lecture8

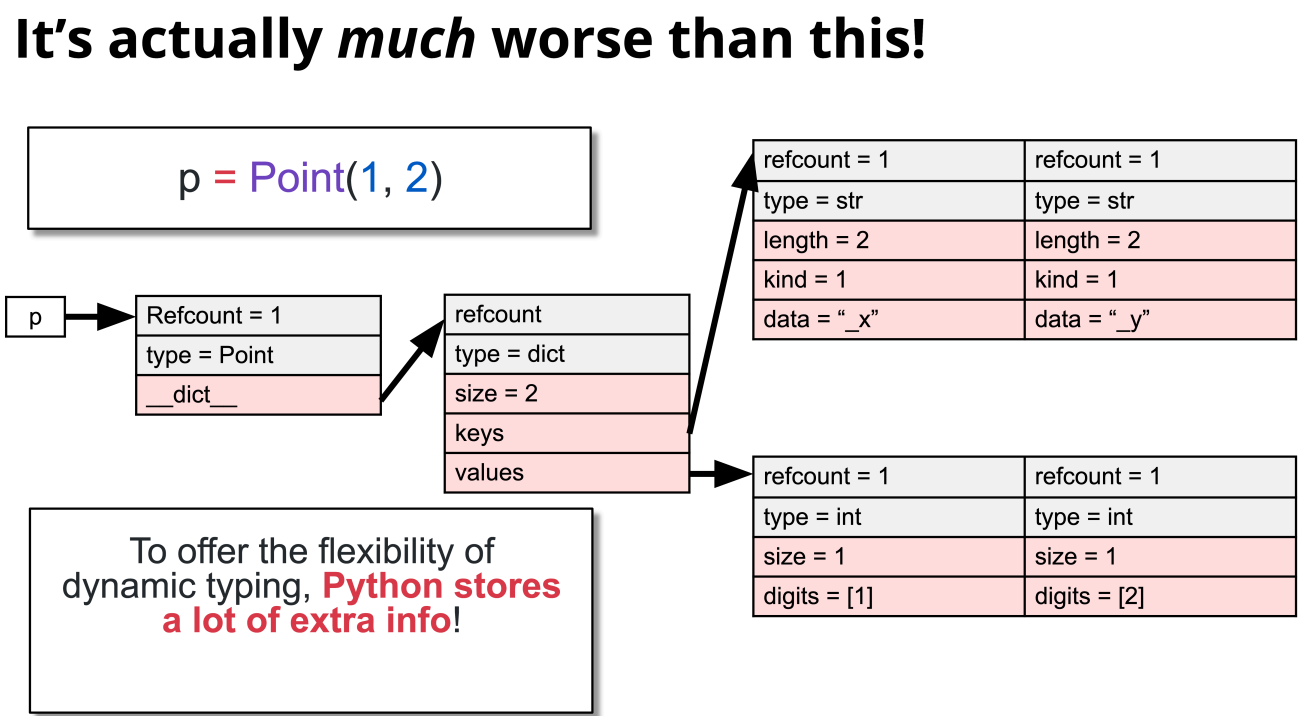
what：Inheritance
what more:
A Recap on classes
Inheriance
virtual functions
closing thoughts

Recap

python与c++在oop的区别：

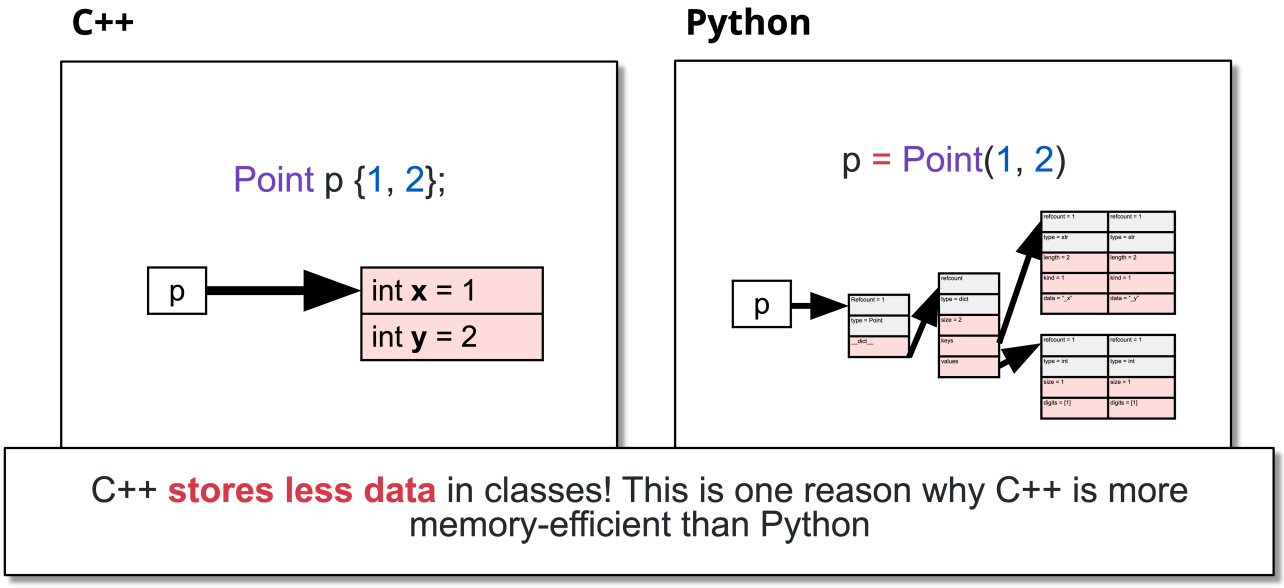
> 2025Fall-08-Inheritance, p.13
>
> > Python stores extra information about the type of the object in its memory footprint!
> > This enables runtime type checking.

