C++

作者：Toki

日志

第一遍整理：只整理了部分重要知识点

基础

函数

函数签名

变量

1. **static**类型变量：执行到它的时候初始化，且只初始化一次；作用域为函数内
    - static全局变量：程序加载时初始化(进入main前)，作用域为文件内，生命周期为整个程序运行时间
2. 全局和静态变量初始化为0，局部变量不初始化

重载

1. 编译器会对重载函数的名称**进行修饰**，所以本质上还是不同名的函数
2. 在**数据类型不匹配**时，会尝试进行类型转换，失败则会报错

内联

声明为内联，但实际是否为内联不一定，这由编译器决定

默认参数

1. 声明写默认参数——做接口
2. 定义不写

C++数据类型

不同的系统大小不一定一样，只保证long long大小 >= long 等等

数据类型转换

不同类型的数据进行混合运算经常出现，某些类型的转换编译器可以隐式的进行

结构体

头文件中定义，有按字长进行**内存对齐**现象

大坑

1. 结构体中有**指针**，对结构体用memset只会清空指针，使其为NULL或其他值，而非清空其指向的内容
2. 结构体成员有**string**等类，memset也不可靠，理由同上

即，memset结构体只适用于所有成员为非指针基本数据类型的结构体

共用体

- 大小为最大成员大小，有内存对齐现象

如：

```
typedef union epoll_data
{
  void *ptr;
  int fd;
  uint32_t u32;
  uint64_t u64;
} epoll_data_t;
```

指针

一个字大小，用于内存中寻址

const与指针

```
char *const p1;
const char *p2;
char const *p3;
const char *const p4;
```

