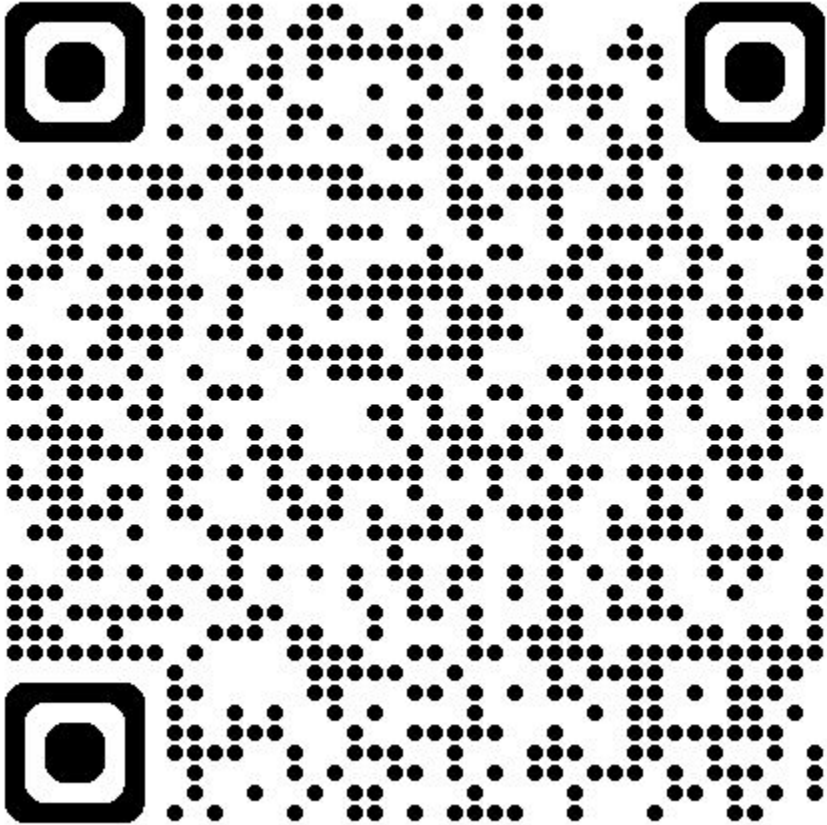


Welcome! Bonus attendance form below



Exercise 1

<https://github.com/cs106l/cs106l-lecture-code/blob/main/lecture08/exercise1.h>

Exercise 2

<https://github.com/cs106l/cs106l-lecture-code/blob/main/lecture08/exercise2.h>

Exercise 3

<https://github.com/cs106l/cs106l-lecture-code/blob/main/lecture08/exercise3>

OPTIONAL

Hi all, everything from here on out is optional
review of inheritance.

Today's Agenda

- A Recap on Classes
 - And more on how they work behind the scenes
- Inheritance
 - Inheritance allows us to reuse features from a parent class
- Virtual Functions
 - Defining function interfaces that can be overridden in sub-classes
- Closing Thoughts
 - Should you always use inheritance?

Last Time: Classes

- What are classes?
 - Turn to the person next to you and think of one thing you remember from Tuesday's lecture!

Last Time: Classes

- What are classes?
 - Turn to the person next to you and think of one thing you remember from Tuesday's lecture!
- A class represents an abstraction — it can model a real-world object, a concept, or any entity you want to organize into data and behavior.
- A class bundles data and methods for an object together

