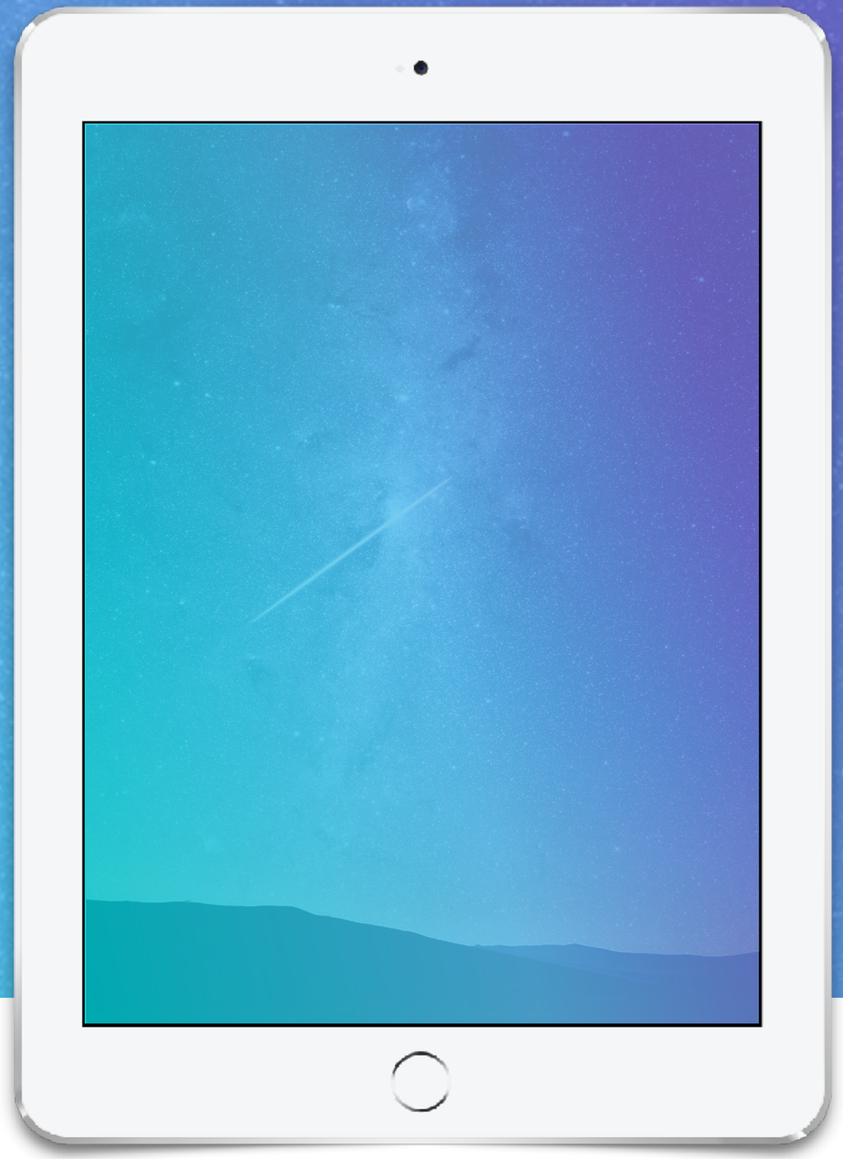
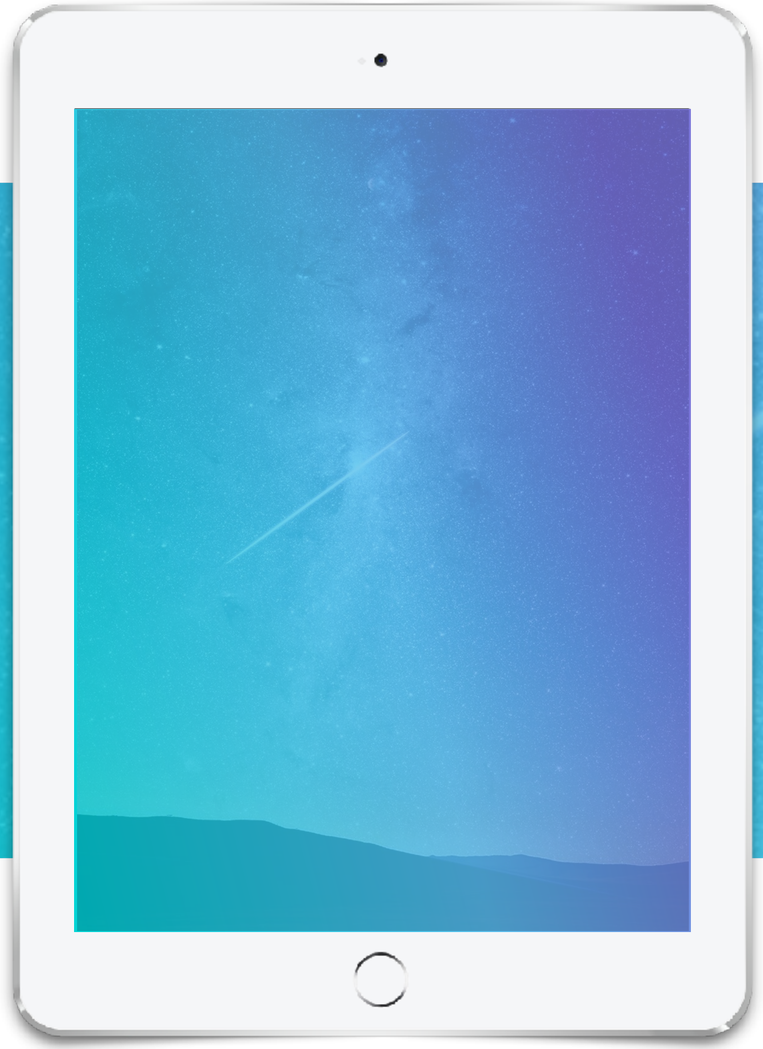
iOS 黑魔法课程

第二课
C语言魔法（上）



StuQueue

一个新的学习方式



本课内容

- 深究面向对象
- C语言实现OOP
- 带来的思考

深究面向对象

深究面向对象

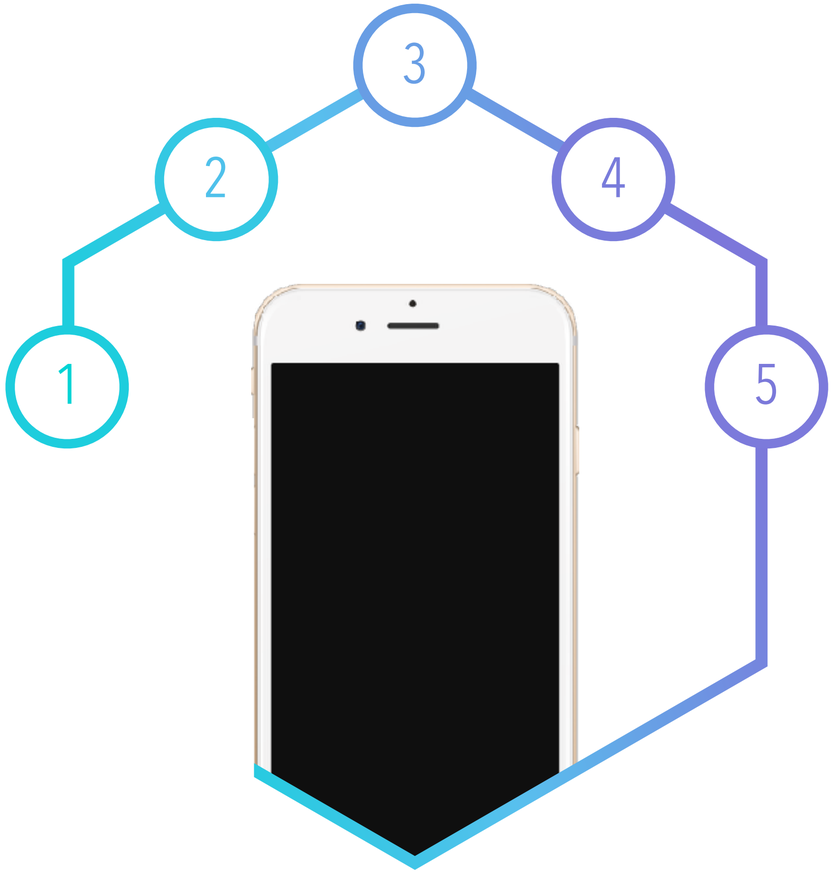
什么是面向对象

- 一种软件开发方法
- 构建业务的方式
- 一种抽象

深究面向对象

五个要素

- 类
- 实例
- 状态（属性集）
- 行为（方法集）
- 关系



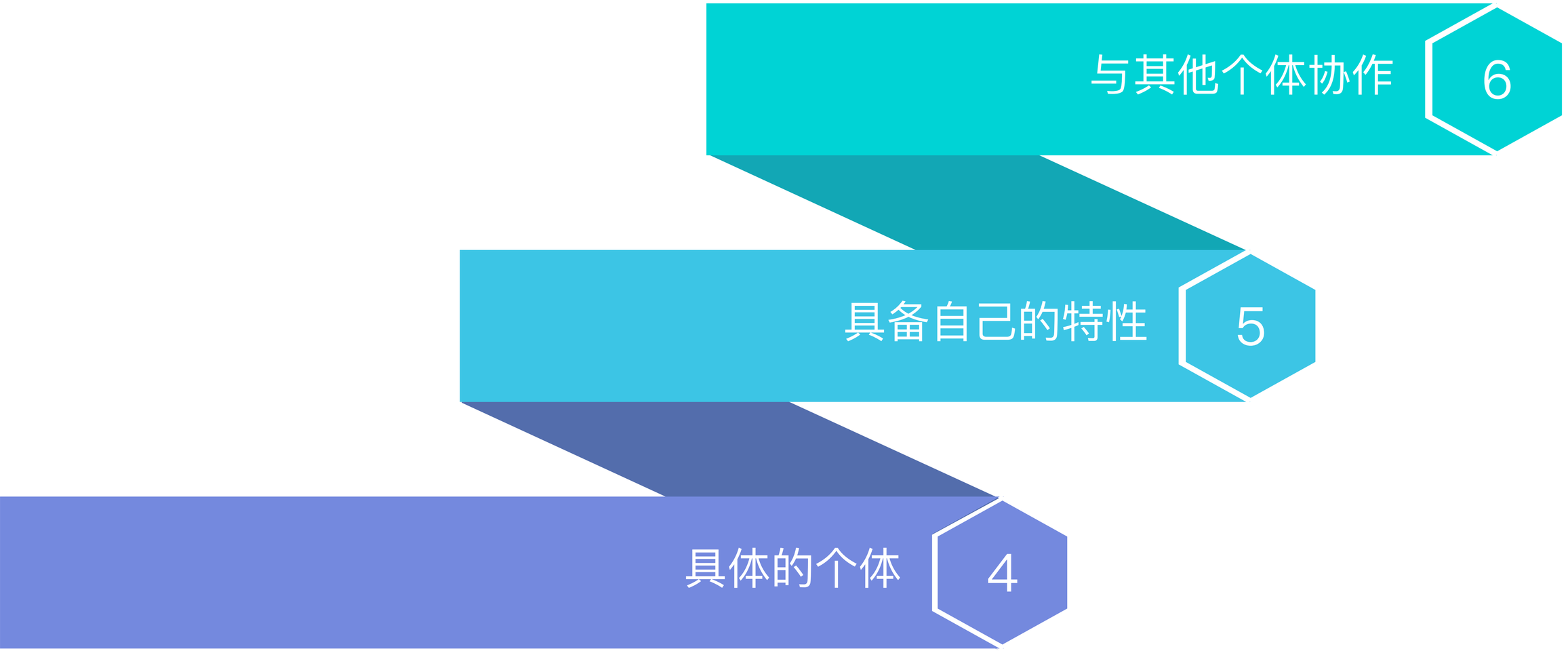
深究面向对象

类

1. 相同物的类别划分
2. 表达一个群体而非个体
3. 实际物和抽象物都可以表达

深究面向对象

对象



深究面向对象

类是否算一种实例?