1. 指针的值不变，但指针指向的内容可变
2. 指针指向的数据不变，指针指向可变
3. 指针指向的数据不变，指针指向可变
4. 俩都不变

void*万能指针

不能直接解引用，转换后再使用

空指针

使用时，程序会崩溃——指向了不属于自己的虚拟地址空间

C++11 nullptr

——NULL可能会被隐式转换为整型，编译器无法判断其类型，故引入nullptr

野指针

指向已经不能使用的地址，造成内存泄露。
可能的原因有：指向free后的堆内存，指向函数栈内存

指针与回调函数

回调函数即使用函数指针，把函数也视为一个对象

```
void (*func_ptr)(); // 函数指针
```

使用回调函数即可实现特色化的功能

大坑：作为参数的数组

——完完全全的退化为了指针，没有任何数组特性，传递时一般需同时传递大小变量

引用

1. 本质为指针的伪装，可以当作正常变量使用
    - 不必为指针or数组建立引用——没有任何意义
2. 需要注意**生命周期**问题，不要引用局部变量

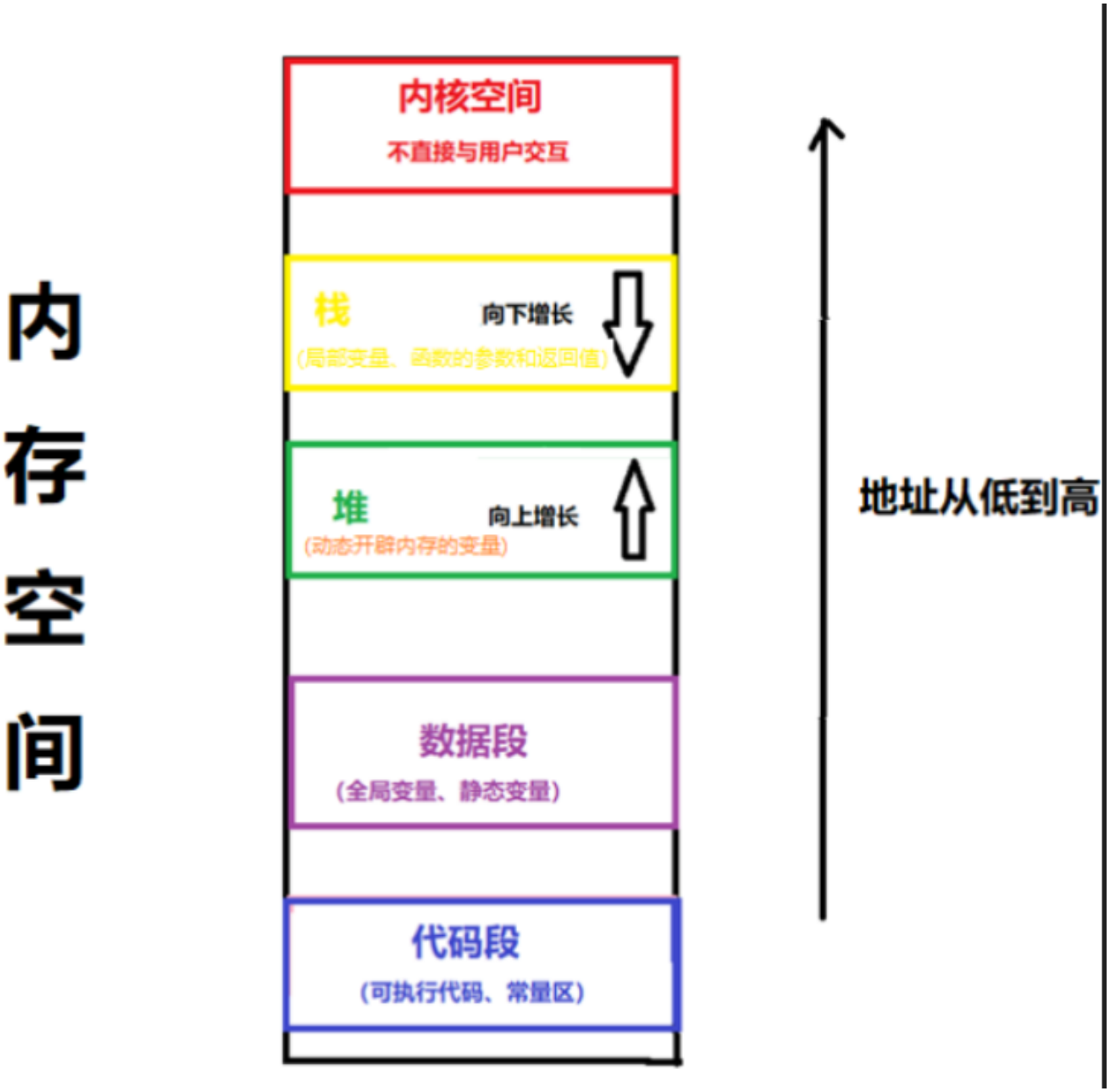
常引用

——参数为常数，参数类型为常引用`const type&`，只要参数类型与 type 类型相同，或可被隐式转换为 type 类型，则创建临时变量，让该参数指向该临时变量

形参定义约定

1. 小实参可以直接值，大实参传指针或引用
2. 需要修改不加const，不需要修改加const

C++内存模型



类与对象

权限

1. 内部成员无权限限制，外部成员只能类中的 public 成员，子类 至多访问 protected 成员
2. 缺省权限为 private，结构体缺省权限为 public

大坑

注意，权限针对**访问**，处于初始化阶段和销毁阶段的代码不受其限制

基本使用规则

1. 一般不用memset，可以专门定义相应函数
2. 类函数自动变为内联函数——仍遵循上文所述规则

构造函数与析构函数

1. 构造函数：创建变量时，自动调用的初始化函数，**禁止手动调用**，但其他类在初始化时，可以**委托构造**，必须为Public
2. 析构函数：销毁类时，自动调用的清理函数，**禁止重载，禁止传参，可手动调用**，必须为Public
3. 创建对象时，**先初始化构造函数形参**，再初始化类的成员
4. 工程经验：不建议在构造/析构函数中写太多的代码，最好只进行必要的初始化工作
5. 注意，**手动调用析构函数**也会自动调用基类析构函数
    - 如果有类似的需求，必须置空指向堆区内存的指针，否则对象销毁时，会因delete野指针而崩溃

构造函数与隐式转换

```
Person person1 = 30;
```

禁止隐式转换：

```
explicit Person(int a){
    //...some code
}
```

有**二义性**的隐式转换会报错，**警惕隐式转换**，极易导致结果发生偏差

构造函数与效率问题

```
Person girl = Person(20); // 一次构造一次析构

Person girl;
girl = Person("西施"20); // 两次构造两次析构
```