说白了python做一个指针的开销很大，都是嵌套的



这张图很好的说明了这点，每个对象P，内部有三个部分：Refcount,type=? **dict**
这个类似字典的变量，包含了更多的内容，比如新的refcount，新的type

Classes: Memory Layout



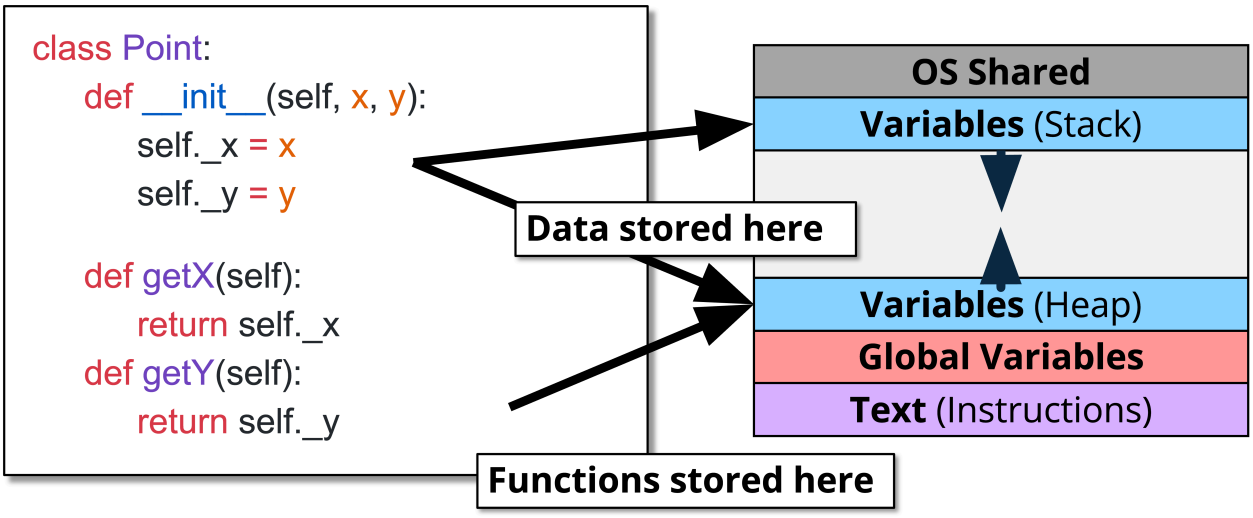
c++相对来说好很多

python与c++的区别：function

python：python把函数实现放在了堆里，这意味这这个函数也看成了类，从这个函数类出发，有新的嵌套

Where are the functions in Python?

Functions are not stored in the object dictionary itself, but separately in a dictionary associated with the class.

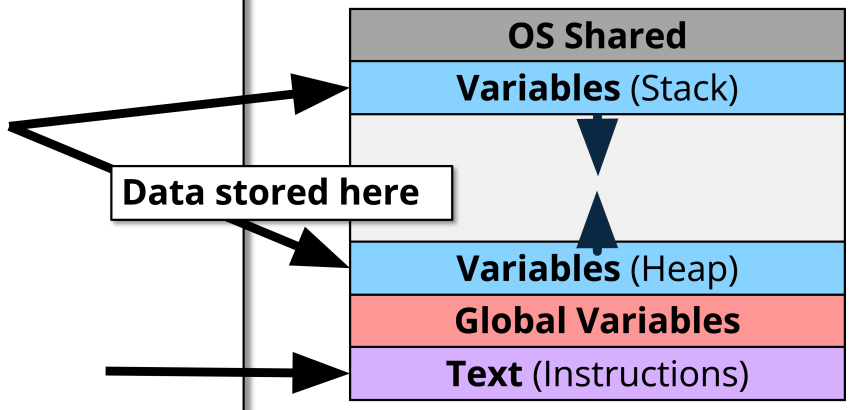


c++：c++把函数实现放在了text底层

Where are the functions in C++?