动物

一只狗

一只鸡

植物

一枝玫瑰

一株郁金香

汽车

一辆宾利

一辆别克

分类

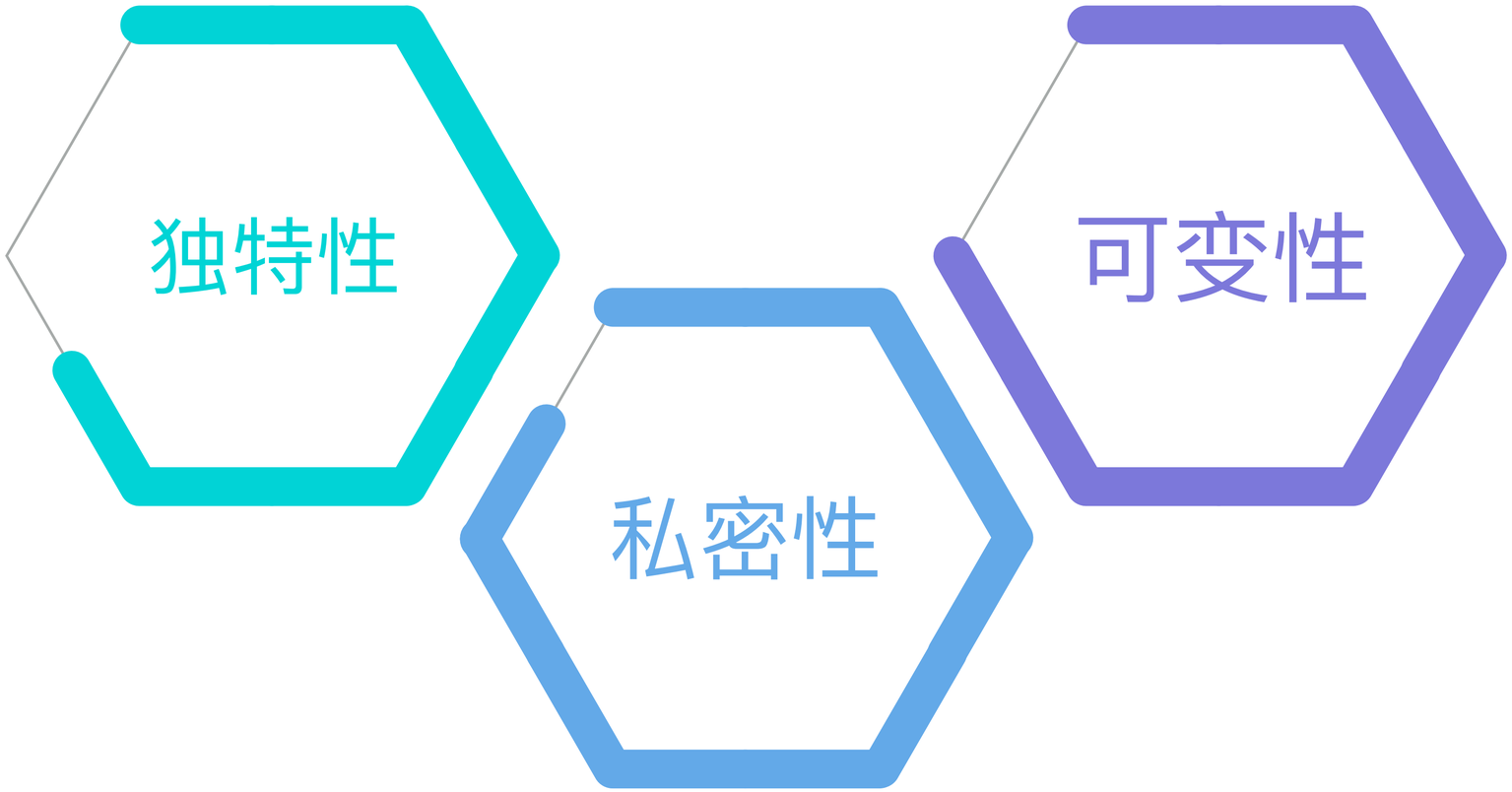
动物

植物

汽车

深究面向对象

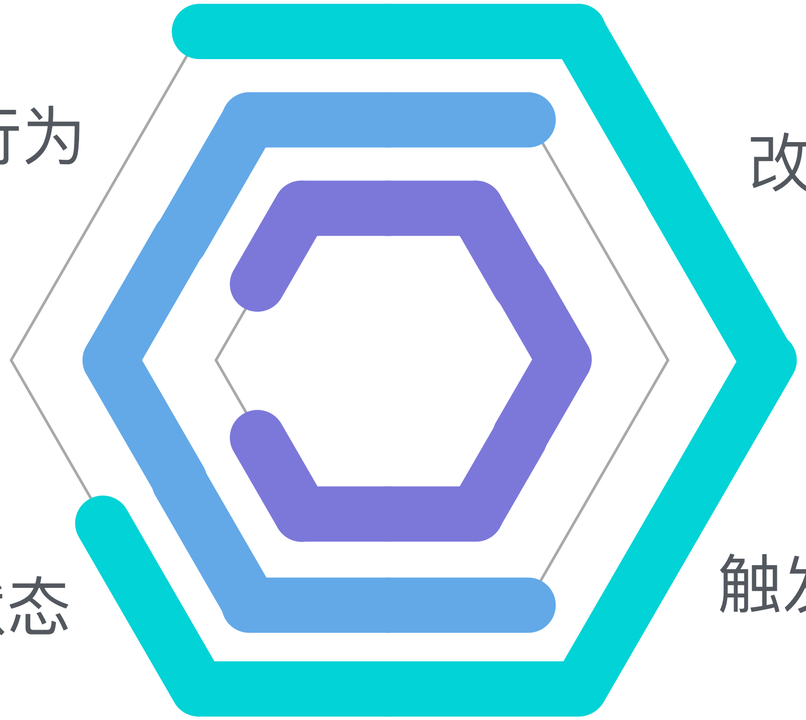
状态



深究面向对象

行为

同一类具备同一行为



改变自己的状态

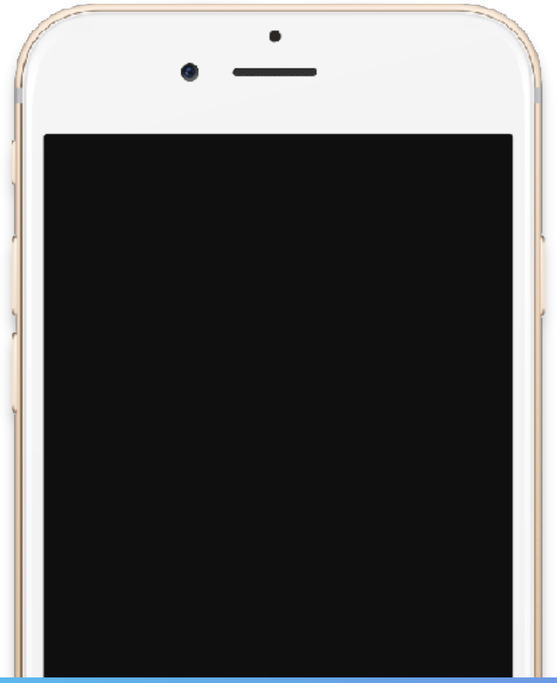
查询自己的状态

触发其他实例的行为

深究面向对象

抽象与具体

抽象



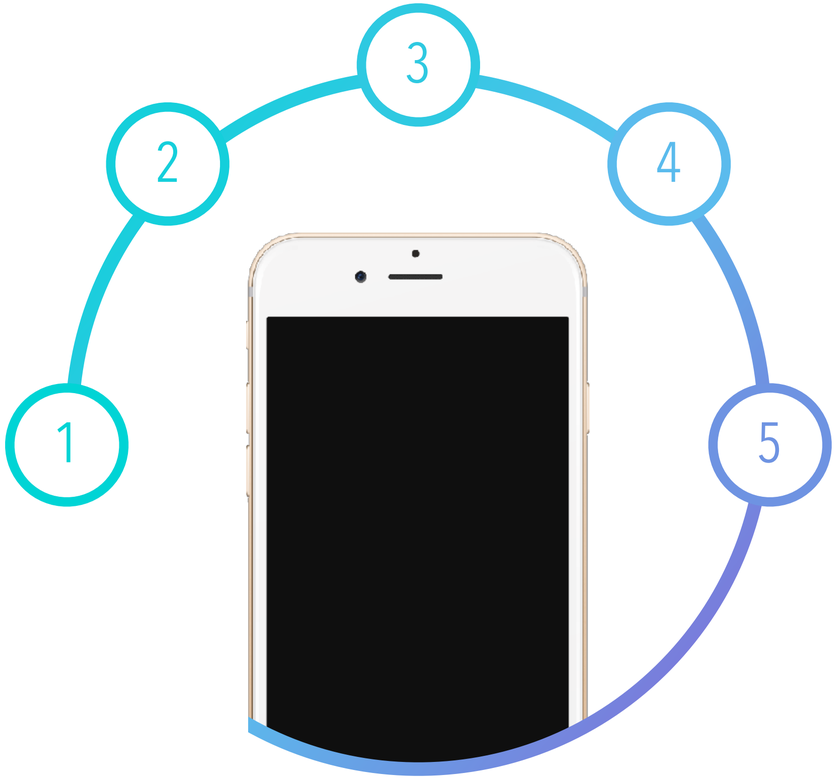
具体

行为　类　关系　实例　状态

深究面向对象

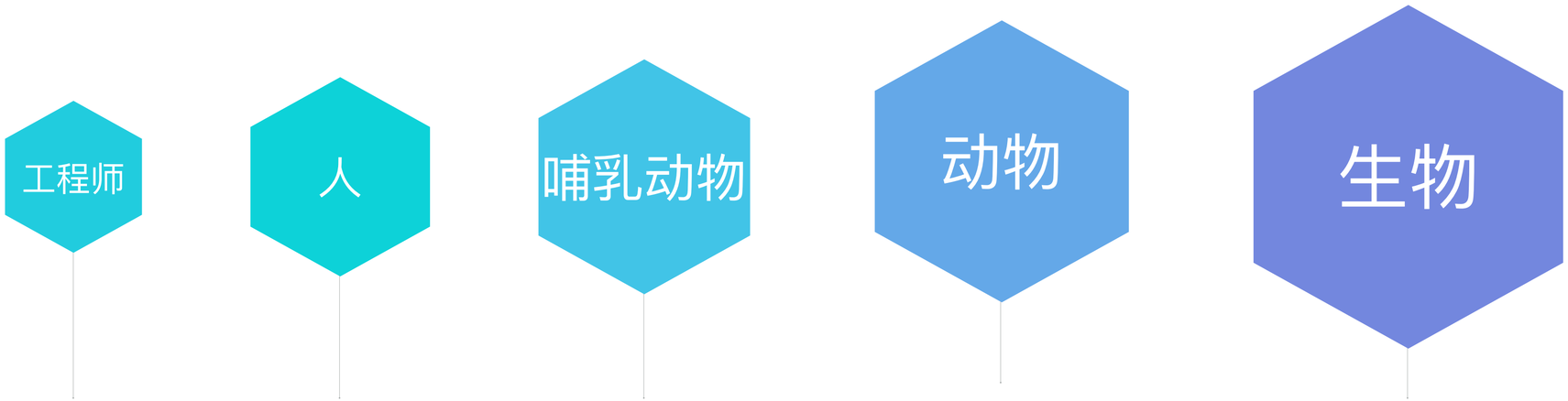
关系

- 类的关系
- 实例的关系
- 行为的关系



深究面向对象

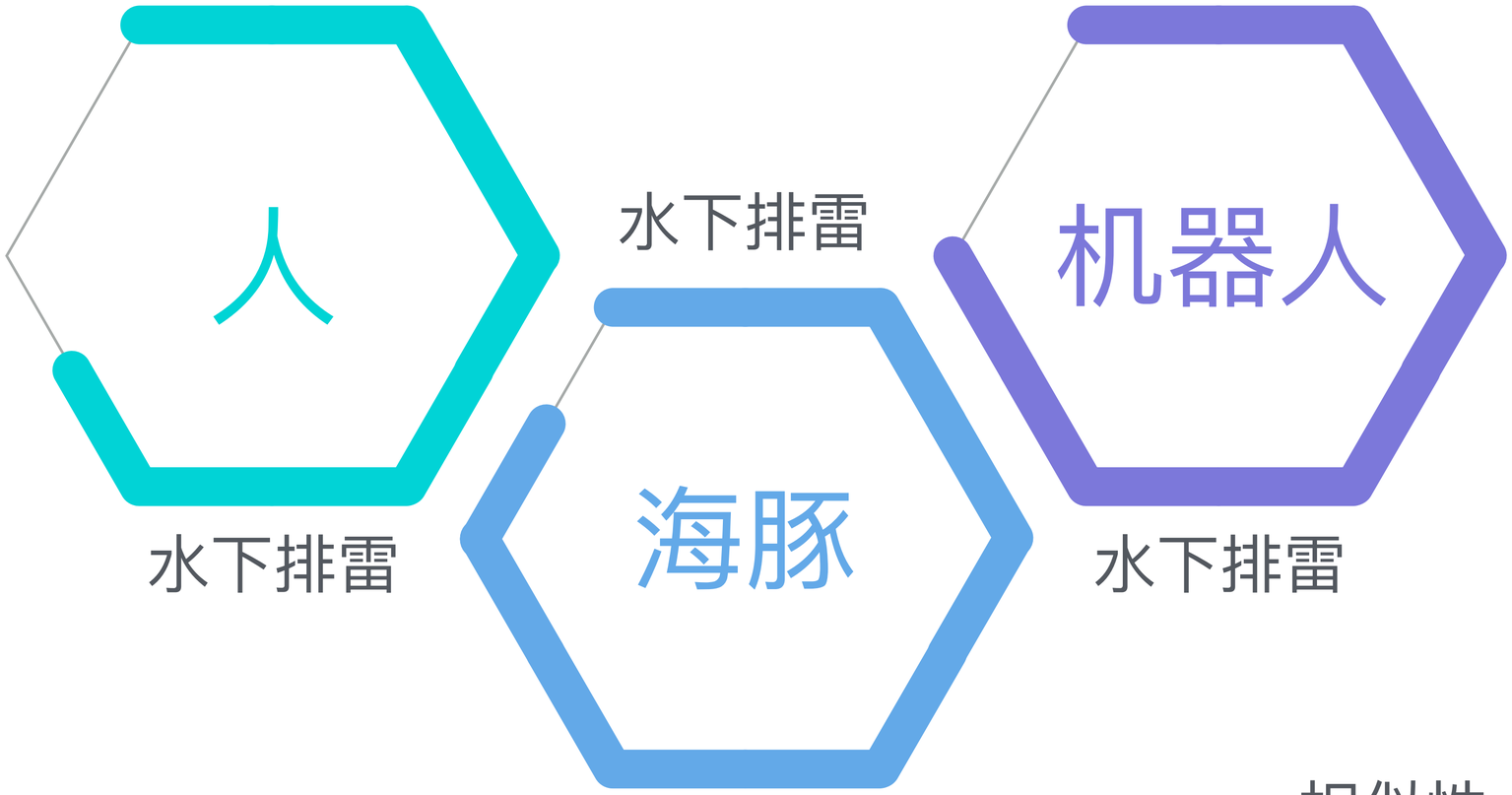
类的关系



派生关系/继承关系

深究面向对象