构造、析构顺序问题

1. 先构造基类，再按顺序构造成员类，最后构造自身类
    - 确保被依赖对象先于依赖者构造
2. 析构过程与之相反
    - 防止无效引用的出现

初始化列表：效率与常量成员的初始化

- 出现在初始化列表中，就不应在函数体中再次赋值

```
Person\(const std::string& name, int age) : name(name), age(age) {}
```

效率问题

注意，初始化列表并非仅仅初始化阶段赋值那么简单
对于代码：

```
class CGirl{
// 构造函数中
   boy.x=pass_boy.x
//...some code
};
//...some code
CBoy boy("子都");
CGirl g1("冰冰",18,boy);
```

1. 若此时的CGirl构造函数的CBoy类型为CBoy，则函数调用过程为：
   - boy构造，boy拷贝构造(值传递)，boy普通构造(构造CGirl成员)，girl构造，然后在构造函数内赋值
   - **多调了一次函数**
2. 若此时的CGirl构造函数的CBoy类型为CBoy&，则函数调用过程为：
   - boy构造--boy普通构造(构造CGirl成员)--girl构造，然后在构造函数内赋值
   - 且boy初始化和赋值为**两步操作**
3. 若为CBoy& + 初始化列表，则为：
   - boy构造--boy拷贝构造(构造CGirl成员)--girl构造
   - 且boy初始化和赋值时**一步操作**

```
CGirl(CBoy& pass_boy): boy(pass_boy){}
```

常量成员初始化

这是ok的：

```
class Person
{
private:
string name_;
const int age_;

public:
Person(string n, int a) : name_("@" + n), age_(a)
{
cout << "name: " << name_ << " age: " << age_ << endl;
}
};
```

但你想在构造函数内给常量成员赋值就不行，因为这已经处于变量**计算阶段**，而非变量构造阶段

- 先创建变量 → 进入函数，这是计算阶段，不能对常量赋值

拷贝构造函数

——用已存在的对象创建新对象时，调用该构造函数，而非普通构造函数

```
Person(const Person& p){}
};
//...some code
Person p2(p1);
Person p2 = p1;
```

常函数

函数中不会修改成员变量的值

```
classname :: my_func() const {}
```

1. **mutable**类型的变量可以突破const限制，在该函数中被修改——灵活性+语义
2. 常函数只能调用常函数

this指针

- 指向调用该函数的对象

static成员

1. **语义**：属于整个类的成员，而非某个对象实例，程序中仅此一份
   - 也因此，静态成员函数只能访问静态成员，而对象可以访问静态成员和对象成员
   - 也因此，静态成员函数无this指针
2. **初始化**：全局区初始化，而非创建某个对象时初始化

友元

——为了使一个函数能够访问类的私有方法，需在类中声明某函数为friend

```
class Person
{
   friend int main(); // main函数可以访问Person的全部成员
   // ...some code
};
```

友元类

使用方法同上，但需要注意一点，友元关系**不能继承**，且是**单向**的

友元成员函数

用法同上，就是代码有些逆天

```
// 类B的成员函数需要使用类A的私有成员

class A; // 先声明A


class B
{
public:
void show_A_info(const A &a);
};


class A
{
friend void B::show_A_info(const A &a); // 友元函数

private:
int age;

public:
A() : age(0) {}
};


void B::show_A_info(const A &a) // 定义A后才能定义该友元函数，因为要用到A中信息
{
cout << a.age << endl;
}
```

大坑

这是声明函数，不是创建对象

```
ppl xm();
```

创建对象时，无参就不用写圆括号

这是创建匿名对象，不是调用构造函数

```
class A;
...
A();
```

构造函数只能在构造函数中进行委托构造调用：

```
AA(int a, int b);
// ...some code
AA(int a, int b, const string &str) : AA(a, b) ;
```

浅拷贝与深拷贝

什么是浅拷贝？

两个指针指向同一块内存，则会产生以下问题：

1. 一个指针被释放时，另一个指针就成了野指针
2. 一个指针对数据的修改，对另外一个指针也可见

怎么产生的?