Functions are not stored in the object itself, but separately

```
class Point {
private:
    int x;
    int y;
public:
    Point(int x, int y)
        : x{x}, y{y} {}
    int getX() { return x; }
    int getY() { return y; }
};
```



对于this和self

Where are the functions?

Functions are stored separately from the object

```
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y

    def getX(self):
        return self._x
    def getY(self):
        return self._y
```

```
p = Point(1, 2)
px = p.getX()

# ...is the same as...

p = Point(1, 2)
px = Point.getX(p)
```

Passing self as parameter

this in C++

this is passed as a parameter to class function behind the scenes

```
int Point::getX() {
    return this->x;
}

// ...gets turned into...

int Point_getX(Point* this)
{ return this->x; }
```

```
Point p {1,2};
int x = p.getX();


// ...gets turned into...

Point p {1,2};
int x = Point_getX(&p);
```

Passing this as parameter

Inheritance

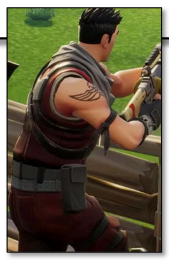
2025Fall-08-Inheritance, p.29

> A mechanism for one class to inherit properties from another

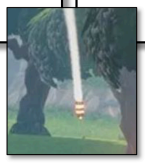
example：其实上一节说过这个，但是这个重新讲一下：游戏的例子

There's a lot of redundancy here!

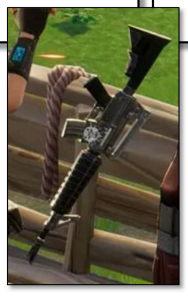
```
class Player {
    double x, y, z;
    HitBox hitbox;
    double hitpoints;
public:
    void damage(double hp);
    void update();
    void render();
};
```



```
class Projectile {
    double x, y, z;
    HitBox hitbox;
    double vx, vy, vz;
public:
    void update();
    void render();
};
```



```
class Weapon {
    double x, y, z;
    HitBox hitbox;
    size_t ammo;
public:
    void fire();
    void update();
    void render();
```

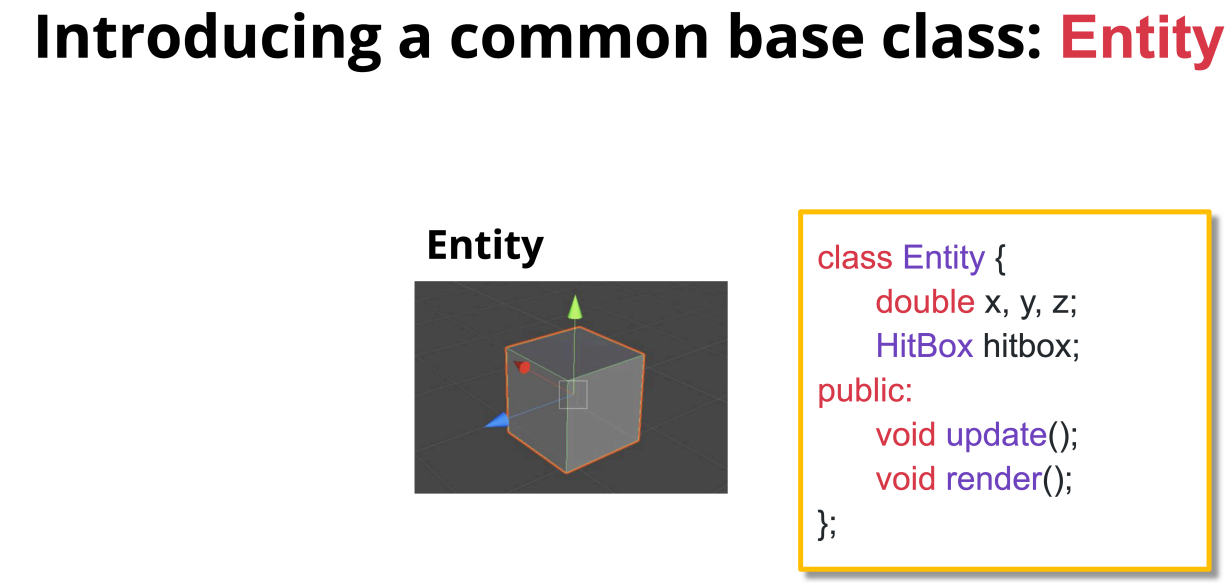


```
class Tree {
    double x, y, z;
    HitBox hitbox;
public:
    void update();
    void render();
};
```

