Вопросы перед лекцией

Глобальные

- Переменные - плохо т.к. она может постоянно изменяться и в моменте она начнёт жить своей жизнью
- Константы - норм

Constexpr

🎯 Для чего нужен `constexpr`

1. Для оптимизации

Компилятор может заранее вычислить значения и убрать лишний код.

2. Для использования в местах, где нужны константы

```
constexpr int size = 10; int arr[size]; // работает, потому что size известен при компиляции
```

`const`	`constexpr`
Значение нельзя менять, но **может быть известно только во время выполнения**	Значение **обязательно должно быть известно на этапе компиляции**
Более гибкий	Более строгий
`const int x = time(0);` — ОК	`constexpr int x = time(0);` — ошибка

extern

💡 Что такое `extern`?

`extern` — это **спецификатор хранения** в C++, который говорит компилятору:

> “Эта переменная (или функция) **объявлена здесь**, но **определена где-то в другом месте**.”

Или проще:

> “Она существует, но не здесь. Найди её в другом файле.”

📦 Приме

р с функцией:

В `math.cpp`:

```
int add(int a, int b) {     return a + b; }
```

В `main.cpp`:

```
extern int add(int, int); // говорим: "эта функция есть где-то ещё" int main()
{     int result = add(2, 3); }
```

Структуры и юнионы (менее удобно чем в C)

Структура - это просто описание того, что будет существовать при создании объекта струкутуры

C

```
struct a
{
...
}

При объявлении структуры можно только так:
struct A x;

typedef struct A
{

} A;
При объявлении структуры с тайпдефом можно только так:
A x;
```

C++

```
struct a
{
...
}

При объявлении структуры можно так:
A x;

typedef struct A
{

} A;
При объявлении структуры с тайпдефом можно так:
struct A x;
```

Функции

C

```
typedef struct A
{

} A;

int f(const A *x)
{
    return x->a * x->b;
}

A y;
int i = f(&y);
```

`C int i = f(&y);` - получает указатель на A

CPP

```
typedef struct A
{
    int a, b;
    int f()
    {
        return a * b;
    }
```

```
} A;

A y;
int i = y.f();
```

int i = y.f(); - по сути то же само, но:
На уровне компилятора:

```
y.f();          // компилятор сам подставляет: f(&y)
```

```
int f(A* this) { return this->a * this->b; }
```

также можно делать this.a и this.b в фугкции

CPP prototype

```
typedef struct A
{
    int a, b;
    int f();
} A;

int A::f()
{
    return a * b;
}

A y;
int i = y.f();
```

f() const:

```
typedef struct A
{
    int a, b;
    int f() const
    {
        return a * b;
    }
} A;
```

```
A y;
int i = y.f();
```

📌 Итого: зачем писать `f() const`?

Зачем	Объяснение
🔒 Гарантия, что метод не изменит объект	Компилятор проверяет, что `f()` не меняет поля
✅ Совместимость с `const A`	Можно вызывать метод даже на `const` объектах
💡 Семантика	Лучше читается: "это просто геттер, не меняет ничего"

static метод

Не получится обратиться к a и b, в статическом нет this

```
typedef struct A
{
    int a, b;
    Не получится обратиться к a и b, в статическом нет this
    static int g()
    {return 25}
    int f() const
    {
        return a * b;
    }
} A;

A y;
int i = y.f();
int z = y.g();
int x = A::g();
```

📌 Табличка: разница

Вызов	Описание	Требования
`y.g()`	Вызов на объекте	`g()` — обычный метод
`A::g()`	Вызов на типе (классе)	`g()` — **static** метод

Вызов	Описание	Требования
Доступ к полям	Да (`this->a`)	Нет (`this` не существует)
Контекст вызова	Конкретный объект	Общий для всех объектов

Глобальные переменные

```
int m;
int main()
{
    float m; - внутри мейна не будет видно int m
    ::m = 3; - вот так можно вызвать глобальную переменную int m
}
```

Перегрузка методов в структуре - они перегруажаются по входным значениям, а не по выходным

Инкапсуляция

! По умолчанию все поля - public !

