extra6. Класс std::variant и идея его реализации. Каким образом происходит выбор подходящего конструктора при создании variant? Каким образом при уничтожении variant вызывается деструктор нужного типа?

pair, tuple, variant, any, optional

optional - в нем либо лежит T либо ничего не лежит Например используется в функциях, которые могут ничего не возвращать, либо сделать поле в классе - которое либо есть либо нет

variant - позволяет хранить чтото из списка - динамически меняется что в нем лежит

any - можно класть что угодно

Проблема работы с union - что все объекты надо создавать самим и уничтожать самим - строки надо явно уничтожать например. Union не могут наследоваться, но могут быть шаблонными.

Но в C++17 сделали адекватную замену - variant.

Шаблон класса **std::variant** представляет собой безопасный union для типов. Экземпляр std::variant в любой момент времени либо имеет значение одного из своих альтернативных типов, либо, в случае ошибки, не имеет значения (такого состояния трудно достичь).

Как и в случае с union, если std::variant содержит значение некоторого объектного типа T, объектное представление T выделяется непосредственно в объектном представлении самого варианта. Варианту не разрешается выделять дополнительную (динамическую) память.

Варианту не разрешается хранить ссылки, массивы или тип void. Пустые варианты также являются некорректными (вместо них можно использовать std::variant<std::monostate>).

Вариант может содержать один и тот же тип более одного раза, а также содержать различные cv-квалифицированные версии одного и того же типа.

```cpp
#include <variant>

int main() {
    std::variant<int, double, std::string> v = 1;
    std::cout << std::get<int>(v); // 1
    std::cout << std::get<double>(); // exception
    v = "abc";
    std::cout << std::get<std::string>(v);  // "abc"
    v = 5.0;
    std::cout << std::holds_alternative<double>(v); // true
    std::cout << v.index(); // 1
}
```

```cpp
template <size_t N, typename T, typename... Tail>
struct get_index_by_type {
    static const size_t value N;
};

template <size_t N, typename T, typename Head, typename... Tail>
struct get_index_by_type {
    static const size_t value = std::is_same_v<T, Head> ? N:
            get_index_by_type<N + 1, T, Tails...>::value;

};

template <typename... Types>
```

```
class variant;

template <typename T, typename... Types>
struct VariantAlternative {
    // CRTP
    using Derived = variant<Types...>;

    VariantAlternative(const T& value) {
        static_cast<Derived*>(this).storage.put<sizeof...(Types)>(value);
    }

    VariantAlternative(T&. value) {
        static_cast<Derived*>(this).storage.put<sizeof...(Types)>(std::move(
    value));
    }

    void destroy() {
        auto this_ptr = static_cast<Derived*>(this);
        if (get_index_by_type<N, T, Types...?::value == this_ptr->current) {
            // this_ptr->storage.destroy
        }
    }
    // определения надо вынести вниз, после variant
};


template <typename... Types>
class variant: private VariantAlternative<Types, Types...>... {
private:
    template <typename... TTypes>
    union VariadicUnion {};

    temaplte <typename Head, typename... Tails>
    union VariadicUnion<Head, Tail...> {
        Head head;
        VariadicUnion<Tail...> tail;

        tempalte <size_t N, typename T>
        void put(const T& value) {
            if constexpr (N == 0) {
                new (&head) T(value);
            } else {
                tail.put<N - 1>(value);
            }
        }
    };

    VariadicUnion<Types...> storage;
    size_t current = 0;
public:
    using VariantAlternative<Types, Types...>::VariantAlternative...;

    /*
    template <typename T>
    variant(const T& value) {
        // static_assert(T is one of types)
        current = get_index_by_type<0, T, Types...>::value;
        storage.put<get_index_by_type<0, T, Types...>::value>(value);
    }
    */
```

```
73
74      size_t index() const {
75          return current;
76      }
77
78      template <typename T>
79      bool holds_alternative() const {
80          return current == get_index_by_type<0, T, Types...>::value;
81      }
82
83      ~variant {
84          (VariantAlternative<Types, Types...>::destroy(), ...);
85      }
86  };
```

Конструктор можно сделать либо через шаблонный конструктор с проверками. Либо можно отнаследвоаться от такой штуки `VariantAlternative<Types, Types...>...` - то есть это N родителей с типами `<T_i, Types>`. При этом сам VariantAlternative - тоже является variant. И когда мы пишем using... то мы подключаем конустркторы родителя а именно конструкторы VariantAlternative и от нужных типов как раз.