类的关系



深究面向对象

实例的关系

深究面向对象

行为的关系

群体行为与个体行为

独立行为与协作行为

多态性

重载

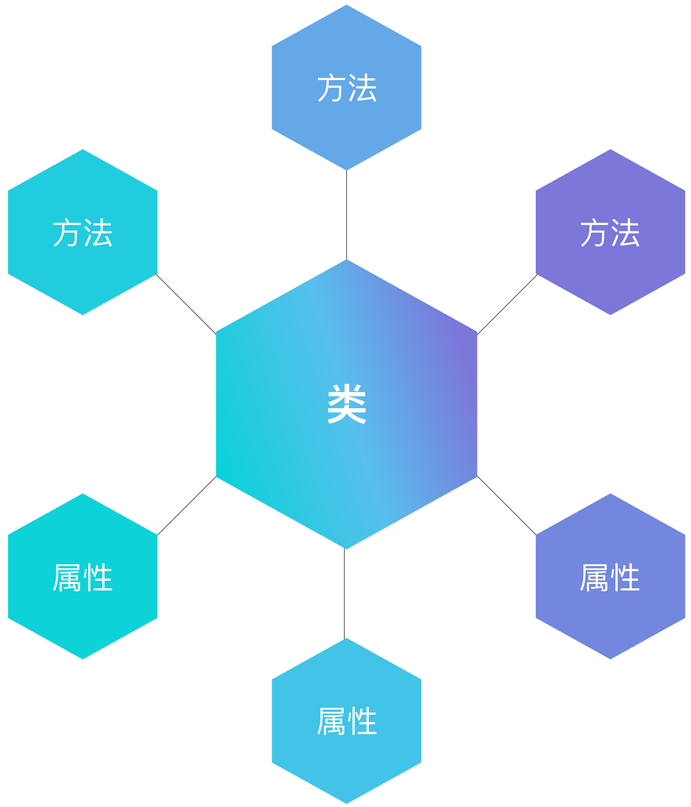
深究面向对象

重要特性

深究面向对象

封装性

- 行为与状态结合形成一体
- 隐藏细节
- 公开接口



深究面向对象

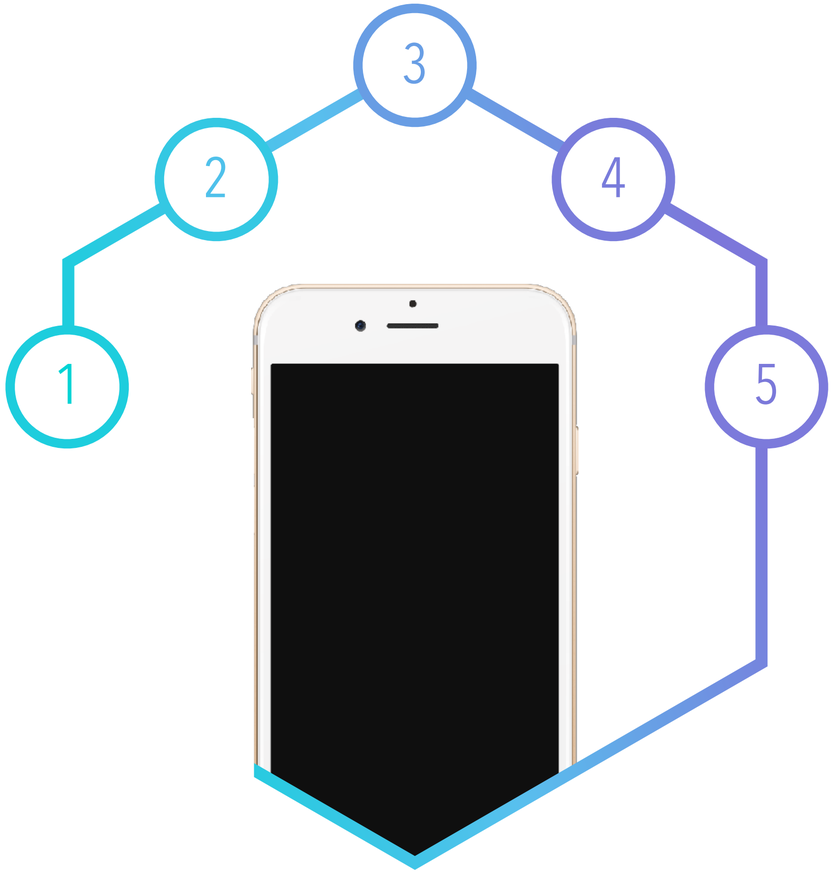
继承性

- 拓展现有封装
- 区别“基于对象”和“面向对象”

深究面向对象

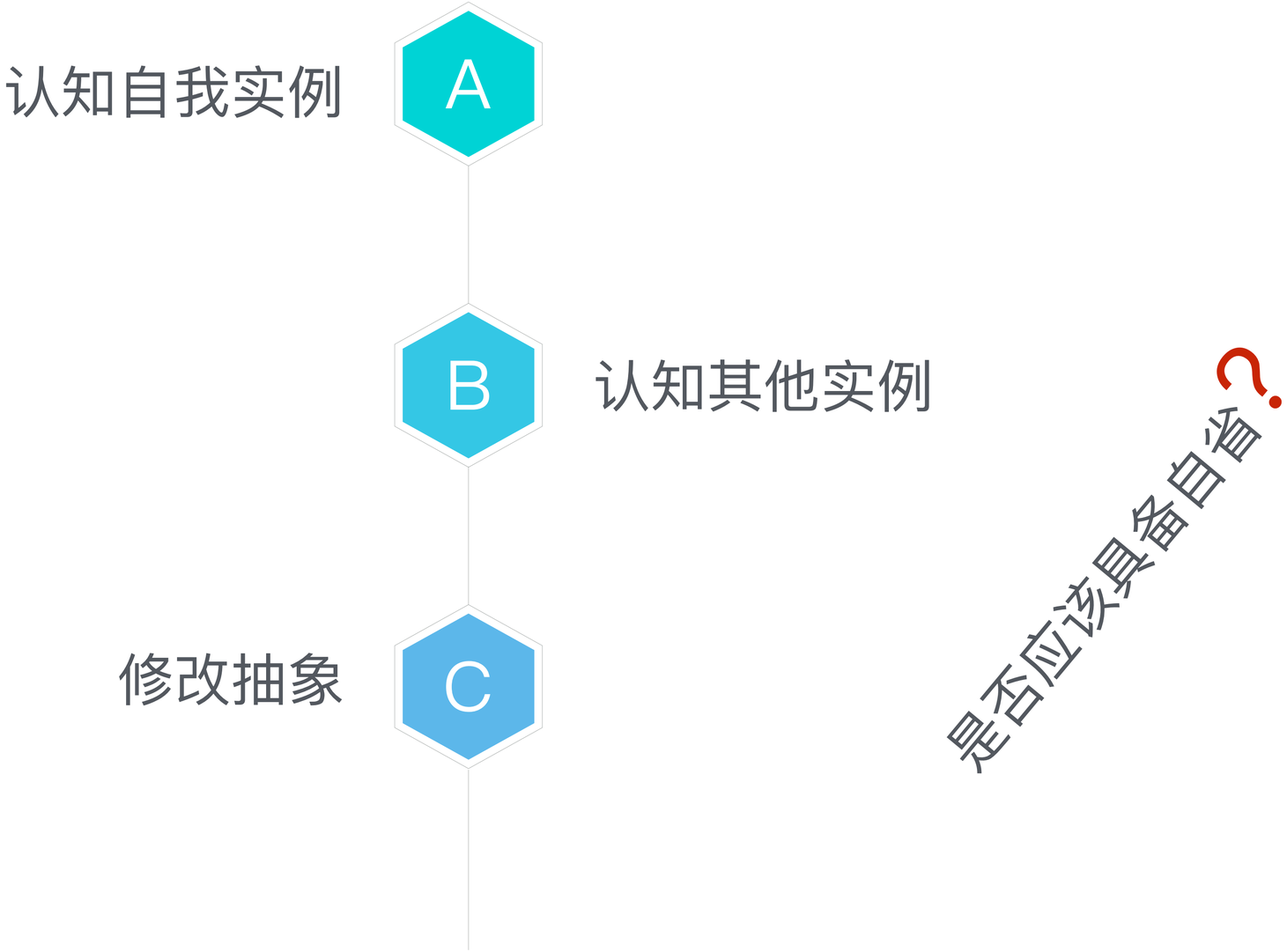
多态性

- 相同行为子类多种实现
- 通过父类抽象访问行为
- 不同子类给与不同响应



深究面向对象

自省

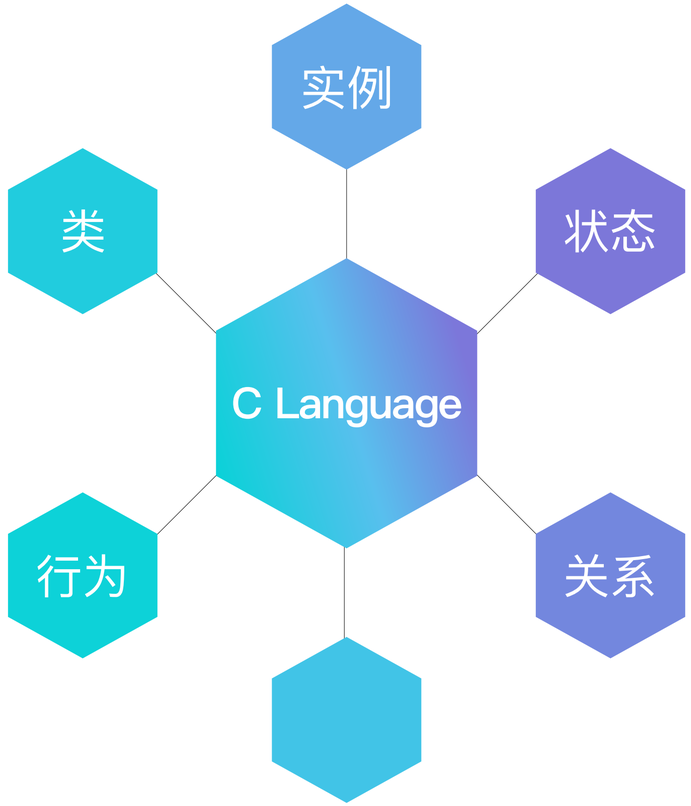


C语言实现面向对象

C语言实现面向对象

五个要素的映射关系

- 类——结构体定义
- 实例——结构体变量
- 状态——结构体的字段
- 行为——函数聚合
- 关系——?