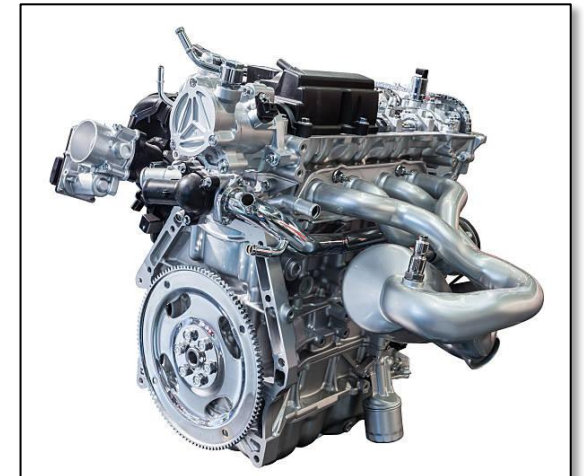
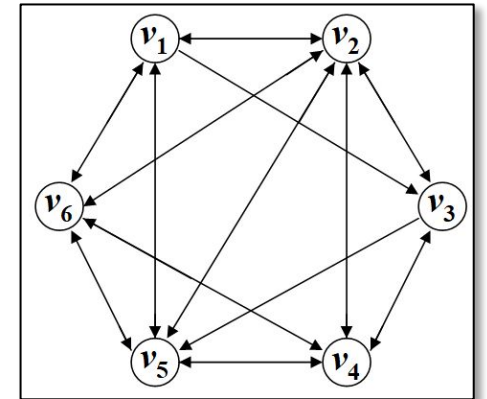
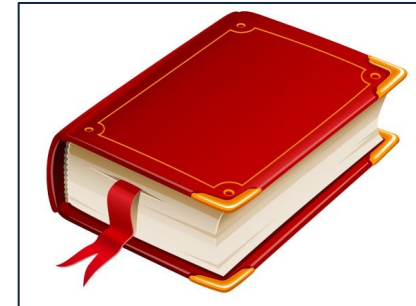
A class as a concept in a program

- You can think of classes as a **real life concept**
 - Car
 - Engine
 - Video Game Character
 - Vectors
 - Graph
 - Book
 - Dog

`std::vector<int>`



$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$



What questions do you have?



bjarne_about_to_raise_hand

A Recap on Classes

A Point on classes

```
class Point {  
public:  
    Point(int x, int y);  
    ~Point();  
    int getX();  
    int getY();  
    void setX();  
    void setY();  
  
private:  
    int x;  
    int y;  
};
```

public:

Accessible by everyone!

constructor

Initializes this class

destructor

Cleans up this class
Usually don't need this

private:

Only visible to us! Implementation details

A Point on classes

```
class Point {  
public:  
    Point(int x, int y);  
    ~Point();  
    int getX();  
    int getY();  
  
private:  
    int x;  
    int y;  
    std::string color;  
}
```

Point.h (header file)

Contains **interface, declarations**

A Point on classes

```
class Point {  
public:  
    Point(int x, int y);  
    ~Point();  
    int getX();  
    int getY();  
  
private:  
    int x;  
    int y;  
    std::string color;  
}
```

Point.h (header file)

Contains **interface, declarations**

```
#include "Point.h"  
  
Point::Point(int x, int y)  
    : x(x), y(y) {}  
  
int Point::getX() {  
    return x;  
}  
  
int Point::getY() {  
    return y;  
}
```

Point.cpp

Contains **implementation, definitions**

Let's try something out...



practice code

Classes: Python vs. C++

```
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
```

```
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; }
```

What does a **Point** look like in memory?

Classes: Memory Layout (Python)

```
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)

p



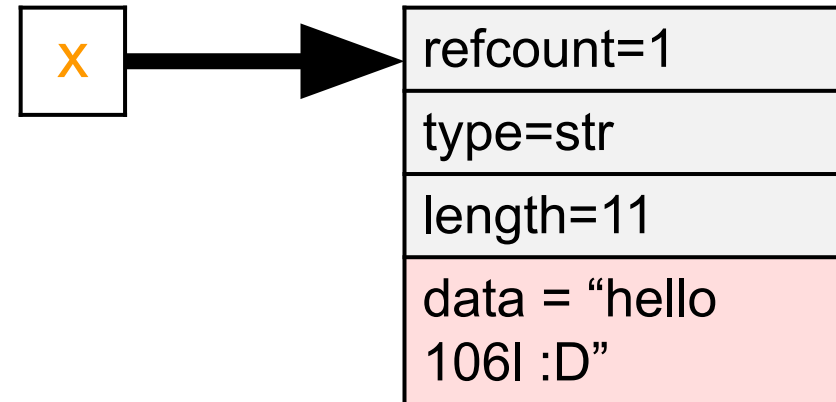
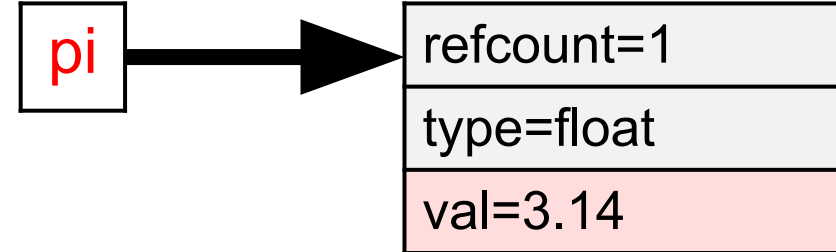
refcount
type
_x = 1
_y = 2

Python **stores extra information** about the type of the object in its memory footprint! This enables runtime type checking.

Python Memory Layout - Examples (Python)

`pi = 3.14`

`x = "hello 106l :D"`



All types in Python are considered classes.

Python Memory Layout - Examples (Python)

