уже вопрос. Есть 4 стратегии:

1. Игнорирование.
2. `std::abort`.
3. Уведомить об этом вызывавшего (исключение или код ошибки).
4. Запись в лог и продолжение работы.

1. Используется намного чаще, чем хотелось бы и чем задумано. Чтобы это написать, надо подумать, а постоянно думать — сложно и энергозатратно. Но не делайте так, пожалуйста. Лучше пишите логи.
2. Сразу при возникновении ошибки мы видим, что происходит. Не можем записать в базу данных некорректные данные, а некорректные данные в базе данных — то, с чем просто невозможно разобраться пост-фактум.
   Но есть и аргументы против. Если у нас сломался какой-то декор (или что-то другое необязательное), не обязательно абортиться.
3. Плюсы исключений мы прекрасно знаем, а давайте обсудим минусы. А минус тут вот какой: когда у вас есть исключения, вы получаете понятие exception-safe. Вам приходится думать о том, что будет, если будет исключение. А если не подумаете, у вас может всё сломаться совсем иначе, и будет как с игнорированием ошибок.
4. В отличие от исключения это нельзя заигнорировать (в исключениях какой-то дурак может написать `try {/*...*/} catch (...) {}`). И всё также при поломке декора всё у нас норм.
   А из минусов: ошибки такого рода чинятся с меньшим приоритетом. Если `abort`, то всё, программа не работает. А если у вас логи, то вы ещё должны заставить вашего коллегу это починить.

Что ещё можно сказать про `assert`'ы? А то, что они выполняют ещё и документирующую функцию. По классу сложно догадаться, какой у него инвариант, а по хорошо написанным `assert`'ам — очень легко.

Контракты. Мы знаем, что такое контракты. Но пока мы не знаем ещё пару терминов.
Есть *narrow contract*: когда мы накладываем ограничения, их нарушение — UB, а есть *wide contract*: всё совершенно чётко определено. Например, мы чётко говорим, что мы кидаем исключение в случае чего. Например, в C++ сложение чисел обладает узким контрактом для знаковых чисел и широким — для беззнаковых.

8 Разные языковые конструкции на примере std::optional.

std::optional. Пусть у нас есть какой-то класс, который позволяет отложить какое-то вычисление. Понятно, что его просто реализовать так: храним значение и флаг, посчитали ли его:

```
struct deferred_value
{
public:
    int get_value() const
    {
        if (!is_initialized)
        {
            value = compute_value();
            is_initialized = true;
        }

        return value;
    }

    int compute_value() const;

private:
    mutable int value;
    mutable bool is_initialized = false;
};
```

В чём проблема? В том, что у нас две переменные. Поля этого класса связаны: можно обращаться к одному, только если второе. Пока класс у нас небольшой, всё хорошо, а когда подобные штуки есть в классах побольше, связь улетучивается. А ещё хочется, чтобы корректное обращение компилятор проверял, а не человек. И третье: нам нужно, чтобы значение было DefaultConstructible, хотя нам это совершенно не нужно. Это легко исправляется при помощи std::aligned_storage_t, но для этого в стандартной библиотеке уже есть тип: std::optional.

```
template <class T>
struct optional
{
public:
    optional() : is_initialized(false) {}
    optional(T value) : is_initialized(true)
    {
        new (&storage) T(std::move(value));
    }
    optional(optional const& other) : is_initialized(other.is_initialized)
    {
        if (is_initialized)
            new (&storage) T(*other);
    }
```

```
    optional& operator=(optional const& other)
    {
        if (is_initialized)
        {
            if (other.is_initialized)
                **this = *other;
            else
            {
                reinterpret_cast<T&>(storage).~T();
                is_initialized = false;
            }
        }
        else
            if (other.is_initialized)
            {
                new (&storage) T(*other);
                is_initialized = true;
            }
    }
    ~optional()
    {
        if (is_initialized)
        reinterpret_cast<T&>(storage).~T();
    }
    operator bool() const
    {
        return is_initialized;
    }
    T& operator*()
    {
        return reinterpret_cast<T&>(storage).~T();
    }
    T const& operator*() const
    {
        return reinterpret_cast<T const&>(storage).~T();
    }

private:
    bool is_initialized;
    std::aligned_storage_t<sizeof(T), alignof(T)> storage;
};
```

И теперь, используя optinal, мы получим явно видимую связь bool и T, все адекватные реализации optional кидают исключение в отладочном режиме, если некорректно обращаться, и вообще всё хорошо.
Что ещё имеет смысл сказать про optional? В нашей реализации мы должны создать объект вне и move'нуть объект внутрь. А это может быть плохо, поэтому ещё optinal умеет в emplace:

```
    template <class... Args>
    void emplace(Args&&... args)
    {
        if (!is_initialized)
            reinterpret_cast<T&>(storage).~T();
        new (&storage) T(std::forward<Args>(args)...);
        is_initialized = true;
```

```
}
```

И ещё у optional'е есть такая же функция, но конструктор, которая просто принимает дополнительный параметр (пустую структуру) своим аргументом.
Ещё у optinal есть такая штука как nullopt_t — пустая структурка, которая при присваивании её в optional сбрасывает его состояние.
Ещё optinal умеет в сравнение (если в сравнение умеет шаблонный тип): при этом nullopt считается меньше все.
Как у vector'а есть at и operator[] с широким и узким контрактом соотвественно, так и у optinal'а есть value и operator*.
А что делать с optional'ом при перемещении? Надо ли разрушать тот, из которого переместили? Нет, не надо, кому надо — сам разрушит. В стандартной библиотеке даже явно написано, что optional, из которого переместили, хранит значение, (из которого переместили).

noexcept. Если всё это примитивно дописать, у полученного класса будет некоторое количество недостатков. Например, у нас есть конструктор перемещения и оператор перемещения. Обычно хотят, чтобы они были noexcept. Сто☒ит ли так написать? Ну, нет, ведь внутренний тип может бросать исключение при перемещении. А почему плохо не писать noexcept? Ну, ничего плохого не случится, но если у нас тип nothrow_move_constructible, то можно применить некоторые оптимизации (чтобы std::function мог применить small-object optimization, тип должен быть именно таким). Ещё пример — реаллокация vector'а. Если у тебя адекватное перемещение, ты можешь просто переместить элементы из старого буфера в новый и не париться. Если же они могут бросить исключение, то тебе придётся их копировать. Поэтому бывает полезно помечать конструктор и оператор перемещения как noexcept.
Кстати, что интересно, ещё одна операция, которую очень полезно делать noexcept — подсчёт хэша. По той же причине: когда у нас происходит перехэширование, нам надо посчитать много хэш-кодов, и если вы при этом можете исключение бросить, перехэширование не работает, поэтому рядом с каждым элементом начинают хранить хэш-код.
Вторая функция noexcept — документирующая, когда нам хочется полагаться на это корректности программы ради. А ещё давайте вспомним zero-cost исключения. На самом деле даже если исключений нет, то их возможность стоит нескольких процентов скорости (почему — вопрос: возможно, нужно меньше переупрядочивать код, возможно, это артефакты и т.д.) и довольно много стоит по размеру файла. Поэтому noexcept позволяет избавляться от этой проблемы.
Но вообще не надо помечать всё как noexcept: имея тело функции компилятор может и сам сообразить, что noexcept, а что нет, но это только с LTO, а без LTO становится грустно:

```
optional(optional&& other) noexcept(std::is_nothrow_move_constructible_v<T>)
    : is_initialized(other.is_initialized)
{
    if (is_initialized)
        new (&storage) T(*other);
}
optional& operator=(optional&& other)
    noexcept(std::is_nothrow_move_constructible_v<T>
          && std::is_nothrow_move_assigneble_v<T>)
{
    if (is_initialized)
    {
        if (other.is_initialized)
            **this = std::move(*other);
        else
        {
            reinterpret_cast<T&>(storage).~T();
            is_initialized = false;
        }
    }
    else
        if (other.is_initialized)
        {
            new (&storage) T(std::move(*other));
            is_initialized = true;
        }
}
```

А вообще в языке существует возможность проверить любое выражение на `noexcept`. Напишите его так: `noexcept(expr)`, и теперь можно написать так: `noexcept(noexcept(expr))`, и ваша функция будет `noexcept` тогда и только тогда, когда `expr`.

Тривиальные операции. Пусть есть два класса:

```
struct foo
{
    int a, b;
}

struct bar
{
    int a, b;

    ~bar() {}
}
```

В чём разница? В том, что на `foo` можно не вызывать деструктор при уничтожении, а в `bar` — надо. С точки зрения компилятора эти два типа отличаются тем, что `foo` не имеет никакого деструктора. А `bar` имеет деструктор, и это обычная функция (но так уж сложилась, что она ничего не делает). Говорят, что класс обладает тривиальным деструктором, если

1. Он не написан (т.е. сгенерирован автоматически) или явно прописан как `default`.
2. Он не виртуальный.
3. Деструкторы всех баз тривиальны.

4. Деструкторы всех нестатических полей тривиальны.

Аналогично определяется тривиальность копирования (обоих копирований), перемещения (обоих перемещений), и создания по умолчанию. Когда нам эти штуки важны? Ну, во-первых, это необходимо для некоторых классов (например, в `std::atomic` вы можете запихать любой созданный вами тип, если он тривиально копируется). Тривиальное копирование — это когда копирование равносильно `memcpy`. И отсюда становится понятно, что, например, при реаллокации вектора можно просто этот `memcpy` и сделать, а не в цикле что-то вызывать.
И на что тривиальность влияет — так это на ABI. Тривиально-разрушаемый тип можно в регистрах вернуть, а нетривиально-разрушаемый придётся передавать параметром в функцию и конструировать на памяти.
Кстати, а что мешает компилятору увидеть `~bar() {}` и удалить его за ненадобностью? А вот мы хотим, чтобы где бы у нас не был написан деструктор, ABI был один, а если он написан в другом месте, мы так сделать не сможем. А проблема тут в том, что в следующем обновлении мы можем сделать деструктор не пустым, и любой код, использующий `bar`, проиграет. И в чём ещё проблема — хочется быть независимым от версии компилятора. Если у нас ABI меняется от оптимизирующих способностей компилятора, то грустно жить. На тему ABI можно ещё посмотреть презентацию «There are no zero-cost abstractions» и увидеть, что у нас даже `unique_ptr` из-за нетривиального разрушения не может быть возвращён из функции через регистры как указатель.
Итак, что нам хочется? Нам хочется, чтобы когда `T` тривиально разрушается, то `optional<T>` — тоже тривиально разрушается. Как это сделать? `enable_if`? Ну, для деструктора это не сработает совсем, а если сделать шаблонные конструктор копирования, то это будет уже не конструктор копирования, и компилятор сгенерирует свои. Можно взять и сделать специализацию `optinal`'а, но у нас 6 операций, которые могут быть тривиальными, и у 5 из них хочется сохранить свойства. Поэтому нам придётся написать 2^5 вариантов `optinal`'а, что совершенно огромное дублирование кода. Что можно с этим сделать? Можно создать несколько шаблонных баз и просто отнаследовать от одной из них в зависимости от тривиальности копирований/разрушения/перемещений...

SFINAE-friendly функции. Что ещё мы можем протаскивать как свойство — копируется ли тип в принципе. В нашей реализации копирование и так не скомпилируется, но нам же хочется, чтобы у нас ещё и `std::is_copy_constructible_v` был правильно задан (например, `std::variant` работает по-разному в зависимости от этого). Так вот если снаружи можно проверить, всё ли будет хорошо при вызове функции с заданным параметром (в нашем случае при вызове метода с заданным параметром шаблона), то такая функция называется SFINAE-friendly. Поэтому, кстати, в `std::atomic` (где условия написаны `static_assert`'ами) всё с этим плохо.
А насколько важно вообще иметь SFINAE-friendly операции? Ну, важно, но очень геморройно (до появления концептов в C++20).
Как можно было бы реализовать это для `optional`'а? Да тривиально, у нас есть два варианта базы, отвечающей за копирование, давайте будет три.

```
template <typename T,
bool CopyConstructible = std::is_copy_constructible_v<T>,
bool TriviallyCopyConstructible = std::is_trivially_copy_constructible_v<T>>
struct optional_copy_constructor_base : optional_storage_base<T>
{
    optional_copy_constructor_base() = default;
    optional_copy_constructor_base(optional_copy_constructor_base const& other) = delete;
    optional_copy_constructor_base(optional_copy_constructor_base&&) = default;
};
```

```
template <typename T>
struct optional_copy_constructor_base<T, true, false> : optional_storage_base<T>
{
    using optional_storage_base<T>::is_initialized;
    using optional_storage_base<T>::storage;

    optional_copy_constructor_base() = default;
    optional_copy_constructor_base(optional_copy_constructor_base const& other)
    {
        is_initialized = other.is_initialized;
        if (is_initialized)
            new (&storage) T(*other);
    }
    optional_copy_constructor_base(optional_copy_constructor_base&&) = default;
};

template <typename T>
struct optional_copy_constructor_base<T, true, true> : optional_storage_base<T>
{};
```

Conditionally-explicit конструктор. И всё бы у нас было хорошо, если бы не одно но. У std::optional<T> есть конструктор от std::optional<U> const&. Причём хочется, чтобы он мог быть явным или неявным в зависимости от того, конвертируется ли U в T явно или нет.
Ну, это тоже тривиально пишется, благо у explicit тоже можно логическое выражение написать:

```
template <typename U,
          std::enable_if_t<std::is_constructible_v<T, const U&>, bool> = true>
explicit(!std::is_convertible_v<const U&, T>)
optional(optional<U> const& other)
{
    if (other)
        emplace(*other);
}
```

constexpr.