C语言实现面向对象

结构体定义表示类

- 结构体定义—>结构体变量
- 类—>实例
- 抽象—>实体

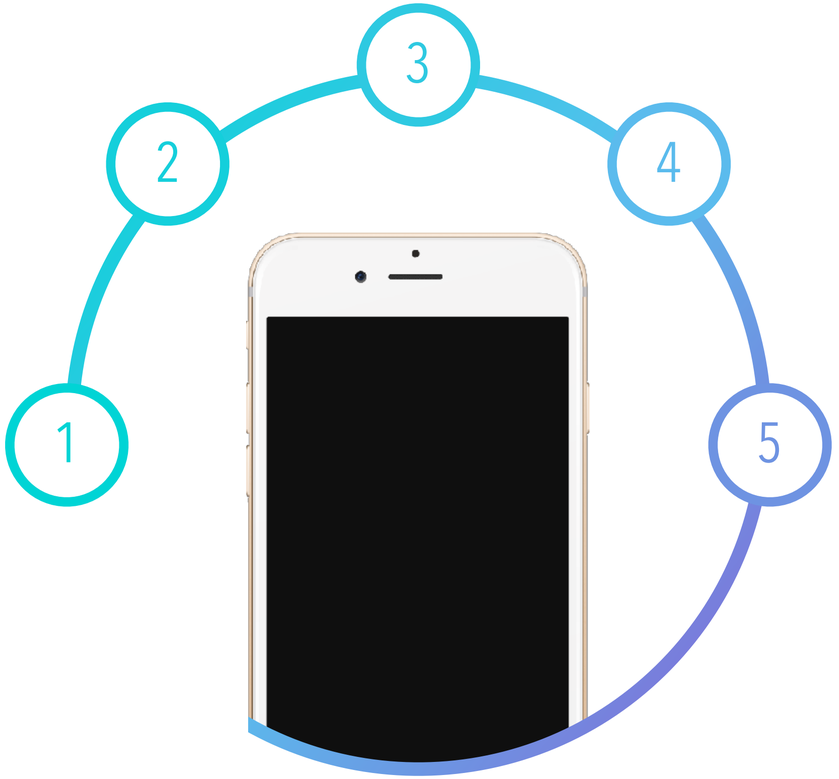
```
typedef struct ClassA {
    int property1;
    float property2;
} ClassA;
```

```
ClassA instanceA = {
    .property1 = 1,
    .property2 = 1.5f
};
```

C语言实现面向对象

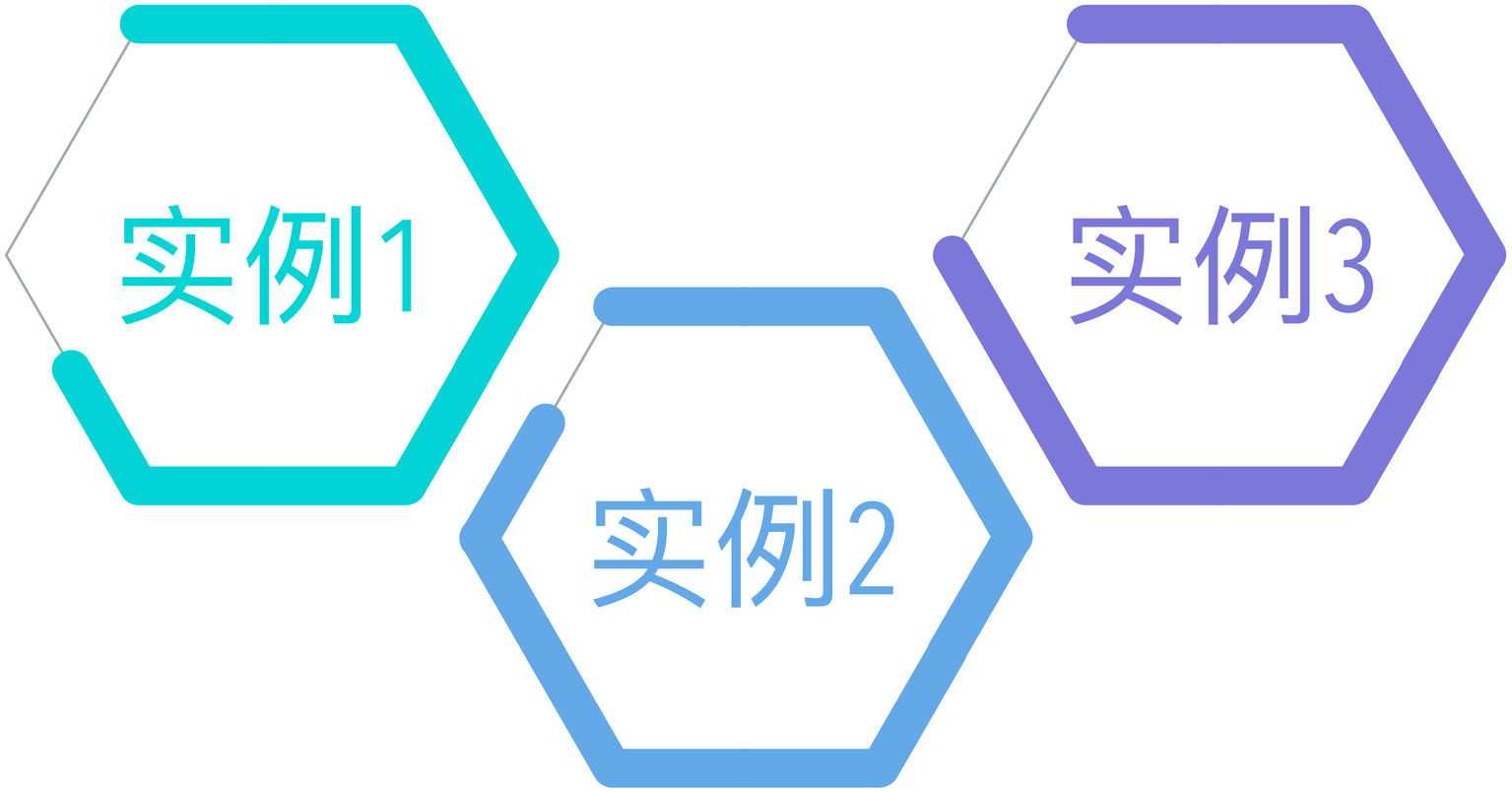
类是否一定是结构体定义?

- 万物皆对象 -> 类也是实例
- 类可以是更高层次的类的实例



C语言实现面向对象

结构体变量表示实例



- 能够表达类型相同
- 能否表达状态不同

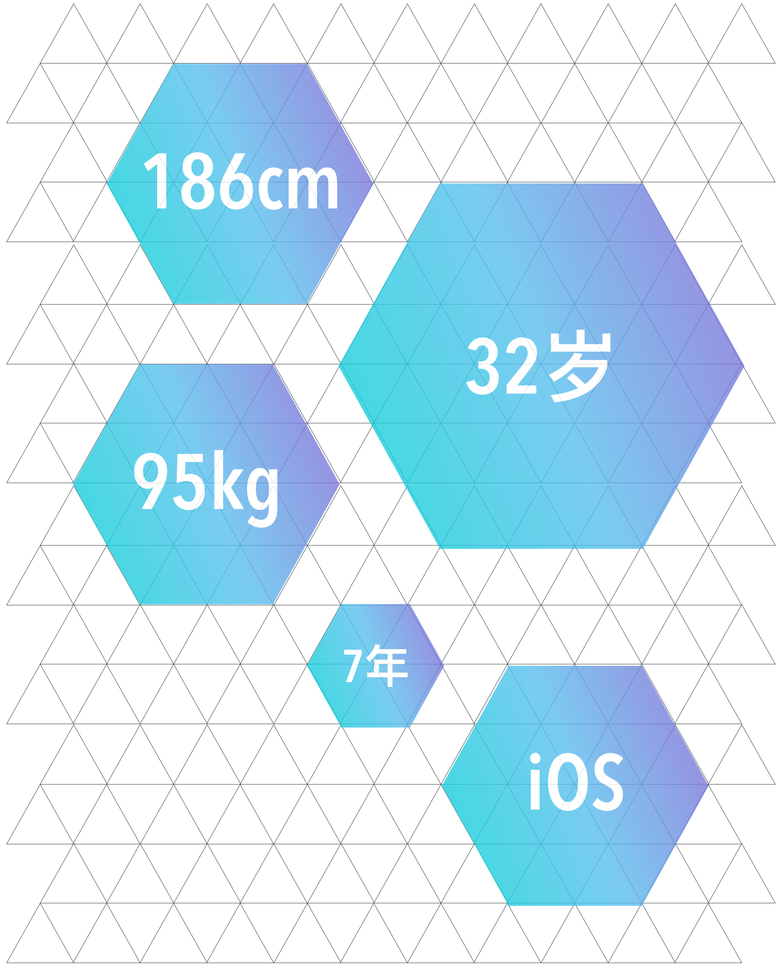
```
ClassA instanceA = {
    .property1 = 1,
    .property2 = 1.5f
};
ClassA instanceB = {
    .property1 = 15,
    .property2 = 2.5f
};
```

C语言实现面向对象

结构体字段表示状态

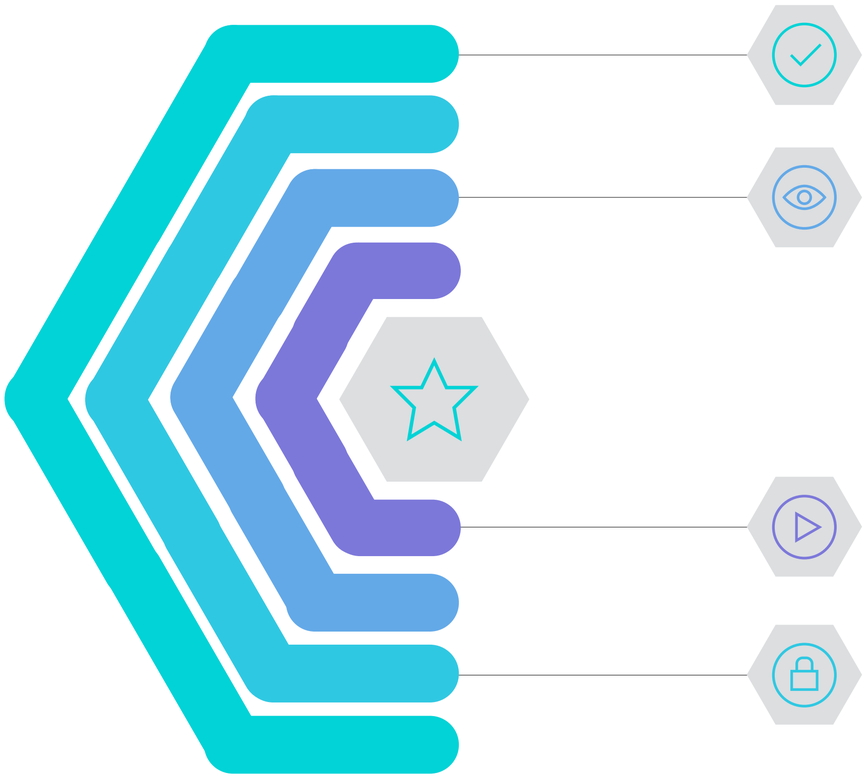
- 能够署名
- 署名不重复
- 可重复设置

```
typedef struct Student {
    int age;
    float height;
    float weight;
    int level;
} Student;
```



C语言实现面向对象

只能用结构体字段来表示状态?