```
y = 42
```

```
z = y
```

y

z

refcount=2

type=int

digits=[42]

refcount increases to 2 since two variables are referencing it now

Classes: Memory Layout (Python)

```
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)

p

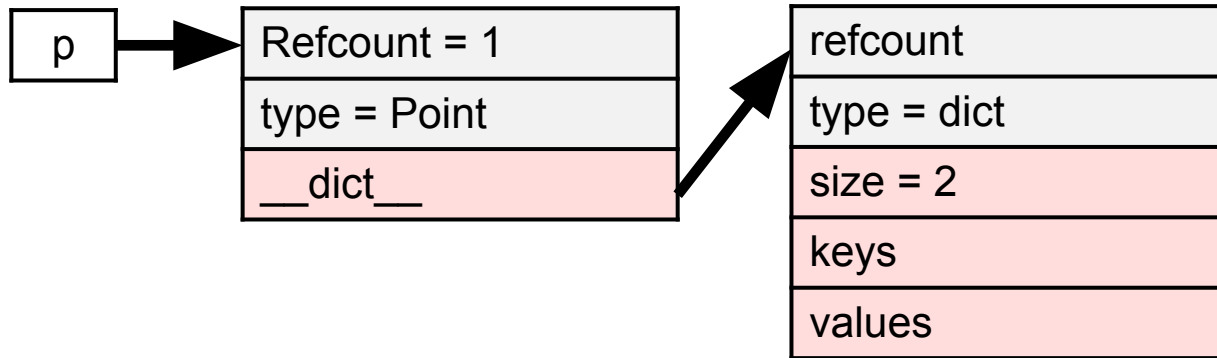


refcount
type
_x = 1
_y = 2

Python **stores extra information** about the type of the object in its memory footprint! This enables runtime type checking.

It's actually *much* worse than this!

```
p = Point(1, 2)
```



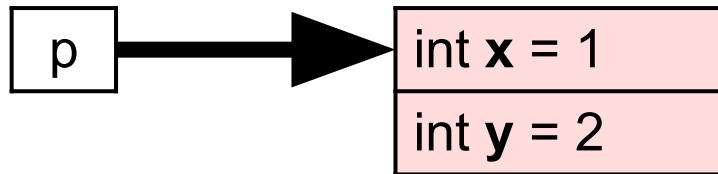
refcount = 1	refcount = 1
type = str	type = str
length = 2	length = 2
kind = 1	kind = 1
data = "_x"	data = "_y"

refcount = 1	refcount = 1
type = int	type = int
size = 1	size = 1
digits = [1]	digits = [2]

To offer the flexibility of dynamic typing, **Python stores a lot of extra info!**

Classes: Memory Layout (C++)

```
Point p {1, 2};
```



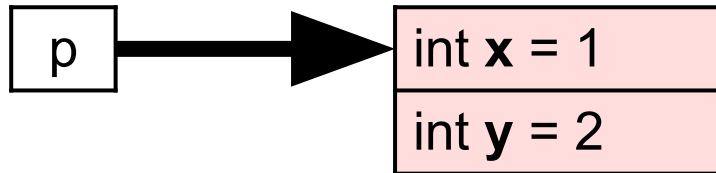
C++ **just stores the data** in the object! The compiler does all of the type checking at compile time!

```
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; }  
};
```