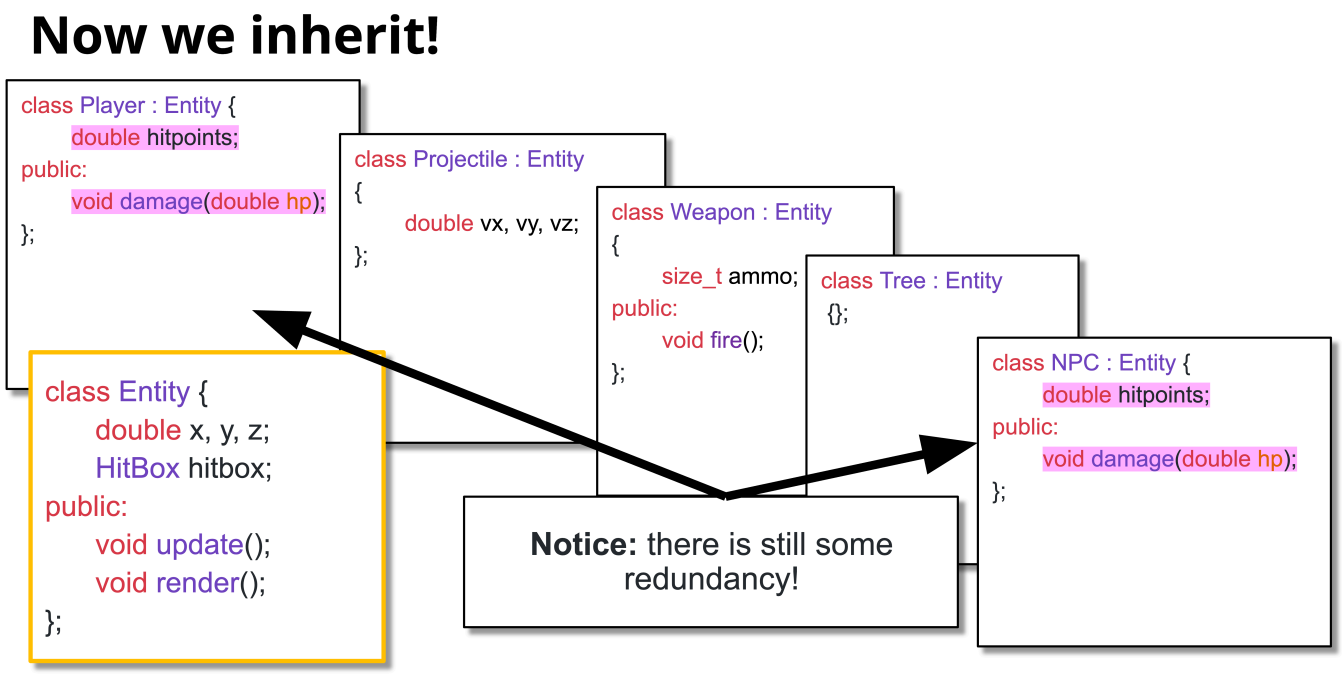
```
class NPC {
    double x, y, z;
    HitBox hitbox;
    double hitpoints;
public:
    void damage(double hp);
    void update();
    void render();
};
```