KV存储

连续地址偏移量

结构体字段不能运行时自省

结构体字段不利于隐藏细节

C语言实现面向对象

函数聚合表示行为

点语法

A

```
a_police_officer
    .arrest(a_student);
```

括号语法

B

```
[policeOfficer
            arrest:student];
```

函数形式

C

```
arrest_student(
    a_police_officer,
    a_student);
```

C语言实现面向对象

方法三要素

C语言实现面向对象

方法的集合存在哪里?

1. C++
   散列存放
2. JavaScript
   存于实例
3. Objective-C、Ruby
   存于类

行为是抽象的么？

C语言实现面向对象

C语言散列存放方法

```
void police_arrest(Police_Officer *police, User *somebody);
void police_call_partner(Police_Officer *police, Police_Officer *another);
int police_level(Police_Officer *police);
```

```
void CGContextBeginPath(CGContextRef cg_nullable c);
void CGContextMoveToPoint(CGContextRef cg_nullable c,
    CGFloat x, CGFloat y);
void CGContextAddRects(CGContextRef cg_nullable c,
    const CGRect * __nullable rects, size_t count);
CGPoint CGContextConvertPointToUserSpace(CGContextRef cg_nullable c,
    CGPoint point);
```

C语言实现面向对象

散列存放方法优劣

优势

- 实现简单
- 理解简单

劣势

- 无法体现封装性
- 无法重载

C语言实现面向对象

C语言方法存放于实例

```
typedef void Arrest_Type(struct Police_Man *self,
                         struct Thief *thief);
typedef void Call_Partner_Type(struct Police_Man *self,
                               struct Police_Man *partner);
```

```
typedef struct Police_Man {
    Arrest_Type *arrest;
    Call_Partner_Type *call_partner;
    int police_id;
} Police_Man;
```

C语言实现面向对象

方法存放于实例优劣

优势

- 实例容易替换方法
- 理解简单
- 查询方法简单

劣势

- 对象占用过多内存
- 父类方法不易找到

C语言实现面向对象

C语言方法存放于类

```
typedef void Arrest_Type(struct Police_Man *self,
                         struct Thief *thief);
typedef void Call_Partner_Type(struct Police_Man *self,
                               struct Police_Man *partner);
```

```
typedef struct Police_Man_MTable {
    Arrest_Type *arrest;
    Call_Partner_Type *call_partner;
} Police_Man_MTable;
```

```
typedef struct Police_Man {
    Police_Man_MTable *mtable;
    int police_id;
} Police_Man;
```

```
Police_Man_MTable global_police_man_mtable;
```

C语言实现面向对象

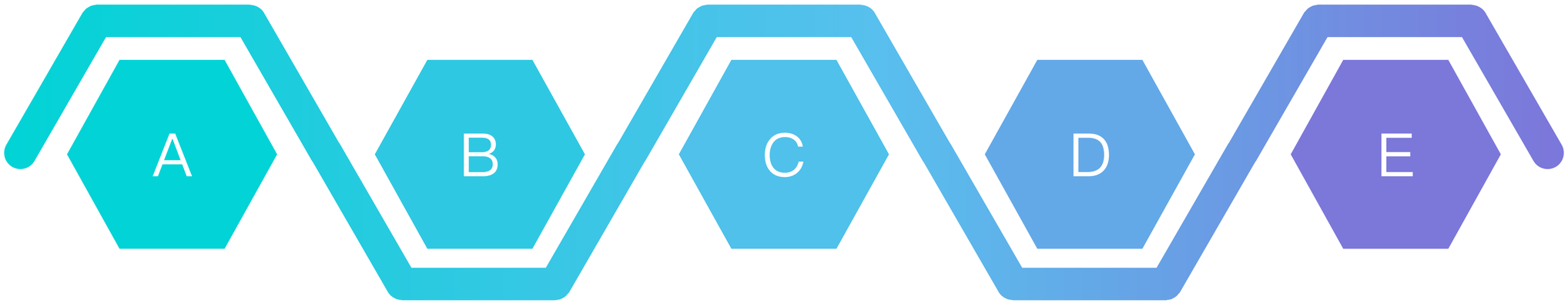
方法存放于类优劣

- 内存更节约
- 便于实现继承链

- 对象和类的模型更复杂

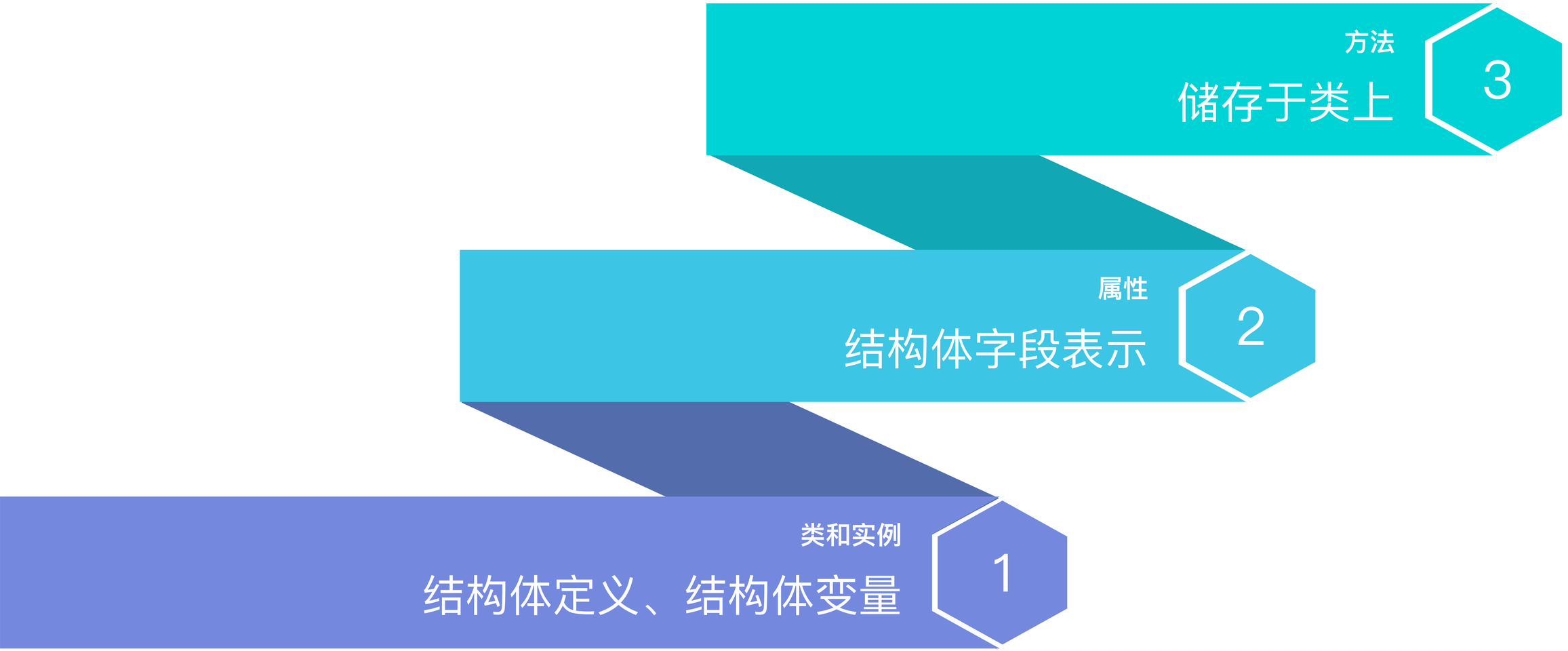
C语言实现面向对象

关系?



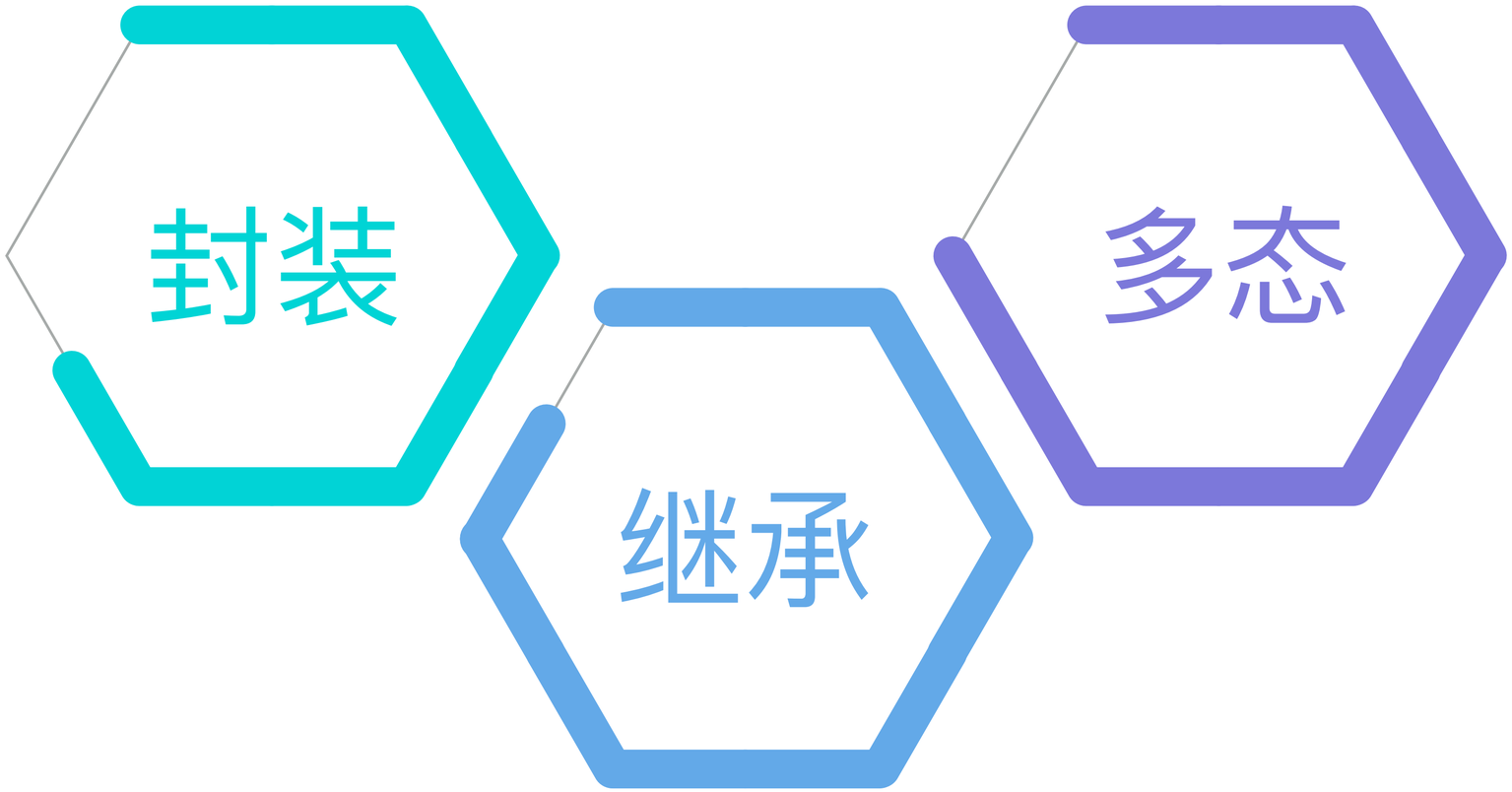
C语言实现面向对象

课上OOP实现方案



C语言实现面向对象

需要实现的三大特性



C语言实现面向对象

实现封装性

- 行为与状态封装到一起
- 通过结构体变量来访问

```
typedef void Arrest_Type(struct Police_Man *self,
                         struct Thief *thief);
typedef void Call_Partner_Type(struct Police_Man *self,
                               struct Police_Man *partner);

typedef struct Police_Man_MTable {
    Arrest_Type *arrest;
    Call_Partner_Type *call_partner;
} Police_Man_MTable;

typedef struct Police_Man {
    Police_Man_MTable *mtable;
    int police_id;
} Police_Man;

Police_Man_MTable global_police_man_mtable;
```

```
police_man->mtable->arrest(police_man, thief);
int id = police_man->police_id;
```