Ничего не поменяется

```
typedef struct A
{
    int a, b;
    public:
        static int g()
        {return 25}
        int f() const
        {
            return a * b;
        }
} A;
```

Часть станет невидимой вне структуры (a и b)

```
typedef struct A
{
    private:
        int a, b;
```

```
    public:
        static int g()
        {return 25}
        int f() const
        {
            return a * b;
        }
} A;
```

Класс <=> структура с видимостью полей private

! Нужно явно указывать где будет public

Ничего не поменяется

```
class A
{
    private:
        int a, b;
        static int g()
        {return 25}
        int f() const
        {
            return a * b;
        }
}
```

Методы станут пабликом

```
class A
{
    int a, b;
    public:
        static int g()
        {return 25}
        int f() const
        {
            return a * b;
        }
}
```

Иерархия классов (наследование)

! Можно наследовать структуру от класса и наоборот

Наследование <=> получение всех паблик и протектед полей от отца +

Private наследование:

- Протектед -> прайват
- Паблик - паблик

Public наследование:

- Протектед -> протектед

Паблик - паблик

```
class A
{
    int a, b;
    public:
        static int g()
        {return 25}
        int f() const
        {
            return a * b;
        }
}

class B : A (B - наследник A)
{
    float c;
    public:
        float f2()
            {return a + c}; НЕ СРАБОТАЕТ Т.К. a - private
}
```

Protected

```
class A
{
    protected:
        int a, b;
    public:
        static int g()
        {return 25}
        int f() const
        {
            return a * b;
        }
}

class B : A (B - наследник A)
{
    float c;
    public:
        float f2()
            {return a + c}; НЕ СРАБОТАЕТ Т.К. a - private
}
```

Переопределение

```
class C : A (C - наследник A и в нём переопределена функция f())
{
    float c;
    public:
        float f()
            {return a * b + c};
}
```

Переопределение с использованием старой

```
class D : A (D - наследник A и в нём переопределенная f() вызывает старый f()
и прибавляет c)
{
    float c;
    public:
        float f()
            {return A::f() + c};
}
```

Тип наследования

```
class D : private/public/protected A
{
    float c;
    public:
        float f()
            {return A::f() + c};
}
```

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

```
class D : A, C (D - наследник A и C в нём переопределенная f() вызывает старый
f() и прибавляет c)
{
    float c;
    public:
        float f()
            {return A::f() + c};
}
```

Проблемы:

В D будет 2 экземпляра A => плохо

```
A -> B
|    |
V    V
C -> D
```

РЕШЕНИЕ:

Virtual

В наследовании

`Создаётся дополнительный указатель на 1 экземпляр A и при случае с D будет передан только 1 указатель на A .` Это полезно, удобно, но дольше по времени для обращения к полям A из-за указателя

```
class B : virtual A
{
...
}
```

В классе (структуре)

РАБОТАЕТ

```
class A
{
    private:
        int a, b;
        static int g()
        {return 25}
        virtual int f() const
        {
            return a * b;
        }
}

A a;
B b;
B *pb = &b;
A *pa = &a;
   pa = &b; - тут b - наследник A => в нём есть код A => можно сделать
указатель типа A на b, который является структурой B - наследник структуры A

pa->f(); - вызовется f() из B т.к. ЕСТЬ VIRTUAL. Если его нет - f() вызовется
из A
```

f() вызывается в зависимости от типа объекта, а не от типа указателя

Интуиция:

Virtual

Virtual говорит вызывать самую новую функцию в соответствии с типом объекта

Без Virtual:

без Virtual говорит вызывать функцию в соответствии с типом указателя

Память

Указатель на таблицу виртуальных функций. Вызов функции происходит в соответствии с таблицей виртуальных функций.