对于这个游戏的设置，每个人物都有自己的坐标和边界判定，这些都可以写进同一个父类里面

设计出一个父类：

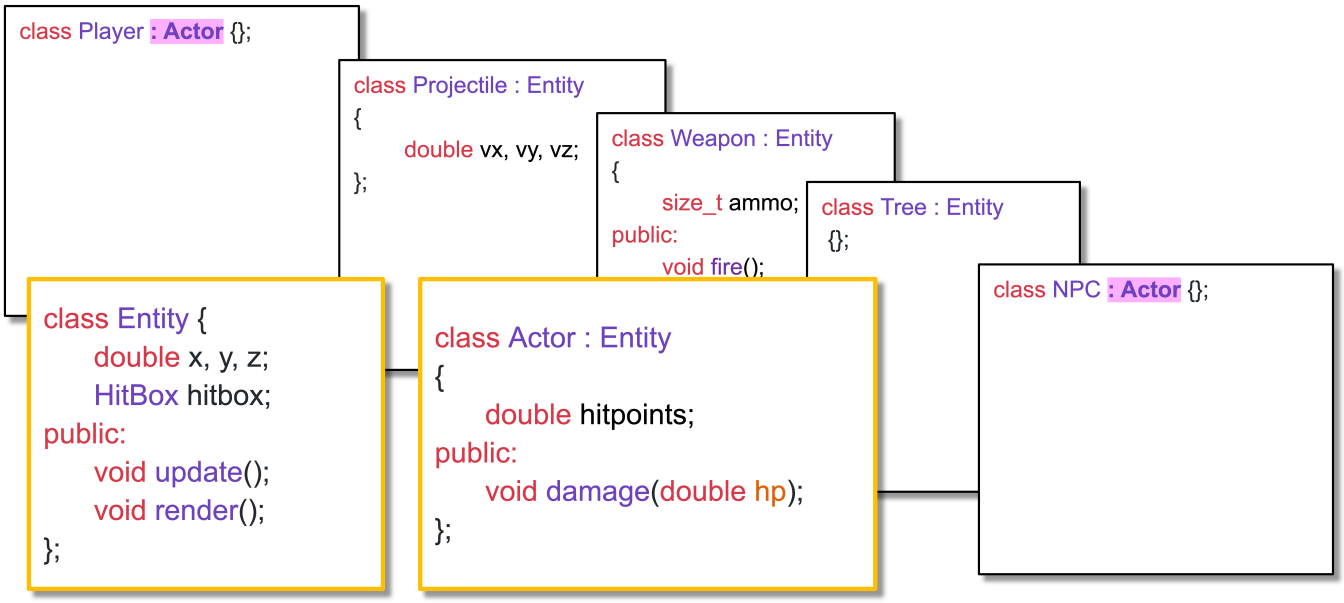


继承第一轮：把Entity继承上



继承第二轮：发现还有冗余，把Actor继承好

More layers of inheritance...



关于protected
public与private在继承上的漏洞

Note: access modifiers

private inheritance (default)

```
class Child : private Parent
```

private in parent	inaccessible in child
public in parent	private in child

public inheritance

```
class Child : public Parent
```

private in parent	inaccessible in child
public in parent	public in child

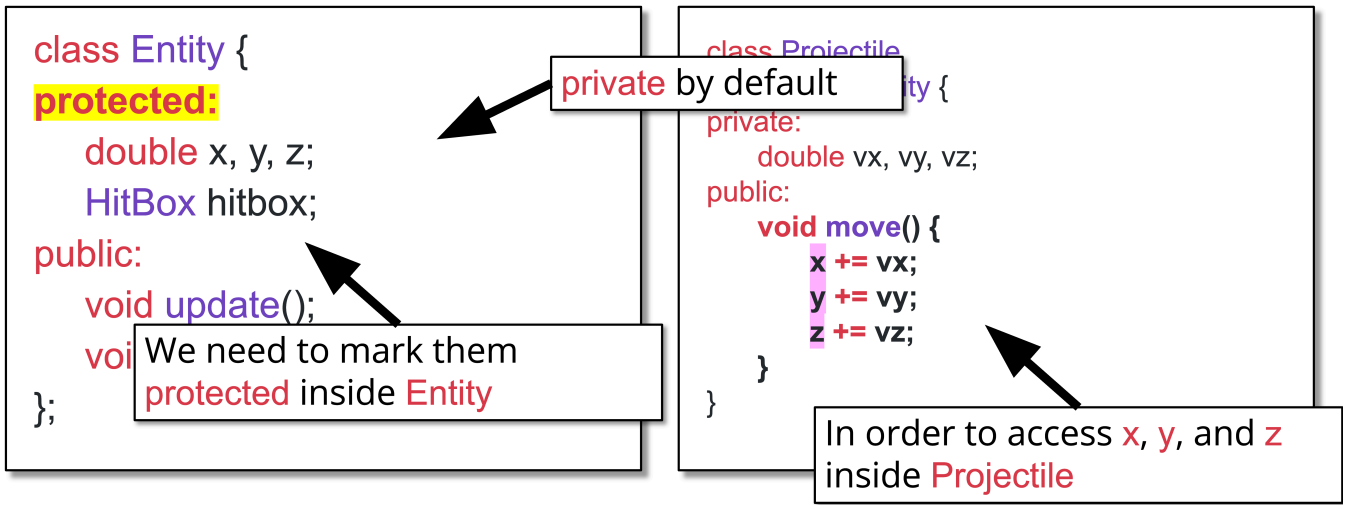
Note: public inheritance better models **is-a** relationships! A Player really *is* an Entity because it exposes all of Entity's functionality publicly

protected可以让继承的子类可以访问父类中的private成员，但是其他外部类无法访问

Note: protected access modifier

Protected members are visible to subclasses, but not the outside!