Мотивация. А давайте сейчас на variant посмотрим. Чем он глобально отличается от optional'а? Ну, у него вместо флага индекс. И std::aligned_storage_t такой, что может хранить любой из типов. Вот, например, std::variant от двух типов:

```
template <class A, class B>
struct either
{
    size_t which;
    std::aligned_storage_t<std::max(sizeof(A), sizeof(B)),
                           std::lcm(sizeof(A), sizeof(B))> storage;
};
```

Но вот в C++03 была бы существенная проблема: std::max не умеет в вычисления на этапе компиляции. Поэтому пришлось извращаться бы:

```
template <size_t a, size_t b>
struct max
{
    static size_t const value = a < b ? b : a;
};
```

И это чудо не только читается сложно и пишется долго, это и для компилятора не так просто: класс — не самая простая штука.
А во время компиляции считать хочется: иногда (как тут) это жизненно необходимо, а иногда можно и во время исполнения, но очень хочется при компиляции. Вспомните `function`: каждый новый объект, который мы подставляем в `function` заводит новую статическую таблицу. А давайте посмотрим, как это работает. Что происходит, если мы напишем что-то такое:

```
int foo() { /* ... */ }
int a = foo();
```

Здесь до запуска `main` запустится `foo` и проинициализирует переменную. И у нас то же самое происходило бы каждый раз в `function`'ах, чем больше бы мы их использовали.
Или вот ещё пример: у нас есть парсер, лексер или ещё какой-то разборщик языка. И мы хотим каждое ключевое слово запихнуть в хэш-таблицу. И очень-очень хочется, чтобы хэш-функция была построена с маленьким количеством бакетов, если все ключевые слова мы знаем заранее. В C для этого использовали кодогенераторы (берём слова, наша программа генерирует нам код той хэш-функции, что нам нужна). Но зачем, можно же сделать так, чтобы на нашем родном языке всё было бы хорошо. И ещё тысячу разных примеров можно придумать, когда хочется вычисления на этапе компиляции: регулярные выражения, например.

`constexpr`-функции. Итак, что сделали? Сделали возможность пометить функцию как такую, что можно посчитать во время компиляции.

```
template <class T>
constexpr T max(T const& a, T const& b)
{
    return a < b ? b : a;
}
```

В C++11 всё было консервативно: Вы не можете написать что угодно в `constexpr`-функции (там должен был быть только один `return`, да и вернуть всё что угодно нельзя...) Но чем дальше в лес, тем больше стало можно делать во время компиляции. Типа, почему тернарный оператор написать можно, а `if` нельзя? Давайте разрешим. Почему рекурсию можно, а `while` нельзя? Давайте тоже разрешим. Это было сделано в C++14. И таким образом у нас есть интерпретатор C++ внутри компилятора C++. В LLVM ещё и компилятор туда вставили, чтобы интерпретировать не код сразу, а ещё переведём его в более удобное представление. Осталось только сделать JIT-компилятор на этапе компиляции. В C++17 лямбды стали `constexpr`, в C++20 стало возможно вызывать виртуальные функции и делать аллокации (правда, не любые, вы обязаны в `constexpr` всё освободить), в C++23 появился `constexpr unique_ptr`, математические функции...
Кстати. А всегда ли обязана функция вычислиться во время компиляции? Ну, объективно, нет

```
int isqrt(int n)
{
    if (n < 0)
        throw new runtime_error("Integer square root of negative value.");
    /*
     * Вычисляем целочисленный квадратный корень.
     */
}

int x = isqrt(-1);
```

Ну, разумеется, нет. И тут даже нет ошибки компиляции, тут просто во время исполнения вы получите исключения. Исключения нельзя выкидывать из `constexpr`. Ровно как и нельзя, чтобы `constexpr`-функция делала UB: иначе вашу функцию просто не посчитают во время компиляции (кстати, этим даже пользуются: UB в `constexpr` ловит вам UB и утечки памяти, напишем всё `constexpr` и хороший

компилятор сам найдёт обращение вне массива, неверное обращение к union'у и кучу всего другого...)
В конце концов вы можете вызвать не-constexpr функцию в constexpr, и тогда, если исполнение дойдёт до этого, во время компиляции вашу функцию считать не будут. Короче, constexpr — то, что можно попробовать вычислить во время компиляции.
Так а какой тогда у constexpr'а смысл, если это разрешает компилятору считать функции во время компиляции? Не сделать ли все функции constexpr по умолчанию? Вон, в GCC есть флаг *-fimplicit-constexpr*, который так делает. Ну, да, но нет. Если Вы знаете, что вашу функцию бессмысленно (но можно) считать во время компиляции, то компиляторы бы тратили кучу времени на то, что пытались.
Кстати, а что нельзя писать в constexpr? goto. Или try. Но это, вероятно, пока никто proposal соответствующий не написал. А вот что делать явно и фундаментально нельзя — это ASM-вставки и вызов функций без тела. Это понятно, почему так. А вот что непонятно, это reinterpret_cast. С ним просто непонятно, его фундаментально нельзя или ещё какая причина, но вот можете почитать тред на эту тему. А вообще есть std::bit_cast, который в хороших компиляторах даже проверяется, что объекты не содержат паддинг.
Кстати, про аллокации. Как уже было сказано, мы можем их использовать только если мы всё освободим в конце (такие аллокации называются *transient*). Но это же не удобно, если мы хотим заполнить vector заранее и вернуть его. И поэтому есть спрос на non-transient аллокации внутри constexpr, но пока совершенно непонятно, как это сделать.
А ещё вот за что поговорим: может ли компилятор проигнорировать new и delete, если видит их рядом и не видит нужны действительно выделять и освобождать память? Ну, может, а почему нет. Но вот что не может, так это вот что:

```
delete new int(42);                // Можно обойти.
operator delete(operator new(8)); // Нельзя обойти.
```

Почему? Ну, потому что operator new и operator delete — обычные функции, **и их можно подменить**. Поэтому лучше использовать вызовы std::allocator, их обходить по стандарту можно. К чему это, собственно, всё? А к тому, что в constexpr разрешено только то, что можно обходить.

constexpr-переменные. Если у нас переменная — compile-time константа, то её значение просто вбивается в бинарник, а если нет, то компилятор создаёт специальную функцию, которая инициализирует переменную. Но иногда мы хотим заставить компилятор посчитать переменную во время компиляции. Для этого и можно пометить переменную как constexpr.
Что ещё важно понимать про constexpr-переменные, так это то, что по умолчанию они видны только в единице трансляции. Почему? А потому что constexpr — это конъюнкция двух вещей: const и наличия статического инициализатора, а const расширяет static. Почему const включает в себя static? Потому что const придумали отчасти для того, чтобы не использовать *#define*'ы, а *#define*'ы часто встречаются в заголовочных файлах. В связи с этим constexpr-переменные статические, а значит, когда вы определяете переменную в заголовочном файле, вам надо явно указать ей inline, если вы не хотите много копий данной переменной.

constinit. А можем ли мы хотеть изменяемую переменную, но со статическим инициализатором? Ну, можем, для этого есть constinit. Нужно это бывает для многопоточности. Есть такой модификатор — thread_local — каждый поток имеет свою копию переменной. Вообще все компиляторы имели свои аналоги thread_local, но они требовали ещё и статический инициализатор (так взаимодействие с ОС проще получалось). Поэтому теперь добавили constinit, который часто встречается вместе с thread_local. Зачем нам для этого ключевое слово, почему просто не if'ать, какой там инициализатор? Потому что Вы можете только объявить переменную где-то, а не определить. И тогда компилятор обязан предполагать худшее. А с constinit'ом в объявлении Вам гарантируют, что всё будет хорошо.

consteval. В C++ есть proposal, связанный с compile-time reflection'ом. Т.е. позволить изнутри программы во время компиляции получать, например, список полей. Понятно, что мы не хотим отправлять это в runtime (иначе нам всю информацию о классах засовывать в бинарник), а во время компиляции — почему нет. Самый простой пример — автоматическая генерация сереализаторов.

И для этого нам нужны функции, которые **обязаны** посчитаться во время компиляции. Для этого есть модификатор `consteval`. Поскольку в данный момент compile-time reflection'а у нас нет, он применяется очень редко, но всё же пара примеров есть. Например, `std::source_location::current()` — функция, являющаяся более хорошим способом во время компиляции указывать кому-нибудь файл и номер строки (нежели макросы `__FILE__` и `__LINE__`).

`is_constent_evaluated. if consteval.` На `constexpr`-функции наложены некоторые ограничения: нельзя, например, делать `memcpy`. И в таком случае есть беда: может быть такая функция, что runtime'овый код, который там написан, нельзя посчитать в compile-time'е, а использовав то, что в compile-time посчитать можно, в runtime'е получим что-то неэффективное. Хотим, например использовать секретную инструкцию Вашего процессора. Тогда нам надо написать разный compile-time и runtime код.
Поэтому в язык добавили функцию `is_constent_evaluated`. Эта функция вернёт вам `true`, если Вы считаете её во время компиляции, иначе `false`. Но у неё есть две проблемы.

1. Даже если вы уже узнали, что сейчас время компиляции, вы не можете соответствующим образом жить (засунуть, например, нечто странное в параметр шаблона).

2. Нельзя написать `if constexpr (is_constent_evaluated())`, потому что в таком случае `is_constent_evaluated` всегда вычислится в `true`.

Отсюда в C++23 появилась такая штука как `if consteval` и `if !consteval`.

`variant.` В чём мотивация этого чуда? Когда у нас был `optional`, мотивация была в том, чтобы не раскидывать по коду `bool` и значение. И конкретно мы привели пример класса, который лениво считает значение

```
struct deferred_value
{
public:
    int get_value() const
    {
        if (!value)
            value = compute_value();
        return value;
    }

    int compute_value() const;

private:
    mutable std::optional<T> value;
};
```

Но обычно мы хотим передавать функцию в конструктор, например:

```
struct deferred_value
{
public:
    deferred_value(std::function<T ()> compute) : compute(std::move(compute)) {}

    int get_value() const
    {
        if (!value)
        {
            value = compute();
            compute = {};
        }
        return *value;
    }

private:
    mutable std::optional<T> value;
    mutable std::function<T ()> compute;
};
```

И функцию стоило бы уничтожить после использования, потому что потенциально в std::function может быть лишнее место.
Но блин, тут же у нас всегда хранится либо одно, либо другое. Это же union и пометка о том, какая альтернатива в данный момент хранится. Иначе говоря std::variant:

```
struct deferred_value
{
public:
    deferred_value(std::function<T ()> compute) : state(std::move(compute)) {}

    int get_value() const
    {
        if (state.index() == 1)
            state = std::get<std::function<T ()>>(value)();
        return std::get<T>(value);
    }

private:
    std::variant<T, std::function<T ()>> state;
};
```

Тут у нас разнесены по времени проверка на индекс и взятие элемента, поэтому у variant'а есть ещё get_if:

```
struct deferred_value
{
public:
    deferred_value(std::function<T ()> compute) : state(std::move(compute)) {}

    int get_value() const
    {
        if (std::function<T ()>* compute_ptr = std::get_if<0>(&state))
           state = (*compute_ptr)();
        return std::get<1>(value);
    }
```

```
private:
    std::variant<T, std::function<T ()>> state;
};
```

Теперь давайте вот что обсудим. В optional'е есть ровно один адекватный вариант создания. В boost'е он был до включения в стандарт и не отличался ничем. А вот с variant'ом есть, *барабанная дробь*, варианты! Вон, в boost'е есть variant и variant2. Они отвечают по-разному на разные вопросы. Например, когда мы присваиваем в variant тот тип, что там уже есть, мы должны разрушить тип и пересоздать или использовать operator=? В optinal'е у нас первое, например. А вот в variant'е были сильные споры на эту тему. В той форме, что он в стандарте — он использует присваивание.
Второй вопрос: вот мы присваиваем в variant не то, что там есть. Если мы разрушим то, что там было, и создадим новое, это непонятная гарантия исключений. Что можно с этим сделать?

1. Добавить в variant некорректное состояние (тогда гарантия исключений будет слабая, но лучше, чем ничего). Это довольно неплохо, но людей очень смущает некорректное состояние. Это как null в Java. С ним можно жить, но хотелось бы в типах информацию кодировать, а не в контракте писать, что variant должен быть не пуст. Ещё базовая гарантия исключений — не очень круто, в следующих вариантах бу

2. Если у того типа, что присваиваем, noexcept-move, то можно это абьюзить, но если нет, то непонятно, как жить. Эта стратегия полностью работает только если мы **требуем** noexcept-move, но это явно не хорошо. Можно доопределить, что тот, кто бросил — сам виноват, и это UB либо std::terminate. Требовать noexcept-move-constructible — совсем плохо (дебажные коллекции вон выделяют память под контрольный блок при любом конструкторе), а terminate или UB не стоит делать там, где мы потенциально можем проверить, что происходит.

3. Стратегия из boost::variant'а (т.е. первой версии). Пусть у нас есть variant<A, B>. Давайте введём не некорректное состояние, а добавим возможность хранить не только A и B, но и A* или B*. И тогда мы перемещаем A у новый указатель, попытаемся присвоить B, если у нас не получится, вернёмся в состояние A, но на самом деле это будет A*. Эта стратегия имеет название: «heap backup». Что тут плохого — разве что потенциально ненужная аллокация (исключения должны быть редкими), происходящая всегда.

4. «Double buffering». То же самое, что и раньше, но копию мы делаем не на куче, а в поля кладём точно такой же storage, и копируем в него. Тут понятно, чего плохо — вдвое больше памяти используем. Если такой variant один, то можно пережить, но если их много, то не надо, пожалуйста.

И вот по этому поводу и происходили зарубы, что же из этого выбрать. Конкретно в std::variant вделано некорректное состояние. Но там его ещё чуть допилили. Некорректное состояние можно сделать по-разному: можно сказать, что некорректное состояние — это полноценное состояние, а можно сказать, что из него работают только ограниченный набор методов. И в стандартной библиотеке пошли вторым путём. Более того, его ещё сильнее спрятали: в конструкторе по умолчанию у нас создаётся первая альтернатива (вместо того, чтобы создавать его в некорректном состоянии). Вообще по этой теме можно посмотреть видос с CppNow
Но тут резонный вопрос, а что делать, если мы таки хотим пустое состояние явно и честно? Ну, создаёте пустую структуру и засуньте её в variant как альтернативу. Точнее,её создали за вас и назвали std::monostate.
Где ещё была заруба — это в сравнении. Есть два варианте: можно сравнить сначала по номеру альтернативы, а потом по значению, но тут возникают такие странности:

```
constexpr variant<int, long> a = 10;
constexpr variant<int, long> b = 5L;
static_assert(a < b);
```

Тогда можно попытаться определить < для каждых пар типов, что у нас есть, но это n^2 кода и должно быть определено сравнение для каждой пары. И как бы ладно с <. А вот что с ==?

```
constexpr variant<int, long> a = 10;
constexpr variant<int, long> b = 10L;
static_assert(a != b);
```

И тут вопрос возникает ещё интереснее, можно ли сделать `veriant<int, int>`?
Так вот, в STL разрешён `variant<int, int>` (мы можем написать шаблонный `variant<mytype, T>`, и так случайно получится, что `T` совпадёт с нашим типом), а `<` и `==` сравнивает сначала по номеру альтернативы, потом по значению. Некорректное состояние считается раньше всех. Оно, кстати, называется `valueless_by_exception`.