万恶之源：默认赋值函数

```
类名 & operator=(const 类名 & 源对象);
```

解决方法：深拷贝

重载上述函数，对指针成员做额外的处理

运算符重载

pass

C++对象模型

类：**关联数据和函数**，对象中维护了**指针表**，存放了成员和地址的对应关系

- 非静态成员变量地址连续，在栈上，成员函数在代码段，静态变量在数据段
- 指针表大小固定，连续

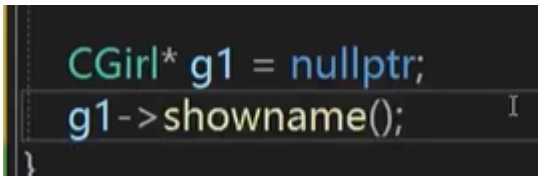
对象实例内存

包含：非静态成员、内存对齐消耗、为支持虚成员产生的额外负担

this

C++与C没什么不同，能够在成员函数中访问成员变量是因为**编译器的特殊处理**，如果一个非静态成员函数用到了成员变量，则编译器会在其前面加个this，指向对象实例，以访问到对象数据，就和**结构体指针**一样

而对于没有用到this指针的非静态成员函数，用**空指针**都可以调用这个函数



继承

——复用基类成员变量与成员函数，并基于此进行功能扩展

权限

1. public无所谓，protected只允许子类访问，private谁都不能访问(除友元)
2. 继承方式：指定基类成员在派生类中，被访问的最高权限，高的会降级否则降级——而这会导致权限混乱，故一般用**public**
3. **using**可以随意改变基类public、protected成员在子类中的被访问权限，但private碰都不能碰

继承对象模型

构造与析构

1. 构造析构函数调用顺序：同上
2. 父子类成员，谁的成员则谁的构造函数负责初始化，不要混淆：代码重用&私有限制

内存使用

只申请一块内存，大小 = 父类上述类型的成员 + 子类上述类型的成员

- **包含父类私有成员**，尽管子类不可见
- **突破私有权限限制**：使用指针，只要知道声明顺序，用指针/memset直接就能访问到这个理论上不可见的父类私有成员

