Особенности языка

const

Переменные

```
//Константный int
const int i = 1;

//Альтернативная запись
int const j = 2;

//Указатель на константный int
const int* pI = &i;

//Константный указатель
int const* const cP = &i;

//Снятие модификатора
const_cast<int&>(i) = 3;

int k = 0;
// Ошибка компиляции, так можно добавить константность
static_cast<const int&>(k) = 1;
```

Функции

```
class A
{
    std::string value = "test";
    int i = 0;

    mutable int j = 0;

    int* p_i; //Предположим что оно проинициализированно

public:

    void f(const int i) {}

    //Ошибка компиляции, сигнатура считается одинаковой
    void f(int) {}

    //Так функции могут быть перегружены
    void f_ref(int& r_i) {}

    //Данная функция будет вызванна от r-value или const int как аргумента
    void f_ref(const int& c_r_i) {}
```

```
    //Возврат константной ссылки
    const string& get_const_ref() { return value; }

    //const функция не даст изменить состояние объекта
    //Из неё могут вызываться только другие const функции
    int const_f() const { return i; }

    //const функции можно перегружать, и они будут выбираться в зависимости от
нашего объекта типа A
    int const_f() { return i };

    //Модификация j разрешена из-за ключевого слова mutable
    void set_const_f() const { j = 1; }


    //Такой вызов разрешён, т.к. сам указатель не меняется
    void update_value() const { *p_i = 0; }

    //Другой метод обойти константность, помимо mutable
    void trick() const
    {
        const_cast<A*>(this)->i = 1;
    }
}
```

#Конструкторы и деструкторы

Базовые

```
class A
{
    std::string str;
    std::vector vec;

public:

    //Конструктор по умолчанию, конструирует str\vec по умолчанию
    A() {}

    //Разрушает str\vec
    ~A() {}

    //Конструктор копирования - может быть сгенерирован, если объект может
копироваться
    //Например наличие ссылки в членах класса не даст ему сгенерироваться
автоматически
    A(const A& other) {}

    //Конструктор перемещения
    A(A&& other) {}
```

```
    //Если требуется только стандартное поведение:
    A() = default;

    //Инициализация в констркторе будет происходить в последовательности членов в классе, т.е. str\vec
    A() : vec({0}), str("hi") {}

};
```

Последовательность

Конструкторы и деструкторы в одном объекте. Конструирование и разрушение из одного скопа. (Принцип один)

Запрещенные

Закрытые конструкторы\деструкторы:

```
class A
{
    //Старый метод запретить
private:
    ~A() {}
    A() {}

public:
    //Современный метод
    ~A() = deleted;
    A() = deleted;
};
```

Так можно запретить конструктор копирования или перемещения. Конструктор можно запретить, например, в синглетоне.

Если конструктор запрещён, то объект можно создавать в статических функциях или классах\функциях friend.

Если деструктор удален есть способ удалить объект:

```
class A
{

public:

    ~A() = deleted;

    void destroy() { delete this; }
```

```
};

Однако такой объект может существовать только в динамической памяти, создать его
на стеке не удастся, т.к. он не способен потом разрушится.


## Виртуальный деструктор

```cpp
class A
{
public:

    virtual ~A() {}

    virtual f() { std::cout << "A::f" << endl; }

};

class B : public A
{
public:
    ~B() {}

    virtual f() { std::cout << "B::f" << endl; }
};


A* a = new B();
//Если не сделать деструктор виртуальным будет вызван только ~A
delete a;

//Есть возможность достичь результата без виртуального деструктора - если создать
объект shared_ptr<B> и вернуть его как shared_ptr<A> он будет разрушен в
деструкторе shared_ptr, но с unique_ptr не выйдет
```

Все классы в STL без виртуального деструктора, потому надо быть осторожным с ними, если возникнет потребность их наследовать.

Исключения в деструкторе

Исключение выброшенное из деструктора наружу может скрашить приложение (terminate). Причём один блок try может обработать только 1 исключение, т.е. если обернуть в try создание двух объектов, и оба выкинут исключения - программа завершится аварийно.

Одно из решений, это обрабатывать исключения внутри деструктора.

Так же может быть хорошей идеей вынести код, который может сгенерировать исключение - в отдельную функцию, и вызвать её явно, в блоке try.
```

Виртуальные функции в конструкторе или деструкторе

```
class A
{
public:

    A() { f(); }
    ~A() { f(); }

    virtual f() { std::cout << "A" << endl; }
};

class B : public A
{
public:

    B() {}

    virtual f() { std::cout << "B" << endl; }
};


//Эта строчка выведет A
B b;

//Эта строчка выведет B
b.f();

//При разрушении B будет выведено тоже A
```

Нужно избегать вызова виртуальных функций, в конструкторе или деструкторе.

Виртуальный конструктор

Возможные ошибки

Для оператора присвоения надо проверять, что объект не присваивается самому себе.

Static intialization fiasco

Проблема возникает при инциализации одной глобальной переменной другой. Этого надо строго избегаить.