Best practice. Помните наш

```
struct deferred_value
{
public:
    deferred_value(std::function<T ()> compute) : state(std::move(compute)) {}

    int get_value() const
    {
        if (std::function<T ()>* compute_ptr = std::get_if<0>(&state))
            state = (*compute_ptr)();
        return std::get<1>(value);
    }

private:
    std::variant<T, std::function<T ()>> state;
};
```

Что у нас в случае `valueless_by_exception`? Ну, всё плохо. Но вообще у нас бывает и другая ситуация: `compute` бросает исключение. И тут тоже вообще логично бы делать некорректное состояние. Поэтому тут можно материализовать некорректное состояние:

```
struct deferred_value
{
public:
    deferred_value(std::function<T ()> compute) : state(std::move(compute)) {}

    int get_value() const
    {
        if (std::function<T ()>* compute_ptr = std::get_if<0>(&state))
            try
            {
                state = (*compute_ptr)();
            }
            catch (...)
            {
                state = computation_failed();
                throw;
            }

        return std::get<1>(value);
    }

private:
    struct computation_failed {};
```

```
    std::variant<T, std::function<T ()>, computation_failed> state;
};
```

Визитеры. Очень хочется pattern-matching в variant. Но фиг нам его кто даст, в языке таких фич нет. А писать постоянно проверки на тип и get не хочется. Поэтому есть такой паттерн как визитеры.

```
struct visitor
{
    void operator()(A const&) const { /* ... */ }
    void operator()(B const&) const { /* ... */ }
    void operator()(C const&) const { /* ... */ }
};

int main()
{
    std::variant<A, B, C> v;
    std::visit(visitor(), v);
}
```

Но писать свои классы – очень не хочется. Формально у нас есть лямбды, но их нельзя перегрузить. Может, существует способ как-то их таки использовать? В один тип, например соединить.

```
template <class... Fs>
struct overload;

template <class F0, class... Frest>
struct overload<F0, Frest...> : F0, overload<Frest...>
{
    overload(F0 f0, Frest ...frest) : F0(fstd::move(f0)), overload<Frest...>(std::move(frest)..

    using F0::operator();
    using overload<Frest...>::operator();
};

template <class F0>
struct overload<F0> : F0
{
    overload(F0 f0) : F0(fstd::move(f0)) {}

    using F0::operator();
};

template <class... Fs>
overload<Fs...> make_overload(Fs&& ...fs)
{
    return overload<std::decay_t<Fs>...>(std::forward<Fs>(fs)...);
}
```

Тут есть очень крупные проблемы с передачей (последний тип переместится N раз), но все мы знаем, как это поправить. Ещё есть проблемы с тем, чтобы передавать указатель на функцию (от него не отнаследоваться), но это тоже поправите сами.

9 Концепты.

Мотивация. Пусть у нас есть `vector`. Когда мы пишем `template <typename T>`, мы можем в качестве `T` подставить всё, что угодно (ссылки, типы-функции, хоть `void`). Но нам же не любой тип подходит: нам нужно nothrow destructible или ещё какие хорошие свойства у типа (вообще в случае с вектором ограничения не на тип `T`, а на аллокатор — второй шаблонный параметр `vector`'а, но не суть). И выразить это ограничение в языке мы никак не могли до C++20.
Ну, хорошо, а зачем? Зачем явно уметь указывать ограничения?
Самый ходовой пример выглядит так: пока у нас программа компилируется, всё хорошо. Но давайте попытаемся подставить в вектор то, что подставить нельзя. Например, нет деструктора. Понятно, что мы увидим ошибку. Но какую? Компилятор скажет, что предпринята попытка ссылки на удалённую функцию (деструктор) и покажет цепочку функций, откуда и кто хочет вызвать этот деструктор. У нас ошибка компиляции где-то глубоко в коде зарыта, да ещё и сообщение не очень содержательное сообщение выводится. В случае `vector<int&>` мы получим сообщение «невозможно создать указатель на ссылку». Если мы подставим `vector<void()>`, то у нас совсем будет что-то несодержательное (arithmetic on pointers to type `void ()`, и совсем непонятно, причём тут это вообще). Причём в разных компиляторах будут разные источники ошибок.
Хочется более адекватных сообщений об ошибках и не где-то, а при подстановке шаблона. Тут нам может помочь `static_assert`, но поможет не везде. Поэтому давайте посмотрим на второй пример:
У `vector`'а есть метод `assign`:

```
void assign(size_t n, T const& value);

template <typename InputIt>
void assign(InputIt first, InputIt last);
```

Тогда мы проиграем, если сделаем

```
vector<int> v;
v.assign(10, 42);
```

Почему? Потому что первая перегрузка требует преобразования типов, а вторая — нет, там `assign(int, int)`. И мы проиграем во время компиляции. Тут нам не поможет `static_assert`, но поможет SFINAE. То есть тоже в целом решаемо, но тут мы получаем не очень приятный код, да ещё и сложно объяснить новичку, что в этих `enable_if`'ах происходит. К тому же SFINAE сложно понять не только человеку, но и компилятору. Мы в целом абьюзим языковые механизмы, которые изначально не для того сделаны. И в-третьих, у нас в SFINAE очень страшно выглядят ошибки компиляции. Ошибка подстановки *какая-то страшная хрень*: не найдён `type` у `enable_if<что-то>`. Поэтому компиляторам пришлось специально проверять, что у нас ошибка подстановки не по какой-то причине, а потому что у нас `enable_if`.
И есть ещё проблемы SFINAE. Вспомним `std::advance`. Он по-разному работает для разных типов итераторов, там какой-то tag dispatching или `if constexpr`, и с этом опять же можно жить, но нам надо либо какие-то функции создавать, какие-то свои теги, что-то ещё. Причём в обоих этих решениях мы не можем просто дописать перегрузку к набору. Ровно как и со SFINAE также не можем просто расширить: если у нас какие-то две сущности, что одна обладает всеми свойствами, что вторая, но ещё какими-то, нам придётся явно выражать условие, что тип обладает одним свойством, но не другим.
Как это с обычными перегрузками работает?

```
struct A {};
struct B : A {};
struct C : B {};

void foo(A&);
void foo(B&);
void foo(C&);
```

```
int main()
{
    C c;
    foo(c);
}
```

Несмотря на то, что у нас во все три перегрузки `c` подходит, из этих трёх мы можем выбрать наиболее специализированную перегрузку, и всё будет хорошо. Когда мы сделаем структуру `struct D : C {}`, мы можем спокойно написать `void foo(D&)`, и всё будет работать хорошо.
А теперь представьте, что у нас `A`, `B` и `C` — это свойства какие-то, а не типы. Тогда, написав что-то подобное, мы проиграем:

```
template <class T>
std::enable_if<A<T>> foo(T&);
template <class T>
std::enable_if<A<T> && B<T>> void foo(T&);
template <class T>
std::enable_if<A<T> && B<T> && C<T>> void foo(T&);
```

Тут мы получим, что у нас в случае `C` всё подходит, поэтому ambiguous. Надо так:

```
template <class T>
std::enable_if<A<T> && !B<T>> foo(T&);
template <class T>
std::enable_if<A<T> && B<T> && !C<T>> void foo(T&);
template <class T>
std::enable_if<A<T> && B<T> && C<T>> void foo(T&);
```

Добавляя новую перегрузку, придётся дописать `&& !D<T>` ко всем уже имеющимся вариантам.
Так вот концепты также умеют в ранжирование компилятором, поэтому с ними нет проблем с расширяемостью.

Немного истории. Дизайн концептов. Ещё Александр Степанов в описании STL ввёл термин «концепт», и ещё тогда, в девяностых, была идея что-то такое сделать. Но тогда и шаблоны были в новинку, считалось, что это уже очень сложно, и добавлять ещё и концепты было бы очень сложно. Но в целом работа над концептами началась уже тогда, когда началась работа над C++11. У людей был прототип компилятора (concept gcc), эти люди концептуализировали STL, и казалось, что всё это будет уже в C++11, в черновиках стандарта уже были концепты, но прямо перед выпуском C++11 концепты решили удалить из стандарта, потому что не у всех была уверенность в том, что дизайн концептов в том виде, как они есть, вызывал вопросы, насколько он долговечен и хорош.
На какую тему шли дискусии?

1. Разница между explicit и implicit концептами.
   Пусть у нас есть класс `my_type`, у которого написан `<`. Чтобы наш тип удовлетворял концепту comparable, этого достаточно, или надо как в Java написать `implements Comparable<T>`? Аргументы есть и за и против. За explicit — не факт, что одинаковые названия операций значат одинаковый смысл. В изначальной реализации были и те, и другие концепты, большая часть была явной, но с течением времени разные люди приходили, говорили, что для данного концепта явность — плохо, есть совершенно тупой пример, который не работает, и со временем всё больше и больше интерфейсов становились неявными. Те концепты, которые вошли в C++20, они уже совершенно неявные. Впрочем, если нам надо, чтобы какой-то концепт был явным, заведите рядом с ним trait. А вообще в жизни это бывает нужно довольно редко. В STL это только итераторы и диапазоны. В первом есть `iterator_traits`, во втором — есть шаблонная переменная `enable_borrowed_range`.

2. Definition checking.

```
void advance(std::input_iterator auto& iter, std::integral auto distance)
{
    // ...
}
```

Тут можно двояко понимать то, что написано. Либо мы обязуемся использовать только то, что написано в `std::input_iterator`, либо нет. Когда компилятор проверяет это, это называется definition checking. В изначальном proposal'е definition checking был (ну, потому что хочется иметь его; почему шаблоны порождают такие длинные сообщения об ошибке? потому что компилятор не знает, где ошибка, и выдаёт все возможные места). Но с concept gcc была проблема: он существенно замедлял компиляцию программы. Это вызывало беспокойство. Авторы утверждали, что это проблема их реализации, а не самой идеи, но прототипа, который работал хорошо, у них не было.
Второй момент — давайте попытаемся залогировать то, что приходит в качестве параметра. Но у нас же definition checking, с чего бы наши типы умели выводиться? Сейчас мы бы просто занесли это под какой-нибудь `if constexpr`, но тогда подобного не было.
И в-третих, возникают тонкости со взаимодействием с обычными шаблонами: они же не проверяют ничего. И это ладно, а если вы из концептуального мира вызываете что-то, что делает tag dispatching.

3. Как концептуализирована стандартная библиотека. Мы для `set` требуем, чтобы типы нуждались в `<`. И когда это концептуализируем, можно требовать либо только `<`, либо все операции. Если требовать все, то сломается куча кода. Поэтому концепты типа `ordered` начали дробиться на более гранулярные: типа нам хватит только сравнения на меньше. Или вот у нас есть `std::advance`: может, нам там не нужен именно итератор, а хватит только инкремента. И тогда теряется смысл у каждого конкретного концепта: если раньше мы знали, что итератор — это понятная сущность, то теперь нет. И суммарно для стандартной библиотеки понадобилось 100 концептов.

В итоге люди отбросили идею концептов из C++11 и решили разработать что-то, что будет не настолько хорошим, но работающим.
И в итоге уже к C++17 концепты были по сути готовы, но у людей были беспокойства на тему, насколько всё продумано. А ещё в STL никакие концепты не добавлялись, что вызывало у людей озабоченность. А ещё в C++17 есть proposal по `auto`, и он там немножко конфликтует с proposal'ом по концептам (была конструкция, что читалась двояко). И ещё были люди, которые сомневались, что можно будет потом допилить definition checking.
И вот в C++20 всё (кроме definition checking'а) порешали, и теперь концепты у нас есть.

Синтаксис. Концепты можно писать так:

```
template <typename T>
concept destructible = std::is_nothrow_destructible_v<T>;
```

Несложно заметить, что концепты очень похожи на `bool`'евые предикаты от своих параметров, но на самом деле они не то что похожи, их можно даже с такой целью использовать.
Самая общая форма использования концептов выглядит как

```
template <class T>
requires destructible<T>
void foo(T&) { /* ... */ }
```

Ещё есть сокращённая форма

```
template <destructible T>
void foo(T&) { /* ... */ }
```

Или даже так:

```
void foo(destructible auto&) { /* ... */ }
```

С последней формой всё понятно, всё как с `auto`, а вот со вторым уже интереснее. У нас вроде был параметр концепта, а тут мы просто написали `destructible`. Как быть, если у нас, например, несколько параметров у концепта? Ну, вот так:

```
template <class T>
requires same_as<T, my_type>
void foo(T&) { /* ... */ }

template <same_as<my_type> T>
void foo(T&) { /* ... */ }
```

Ещё можно писать вот такие вещи:

```
template <class T>
concept ordered = requires (T x)
{
    {x < x} -> std::same_as<bool>;
    // В жизни не надо проверять наличие только меньше, проверьте наличие всего.
};
```

То есть это говорит нам о том, что сравнение двух `T` работает и возвращает `bool`.
Вообще в `requires` можно писать 4 разных штуки:

```
template <class T>
concept something = requires (T x)
{
    // simple
    x.foo(); // В x есть функция foo без параметров.

    // type
    typename T::value_type; // В T есть тип value_type.

    // compound
    {x < x} noexcept -> std::same_as<bool>;
    // Выражение компилируется и результат удовлетворяет концепту. И noexcept ещё.
    // noexcept и результирующий концепт можно не писать, если не надо.

    // nested
    requires destructible<T>; // Выпослнены какие-то ещё концепты.
}
```

Компилятор про концепты знает чуть больше, чем про обычные `constexpr bool`-переменные. Он знает, когда один концепт более специализирован, нежели другой. Как это работает?

```
template <typename T>
concept destructible = std::is_nothrow_destructible_v<T>;

template <typename T>
concept ordered = destructible<T> && requires (T x)
{
    {x < x} -> std::same_as<bool>;
}
```

Как понять, что одно — подмножество другого? На самом деле по произвольному предикату понять, что одно влечёт другое, нельзя. Поэтому в C++ определено нечто более простое. Точно не будет работать такое:

```
template <size_t N>
requires (N > 5)
void foo(std::array<int, N>&);

template <size_t N>
requires (N > 7)
void foo(std::array<int, N>&);
```

Компилятор считает эти штуки одинаковыми.
Компилятор строит нормальную форму: он берёт `&&`, `||` и концепты, раскрывает все их, а всё остальное считает atomic constraint (т.е. их он не сравнивает). И дальше когда компилятору надо сравнить, верно ли, что чисто по атомарным constraint'ам из одного следует другое. Например, `std::random_access_iterator` просто начинается с `std::bidirectional_iterator<T> &&`, и поэтому он более специализирован, нежели `std::bidirectional_iterator`. Что важно понимать — `requires`-expression считается атомарным constraint'ом. Не конъюнкцией.
Этот механизм приводи к интересным спецэффектам:

```
template <typename T>
requires destructible<T>
void foo(T&) {}

template <typename T>
requires destructible<T> && true
void foo(T&) {}
```

Нижняя функция более специализирована, нежели верхняя.
И такое — ещё половина беды.

```
template <typename T>
requires destructible<T> && true
void foo(T&) {}

template <typename T>
requires ordered<T> && true
void foo(T&) {}
```

Не компилируется, потому что `true` здесь — два разных atomic constraint'а, которые никак не связаны.
Но вообще концепты очень сильно помогают в жизни. Желающие посмотреть что-нибудь по теме могут посмотреть презентацию Андрея Давыдова.