C语言实现面向对象

代码实现

```c
void arrest_imp(struct Police_Man *self,
                struct Thief *thief) {
    printf("arrest a thief!\n");
}
```

```c
void call_parnter_imp(struct Police_Man *self,
                      struct Police_Man *partner) {
    printf("call another police man!\n");
}
```

```c
Police_Man_Mtable global_police_man_mtable = {
    .arrest = arrest_imp,
    .call_partner = call_parnter_imp
};
```

```c
Police_Man *new_police_man() {
    Police_Man *new_obj = (Police_Man *)malloc(sizeof(Police_Man));
    new_obj->mtable = &global_police_man_mtable;
    static int auto_increase_id = 0;
    new_obj->police_id = ++auto_increase_id;
    return new_obj;
}
```

C语言实现面向对象

实现封装性

- 隐藏细节?

```
typedef struct Thief {
    struct Thief_MTable *mtable;
    float money;
    void *private;
} Thief;
```

```
typedef struct Thief_Private_Data {
    int tools_count;
} Thief_Private_Data;
```

```
typedef struct Thief_Private {
    struct Thief_MTable *mtable;
    float money;
    Thief_Private_Data *private;
} Thief_Private;
```

C语言实现面向对象

代码实现

```
Thief *new_thief() {
    Thief_Private *new_obj = (Thief_Private *)malloc(sizeof(Thief_Private));
    new_obj->mtable = &global_thief_mtable;
    new_obj->private = (Thief_Private_Data *)malloc(sizeof(Thief_Private_Data));
    return (Thief *)new_obj;
}
```

```
typedef void Dealloc_Type(struct Thief *self);
typedef int Tools_Count_Type(struct Thief *self);
```

```
typedef struct Thief_MTable {
    Dealloc_Type *dealloc;
    Tools_Count_Type *tools_count;
} Thief_MTable;
```

```
void thief_dealloc(struct Thief_Private *self) {
    free(self->private);
    free(self);
}
```

```
int thief_tools_count(struct Thief_Private *self) {
    return self->private->tools_count;
}
```

```
Thief_MTable global_thief_mtable = {
    .dealloc = (Dealloc_Type *)thief_dealloc,
    .tools_count = (Tools_Count_Type *)thief_tools_count
};
```

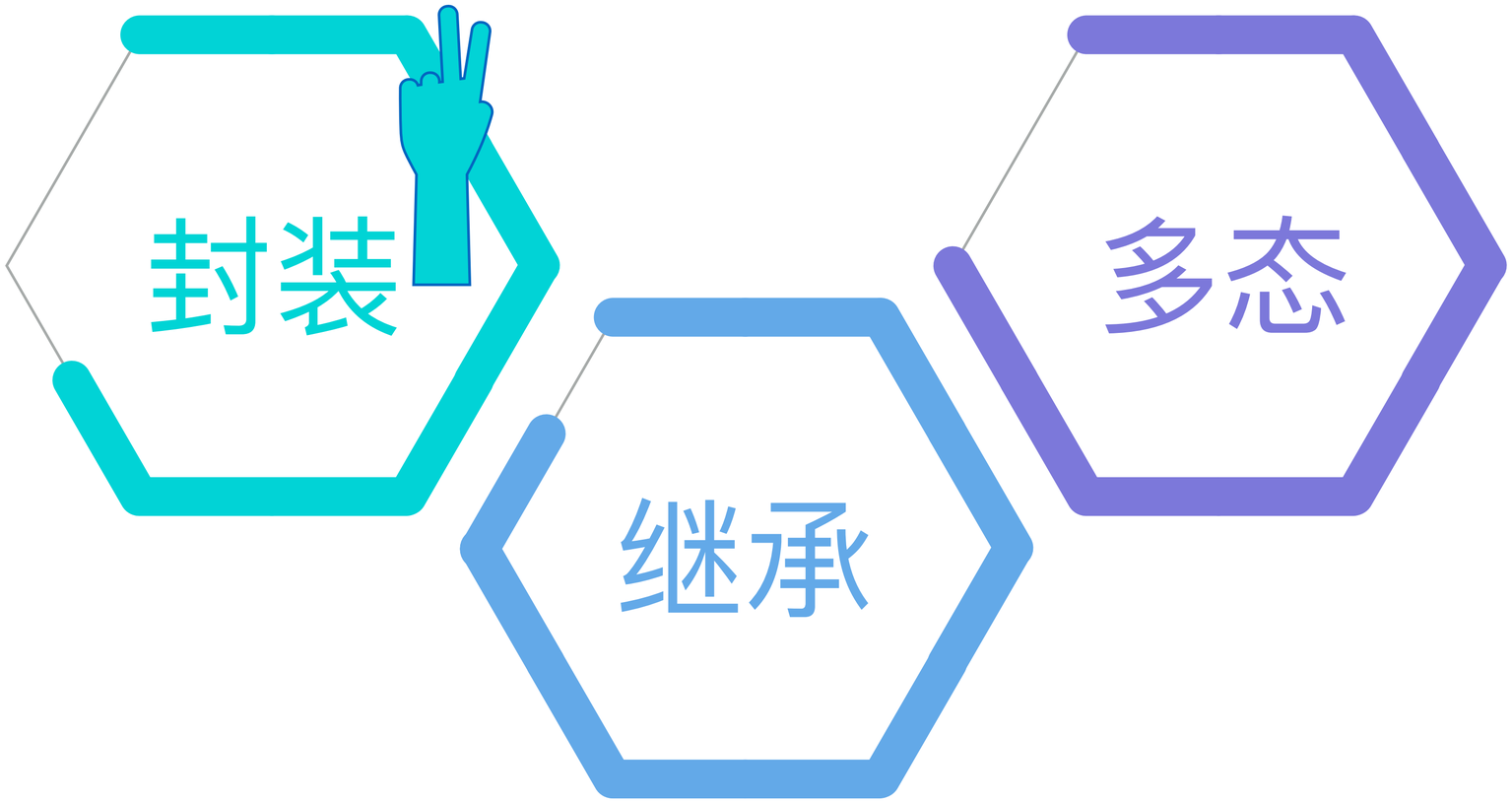
C语言实现面向对象

私有方法?

```
void private_thief_method(struct Thief_Private *self) {
    printf("I have %d tools, it's a secret", self->private->tools_count);
}
```

C语言实现面向对象

需要实现的三大特性



C语言实现面向对象

实现继承性

- 重点&难点
- 子类需要拥有父类的状态

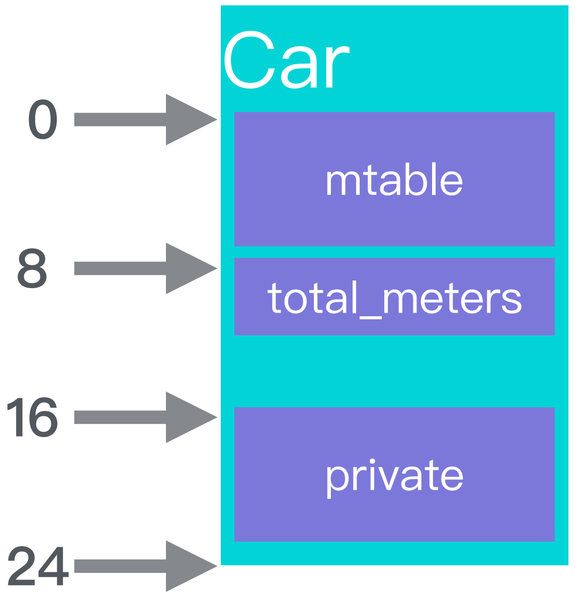
```
struct Car;
typedef void Car_Dealloc_Type(struct Car *self);
typedef void Drive_Type(struct Car *self, int meter);
typedef int DriveTimes_Type(struct Car *self);
typedef const char *Color_Type(struct Car *self);

typedef struct Car_MTable {
    Car_Dealloc_Type *dealloc;
    Drive_Type *drive;
    DriveTimes_Type *drive_times;
    Color_Type *color;
} Car_MTable;

Car_MTable global_car_mtable;

typedef struct Car {
    Car_MTable *mtable;
    int total_meters;
    void *private;
} Car;
```

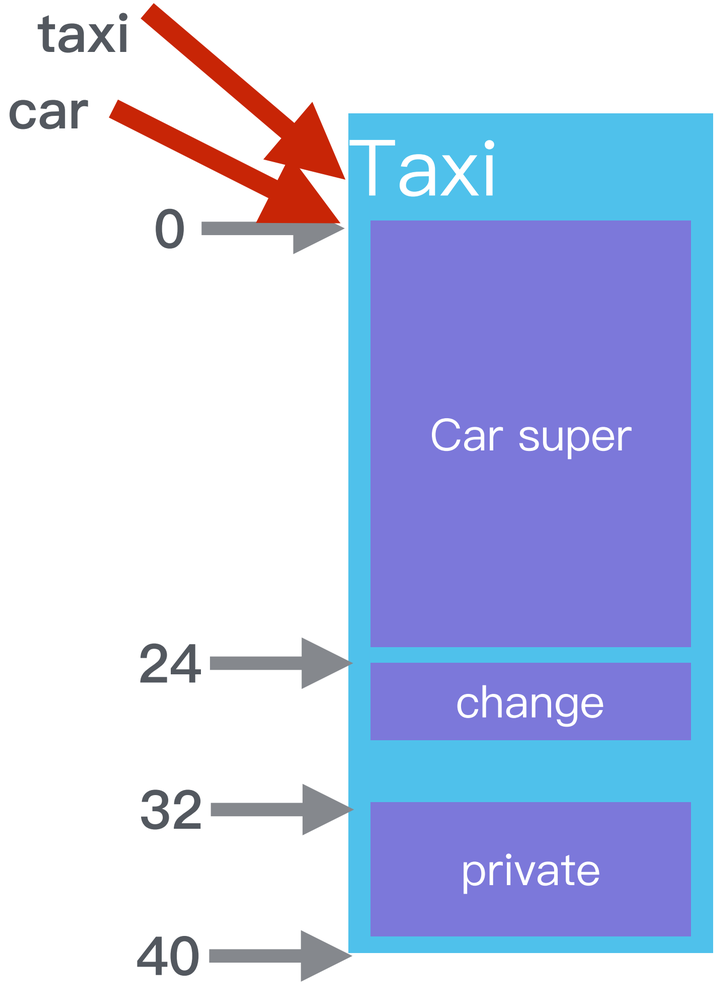
C语言实现面向对象

状态的扩展

```
typedef struct Car {
    Car_MTable *mtable;
    int total_meters;
    void *private;
} Car;
```

```
typedef struct Taxi {
    Car super;
    float change;
    void *private;
} Taxi;
```






```
Taxi *taxi = (Taxi *)malloc(sizeof(Taxi));
Car *car = (Car *)taxi;
```

C语言实现面向对象

行为扩展

- 扩展新的行为

```
typedef struct Car_MTable {
    Car_Dealloc_Type *dealloc;
    Drive_Type *drive;
    DriveTimes_Type *drive_times;
    Color_Type *color;
} Car_MTable;

Car_MTable global_car_mtable;

typedef struct Car {
    Car_MTable *mtable;
    int total_meters;
    void *private;
} Car;
```

C语言实现面向对象

扩展新的行为

```
typedef struct Car_MTable {
    Car_Dealloc_Type *dealloc;
    Drive_Type *drive;
    DriveTimes_Type *drive_times;
    Color_Type *color;
} Car_MTable;

Car_MTable global_car_mtable;
```

```
typedef struct Taxi_MTable {
    Car_MTable super;
    Taxi_Init_Type *init;
    Available_Type *available;
    Pick_Up_Type *pick_up;
    Set_Off_Type *set_off;
} Taxi_MTable;

Taxi_MTable global_taxi_mtable;
```

```
typedef struct Car {
    Car_MTable *mtable;
    int total_meters;
    void *private;
} Car;
```

```
typedef struct Taxi {
    Car super;
    float change;
    void *private;
} Taxi;
```