作用域遮蔽

重名不覆盖，不重载——但遮蔽：优先用子类，可以通过AA::的方式指定调用对象

B is an A

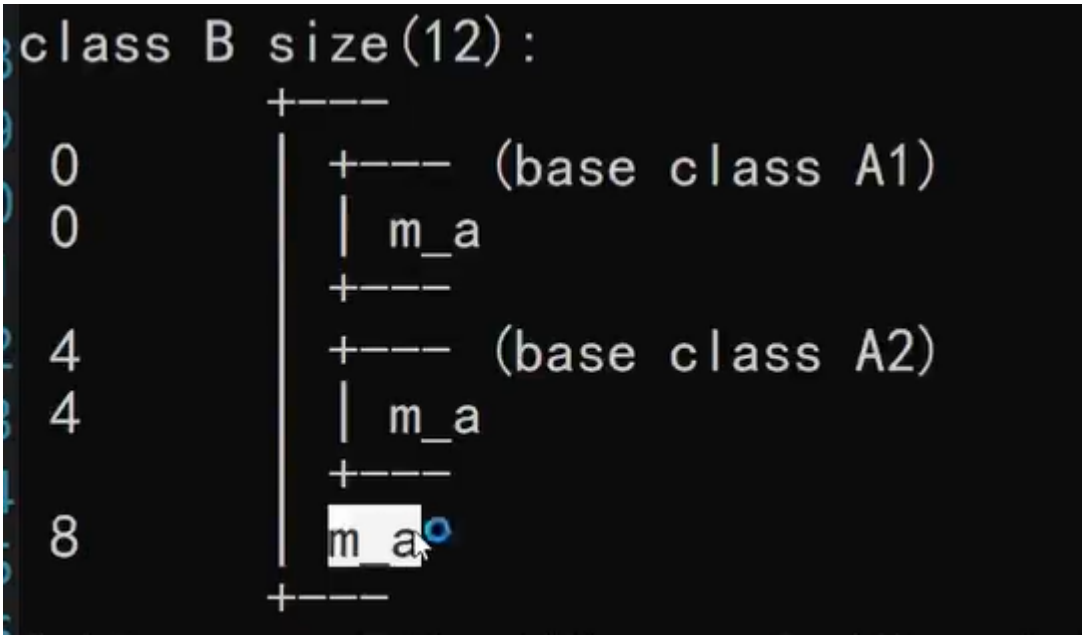
允许使用基类对象/指针/引用，指向派生类对象——但会舍弃非基类成员

- 有啥用？派生类构造基类，派生类传递给基类参数
- 但这种关系是单向的
- **多态原理**

多继承与虚继承

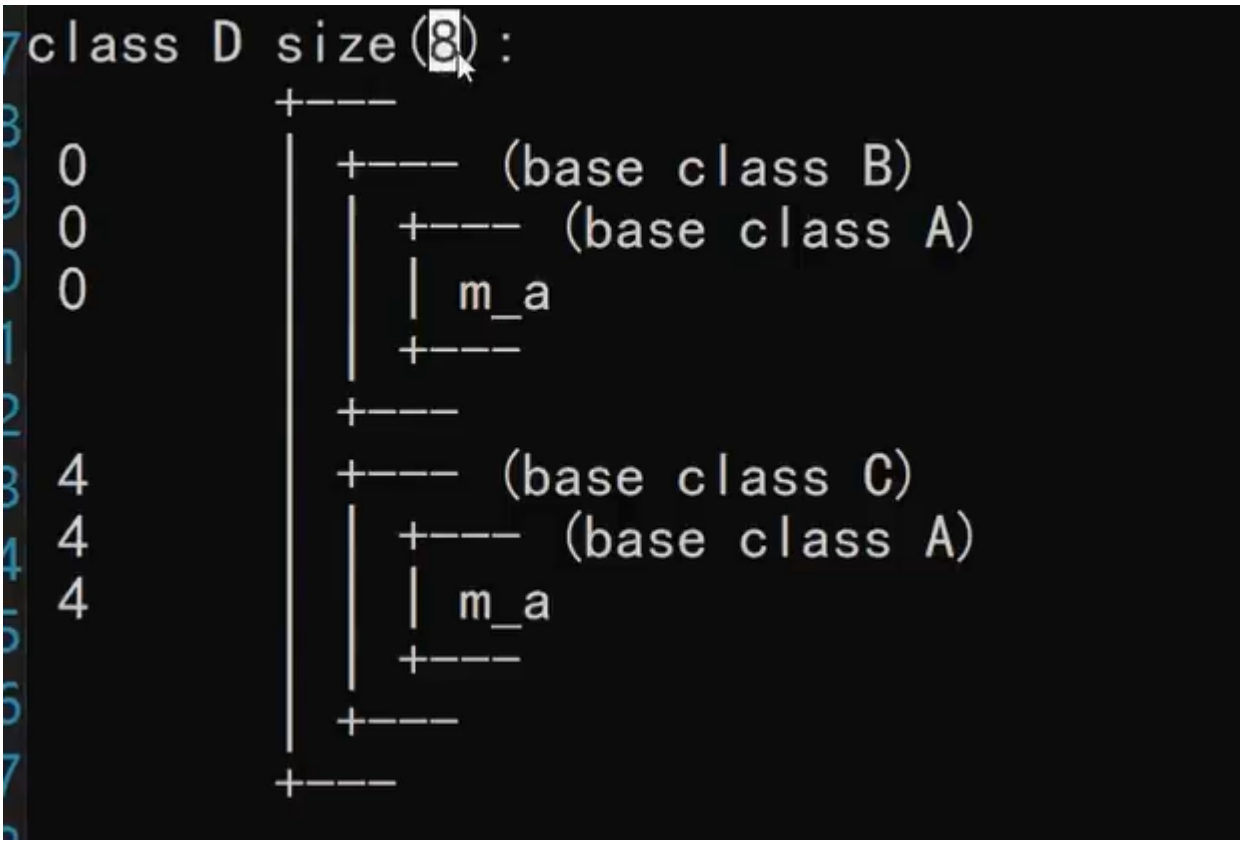
典型多继承

——继承的地方写俩继承罢了，内存模型为：