10 Кодировки.

В основном мы, конечно, поговорим про Unicode. Юникод — это стандарт, ставящий числу в соответствие символ. Числа при этом называются codepoint'ами, и текст по сути представляется последовательностью codepoint'ов. Сами числа при этом лежат в диапазоне от `0` до `0x10FFFF`. Очень весёлый диапазон.
Рядом с Юникодом лежат кодировки, ассоциированные с ним: `UTF-8`, `UTF-16` (+ `UCS2`) и `UTF-32`. При этом последние два также могут быть little- или big-endian. `UTF-8` состоит из 8-битных чисел, `UTF-16` и `USC2` — 16-битных, `UTF-32` — 32-битных. Эти самые числа называются code unit'ами. Кодировки эти занимаются тем, что определяют, как преобразовать code point в code unit.

Кодировки.

UTF-32. `UTF-32` делает всё очень просто: каждый code point является одним code unit'ом. У этого есть существенный недостаток: 4 байта — безмерно много. Особенно учитывая тот факт, что символы из расширенных плоскостей (то, что вне `0`–`0xFFFF`) встречается довольно редко.

UCS2. UCS2 не поддерживает расширенные плоскости. Совсем. Если вы хотите закодировать code point до 0xFFFF, вы просто записываете code unit с этим числовым значением, а если code point выше, то вы проиграли. Поэтому UCS2 не может закодировать всё, не используется, и вместо него применяется UTF-16.

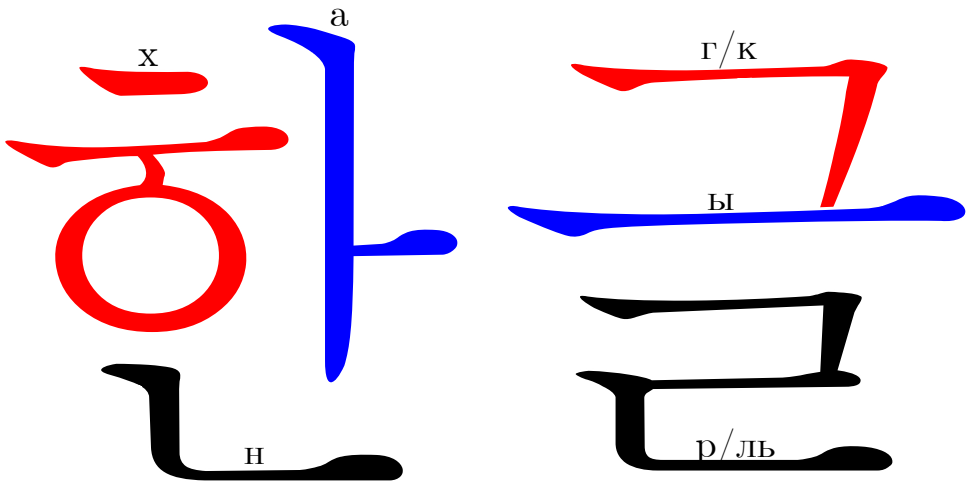
UTF-16. В Unicode есть мем. Коды в интервале 0xD800–0xDFFF некорректны. Поэтому эти коды используется для UTF-16. И вот как он работает: если code point помещается в 2 байта, он пишется as is. Если нет, то из него вычитается 0x10000, и получается 10 бит. Эти самые 10 бит кодируются парой code unit'ов: первый получается сложением 0xD800 и старших 5 бит, второй — сложением DC00 и младших 5 бит. Итого базовая плоскости кодируется 2 байтами, а дополнительные — парой из двухбайтовых слов. Такая пара называется **суррогатной парой**.

UTF-8. Традиционно программы использовали 8-битные кодировки. И не хочется жить с UTF-16. А представлять Юникод хочется. и придумали вот что.
Если у вас code point лежит в диапазоне ASCII (0–0x7F), то его пишут как 0xxxxxxx. Если от 0x80 до 0x07FF, то 110xxxxx_10xxxxxx, если от 0x8000 до 0xFFFF, то 1110xxxx_10xxxxxx_10xxxxxx, иначе 1110xxxx_10xxxxxx_10xxxxxx_10xxxxxx.
Code unit'ы в UTF-8 называют октетами, и каждый октет, как видно из описания, может быть trailing (10xxxxxx) и leading — всё остальное.
Но подождите. Значит ли то, что написано выше, что мы можем закодировать один символ разными способами? Ведь 11000000_10000000 — это вроде как '\0'. Так вот нет. Это называется overload sequences, и кодировщики UTF-8 не должны их создавать, а декодировщики должны считать это некорректным.
Кстати, интересный факт. UTF-8-строки можно лексикографически сравнивать пооктетно и получится, что у нас нормально отсортированы строки и с точки зрения code point'ов. В UTF-16 это не верно.

Byte-order mark. Он же «неразрывный пробел нулевой ширины». Это специальный codepoint, который вставляется в начало файлов для того, чтобы можно было определить, UTF-8 у вас, UTF-16LE или UTF-16BE.

Задачи, которые сложно решать в Unicode'е.

Длина и обращение по индексу. Казалось бы, в UTF-32 можно обращаться по индексу за $O(1)$. Но всё было бы хорошо, если бы не сам стандарт Unicode. Посмотрим на вот такую букву: Å. Есть специальная буква: «LATIN CAPITAL A WITH RING ABOVE», 00C5. Это один code point, и всё хорошо. Но есть обычная буква A и отдельно символ «COMBINING RING ABOVE». При их совмещении получается тот же самый глиф, но это два code point'а: 0041_030A. А ещё есть знак ангстрема, и это тоже отдельный символ с codepoint'ом 212B.
А есть ещё корейский алфавит «Хангыль», где буквы пишутся где попало.



И вот что вы хотите получить, когда просите длину строки для корейского текста?
Ну, хорошо, а как жить-то с этим? А на самом деле вам не бывает так часто нужно брать длину и обращаться по индексу. Вам бывает нужна конкатенация или поиск по строке. И это работает очень хорошо (поскольку `UTF-8`, например, специально помечает trailing-октеты особым началом). Но иногда, когда вы пишете растеризацию, например, то да, вам придётся страдать, увы и ах.

Поиск подстроки. Кстати. По поводу поиска подстрок. Когда нам дали Å, мы должны её сопоставлять только с конкретным символом или с любой фигнёй? Вообще если с любым возможным A с кружком, то тут нужна нормализация, но на самом деле, возможно, вы хотите делать соответствие ещё и «ё» и «е». Короче, если вы делаете user-friendly, то у вас всё грустно.

Приведение к верхнему/нижнему регистру. Ещё у любого Юникода есть проблемы с приведением в нижний и верхний регистр. Например, вот у буквы «`i`» какая заглавная? «`I`»? Ну, в английском да. А в турецком есть строчная `ı` («`i` без точки») и заглавная «`İ`» («`I` с точкой»).
Есть греческая буква «сигма», у которой есть две строчные версии: одна встречается на концах слов («`ς`»), другая — в серединах («`σ`»), и поэтому для приведения в нижний регистр надо смотреть на границы слов.
Есть голландский, в котором есть дираф `ij`, и его с точки зрения грамматики надо капитализировать всей парой целиком (несмотря на то, что обычно его пишут именно как два символа, а не как дираф, и, более того, это Юникод не рекомендует этот дираф).
Мораль: капитализировать буквы — порочное занятие. Большие и маленькие буквы вообще встречаются в не таком большом количестве систем письма. Вот, грузины живут без регистра вообще, и всё у них хорошо.

Лексикографическая сортировка. Ещё одна задача: лексикографическая сортировка. Тоже, на самом деле, очень странное занятие. Буква «`Ё`» находится до всех заглавных букв, а «`ё`» — после. Да более того, в разных языках разные буквы могут стоять в разных местах. В немецком языке есть умляуты (Ä, Ö, Ü). Они должны сортироваться так, как будто они — одно и то же. А в эстонском эти же буквы находятся в конце алфавита.

Взаимодействие с операционными системами. Пока вы сидите на Linux'е, там почти всё живёт с UTF-8, и всё у вас хорошо. Есть исключения (QT, например, использует UTF-16, так как она была создана давно, когда Юникод был 16-битным). Впрочем, в Linux можно использовать другую восьмибитную кодировку (при помощи переменной окружения `LANG`), но некоторые программы полностью это игнорирует, считая, что у нас всё в UTF-8.
Про MacOS так много не расскажем, но там тоже есть какое-то API с UTF-8, какое-то с UTF-16, но тут у нас нет столько информации, так что не будем больше ничего говорить.
А вот с Windows ситуация интересная. Она внутри себя использует UTF-16. Но у них всё ещё есть совместимость с восьмибитными кодировками, правда, долгое время нельзя было в качестве неё поставить UTF-8. И приводило это к совершенно упоротым эффектам: если вы пользуетесь набором функций с 8-битной кодировкой (которая 1251), то не существует способа конвертировать, например, название файла, так, чтобы оно конвертировалось в то, что вам хочется. Поэтому Microsoft советовали использовать их функции формата `_wfopen`, с которой всё хорошо, но тогда программа теряет в платформонезависимости. Поэтому многие чуваки делали свои библиотеки (см. Boost.Nowide), которые на Windows конвертировали строки в 16 бит и вызывали `_wfopen`, а на Linux делали просто `fopen`. В Windows 10, впрочем, таких приколов нет, так что вам с этим заморачиваться не надо.
Второй прикол в Windows — наличие двух разных восьмибитных кодировок. Первая — ACP (ANSI code page) (для русской Windows — 1251), вторая — OEMCP (OEM code page) (для русской Windows — 866). И разные восьмибитные функции работали в разных кодировках. Поэтому получив строку из одной функции и отдав в другую, вы могли увидеть, что строка попортилась. Что ещё интереснее, некоторые функции могут использовать одно или другое (функция `SetFileApisToOEM` переключает кодировку для работы с файлами, например).
Мораль: на Windows ставьте обеими текущими кодировками UTF-8 и живите счастливо. А если уж вам нужно что-то старое, то вам можно только посочувствовать (но вообще выберите одну кодировку для своей программы и при необходимости конвертируйте туда-обратно).

Имена файлов. Впрочем, это не поможет вам делать `fopen`'ы. Потому что имена файлов во многих ОС (и Windows, и Linux) не обязаны являться корректными строками. В Linux, как мы знаем, файлы могут содержать любые символы, кроме `'\0'` и `'/'`. Например, файл может быть назван `"Ha\bello"`, и многие утилиты будут выводить название файла как `"Hello"`. И это то ладно. У нас может быть некорректные октеты (head octet без tail octet'а или наоборот или ещё чего). В Windows ситуация такая же: у вас может быть первый символ суррогатной пары без второго. Мораль: не пытайтесь преобразовать путь в строку и обратно, надеясь, что он не испортится. И не все пути можно нормально в строку преобразовать вообще.
А теперь вспомним, что имена файлов на Windows не зависят от регистра. Некорректные UTF-16 строки, не зависящие от регистра. Это, простите, что за срань? А ситуация такая: правило сопоставления маленьких букв большим прошито в файловой системе NTFS. На NTFS создаётся файлик (называется UpCase), в котором и хранится информация о сопоставлении маленьких букв большим. И в разных файловых системах (а значит и в разных разделах вашего компьютера) это сопоставление может быть разным. Да и более того. В EXT4 вы можете повесить себе на каталог опцию, чтобы в нём имена были регистронезависимы, а потом и в Windows — обратную опцию.

11 Многопоточность.

Многопоточность в терминах C++ довольно важна многопоточность не только для того, чтобы знать, как она есть, но и, например, чтобы понимать, почему аллокаторы или shared pointer работают так, как работают, потому что сильно связаны с дизайном многопоточки.
Зачем многопоточка, можно понять из курса АрхЭВМ и OS-lite. Но чего там нет, так это то, что вообще для использования ядер не нужно использовать потоки. Мы можем просто запускать процессы несколько раз (так компиляторы работают, например).

Создание и завершение потоков на примерах.

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

int main()
{
    std::thread th([]
    {
        std::cout << "Hello, world!\n";
    });
    th.join();
}
```

В стандартной библиотеке есть класс `thread`. В конструкторе он стартует поток и потом можно дождаться, пока он закончит (`join`). Помимо к ней есть операция `detach`. Она отвязывает наш `thread` от потока ОС, но поток ОС никто не завершает. Такое бывает нужно очень редко. Например, вот такая программа просто некорректна:

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

int main()
{
    std::thread th([]
    {
        std::cout << "Hello, world!\n";
    });
    th.detach();
}
```

Поток быдет выполняться тогда, когда мы уже будем при выходе из программы вызывать деструкторы. В частности, деструкторы `std::cout`. И отвязанный поток будет писать в то место, чего нет.
Хорошо, а если мы не напишем ни `join`, ни `detach`, что будет? Ну, будет деструктор `thread`, объективно. А что там? А там `std::terminate`. Возникает вопрос: зачем? А потому что у нас бывают потоки, которые работают бесконечно долго. Поэтому по умолчанию сделать `join` — не очень затея. Но если вы всё же хотите `join` по-умолчанию, то ваш выбор — `std::jthread`. Он в деструкторе делает `join`. Но на самом деле у него ещё есть способ сказать потоку «так, ты это, выходи пожалуйста», и после этого делает `join`.
Ну, хорошо, а если поток не хочет с вами кооперироваться и выполнять вашу просьбу о завершении? Тогда вы проиграли. В C++ никак нельзя заставить поток умереть, потому что иначе могут быть неконсистентные данные. Несмотря на то, что в операционных системах есть какие-нибудь операции по типу TerminateThread, на всех них написано «опасная функция, которую стоит использовать в совсем экстренных случаях». Если вы убьёте поток в критической секции, вы проиграли. Если вы убили поток в тот момент, когда он выделял память, то вы совсем больше не сможете выделять память в вашей программе.