C语言实现面向对象

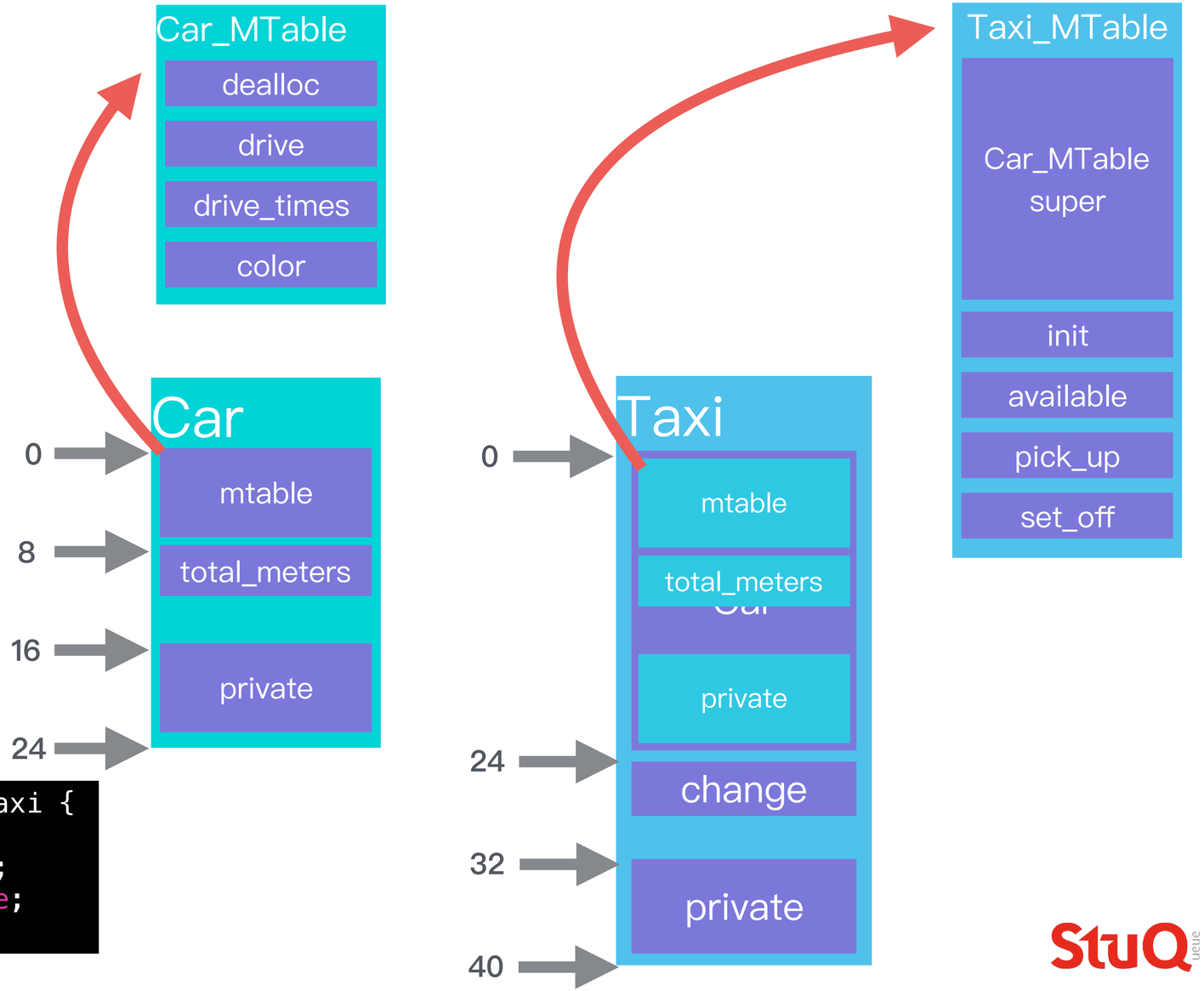
扩展新的行为

```
typedef struct Car_MTable {
    Car_Dealloc_Type *dealloc;
    Drive_Type *drive;
    DriveTimes_Type *drive_times;
    Color_Type *color;
} Car_MTable;

Car_MTable global_car_mtable;
```

```
typedef struct Taxi_MTable {
    Car_MTable super;
    Taxi_Init_Type *init;
    Available_Type *available;
    Pick_Up_Type *pick_up;
    Set_Off_Type *set_off;
} Taxi_MTable;

Taxi_MTable global_taxi_mtable;
```

```
typedef struct Car {
    Car_MTable *mtable;
    int total_meters;
    void *private;
} Car;
```

```
typedef struct Taxi {
    Car super;
    float change;
    void *private;
} Taxi;
```



C语言实现面向对象

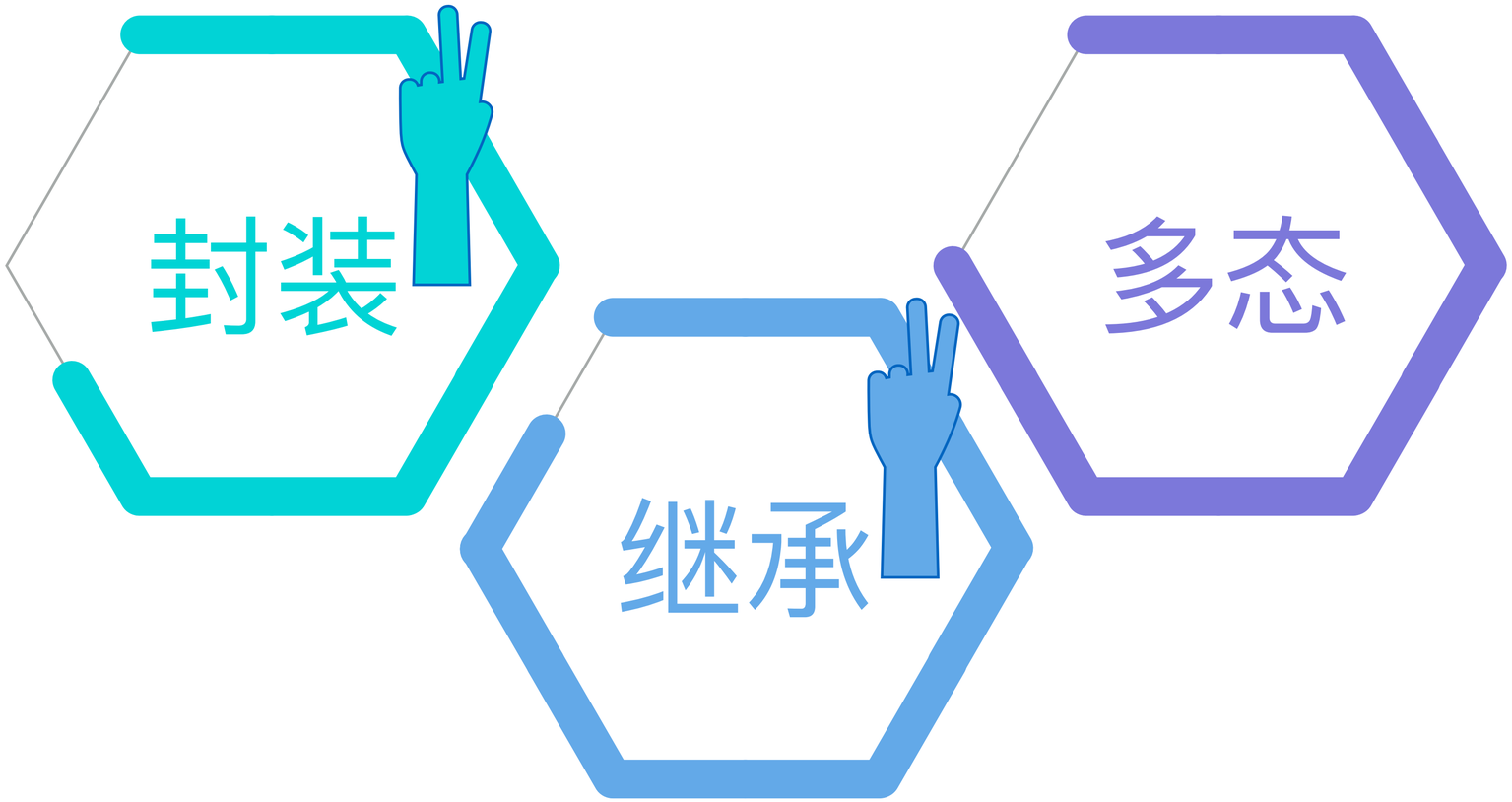
代码实现

```
Taxi *new_taxi() {
    Taxi *new_obj = (Taxi *)malloc(sizeof(Taxi));
    ((Car *)new_obj)->mtable = &global_taxi_mtable;
    return new_obj;
}
```

```
void test2() {
    Taxi *taxi = (Taxi *)malloc(sizeof(Taxi));
    ((Car *)taxi)->mtable->drive((Car *)taxi, 15);
    ((Taxi_MTable *)((Car *)taxi)->mtable)->pick_up(taxi);
}
```

C语言实现面向对象

需要实现的三大特性



C语言实现面向对象

实现多态性

- 方法重载

```
typedef struct Car_MTable {
    Car_Dealloc_Type *dealloc;
    Drive_Type *drive;
    DriveTimes_Type *drive_times;
    Color_Type *color;
} Car_MTable;

Car_MTable global_car_mtable;
```

```
typedef struct Car {
    Car_MTable *mtable;
    int total_meters;
    void *private;
} Car;
```

```
typedef struct Taxi {
    Car super;
    float change;
    void *private;
} Taxi;
```

C语言实现面向对象

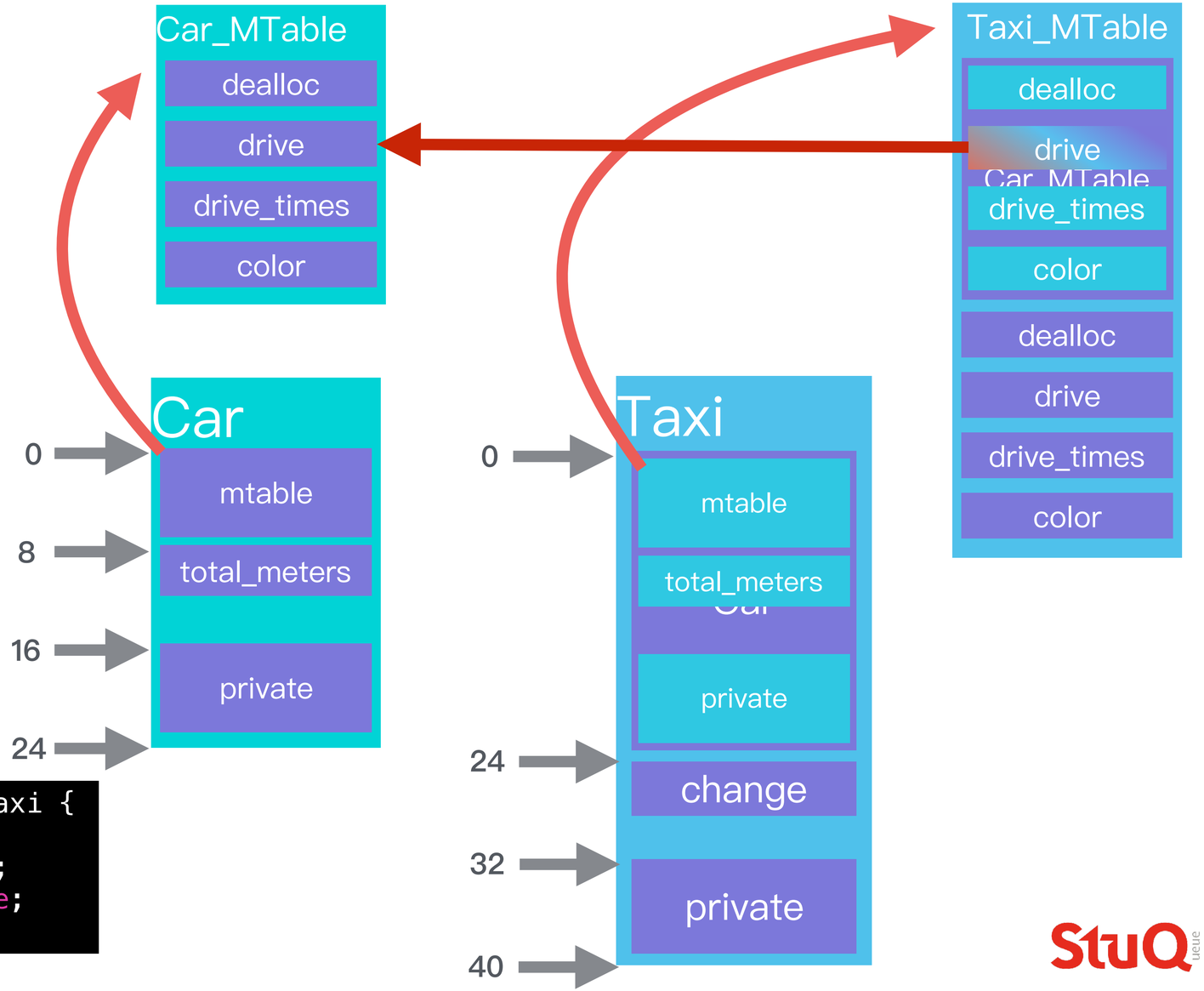
方法重载

```
typedef struct Car_MTable {
    Car_Dealloc_Type *dealloc;
    Drive_Type *drive;
    DriveTimes_Type *drive_times;
    Color_Type *color;
} Car_MTable;

Car_MTable global_car_mtable;
```

```
typedef struct Taxi_MTable {
    Car_MTable super;
    Taxi_Init_Type *init;
    Available_Type *available;
    Pick_Up_Type *pick_up;
    Set_Off_Type *set_off;
} Taxi_MTable;

Taxi_MTable global_taxi_mtable;
```

```
typedef struct Car {
    Car_MTable *mtable;
    int total_meters;
    void *private;
} Car;
```

```
typedef struct Taxi {
    Car super;
    float change;
    void *private;
} Taxi;
```