- Remember, class members are *private by default*



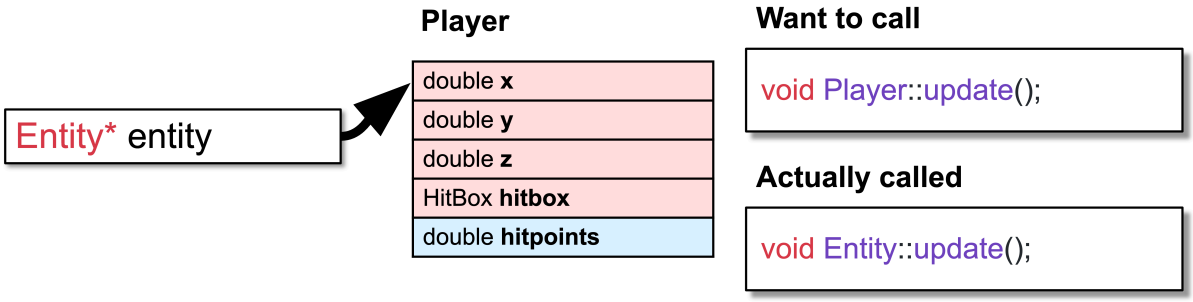
virtual function

起因：

父类继承下很多子类，这些子类的成员函数与父类的成员函数相同，具体调用时分不清是调用父类的成员函数还是子类的成员函数

Problem: Which one is called?

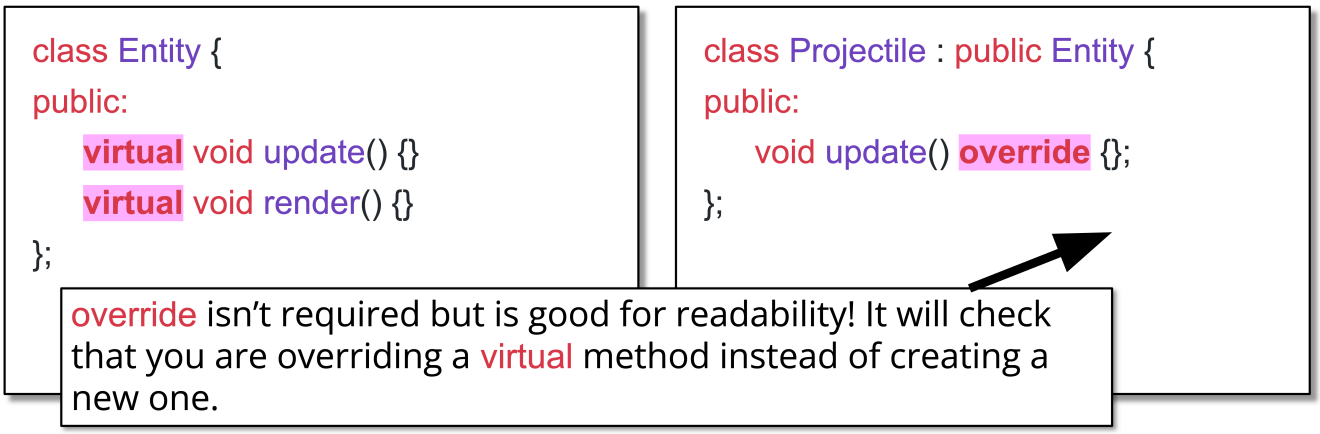
- We should call the update method which matches the type of the object that entity points to
  - If entity points to a Player, we should call Player::update()
  - If it points to a Projectile, we should call Projectile::update() and so on
- But an Entity* alone doesn't tell us any information about the type!