Проблемы в использовании потоков и способы с ними жить.

Гоночки. Пусть мы делаем банковское приложение и у нас есть 10000 счетов. И мы хотим сделать операцию `transfer`, которая переводит заданную сумму со счёта на счёт.

```cpp
std::array<uint32_t, 10'000> accounts;

void transfer(size_t from, size_t to, uint32_t amount)
{
    if (accounts[from] < amount)
```

```
            throw std::runtime_error("insufficient funds");

        accounts[from] -= amount;
        accounts[to] += amount;
    }
```

Чудесная функция для однопоточного случая. И даже если не иметь проблем с `+=` и `-=`, то мы можем переводить деньги с одного счёта на два разных, и тогда мы можем два раза подряд проверить, потом два раза вычесть. А денег изначально только на один перевод хватало. Ой.
Ещё мы не знаем, как работает `+=`. Это может быть единая неделимая операция, а может быть сначала чтение, потом запись. И тут у нас ещё одна гонка: когда мы сначала читаем два раза, потом оба вычитают и записывают. И эффект от одного `+=` не применяется.
Это чудо называется **состояние гонки**, **race condition**. Если воспринимать это чудо как псевдокод, если же написать это честно на C++, то это жёсткое UB.

`std::mutex`. Хто это такой? Это класс, поддерживающий две операции: `lock` и `unlock`. Первая ждёт, пока `std::mutex` не разблокируют, после чего блокирует сам, вторая — разблокирует:

```
std::mutex accounts_mutex;
std::array<uint32_t, 10'000> accounts;

void transfer(size_t from, size_t to, uint32_t amount)
{
    accounts_mutex.lock();

    if (accounts[from] < amount)
            throw std::runtime_error("insufficient funds");

    accounts[from] -= amount;
    accounts[to] += amount;

    accounts_mutex.unlock();
}
```

Что важно понимать: `mutex` идеологически ассоциирован с данными, а не функциями. Если бы мы написали ещё какую-то операцию с `accounts`, надо там тоже работать с `mutex`'ом.
Что с данной программой не так? `throw` не разблокирует `std::mutex`. Потому что не RAII. Наше счастье, что в стандартной библиотеке есть RAII-шная обёртка: `std::lock_guard`:

```
std::mutex accounts_mutex;
std::array<uint32_t, 10'000> accounts;

void transfer(size_t from, size_t to, uint32_t amount)
{
    std::lock_guard<std::mutex> lock(accounts_mutex);

    if (accounts[from] < amount)
        throw std::runtime_error("insufficient funds");

    accounts[from] -= amount;
    accounts[to] += amount;
}
```

Кстати, `std::lock_guard<std::mutex>` в данном коде можно заменить на просто `std::lock_guard` (начиная с C++17). Это называется CTAD (class template argument deduction), но это сейчас не важно.

Давайте вот ещё что скажем: у нас была хорошая однопоточная функция `transfer`. А теперь её можно вызывать их разных потоков, но параллельной эта функция не стала. Не проблема ли это? Ну, разумеется, нет, если у нас программа не только `transfer` делает, а ещё что-то делает, то всё остальное можно делать многопоточно, однопоточный у нас только `transfer` — скорее всего, малая часть однопоточной программы.

Закон Амдала. У нас есть части кода, которые параллелятся, и части, которые нет. И чем больше процент первого, тем больше ускорение программы мы можем получить. Если у нас 50% программы можно распараллелить, то как много у нас потоков не было бы, ускорение больше чем вдвое мы не получим никак. Хотя на самом деле сто⊠ит ещё учитывать то, что чем больше у нас потоков, тем больше накладных расходов у нас есть.

Deadlock. У нас однопоточный `transfer`. Это плохо и совершенно не нужно. Давайте просто положим `std::mutex` в account:

```
struct account
{
    std::mutex m;
    uint32_t amount;
};
std::array<account, 10'000> accounts;

void transfer(size_t from, size_t to, uint32_t amount)
{
    std::lock_guard lock1(accounts[from].m);
    std::lock_guard lock2(accounts[to].m);

    if (accounts[from] < amount)
        throw std::runtime_error("insufficient funds");

    accounts[from] -= amount;
    accounts[to] += amount;
}
```

Здравствуйте, обедающие философы. Если у нас цикл из transfer'ов и каждый успеет взять первый нужный ему `mutex`, и будет ждать второго. Все стоят, ждут. Ничего нельзя сделать. **Deadlock** называется.
Как решить? Ну, вот так:

```
void transfer(size_t from, size_t to, uint32_t amount)
{
    std::lock_guard lock1(accounts[std::min(from, to)].m);
    std::lock_guard lock2(accounts[std::max(from, to)].m);

    if (accounts[from] < amount)
       throw std::runtime_error("insufficient funds");

    accounts[from] -= amount;
    accounts[to] += amount;
}
```

Почему такое корректно? Тут можно нарисовать граф, где вершины — `mutex`'ы, а рёбро есть, если в программе есть случай, когда мы взяли один `mutex` и пытаемся взять другой. Если в таком графе нет цикла, мы победили. И у нас его в нашем случае действительно нет.

std::recursive_mutex. Но жизнь — боль. Так как то написано, у нас UB. Когда поток уже владеет std::mutex'ом и пытается получить его ещё раз, это UB. Поэтому

```
void transfer(size_t from, size_t to, uint32_t amount)
{
    if (from == to)
        return;

    std::lock_guard lock1(accounts[std::min(from, to)].m);
    std::lock_guard lock2(accounts[std::max(from, to)].m);

    if (accounts[from] < amount)
        throw std::runtime_error("insufficient funds");

    accounts[from] -= amount;
    accounts[to] += amount;
}
```

И это в данном случае всё легко правится, но вообще, иногда нам очень хочется уметь брать mutex несколько раз, и в стандартной библиотеке есть такой вид mutex'ов: std::recursve_mutex. И вообще в стандартной библиотеке есть много интересных mutex'ов, и ещё больше есть в сторонних библиотеках.

Другие примитивы синхронизации в попытках написать многопоточную очередь. Представьте, что у нас есть какие-то многопоточные запросы, и мы хотим поддерживать очередь из них.

```
template <class T>
struct concurrent_queue
{
private:
    std::mutex m;
    std::deque<T> queue;

public:
    void push(T val)
    {
        std::lock_guard lock(m);
        queue.push_back(std::move(value));
    }

    bool empty() const
    {
        std::lock_guard lock(m);
        return queue.empty();
    }

    T pop()
    {
        std::lock_guard lock(m);
        T result = queue.back();
        queue.pop();
        return result;
    }
};
```

Нормально ли такое? Да нифига. Потому что мы не можем вызывать empty и сразу после этого pop. Потому что между этими вызовами у нас нет замка. Поэтому правильнее будет изменить интерфейс.

```
template <class T>
struct concurrent_queue
{
private:
    std::mutex m;
    std::deque<T> queue;

public:
    void push(T val)
    {
        std::lock_guard lock(m);
        queue.push_back(std::move(value));
    }

    std::optional<T> try_pop()
    {
        std::lock_guard lock(m);
        if (queue.empty())
            return std::nullopt;
        std::optional<T> result = queue.back();
        queue.pop();
        return result;
    }
};
```

В чём тут ещё проблема? В том, что у нас есть чуваки, которые сидят и ждут, пока будут данные. Они не хотят писать `while`, это активное ожидание. Давайте сделаем `pop`, который ждёт.

`std::condition_variable`. Вот какая эта штука. У него есть 3 операции: `wait`, `notify_one` и `notify_all`. Первая ждёт, вторая пробуждает одного ждущего, третья — пробуждает всех. Пока что наш код не будет компилироваться, но давайте всё равно его напишем:

```
template <class T>
struct concurrent_queue
{
private:
    std::mutex m;
    std::deque<T> queue;
    std::condition_variable cv;

public:
    void push(T val)
    {
        std::lock_guard lock(m);
        queue.push_back(std::move(value));
        cv.notify_one();
    }

    T pop()
    {
        std::lock_guard lock(m);
        if (queue.empty())
           cv.wait();
        T result = queue.back();
        queue.pop();
```

```
        return result;
    }
};
```

Тут понятно какая проблема, мы спим с `mutex`'ом. Исправляем:

```
T pop()
{
    std::unique_lock lock(m); // Как lock_guard, но можно делать lock и unlock.
    if (queue.empty())
    {
        lock.unlock()
        cv.wait();
        lock.lock();
    }
    T result = queue.back();
    queue.pop();
    return result;
}
```

И тут уже сильно лучше, но если один поток пробудился и ещё не взял блокировку, а другой тем временем зашёл, увидел непустую очередь и опустошил её. И всё, мы проиграли. Поэтому вместо `if`'а надо взять `while`.

```
T pop()
{
    std::unique_lock lock(m);
    while (queue.empty())
    {
        lock.unlock()
        cv.wait();
        lock.lock();
    }
    T result = queue.back();
    queue.pop();
    return result;
}
```

Кстати, есть ещё одна причина, почему `while`: спонтанные пробуждения. `condition_variable` может сам внезапно пробудиться ни с чего.
И эту проблемы мы починили, но ещё есть другая: если мы уже сделали `unlock`, нам в это время сделали `push`, но `notify_one` произошёл до того, как мы стали ждать. И всё, мы ждём при непустой очереди. Поэтому `wait` на самом деле принимает один аргумент: `unique_lock`, чтобы сделать `unlock` и `wait` неделимо:

```
T pop()
{
    std::unique_lock lock(m);
    while (queue.empty())
        cv.wait(lock);
    T result = queue.back();
    queue.pop();
    return result;
}
```

На самом деле такой паттерн с `while`'ом настолько частый, что под это специальная перегрузка есть:

```
T pop()
{
    std::unique_lock lock(m);
    cv.wait(lock, [&]{ return !queue.empty(); });
    T result = queue.back();
    queue.pop();
    return result;
}
```

Thundering herd problem. Общего развития ради: не надо использовать `notify_all` там, где можно `notify_one`. Такое даже название имеет: Thundering herd problem.

Worst practices. Может показаться, что `notify` — штука долгая, и её можно пытаться оптимизировать. Давайте не будем никого уведомлять, если очередь не была пуста.

```
void push(T val)
{
    std::lock_guard lock(m);
    bool was_empty = queue.empty();
    queue.push_back(std::move(value));
    if (!was_empty)
       cv.notify_one();
}
```

Кто видит проблему? А она есть.
Пусть у нас есть несколько спящих потоков на `pop`'е. К ним приходит несколько `push`'ей сразу. Первый уведомляет одного, остальные — нет, все спящие потоки, кроме одного, продолжат спать при непустой очереди.

Best practices. Если производителей данных больше, чем потребителей, очередь может бесконечно расти. Давайте ненадо. Ну, тривиально правится: заводим ещё `std::conditional_variable`, который будем ждать в `push` и сообщать в `pop`. Это, кстати, объясняет, зачем нам отдельно `mutex`'ы и отдельно `condition_variable`.

Реализация. Понятно, что для всего этого необходимы специальные инструкции со стороны процессора. А ещё что `mutex`'у, что `condition_variable`'у надо уметь засыпать, и это уже поддержка со стороны ОС.

Worst practices. Во-первых, не надо делать `std::this_thread::yield`. Никогда. Любой хоть сколько-нибудь продвинутый планировщик старается минимально мигрировать задачи между ядрами (кэши охлаждаются, например). Поэтому планировщики могут игнорировать `yield`. Зачем эта функция существует — ничуть не для примитивов синхронизации, а только если вы знаете всё про планировщик и про приоритеты.
А что использовать для реализации? То, что ожидает ОС, объективно. И то что разработчики ОС рекомендуют. В POSIX есть такая штука как futex, и всё реализуются через него.

Busy wait/spinlock. Может нам не надо идти в ядро для каждого `mutex`'а, может его только вот прямо сейчас отпустят? Такая стратегия называется spinlock, и обычно `mutex`'ы реализуют как комбинацию этого и нормальных `mutex`'ов.
Кстати, многие процессоры имеют в себе инструкции-подсказки для ОС (`PAUSE` на x86, например).

Атомики. Про это можно говорить очень много, но этим мы будем заниматься на специальном курсе про многопоточность.

Мотивация. В прошлый раз у нас была программа с банковскими счетами. Там мы делали `mutex`'ы на каждый счёт. Мы помним, что у нас куча памяти тратится, возможно, стоило бы сделать что-то другое? А вот да: в процессоре чтения и записи даже из разных потоков как-то определены. Тогда почему в C++ не определено многопоточное поведение для встроенных типов? Ну, потому что на разных процессорах чтения и записи определены по-разному. И можно наблюдать самые разные эффекты, смотря в ассемблер.
Например, поскольку процессор может переупорядочивать инструкции, он может переупорядочивать записи и чтения. И на разных процессорах можно получить самые разные мемы. Есть даже весёлая статейка, в которой на 16 странице есть табличка о том, какие архитектуры что могут переупорядочивать. И эта табличка очень сильно упрощена, потому что почти все архитектуры имеют разные команды, которые позволяют разную степень переупорядочивания.
Так вот, поэтому нельзя специфицировать многопоточное поведение для встроенных типах: когда процессор переупорядочит инструкции, ваше специфицирование вам же выйдет боком. Причём компилятор-то не знает, многопоточное ли у вас приложение, или нет, ему придётся использовать тяжёлые инструкции всегда.
Чтение и запись — это, конечно, хорошо, но разве мы не хотим делать инкремент неделимо. То есть так, что весь остальной мир не видит промежуточных состояний: либо операция не началась, либо закончилась (это *атомарностью* и называется). Такие операции, кстати, и в процессорах нередко есть (так атомарный инкремент называется `lock add` на x86). И вообще в `std::atomic` есть много весёлых сложных операций, например обмен со сравнением.
Кстати, ещё важный момент: в атомарности очень важно выравнивание. В процессоре нельзя атомарно поменять значение, лежащее на границе кэш-линий. Почти во всех архитектурах. В x86 можно, но это работает медленно (существенно замедляет всю систему). Если вам зачем-то это нужно, то знайте, что это называется split lock'ами, почитать про них можно тут, но оно вам таки не нужно.