Функции не хранятся внутри функции, они хранятся в таблице.

Чисто виртуальная функция - можно создать что-то типа интерфейса из Java. НЕЛЬЗЯ СОЗДАТЬ ОБЪЕКТ С ВИРТУАЛЬНЫМИ ПОЛЯМИ

```
Её невозможно создать, сначала нужно определить f()
class A
{
    protected:
        int a, b;
    public:
        virtual int f() = 0;
}
```

БОНУС нахождения внутри класса

```
class A
{
friend void f(A **);
int a, b;
public:
    int f():
    {
    return a + b;r
    }
}

class B : A (B - наследник A)
{
    float c;
    public:
        float f2()
            {return a + c}; СРАБОТАЕТ, a - private, но f - ЭТО ДРУГ
}
```

Специальные методы классов

Конструктор

! Вызывается при создании объекта

- Имеет аргументы
- Может быть перегружен
- Не наследуется (но внутри реализации наследника можно вызвать конструктор из предка)
- Ни что не может вернуть, даже void
- Конструкторы полей вызываются в соответствии порядка полей ВНУТРИ класса

```
class A
{
int a, b;
public:
    A() {a = 0; b = 0;}
    A(int z) {a = z; b = z;}
}

Вызов:
A x;
A y{4}; - вызов конструктора с аргументами
A z{}; - вызов конструктора без аргументов
```

Всё что может быть прототипом - является прототипом

НЕ БУДЕТ КОМПИЛИТЬСЯ

```
class A
{
int a, b;
public:
    A(int z) {a = z; b = z;}
}

Вызов:
A x; - ОШИБКА, нет конструктора т.к. конструктор требует 1 аргумент
A y{4}; - вызов конструктора с аргументами
```

НЕ БУДЕТ РАБОТАТЬ

```
class A
{
```

```
const int c;
int a, b;
public:
    A(int z) {a = z; b = z;}
}

Вызов:
A y{4}; - вызов конструктора с аргументами, НО константа не инициализирована
```

РЕШЕНИЕ:

```
class A
{
const int c;
int a, b;
public:
    A(int z) : c(z), a(0) {b = 5} - инициализация КОНСТАНТЫ
}

Вызов:
A y{4}; - вызов конструктора с аргументами, РАБОТАЕТ, и запускаются несколько
конструкторов
```

```
class A
{
const int c;
int a, b = 33; - явная инициализация полей в конструкторе <=> неявный вызов
конструктора для данных полей с таким аргументом (также можно переопределить
стокове значение поля в конструкторе)
public:
    A(int z) : c(z), b(0) { } - инициализация КОНСТАНТЫ + переменных
}

Вызов:
A y{4}; - вызов конструктора с аргументами, РАБОТАЕТ, и запускаются несколько
конструкторов
```

Диструктор 💀

🧠 Когда вызывается деструктор

- **Когда объект выходит из области видимости** (например, функция закончилась).
- **Когда объект удаляется через** `delete` (если он был создан через `new`).

- **При завершении программы** — для глобальных и статических объектов.

```
class MyClass {
public:
    ~MyClass() {
        // код для "уборки" объекта
    }
};
```

Виртуальный конструктор и дистуктор

Конструктор - НЕ может быть виртуальным
Деструктор - может (стоит использовать при динамическом выделении памяти)

📌 Когда делать деструктор виртуальным?

Ситуация	Нужно ли делать виртуальным
Есть наследование и удаление через базовый указатель	✅ ОБЯЗАТЕЛЬНО
Класс не используется как базовый	❌ Нет нужды
Интерфейс с чистыми виртуальными методами	✅ Да, это хорошая практика

🖤🖤🖤🖤💚💚💚💚💚💙💙💙💙💙Виктория🖤🖤🖤🖤💚💚💚💚💚💙💙💙💙💙