引入虚函数重写

Virtual functions

- Marking a function as virtual enables dynamic dispatch
- Subclasses can override this method



原函数：

```
#include <iostream>
#include <vector>

class HitBox {
public:
    double x, y, width, height;
};

class Entity {
protected:
    double x, y, z;
    HitBox hitbox;
public:
    virtual void update() {};
    virtual void render() {};
};

class Player : public Entity {
    double hitpoints = 100;
public:
    void damage(double hp) {
        hitpoints -= hp;
    }

    void update() override {
```

```
        std::cout << "Updating Player!" << std::endl;
    }

    void render() override {
        std::cout << "Rendering Player!" << std::endl;
    }
};

class Tree : public Entity {
public:
    void update() override {
        std::cout << "Updating Tree!" << std::endl;
    }

    void render() override {
        std::cout << "Rendering Tree!" << std::endl;
    }
};

class Projectile : public Entity {
    double vx, vy, vz;
public:
    void update() override {
        std::cout << "Updating Projectile!" << std::endl;
    }

    void render() override {
        std::cout << "Rendering Projectile!" << std::endl;
    }
};



int main() {
    Player player;
    Tree tree;
    Projectile proj;

    std::vector<Entity*> entities { &player, &tree, &proj };
    while (true) {
        std::cout << "Rendering frame..." << std::endl;
        for (auto& entity : entities) {
            entity->update();
            entity->render();
        }
    }
```

```
    return 0;
}
```

虚函数virtual function就是多态的表现

纯虚函数的写法

Pure virtual functions

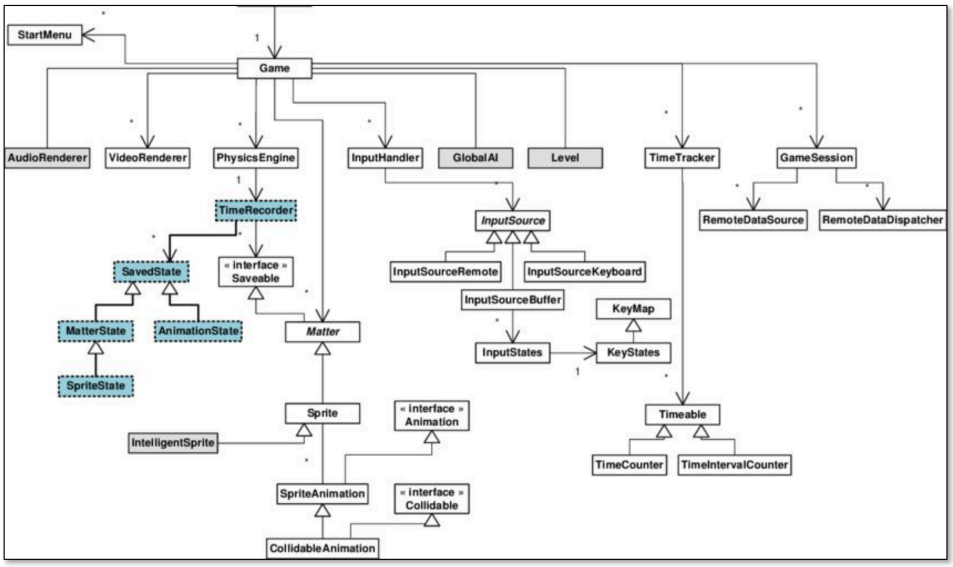
```
class Entity {
public:
    virtual void update() = 0;
    virtual void render() = 0;
};
```

Mark a virtual function as pure virtual by adding = 0; instead of an implementation!

closing thoughts

继承的局限性：

Sometimes inheritance can get out of hand



Sometimes inheritance can get out of hand

- Big inheritance trees tend to be **slower** and **harder to reason about**
  - In video games, approach of subclassing for every different object type is uncommon among modern game engines
  - Composition is often more flexible and just makes sense

这张图的核心观点是：**“过度使用继承（Inheritance）会导致系统变得极其复杂且难以维护。”**

在软件工程（尤其是游戏开发）中，这是一个非常经典的讨论点。我们可以从以下几个维度来理解这张图的含义：

1. 继承的“恐怖森林” (Inheritance Hell)

图中的 UML 类图展示了一个极其复杂的层级结构。

- **深层嵌套：** 想要理解底部的 `CollidableAnimation`，你必须先搞清楚它继承的 `SpriteAnimation`、`Sprite`、`Matter` 等一系列父类的逻辑。
- **脆弱性：** 如果你修改了顶层的某个基类（比如 `Matter`），可能会产生“蝴蝶效应”，导致所有子类崩溃。这就是所谓的“牵一发而动全身”。

2. 为什么在游戏开发中被嫌弃?