Пример. Давайте переделаем наш пример с банковскими счетами и переделаем его на атомарные операции.

```
std::array<std::atomic<int32_t>>, 10'000> accounts;

void transfer(size_t from, size_t to, uint32_t amount)
{
    if (from == to)
        return;

    if (accounts[from] < amount)
        throw std::runtime_error("insufficient funds");

    accounts[from] -= amount;
    accounts[to] += amount;
}
```

Пока что оно не работает. По понятным причинам, нам нужно не только атомарно делать `-=`, нам надо атомарно сделать проверку и вычитание. Ну, это можно. Есть очень весёлая операция `compare_exchange`, которая сравнивает значение атомарной переменной с заданным и если совпадает, то происходит присваивание:

```
std::array<std::atomic<int32_t>>, 10'000> accounts;

void transfer(size_t from, size_t to, uint32_t amount)
{
    if (from == to)
        return;

    int32_t old = accounts[from];
```

```
        do
        {
            if (old < amount)
                throw std::runtime_error("insufficient funds");
        } while (!accounts[from].compare_exchange_weak(old, old - amount))

        accounts[to] += amount;
    }
```

То есть мы берём значение, проверяем, если оно не поменялось, вычитаем. Иначе в переменную `old` присваивается новое значение `accounts[from]` и мы пытаемся снова.
Хорошо, а почему `compare_exchange_weak` назван «weak»? Потому что он может спонтанно обвалиться. Зачем он такой? Потому что на некоторых процессорах (POWER, например) нет инструкции под «strong», поэтому там в ассемблере будет написан цикл. А если у нас уже цикл, зачем нам ещё один внутри.

Relaxed atomic. Вот есть у нас чтение `atomic`'а. Там есть метод `load`, который принимает `std::memory_order`. Это хто? Подробно об этом тут, но мы вкратце обсудим.
Проблема с обычными операциями (которые работают с `std::memory_order_seq_cst`) в том, что все операции гарантируют, что их можно чётко расположить на временной оси и определить порядок. Но реальное железо работает совсем иначе, и поддержание иллюзии последовательности стоит недёшево. И только на x86 `std::memory_order_seq_cst` работает нормально. Там вообще почти все `std::memory_order`'ы работают почти одинаково. Но вот на POWER'ах или ARM'ах все они порождают разные операции.
Окей, так что такое не sequential-consistent операции? Какие могут быть эффекты?

```
int x = 0, y = 0;

void thread_1()
{
    x = 1;
}
void thread_2()
{
    x = 1;
}
void thread_3()
{
    int x3 = x;
    int y3 = y;
    if (x3 == 1 && y3 == 0)
        std::cout << "x was written to before y" << std::endl;
}
void thread_4()
{
    int y4 = y;
    int x4 = x;
    if (x4 == 0 && y4 == 1)
        std::cout << "y was written to before x" << std::endl;
}
```

Так вот в relaxed-модели вполне возможна ситуация, когда поток 3 увидит, что `x` был записан до `y`, а поток 4 — что `y` до `x`. Потому что у нас есть кэши, они портят нам наивное представление о работе памяти. В sequential-consistent модели такого тоже не бывает, а в relaxed можно интерпретировать ситуацию так, что данные о записи в переменную не доставляются всем сразу.
Так а что, в x86 не бывает такого? Нет, там тоже бывает. Конкретно такой пример (если его на

ассемблере написать) будет работать, но на x86 сами потоки, которые пишут, могут разойтись во мнениях, кто записал раньше (см Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4? страница 3094, раздел 8.2.3.5, пример 8-5).
Хорошо, давайте разбираться в `std::memory_order`'ах.

- Если один поток сделал запись с `release`, а второй — чтение с `acquire`, то все чтения, сделанные до записи, будут видны во втором потоке. Можно применить например тогда, когда мы имеем две переменных: значение и флаг, посчитано ли оно.

```
int computation_result;
atomic<bool> computation_result_ready;

void thread_1()
{
    computation_result = 42;
    computation_result_ready.store(true, std::memory_order::release);
}
void thread_2()
{
    if (computation_result_ready.load(std::memory_order::acquire)
    {
        int tmp = computation_result;
        // ...
    }
}
```

- Представим себе вот такой пример:

```
atomic<uint64_t> event_counter;

void thread_proc()
{
    while (true)
    {
        // ...
        event_counter++;
        // ...
    }
}

int main()
{
    std::thread th1(&thread_proc);
    std::thread th2(&thread_proc);
    th1.join();
    th2.join();

    uint64_t tmp = event_counter;
}
```

Насколько нам важен конкретный порядок инкрементов `event_counter`? Да ни насколько не важен. Важно только то, что все они произойдут. Ну и хорошо:

```
atomic<uint64_t> event_counter;

void thread_proc()
```

```
{
    while (true)
    {
        // ...
        event_counter.fetch_add(1, std::memory_order::relaxed);
        // ...
    }
}

int main()
{
    event_counter.store(0, std::memory_order::relaxed);

    std::thread th1(&thread_proc);
    std::thread th2(&thread_proc);
    th1.join();
    th2.join();

    uint64_t tmp = event_counter;
}
```

- `std::memory_order::acq_rel` — комбинация `acquire` и `release`, применяется в некоторых операциях, которые и читают, и пишут.
- А кто же такой `std::memory_order::consume`? Вообще это нечто среднее между `acquire` и `relaxed`. Нужен он бывает редко, поддерживается не всеми, так что фиг с ним, неважно, кто это.

Worst practices. Никогда, слышите, никогда не используйте один `memory_order` вместо другого, опираясь на то, что на вашей архитектуре они одно и то же. Это не только очень платформозависимо, это ещё и от компилятора зависит, потому что он тоже имеет право инструкции переставлять. Поэтому руководствуйтесь только свойствами языка.

Ненативные атомики. Вообще в `std::atomic` можно запихнуть любой тривиально копируемый (и просто перемещаемый) тип. Но не для любого же размера процессор имеет инструкции. Ну, да. Поэтому для некоторых типов `std::atomic` будет скрафчен руками на основе `mutex`'ов. Это даже проверить можно.

Цена атомиков. Sharing. Рассмотрим следующую программу:

```
#include <boost/program_options.hpp>
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>
#include <memory>

namespace po = boost::program_options;

void thread_proc(std::atomic<int>& v, std::atomic<bool>& finish)
{
    while (!finish.load())
        v += 1;
}

int main(int argc, char* argv[])
```

```
{
    try
    {
        po::options_description desc("Allowed options");
        desc.add_options()
            ("threads,j", po::value<size_t>(), "set number of threads")
            ;

        po::variables_map vm;
        po::store(po::parse_command_line(argc, argv, desc), vm);
        po::notify(vm);

        size_t number_of_threads = 1;
        if (vm.count("threads") != 0)
            number_of_threads = vm["threads"].as<size_t>();

        for (;;)
        {
            std::cout << number_of_threads << " threads";

            std::atomic<int> val(0);
            std::atomic<bool> finished(false);

            std::vector<std::thread> threads;
            for (size_t i = 0; i != number_of_threads; ++i)
                threads.emplace_back(&thread_proc, std::ref(val), std::ref(finished));

            std::this_thread::sleep_for(std::chrono::seconds(2));
            finished.store(true);

            for (auto i = threads.rbegin(); i != threads.rend(); ++i)
            {
                auto& th = *i;
                th.join();
            }

            std::cout << ", " << val.load() << " iterations" << std::endl;
        }
    }
    catch (std::exception const& e)
    {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }
}
```

Как думаете, как будет меняться вывод в зависимости от количества потоков? А вот будет сильно уменьшаться. Например, на двух потоках значение вчетверо меньше, чем на одном. Причём что ещё хуже, в зависимости от того, на каких ядрах мы запускаем программу, получаются существенно различающиеся результаты.

И тут можно догадаться, что дело всё в кэшах. Её периодически бывает нужно забрать с другого ядра. Тут у нас одна переменная на кучу потоков, что порождает проблему с производительностью, которая называется sharing. И решается она обычно созданием разных переменных под каждый поток и последующим объединением результатов.

Хорошо, фикс на 100 рублей: вместо одной переменной `val` массив. Поможет? Ну, разумеется, нет,

массив не пораскидаешь по разным кэш-линиям. Такое false-sharing называется (как и в принципе если две переменные в одной кэш-линии лежат).

Аллокаторы памяти. На самом деле дизайн аллокаторов памяти сильно определяется наличием нескольких потоков. В нулевых аллокаторы были однопоточными, просто рядом с ними лежал `mutex`. Но по мере развития многоядерности аллокаторы стали узким местом. И там появились разные решения по типу hoard'а, который считает хэш от номера потока и блокирует не всё вообще после этого. И уже на этих идеях создали свои аллокаторы.

`volatile`.

Исторический контекст. Многопоточка появилась в C++11. Но многопоточные программы писались и до этого. Как? Конкретные ОС, библиотеки могли предоставлять свои способы делать многопоточку. И некоторые библиотеки для разных целей использовали `volatile`. В интерпретации стандарта `volatile` к многопоточке не имеет никакого отношения. А к чему имеет?

Применения `volatile`. У нас могут быть какие-то устройства, которые тупо отображаются в память по каким-то адресам. Вы делаете запись, а это не честная запись в память, а интерфейс для взаимодействия с устройством. И вполне нормальное применение — читать несколько раз подряд. Любой нормальный компилятор охуеет от такого и заменить все чтения на одно. Для обычной памяти это норм, но вот в случае с устройствами нам такое не надо. Это первое применение `volatile`.
Второй случай — `setjmp` и `longjmp`. На C++ вам это не надо, но вообще это в некотором смысле С'шный аналог исключений. И там тоже при записи в переменную после `setjmp` может иметь смысл делать `volatile`.
И третий случай — UNIX-сингалы (ничего общего с сигналами, которые мы обсуждали). Вас на любой инструкции могут дёрнуть и сказать разные интересные вещи. И поскольку вас могут позвать на любую инструкцию, делать обработчики UNIX-сигналов могут не всё. Например, могут обращаться к `volatile`-переменным. Это более применимый случай, но тоже довольно нишевый.

Почему это не атомики. Ни для каких других случаев `volatile` не используется, не надо пытаться использовать из вместо `std::atomic`'ов. У них есть некоторые схожие гарантии, но они во многих местах различны.

```
volatile int x;
int y;

void foo()
{
    int x_copy = x;
    if (x_copy)
        int y_copy = y;
}
```

Тут с `volatile`-переменными мы можем их переупорядочить, как нам хочется (ну а какая разница, если первое из памяти устройства, а второе — из памяти). А с атомиками это не так.
Есть и обратный пример:

```
volatile int z;

void bar()
{
    z = 1;
    z = 2;
}
```

В случае с `volatile` мы никак не можем убрать первую инструкцию, а в случае `atomic`'ов — можем, почему нет. Другое дело, что компиляторы её не делают, но могут: вполне возможна ситуация, когда никто не увидит первую запись, такая она быстрая. На тему того, что компилятор не оптимизирует `std::atomic`'и есть даже статья и другая статья, в которых написано, что компилятор не будет пытаться оптимизировать атомики, хотя на самом деле есть несколько случаев, когда этого хочется и это будет корректно.

Мораль. Применим ли `volatile` в потоках в C++?

Ещё немного про не sequential-consistent атомики. Есть версия, что их не надо использовать по причине их неинтуитивности. Причём иногда максимально неинтуитивная. Можете открыть презентацию о том, как атомики пихали в CUDA и найти там несколько примеров. Там модель памяти чем-то похожа на POWER и 32-битный ARM. И эти чуваки из NVIDIA начали проверять свою трансляцию формальными методами: они брали пример плюсовой программы, смотрели, как позволяет стандарт и сравнивали со своей трансляцией. И столкнулись они с тем, что стандарт и трактовка большинства компиляторов не согласована (компиляторы слишком много себе позволяют). После этого стандарт пофиксили, но пример всё ещё очень показателен: куча чуваков, которые очень долго шарили в многопоточке, не замечали это очень долго. Поэтому очень сильно думайте, насколько вам нужны relaxed-атомики, особенно в комплексных случаях.
Да и вообще, писать многопоточку сложно. Вы не можете найти случай, когда программа не работает, например. Тут вам помогут thread-sanitizer'ы, cppmem и средства формальной проверки, но боже, оно вам точно надо?

Cancellation. Представьте себе какую-нибудь UI-ную программу, которая что-то делает по подсчёту кнопки. Обычно в таком случае пользователь хочет видеть какие-то результаты. Но бывают и ситуации, когда посреди вычисления мы понимаем, что результат нам не нужен. Если мы реализуем долгую операцию как вычисление в потоке, надо думать о том, как можно отменять. Мы помним, что операции `terminate` у потока нет. Поэтому было бы неплохо предусмотреть механизм «выходи, пожалуйста».
Если мы не пользуемся никакими библиотеками, мы могли бы завести `bool`'овый флажок, который в потоке периодически опрашивать.
Ещё есть прерывания, которые вообще говоря встречаются на низком уровне (те же UNIX'овые сигналы). Однако внутри обработчиков этих прерываний можно делать не так много полезного. Когда вас прервали посреди непонятно какой инструкции, у вас с инвариантами происходит лютый ужас. Вы, конечно, могли быть просто считать, что из каждой инструкции процессора может вылететь исключение, но с таким жить ой как непросто.

`std::jthread`, `std::stop_token`. В C++20 есть `std::jthread`, в который уже встроенный механизм «остановись, пожалуйста». Там есть `std::stop_token`, он позволяет проверять, что остановили. И есть ещё `std::stop_source`, при помощи которого и можно останавливать. Но второй вам не нужен, вам нужен метод `request_stop` у класса `std::jthread`.
Но в чём проблема? В том, что у нас есть блокирующие операции. Любая операция с файловыми дескрипторами сидит и ждёт, пока ей дадут данные. А пока мы сидим и ждём данные, мы не можем проверять `std::stop_token`. И ладно ввод-вывод, но у вас есть совершенно естественные для потоков операции — `std::thread::join`, `std::mutex::lock` и `std::condition_variable::wait`. Для `std::mutex::lock` вообще неестественно, что он не прерывается.
В общем случае с блокирующими операциями вам нечего делать. Но в некоторых специфичных случаях, зная предметную область блокирующей операции, есть способы что-то сделать. Если вы пользуетесь какими-то библиотеками, то в них написано, что происходит. Если библиотека претендует на совместимость с STL, в них будет появляться поддержка `std::stop_token`, и всё у вас будет хорошо. Да и в самой стандартной библиотеке есть способы. Например, у `std::condition_variable` есть перегрузка `wait`, которая принимает `std::stop_token`. А ещё есть такая штука как `std::stop_callback`, которая регистрирует обработчик на `std::stop_token`. Если вам чем-то это поможет, конечно.