Classes: Memory Layout

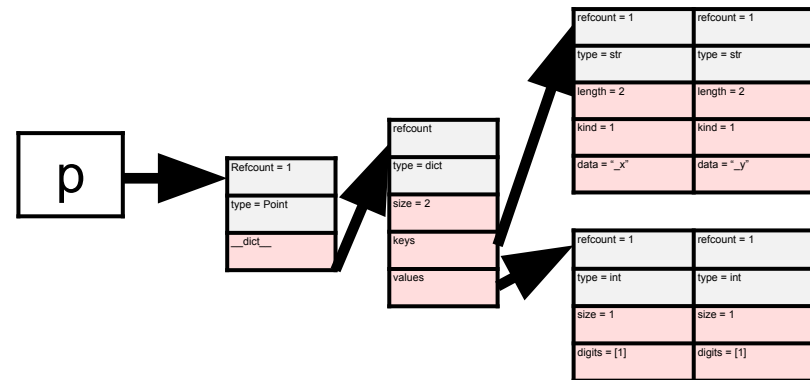
C++

```
Point p {1, 2};
```



Python

```
p = Point(1, 2)
```



C++ **stores less data** in classes! This is one reason why C++ is more memory-efficient than Python

What questions do you have?



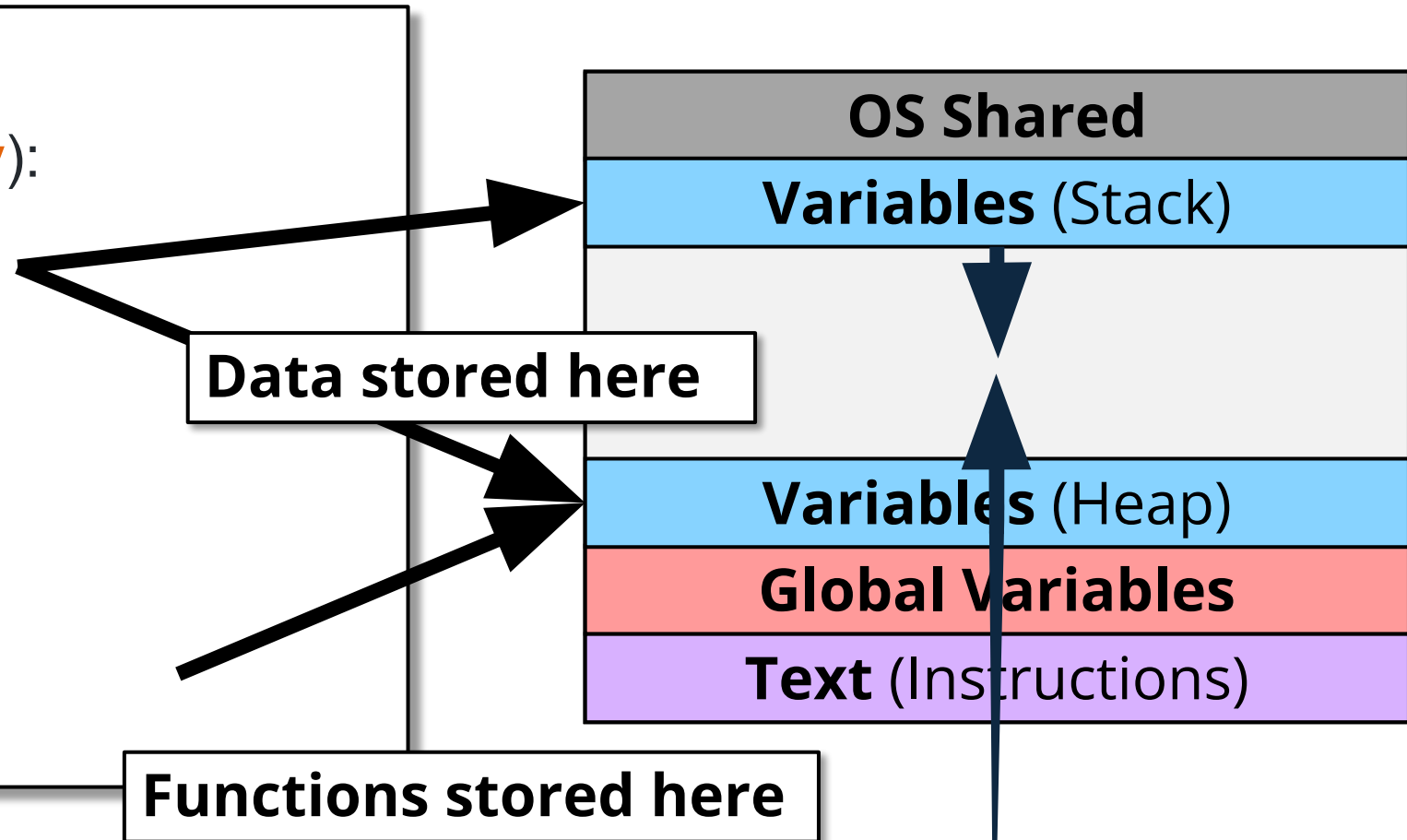
bjarne_about_to_raise_hand

Where are the functions?

Where are the functions in Python?