- 三个变量重名，可以使用域解析符
- 99%的情况

多继承的问题：菱形继承

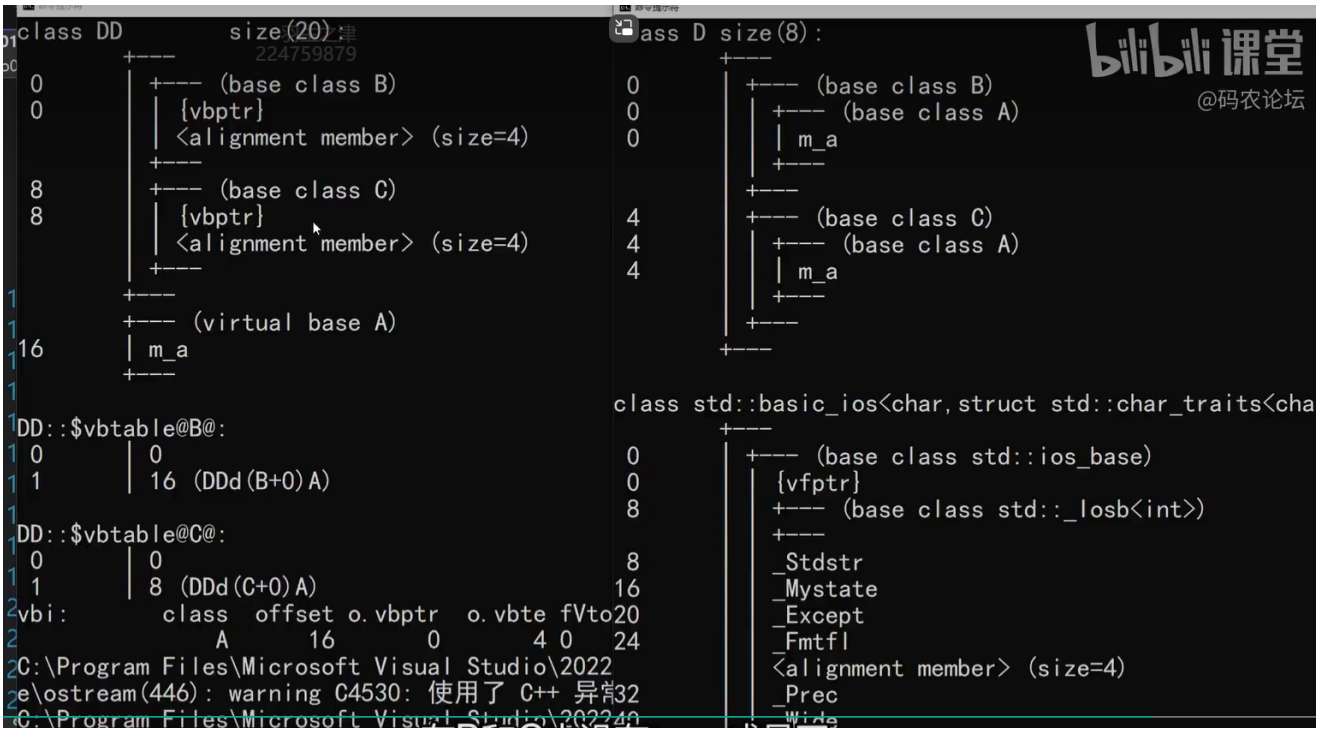


- **数据冗余**和**二义性**问题

解决方法：虚继承

```
class A{ public: int m_a; };
class B: virtual public A{};
class C: virtual public A{};
class D: public B, public C{};
```