Прикол про pthread cancellation. Смотрите, если взять C++20, то нам надо таскать `std::stop_token` во все функции. Это не звучит очень удобно. Поэтому в pthread'ах сказали, что это `thread_local`-переменная. Чтобы с ней взаимодействовать — есть `pthread_testcancel`, которая проверяет их аналог `std::stop_token`'а и делает нечто похожее на бросок исключения. glibc, например, именно что бросает `abi::__forced_unwind`. Его, кстати, нельзя ловить, как помнят читатели конспектов. Если вы его поймаете и зажуёте, вы совсем проиграете. Ещё в POSIX можно сделать `pthread_set_cancelstate`, который позволяет сделать вам аналог `noexcept` (т.е. сделать так, чтобы какой-то кусок кода не реагировал на прерывание). В C++, кстати, для этого есть конструктор `std::stop_token` по умолчанию. В POSIX, кстати, что удобно, так это то, что все системные операции там прерываемые. Это, с одной стороны, замечательно, а с другой — а насколько это поддерживается? И этому есть причина. Поскольку прерывание `pthread`'ов работает как будто у вас исключения в C, кучка программистов на C сидят без знаний о гарантиях безопасности, RAII и всего остального такого. Разумеется, им не нравятся исключения. Потому что, блин, у них даже деструкторов нет нормальных, вам придётся руками их делать через `pthread_cleanup_push` и `pthread_cleanup_pop`.

Асинхронщина. Представьте, что у нас сотня клиентов. Мы не знаем, как они посылают данные. И мы не можем выбрать одного и начать получать данные от него, потому что это блокирующая операция. Если этот ничего не отправляет, значит другие ждут, даже если данные они уже отправили. Вряд ли это то, чего мы хотим.
Можно создать по одному потоку на каждого нашего клиента, но это как-то не то, чего мы хотим. Мы создаём поток, чтобы он ждал. Как-то не очень логично. Да и клиентов у нас может быть очень много, намного больше, чем логичнее будет создавать потоков.
Есть другой вариант — в некоторых местах есть возможность сделать неблокирующее чтение, которое просто вернёт вам что-то, что скажет вам, что данных нет. Только вот что вы будете с этим делать? В цикле спрашивать? Ну, тогда вы выжираете ядро подчистую, а значит и когда вы дойдёте до того, у кого есть данные, куча инструкций (следовательно и времени) пройдёт.
Окей, а как это в операционной системе происходит? Там есть несколько механизмов. Например функция `poll`, которой вы передаёте массив файловых дескрипторов, а она говорит вам, у кого из них есть данные. Это в целом ок, но честно опрашивать все десять, когда данные приходят из двух – трех — не очень. Да ещё и дальше надо самому по массиву проходиться. Проблему с производительностью побороли при помощи `epoll`, там список не снаружи, а внутри, и он сам даёт вам список готовых, так что вы не имеете холостых проходов.
Такая концепция, когда у нас есть набор чуваков, которых мы ждём, и мы периодически опрашиваем, чо там готово, она очень сильно переплетается с GUI. GUI очень много ждёт. Ждёт каждую из возможных кнопок, например.

12 Примеры с использованием QT.

```
#include <QApplication>
#include <QWidget>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    w.show();

    return a.exec();
}
```

Что тут происходит? `QApplication` — глобальная инициализация/деинициализация, надо создать его тогда, когда хотите использовать QT и не использовать QT после его удаления. `QWinget` — любая хрень, любые кнопочки, например, наследуются от него. А вот `a.exec()` — запуск цикла обмена сообщениями. Завершается тогда, когда мы закрываем все окна.
Давайте сделаем что-то ещё:

```
#include <QApplication>
#include <QWidget>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget w;
    QPushButton button(&w);
    button.setGeometry(50, 50, 100, 100);
    button.setText("Hello");
    w.show();

    return a.exec();
}
```

Ура, у нас кнопочка.
Так, ладно. Давайте сделаем что-то полезное. Обычно, например, в тестовых проектах не в `main`'е что-то пишут, а наследуются от, например, `QMainWindow`. Что можно заметить, так это то, что помимо классов пишется XML-ка, в которой вы пишете, какие сущности у вас есть, и вот эта XML-ка уже компилируется в код на C++. Нужно это, чтобы XML-ки всякие дизайнеры делали.
Хорошо, но вот эти XML-ки — не действие. А надо на кнопочки ещё действие привязать. И делается это при помощи таких сигналов, что мы обсуждали ранее. Нам интересно `QAbstractButton::clicked` — сигнал, который кнопка emit'ит, когда кнопку нажимают. Что про это стоит знать, так это то, что сигналы там появились раньше, чем STL. Итак, там ест функция `connect`, куда вы передаёте объект, его сигнал, слотовый объект и саму функцию-слот. То есть традиционно слоты — методы. В QT5 стало возможно передавать лямбду вместо метода.
Ещё в QT можно увидеть кучу сырых указателей. Возникает вопрос: «кто удаляет объекты»? На самом деле каждый QT-шный класс наследуется от `QObject`'а. И в нём можно заметить, что у каждого QT-шного объекта есть родитель, и каждый родитель знает всех всех детей. И родитель сам удаляет своих детей. Если вы свой объект привязали к родителю, то у вас нет никаких проблем с `delete`'ом. Если не привязали, то применимо всё, что мы говорили про RAII.
Кстати. Зачем в `connect` передают `this`? Потому что есть внезапно приёмник или владелец сигнала умирают, `connection` моментально рвётся. Если вы не хотите привязываться к какому-то объекту, то вообще есть перегрузка `connect` и без этого.
Как, кстати, пишутся слоты и сигналы? Ну, вот так:

```
// inside a class
signals:
    void something_changed();
private slots:
    void button_clicked();
```

Эти макросы созданы для того, чтобы специальный QT-штый компилятор прошёлся по этим макросами и нагенерировал кода сбоку. Этот компилятор называется MOC.

Многопоточка. Рассмотрим вот такую https://github.com/sorokin/qfactorпростую программку:

```
#include "main_window.h"
#include "ui_main_window.h"
#include <cassert>
#include <sstream>
#include <QEventLoop>

main_window::main_window(QWidget *parent)
    : QMainWindow(parent)
```

```
    , ui(new Ui::main_window)
{
    ui->setupUi(this);
    connect(ui->input_edit, &QLineEdit::textChanged, this, &main_window::input_changed);
}

main_window::~main_window()
{}

void main_window::input_changed()
{
    bool ok;
    unsigned long long val = ui->input_edit->text().toULongLong(&ok);
    if (!ok || val == 0)
    {
        QPalette p;
        p.setColor(QPalette::Text, Qt::red);
        ui->input_edit->setPalette(p);
        ui->output_label->setText("");
        return;
    }
    else
    {
        QPalette p;
        ui->input_edit->setPalette(p);
    }

    uint64_t uval = static_cast<uint64_t>(val);
    ui->output_label->setText(format_output(uval, factor(uval)));
}

std::vector<uint64_t> main_window::factor(uint64_t val)
{
    assert(val >= 1);
    std::vector<uint64_t> result;

    while ((val % 2) == 0)
    {
        QApplication::processEvents(QEventLoop::AllEvents);
        result.push_back(2);
        val /= 2;
    }

    uint64_t current_divisor = 3;

    for (;;)
    {
        QApplication::processEvents(QEventLoop::AllEvents);
        if (val == 1)
            break;

        if ((val % current_divisor) == 0)
        {
            result.push_back(current_divisor);
```

```
                val /= current_divisor;
            }
            else
                current_divisor += 2;
        }

        if (result.empty())
        {
            assert(val == 1);
            result.push_back(1);
        }

        return result;
    }

    QString main_window::format_output(uint64_t val, std::vector<uint64_t> const& factors)
    {
        assert(!factors.empty());

        std::stringstream ss;
        ss << val << " =\n";

        for (size_t i = 0; i != factors.size(); ++i)
        {
            ss << "\t";
            ss << factors[i];
            if ((i + 1) != factors.size())
                ss << " ⊠";
            ss << '\n';
        }

        return QString::fromStdString(ss.str());
    }
```

Она факторизует число в текстовой панели, когда оно меняется. Понятно, что если число поменялось на большое, то факторизовать его мы будем долго, и программа будет делать вид, что зависнет.
Что говорят делать, чтобы программа не зависала? Говорят сделать отрисовку в отдельном потоке. Каждый QT-шный объект не только имеет родителя, но ещё и принадлежит определённому потоку (тому же, что его родитель). Важно это потому, что сигналы и слоты работают по-другому, если сигнал и слот находятся в разных потоках. Тогда при emit'е отсылается событие потоку, и это самое событие обрабатывается в цикле обработке сообщений. А вообще в `connect` есть пятый параметр: тип connection'а. Например, `DirectConnection` — то, что происходит обычно, а `QueuedConnection` — когда объекты в разных потоках.

```
    #include "main_window.h"
    #include "ui_main_window.h"
    #include <sstream>

    main_window::main_window(QWidget *parent)
        : QMainWindow(parent)
        , ui(new Ui::main_window)
    {
        ui->setupUi(this);
        worker_obj.moveToThread(&worker_thread);
        connect(ui->input_edit, &QLineEdit::textChanged, this, &main_window::input_changed);
```

```
    connect(&worker_obj, &worker::factoring_finished, this, &main_window::factoring_finished);
    worker_thread.start();
}

main_window::~main_window()
{
    worker_thread.quit();
    worker_thread.wait();
}

void main_window::input_changed()
{
    bool ok;
    long long val = ui->input_edit->text().toLongLong(&ok);
    if (!ok || val <= 0)
    {
        QPalette p;
        p.setColor(QPalette::Text, Qt::red);
        ui->input_edit->setPalette(p);

        val = 0;
    }
    else
    {
        QPalette p;
        ui->input_edit->setPalette(p);
    }

    uint64_t uval = static_cast<uint64_t>(val);
    QMetaObject::invokeMethod(&worker_obj, "factor", Qt::QueuedConnection, Q_ARG(uint64_t, uval)
}

QString main_window::format_output(uint64_t val, std::vector<uint64_t> const& factors)
{
    assert(!factors.empty());

    std::stringstream ss;
    ss << val << " =\n";

    for (size_t i = 0; i != factors.size(); ++i)
    {
        ss << "\t";
        ss << factors[i];
        if ((i + 1) != factors.size())
        ss << " ⊠";
        ss << '\n';
    }

    return QString::fromStdString(ss.str());
}

void main_window::factoring_finished(factoring_result result)
{
    if (!result.success)
```

```
        {
            ui->output_label->setText("");
            return;
        }
        ui->output_label->setText(format_output(result.input_val, result.factors));
    }
```

Это штука замечательна всем, кроме того, что для больших чисел она не печатает результат. Что логично, оно тупо долго считается. Но есть проблема: мы меняем число. Дописываем к нему 2 цифры. Он уже на первой начинает вычисляться (хотя оно не надо), а вычисление второго ждёт, пока посчитается ненужное первое. На эту тему и было cancellation.

```
#include "factoring_worker.h"
#include <cassert>

factoring_result::factoring_result(uint64_t input, std::vector<uint64_t> factors, bool incomple
    : input(input)
    , factors(std::move(factors))
    , partial(incomplete)
{}

factoring_worker::factoring_worker()
    : input_version(INPUT_VERSION_QUIT + 1)
    , worker_thread([this] { thread_proc(); })
{}

factoring_worker::~factoring_worker()
{
    input_version = INPUT_VERSION_QUIT;
    input_changed.notify_all();
    worker_thread.join();
}

void factoring_worker::set_input(std::optional<uint64_t> val)
{
    {
        std::lock_guard lg(m);
        input = val;
        ++input_version;
    }
    input_changed.notify_all();
}

std::optional<factoring_result> factoring_worker::get_output() const
{
    std::lock_guard lg(m);
    return output;
}

void factoring_worker::thread_proc()
{
    uint64_t last_input_version = 0;
    for (;;)
    {
        std::optional<uint64_t> input_copy;
```

```
            {
                std::unique_lock lg(m);
                input_changed.wait(lg, [&]
                {
                    return input_version != last_input_version;
                });

                last_input_version = input_version;
                if (last_input_version == INPUT_VERSION_QUIT)
                    break;

                input_copy = input;
            }

            std::optional<factoring_result> result;
            if (input_copy)
                factor(last_input_version, *input_copy);
            else
                store_result(std::nullopt);
        }
    }

    void factoring_worker::factor(uint64_t last_input_version, uint64_t val)
    {
        assert(val >= 1);
        uint64_t const initial_val = val;
        std::vector<uint64_t> factors;
        store_result(factoring_result(initial_val, factors, true));

        while ((val % 2) == 0)
        {
            if (last_input_version != input_version)
                return;

            factors.push_back(2);
            store_result(factoring_result(initial_val, factors, true));
            val /= 2;
        }

        uint64_t current_divisor = 3;

        for (;;)
        {
            if (last_input_version != input_version)
                return;

            if (val == 1)
                break;

            if ((val % current_divisor) == 0)
            {
                factors.push_back(current_divisor);
                store_result(factoring_result(initial_val, factors, true));
                val /= current_divisor;
```

```
            }
            else
                current_divisor += 2;
        }

        if (factors.empty())
        {
            assert(val == 1);
            factors.push_back(1);
        }

        store_result(factoring_result(initial_val, factors, false));
    }

    void factoring_worker::store_result(std::optional<factoring_result> const& result)
    {
        std::lock_guard lg(m);
        output = result;

        if (!notify_output_queued)
        {
            QMetaObject::invokeMethod(this, "notify_output");
            notify_output_queued = true;
        }
    }

    void factoring_worker::notify_output()
    {
        {
            std::lock_guard lg(m);
            notify_output_queued = false;
        }
        emit output_changed();
    }
```