C语言实现面向对象

代码实现

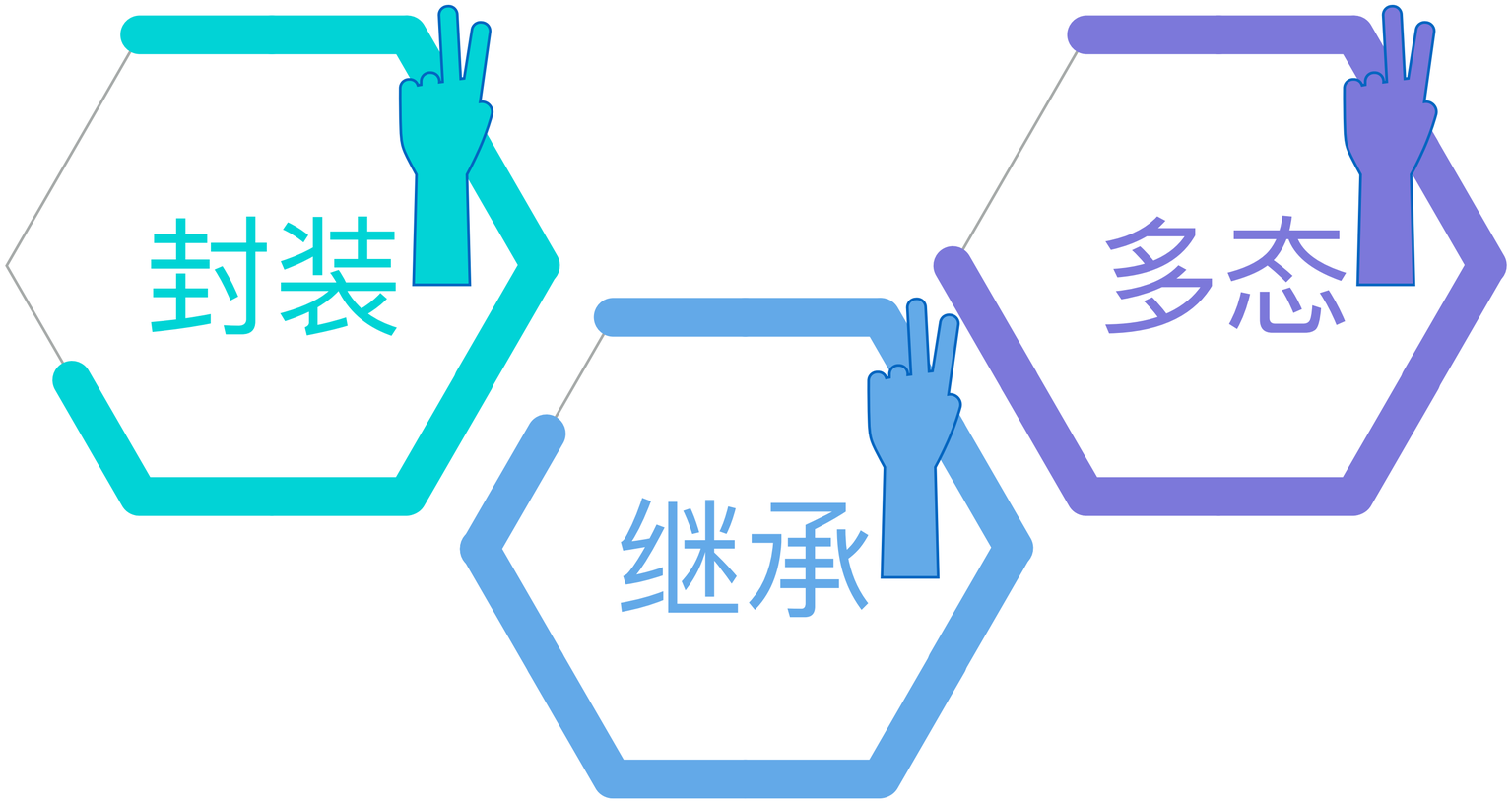
```
void taxi_drive(Taxi *self, int meters) {
    global_car_mtable.drive((Car *)self, meters);
    self->change += meters;
}
```

```
Taxi *new_taxi() {
    Taxi *new_obj = (Taxi *)malloc(sizeof(Taxi));
    ((Car *)new_obj)->mtable = (Car_MTable *)&global_taxi_mtable;
    global_taxi_mtable.super.drive = (Drive_Type *)taxi_drive;
    return new_obj;
}
```

```
void test3() {
    Taxi *taxi = new_taxi();
    Car *car = new_car();
    ((Car *)taxi)->mtable->drive((Car *)taxi, 15);
    car->mtable->drive(car, 17);
}
```

C语言实现面向对象

需要实现的三大特性



C语言实现面向对象

Is it enough?

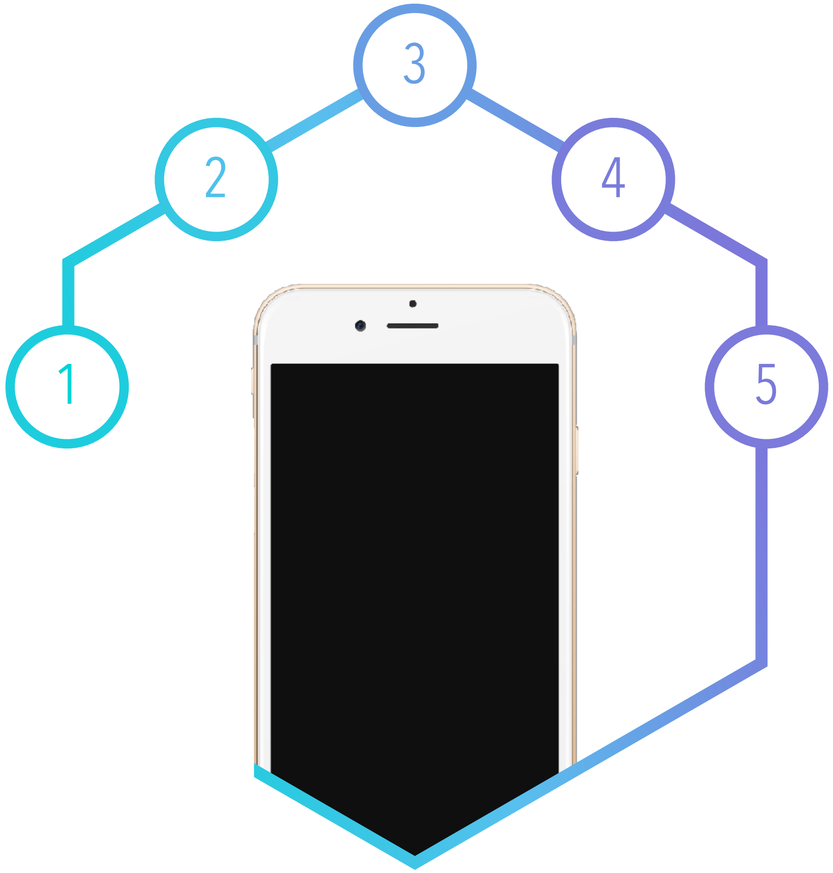
带来的思考

StuQ

带来的思考

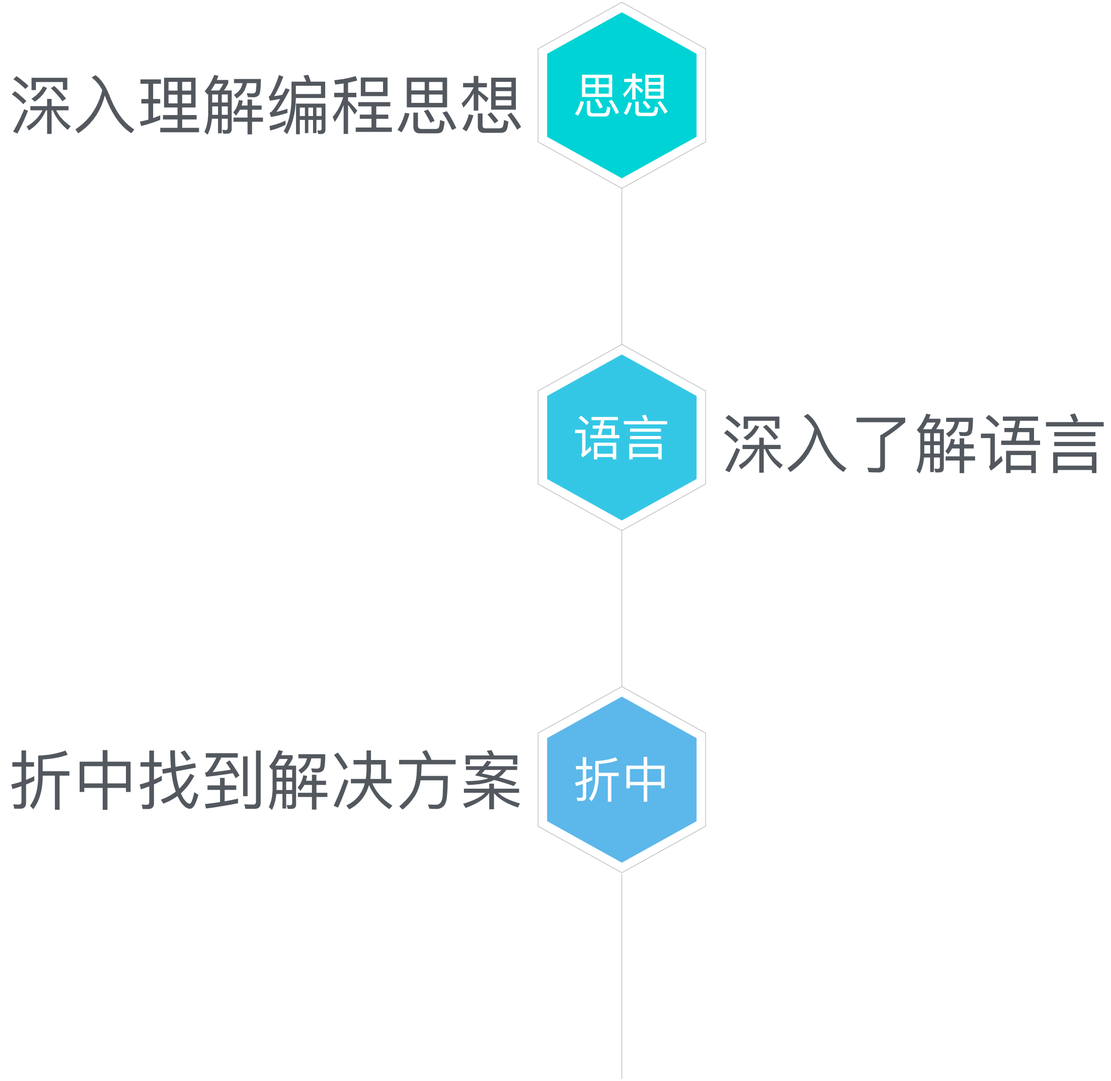
图灵等价

- 编程思想 > 编程语言
- 能力 > 形式



带来的思考

如何实现一种编程思想



Q&A

感谢聆听