——B::m_a与C::m_a变为同一个变量，即解决上述问题
虚继承内存模型：变量替换为虚指针，对象模型中额外添加结构体



多态

基类指针指向子类对象，使用子类重写父类的函数

- 重写——**函数特征**相同，不相同的就会直接使用基类函数

虚函数

——被重写的函数仅作为接口存在，不会被直接调用，也就无需实现

- 虚函数声明要写virtual，但定义不用；如果只有定义则需要(类中定义)

case 1：非纯虚函数

为声明为虚函数的函数提供定义

```
class Hero //基类
{
public:
    virtual void skill1() { cout << "hero 释放了技能" << endl; } //非纯虚函数
};
```

case 2：纯虚函数，抽象类

不提供定义，直接=0，也因此抽象基类**不能实例化**，派生类必须重写该函数才可实例化

```
class Hero //基类，抽象类
{
public:
    virtual void show() = 0; //纯虚函数
};
```

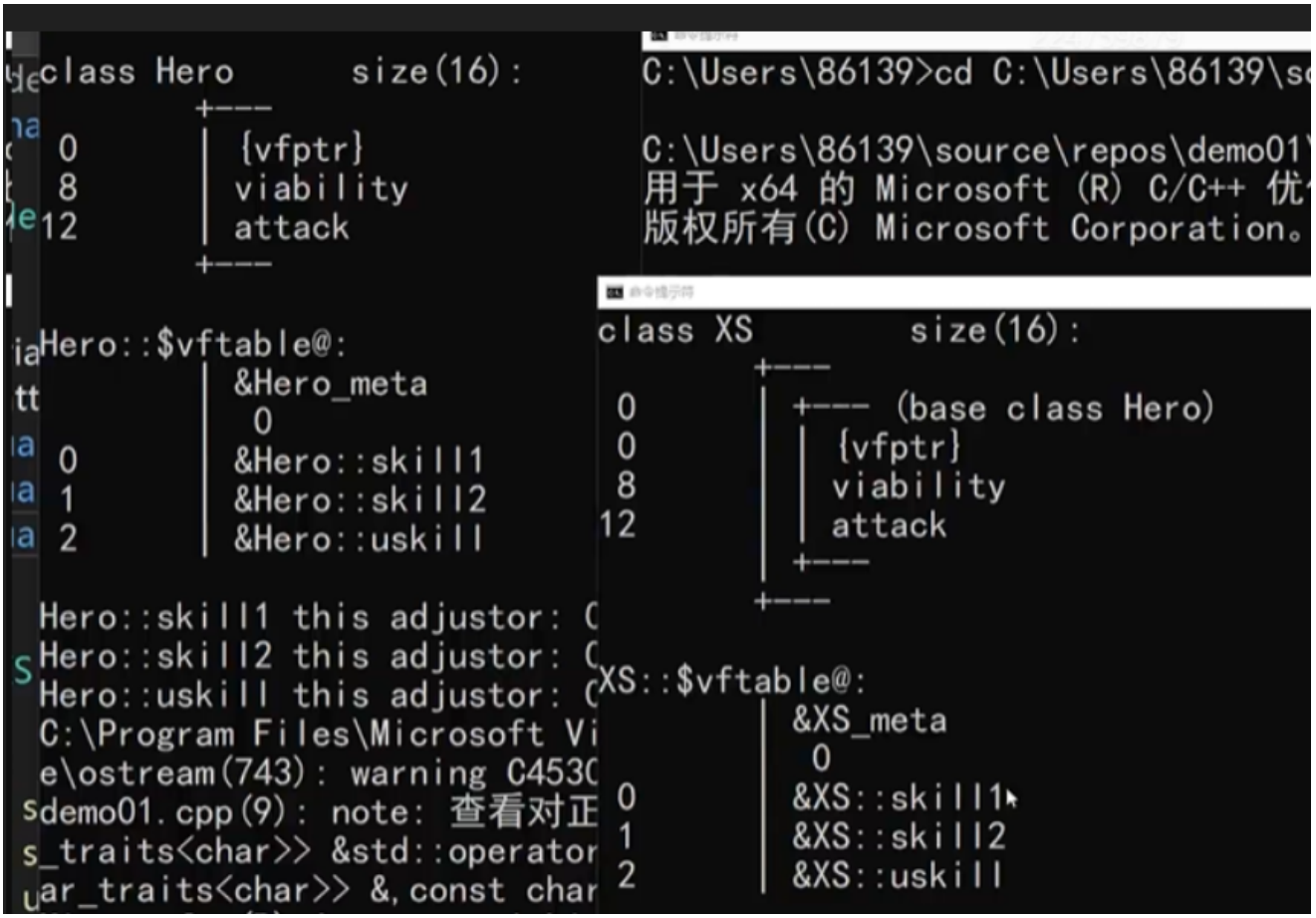
多态内存模型

结构体中多了个**虚函数指针**，指向一张存放虚函数地址的表

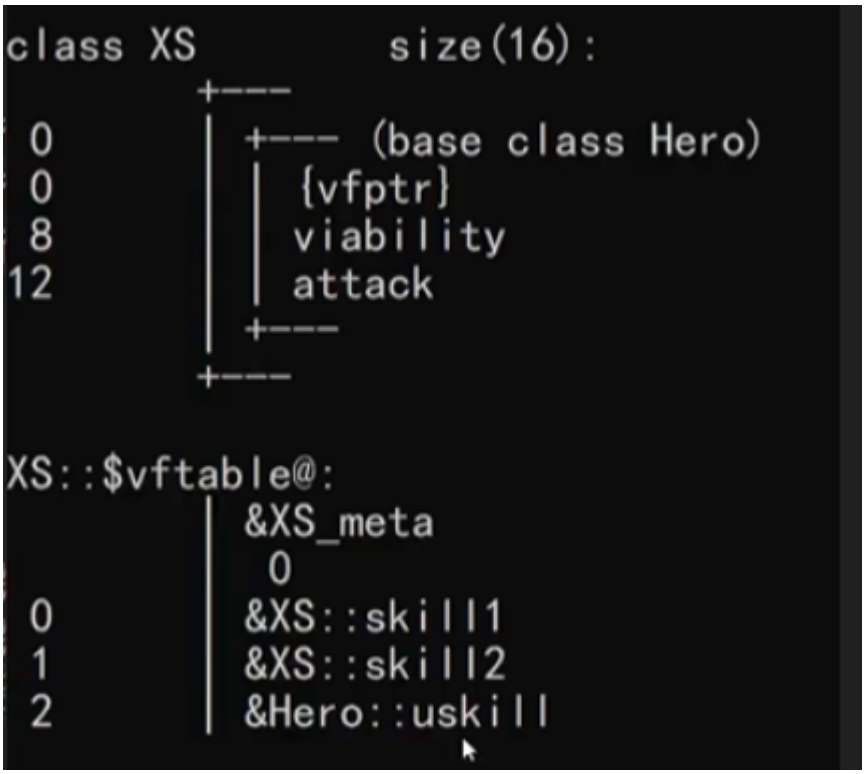
- 虚函数表**不会包含非虚函数的地址**，其地址在编译器确定

多态原理