Эта программа, кстати, имеет ещё одно преимущество: она показывает, что мы в данный момент досчитываем.
Мораль: от того, что у вас потоки, не значит, что ваша программа не виснет. Но более того, это и в обратную сторону работает: программа может и в одном потоке не виснуть. Всё, что нам надо — сделать обновление быстрым. Всё, что нам надо — чтобы мы регулярно возвращались в цикл обработки сообщений. Как это сделать в 1 потоке? Мы могли бы сделать часть работы и записать в цикл обработки сообщение о том, что нам нужно доработать. А есть другой вариант:

```
    #include "main_window.h"
    #include "ui_main_window.h"
    #include <cassert>
    #include <sstream>

    main_window::main_window(QWidget *parent)
        : QMainWindow(parent)
        , ui(new Ui::main_window)
    {
        ui->setupUi(this);
        connect(ui->input_edit, &QLineEdit::textChanged, this, &main_window::input_changed);
    }
```

```
main_window::~main_window()
{}

void main_window::input_changed()
{
    bool ok;
    long long val = ui->input_edit->text().toLongLong(&ok);
    if (!ok || val <= 0)
    {
        QPalette p;
        p.setColor(QPalette::Text, Qt::red);
        ui->input_edit->setPalette(p);
        val = 0;
        return;
    }
    else
    {
        QPalette p;
        ui->input_edit->setPalette(p);
    }

    bool has_job = current_job.has_value();
    current_job = static_cast<uint64_t>(val);

    if (!has_job)
    {
        try
        {
            factor();
        }
        catch (...)
        {
            current_job = std::nullopt;
            throw;
        }
        current_job = std::nullopt;
    }
}

void main_window::factor()
{
    retry:
    uint64_t const initial_val = *current_job;
    uint64_t val = initial_val;
    if (val == 0)
    {
        ui->output_label->setText("");
        return;
    }

    std::vector<uint64_t> result;
    ui->output_label->setText(format_output(initial_val, result, true));
```

```
        while ((val % 2) == 0)
        {
            QApplication::processEvents(QEventLoop::AllEvents);
            if (initial_val != current_job)
                goto retry;

            result.push_back(2);
            ui->output_label->setText(format_output(initial_val, result, true));
            val /= 2;
        }

        uint64_t current_divisor = 3;

        for (;;)
        {
            QApplication::processEvents(QEventLoop::AllEvents);
            if (initial_val != current_job)
                goto retry;

            if (val == 1)
                break;

            if ((val % current_divisor) == 0)
            {
                result.push_back(current_divisor);
                ui->output_label->setText(format_output(initial_val, result, true));
                val /= current_divisor;
            }
            else
                current_divisor += 2;
        }

        if (result.empty())
        {
            assert(val == 1);
            result.push_back(1);
        }

        ui->output_label->setText(format_output(initial_val, result, false));
    }

    QString main_window::format_output(uint64_t val, std::vector<uint64_t> const& factors, bool par
    {
        assert(!factors.empty() || partial);

        std::stringstream ss;
        ss << val << " =\n";

        for (size_t i = 0; i != factors.size(); ++i)
        {
            ss << "\t";
            ss << factors[i];
            if ((i + 1) != factors.size() || partial)
            ss << " ×";
```

```
            ss << '\n';
        }

        if (partial)
            ss << "\t…\n";

        return QString::fromStdString(ss.str());
    }
```

Тут можно найти `QApplication::processEvents(QEventLoop::AllEvents)`. Это что?
Посмотрим на вот что. У нас могут быть сложенные циклы обработки сообщений. Вот открыли мы диалог «открыть файл». Там запускается свой цикл обработки сообщений, который обрабатывает только её сообщения, но не сообщения основного окна. Но ведь крутить циклы обработки мы можем не только в подобных (модальных) диалогах. Мы можем сказать «а давайте запустим обработку вот в этом месте». Главное, чтобы вы не обрабатывали такое же событие, как мы сами, иначе stack overflow. И ещё важно в этом моменте очень много думать про реентрабельность.

Рисование в окне на Qt (для ДЗ). Если мы хотим нарисовать в окне множество Мандельброта, то мы пока не занем, что делать. А делать вот что: надо переопределить paint event. Paint event — это то, что вызывается, когда содержимое окна надо перерисовать. То есть когда вы изменили его размер, например.

```
void main_window::paintEvent(QPaintEvent* event)
{
    QPainter p(this);
    p.drawLine(QLine(50, 50, 100, 100));
}
```

Конкретно нам надо рисовать попиксельно. Делается это так:

```
void main_window::paintEvent(QPaintEvent* event)
{
    QPainter p(this);
    QImage img(width(), height(), QImage::Format_RGB888);

    for (int y = 0; y < img.height; y++)
        for (int x = 0; x < img.width; x++)
            img.bits[...] = ...;

    p.drawImage(0, 0, img);
}
```

`bits` — массив битов, в котором ещё и дырки между строками могут быть. Размер строки целиком — `img.bytesPerLine()`. То есть заполнение выглядит примерно так:

```
void fill(QImage& img)
{
    uchar* image_start = img.bits();
    for (int y = 0; y < img.height; y++)
    {
        uchar* p = image_start + y * img.bitsPerLine();
        for (int x = 0; x < img.width; x++)
        {
            *p++ = red_component;
            *p++ = green_component;
            *p++ = blue_component;
```

```
            }
        }
    }
```

Очевидно, если честно рисовать мандельброта, нам никакого времени не хватит. Поэтому имеет смысл сначала рисовать его очень пиксельно, а потом уточнять.
Ещё нам понадобится `wheelEvent`, `mousePressEvent`, `mouseMoveEvent` и `mouseReleaseEvent`, чтобы реализовать перетаскивание и масштабирование, а также `resiseEvent`, чтобы масштабирование работало.

13 Модули.

```cpp
// test.ixx
export module test;
export int sum(int a, int b)
{
    return a + b;
}

// main.cpp
#include <iostream>
import test;

int main()
{
    std::cout << sum(2, 3) << std::endl;
}
```

Мотивация. Мы помним, что при работе с заголовочными файлами надо соблюдать осторожность. Например, нам очень надо предотвращение повторного включения. Или ещё мы не можем писать определение функции в заголовочном файле (или можем, но помечая её как `inline`, со всеми вытекающими последствиями). И на крупных проектах текст заголовочного файла проходится в каждом *cpp*-файле. И какое-то время компиляции отжирает на это. Если у нас n cpp-файлов и m заголовочных, то в худшем случае у нас время компиляции равно $O(nm)$. Каждая новая версия стандартной библиотеки увеличивает размер заголовочных файлов STL, а значит увеличивает время компиляции.
Отсутствие такой проблемы — не единственный бонус модулей. Написав какой-то код в заголовочных файлах очень легко получить нарушение ODR:

```cpp
// header.h
#pragma once
#include <cassert>
#include <cstdint>

inline int32_t isqrt(int32_t arg)
{
    assert(arg >= 0);

    // ...
}
```

Уже даже тут можно получить нарушение ODR, если мы скомпилируем несколько файлов отдельно, и в одном включим `assert`'ы, а в другом — нет. Тут нарушение ODR, конечно, относительно безобидное, но UB есть UB. Понятно, что *g++ -flto -Wodr* может нам это найти, но знаете ли, LTO не всегда применимо. А ещё можно получить значительно более жёсткое нарушение ODR, если мы не у себя `assert` будем включать-выключать, а если у нас аналогичная ситуация будет с дебажными версиями контейнеров STL.

Да и вообще: препроцессор — зло.
А ещё подумайте, насколько удобно жить, когда определение и объявление постоянно разделены. Да, иногда это полезно, но всегда ли?..
Ещё в заголовках нельзя писать `using`-директиву (иначе она повлияет на всё, что написано во всех файлах, которые данный заголовок подключают).
Ещё когда у нас есть шаблон, который мы хотим сделать от конкретных параметрах всегда одинаково, нам может захотеться в заголовочном файле подавить инстанцирование, а в cpp-файле явно проинстанцировать. Оно вам надо так заморачиваться?

Visible vs reachable. В целом понятно: пишем `export` в модулях, и хорошо. Давайте обсудим некоторые детали.

```
// test.ixx
export module test;

export struct pair
{
    int x;
    int y;
};

export pair make_pair(int a, int b)
{
    return { a, b };
}
```

Тут всё понятно, у нас есть `pair`, и хорошо. Но вот вопрос: а что если у нас не будет экспортироваться `pair`? Ошибка компиляции? Ну, могло бы быть и так, но заметьте вот какой пример: у нас есть `std::bind`. Он возвращает неизвестно какой функциональный объект. И вы не хотите давать пользователю самому создавать объекты такого типа, он вам просто не нужен. И тут вы вынуждены прятать его в пространство имён `__detail` или называть как-нибудь типа `_Binder`.
И вообще у нас нередко бывает ситуация, когда мы не можем обратиться к классу по имени: у нас есть лямбды, у нас есть локальные классы, объект которого мы можем вернуть из функции. Бывают, короче, ситуации такие.
Поэтому вы вполне можете не экспортировать структуру `pair`, а экспортировать только `make_pair`.
Надо понимать, что все свойства имени известны. Просто вы не можете по имени на него сослаться. Научно это называется visible и reachable. Visible — когда вы можете сослаться по имени. Reachable — когда вы можете сделать какой-то `decltype` или ещё чего, чтобы добраться до типа.

Интересный факт. Кстати, в именах модулей допускаются точки. Но, что стоит сказать, с точки зрения языка модули `info` и `info.kgeorgiy` не связаны никак.

Поддержка модулей. Заявляется, что модули полностью поддерживаются в MSVC и частично пожжерживаются в GCC и clang. В GCC работают какие-то простые примеры, а комплексные примеры не работают и весь трекер у них завален примерами, которые не работают. В clang'е ещё более интересная ситуация: там были модули ещё до C++20 (потомум что у них один фронтенд для C, C++, Objective-C, CUDA и ещё куча всего) и ещё из Objective-C модули были. Когда появился C++20, стало надо доделать ещё много всего (так как они в C++20 там отличаются от Objective-C), но что конкретно там происходит, не расскажем.

Кто такой модули и как они работают. У нас были мир с заголовками, а сейчас мир с модулями. Что происходит при компиляции модуля? А вот помимо объектного файла создают IFC-файл (так он в MSVC называется), в котором хранится публичный интерфейс модуля. И компиляторы умеют из этих файлов тянуть информацию об объявлениях (соответственно, тянут они только те объявления,

что нужны).
То есть модуль — это обычный cpp-файл, который просто создаёт не только объктник, а ещё и информацию о публичном интерфейсе. При этом у нас ещё и inlin'инг бывает: компилятор может позволить себе в IFC-файл записать само тело функции, чтобы inlin'ить.
Вопрос: правда ли, что модули всегда дают нам более быструю компиляцию, чем заголовки? А вот не факт. В старом мире файлы можно было бы компилировать параллельно. А тут у нас могут быть зависимости, так что такой прикол не работает. Может ли это привести к тому, что программа в заголовками будет компилироваться быстрее? Да, может. Представим себе функцию с очень простым интерфейсом и длинным телом. Тогда с заголовками у нас заголовок содержал бы только объявление, и мы бы параллельно компилировали бы два больших файла. А в случае с модулями параллельность умирает.
Что с этим делать? Ну, во-первых, есть вероятность, что ничего: просто с развитием компиляторов, всё хорошо будет. Правда, тут нужно исправить системы сборки, чтобы они могли в разные времена выдавать IFC и OBJ. Но вообще, если вам очень важна параллельность компиляции, есть...

Разделение модулей на куски. Для каждого модуля надо написать «главный файл» — primary module interface unit. Там надо объявить, что модуль будет экспортировать. А дальше можно написать implementation unit.

```
// test.ixx
export module test;

export int sum(int a, int b);

// test_implementation.ixx
module test;
int sum(int a, int b)
{
    return a + b;
}
```

Что ещё нужно бывает нужно иногда — экспортировать структуру как incomplete-тип:

```
// test.ixx
export module test;

export struct foo;

// test_impl.ixx
struct foo {
    // implementation.
};
```

export import. Ещё про разделение модулей на куски: если мы хотим и публичный интерфейс разделить на куски, что делать? А вот смотрите как можно. Представьте, что в модуле `inner` у нас есть кусок модуля. Тогда мы можем написать такое:

```
// test.ixx
export module test;

export import inner; // добавить в публичный интерфейс всё, что есть в публичном интерфейсе inn
export int sum(int, int);
```

Но вообще так не надо, потому что если у вас была функция в модуле `test`, а вы перенести её в `inner`, то у вас сломается ABI-совместимость, потому что декорированное имя этой функции изменится. Поэтому есть другой механизм

Partition'ы

```
export module test:part1;
```

Подобная штука позволяет вам распилить модуль на несколько кусков и импортировать в одном модуле partition'ы:

```
export modute test;

export import :part1;
```

При этом если у нас есть **implementation partition** (он пишется без `export` перед `module`), его нельзя пере-`export`'ировать, а если есть **interface partition** (который с `export module`), то его именно что необходимо пере-`export`'ировать.

Приватный фрагмент модуля. Помните экспортирование incomplete-типа из модуля? Мы для этого создавали два файла, и в одном писали `export struct foo;` в одном файле, а `struct foo { /* ... */ };` в другом. Но можно и не так: можно написать `module :private;`, и всё до конца файла не будет экспортироваться совсем никак. Соответственно, вам поможет что-то такое

```
export struct foo;

module :private;

struct foo
{
    // implementation.
};
```

Transition. Global module fragment и `import` header unit. Прекрасные модули будущего — это, конечно, хорошо, но хочется же ещё уже имеющиеся кодовые базы переводить на модули. И не хочется при этом останавливать разработку.
Так что нам надо сделать что-то, чтобы разработчики boost'а не писали две копии своей библиотеки: для тех, кто на модулях, и для тех, кто нет.
Значит, смотрите. Использовать модули где попало — на здоровье. Но вот использовать заголовочные файлы там, где уже всё переписано на модули — непонятно как. Мы не можем написать

```
export module test;

#include <vector>

// sth
```

Это просто не сработает, потому что у нас в каждом модуле будет свой вектор. Это не то, чего мы хотим. Поэтому у нас есть **global module fragment**:

```
module;
#include <vector>

export module test;
export std::vector<int> foo();
```

Так можно было бы создать модульную обёртку стандартной библиотеки.
В global module fragment'е можно писать только препроцессорные директивы, если что.
Второе, что есть для переноса — header unit'ы. Вы можете сказать, что у нас есть заголовочный файл и вы хотите импортировать его как модуль:

```
import <vector>;
```

При этом если вы хотите сделать так со своими файлами, вы тоже можете так написать, правда у MSVC надо в настройках проекта пометить свой файл как header unit. При этом у нас нет никаких проблем с ODR или ещё чем, если мы делаем `import` нескольких header unit'ов, в которых внутри где-то определено одно и то же.

Модули в стандартной библиотеке. В C++23, кстати, даже стандартная библиотека начала пилиться на модули, все ругались на тему, как же её распилить и в итоге решили сделать очень просто — вся стандартная библиотека один модуль — `std`. Попутно они решили проблему, что в C++ на самом деле нет `size_t`, а есть только `std::size_t`, но все это игнорируют. Это решено так: есть модуль `std`, а есть модуль `std.compat`, который выдаёт не только имена через `std::`, но и C-шные `typedef`'ы.