Деструктор тоже можно сделать в классе "родителя";

cppreference:

Destructor:

Example:

```
1  #include <variant>
2  #include <cstdio>
3
4  int main()
5  {
6      struct X { ~X() { puts("X::~X();"); } };
7      struct Y { ~Y() { puts("Y::~Y();"); } };
8
9      {
10         puts("entering block #1");
11         std::variant<X,Y> var;
12         puts("leaving block #1");
13     }
14
15     {
16         puts("entering block #2");
17         std::variant<X,Y> var{ std::in_place_index_t<1>{} };
18         // constructs var(Y)
19         puts("leaving block #2");
20     }
21 }
22
23 /*
24 Output:
25 entering block #1
26 leaving block #1
27 X::~X();
28 entering block #2
29 leaving block #2
30 Y::~Y();
31 */
```

If valueless_by_exception() is true, does nothing. Otherwise, destroys the currently contained value.

This destructor is trivial if std::is_trivially_destructible_v<T_i> is true for all T_i in Types.

Constructors:

Constructs a new variant object.

1) Default constructor. Constructs a variant holding the value-initialized value of the first alternative (index() is zero). This constructor is constexpr if and only if the value initialization of the alternative type T_0 would satisfy the requirements for a constexpr function.

2) Copy constructor. If other is not valueless_by_exception, constructs a variant holding the same alternative as other and direct-initializes the contained value with std::get<other.index()>(other). Otherwise, initializes a valueless_by_exception variant.

This constructor is defined as deleted unless std::is_copy_constructible_v<T_i> is true for all T_i in Types....

3) Move constructor. If other is not valueless_by_exception, constructs a variant holding the same alternative as other and direct-initializes the contained value with std::get<other.index()>(std::move(other)). Otherwise, initializes a valueless_by_exception variant.

4) Converting constructor. Constructs a variant holding the alternative type T_j that would be selected by overload resolution for the expression F(std::forward<T>(t)) if there was an overload of imaginary function F(T_i) for every T_i from Types... in scope at the same time.

```
    std::variant<std::string> v("abc"); // OK
    std::variant<std::string, std::string> w("abc"); // ill-formed
    std::variant<std::string, const char*> x("abc");
    // OK, chooses const char*
    std::variant<std::string, bool> y("abc");
    // OK, chooses string; bool is not a candidate
    std::variant<float, long, double> z = 0;
    // OK, holds long
    // float and double are not candidates
```

И ещё много пунктов (см. оригинал)

extra7. Идиома type erasure. Реализация type erasure на примере класса std::any. Основные методы этого класса. Каким образом работают конструкторы этого класса, копирование и перемещение, присваивание?

```
#include <any>

int main() {
    std::any a = 5;
    std::cout << std::any_cast<int>(a); // 5
    std::cout << std::any_cast<double>(a); // Exception

    a.type(); // std::type_info
    std::cout << a.type().name();

    a = "abcde";
    // std::any_cast<std::string>(a);
    std::any_cast<const char*>(a);
}
```

Implementation of std::any

```
class any {
private:
    void* storage;

    struct Base {
        virtual Base* get_copy();
        virtual ~Base() {}
    };

    template <typename T>
    struct Derived: public Base {
        T value;

        Derived(const T& value): value(value) {}

        Base* get_copy() {
            return new Derived<T>(value);
        }
    };

    Base* storage = nullptr;
public:
    template <typename U>
    any(const U& value): storage(new Derived<U>(value)) {}

    ~any() {
        delete storage;
    }

    any(const any& a): storage(a.storage->get_copy()) {}

    tempalte <typename U>
    any& operator=(const U& value) {
        delete storage;
        storage = new Derived<U>(value);
    }
};
```

Проблема возникает в деструкторе - потому что надо разрушить тип предыдущий, но не понятно как достать его тип. Благодаря такому полиморфизму мы можем подменять один тип на другой, и все будет корректно вызываться. Такой приём называется **идиома Type erasure**.

extra8. Идиома type erasure. Реализация type erasure на примере нестандартного Deleter у класса shared_ptr.

Type Erasure: см. выше.

```
private:
    struct DeleterBase {
    virtual void operator()(T*);
        virtual ~DeleterBase() {}
    };

    template <typename U>
    struct DeleterDerived: public DeleterBase {
        U deleter;
```

```
10          DeleterDerived(const U& deleter):deleter(deleter) {}
11          void operator()(T* ptr) override {
12              deleter(ptr);
13          }
14      };
15
16      DeleterBase* deleter = nullptr;
17  public:
18      ~shared_ptr() {
19              --*counter;
20      }
```

У shared_ptr может быть свой deleter - в случаях если мы хотим не удалять а что-то еще сделать например. А Аллокатор нужен на свои нужды, выделять указатели на контрольные блоки.