重写了哪个虚函数，就修改虚函数表中的哪个函数的地址



部分重写：



效率

普通成员函数调用：已链接函数地址，直接调用
虚函数调用：**先查虚表**，找到函数地址然后调用，效率低

静态多态与动态多态

1. 静态多态：编译期确定调用函数地址
    - 函数重载、函数模板
2. 动态多态：运行时确定调用函数地址
    - 类多态

多态与析构函数

多态情境下，销毁时，只调用基类析构函数，`没有释放子类资源

- 解决办法：**虚析构函数**，此时虚函数表中的析构函数为子类的析构函数，而子类的析构函数又会自动调用父类析构函数，也就解决了这个问题
- 虽说这俩函数特征不可能相同，但是没问题，相信编译器

因此，多态情境下，即便不使用基类的析构函数，也必须**提供空虚析构函数**

纯虚析构函数?

听起来不太靠谱，子类总自动调用基类析构函数，若无相应实现则会报错

那加一个实现?

ok，问题解决，但有什么必要吗?

有什么必要吗?

想使一个类成为抽象类，但没有可用的纯虚函数

运行阶段类型识别 dynamic_cast

新需求：想调用子类的非虚函数咋整?

- 基类指针转换为原类型指针!
- 这个方法需要由程序员确保类型正确，可以结合if + 额外的信息
- 加个dynamic_cast...pass

type_id...pass

模板

编译器自动推导参数类型，并生成函数定义

- **静态多态，编译期确定类型**

auto 类型必须定义时初始化