图中文字提到，现代游戏引擎已经很少使用这种“为每种类型创建一个子类”的方法了：

- **推理困难：** 开发者很难一眼看出一个对象到底具备哪些功能。
- **性能问题：** 过深的继承树在某些语言中会增加内存开销和寻址时间（例如虚函数表的查找）。
- **灵活性差：** 如果你想要一个“既能动（Animation）又是可破坏的物理物体（Matter）”的对象，在纯继承体系下，你可能需要复杂的“多重继承”，这通常是灾难的开始。

3. 替代方案：组合优于继承 (Composition over Inheritance)

图中最后提到 **“Composition is often more flexible”**（组合通常更灵活）。

这就是现代游戏引擎（如 Unity 或 Unreal）广泛采用的 **ECS（Entity Component System，实体组件系统）** 模式：

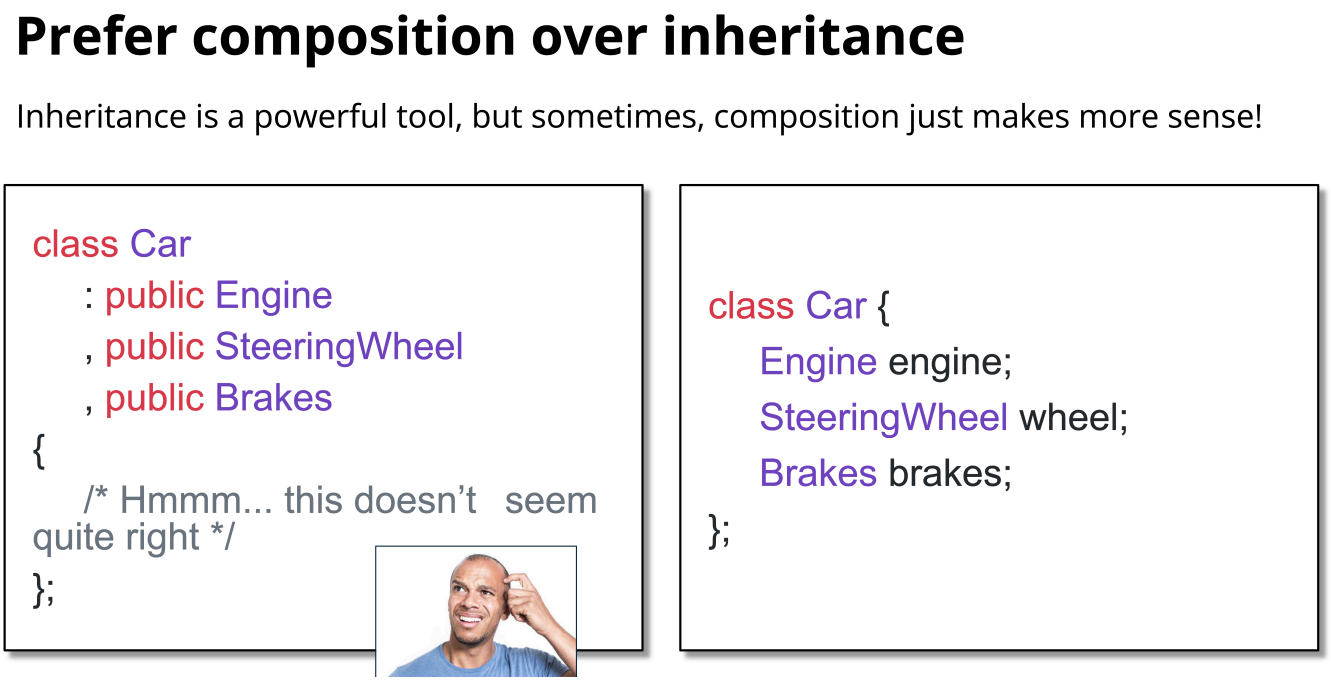
模式	逻辑	比喻
继承 (Inheritance)	它**是**什么 (Is-a)	“它是一个会飞的、会喷火的、红色的龙”
组合 (Composition)	它**有**什么 (Has-a)	“它有一个‘飞行组件’、一个‘喷火组件’、一个‘红色外观’”

总结

这张图是在提醒开发者：**不要试图用继承去描述世界上的万事万物。** 保持类结构的扁平化，通过把不同的功能“插件”（组件）拼装在一起，比构建一个庞大的“族谱”要高效得多。

> 2025Fall-08-Inheritance, p.99
>
> Prefer composition over inheritance

组合比继承好



> 2025Fall-08-Inheritance, p.100
>
> Prefer composition and inheritance

二者一起用会更好

Prefer composition and inheritance

Combining both of these ideas can give the best of both worlds!

```
class Car {
    Engine* engine;
    SteeringWheel* wheel;
    Brakes* brakes;
};

class Engine {};
class CombustionEngine : public Engine {};
class GasEngine : public CombustionEngine {};
class DieselEngine : public CombustionEngine{};
class ElectricEngine : public Engine {};
```

If you want to see one place this technique is used in C++, look up the PIMPL idiom!