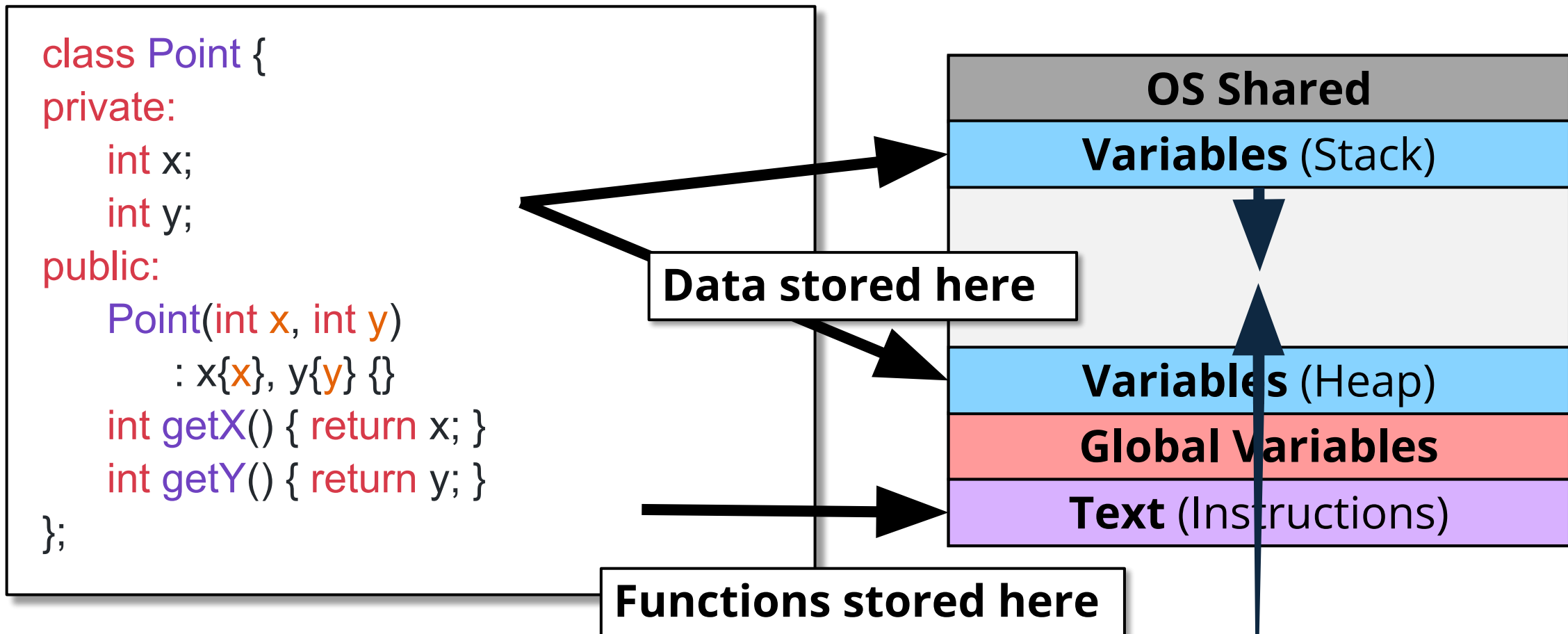
Functions are not stored in the object dictionary itself, but separately in a dictionary associated with the class.

```
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
```



Where are the functions in C++?

Functions are not stored in the object itself, but separately



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...
```

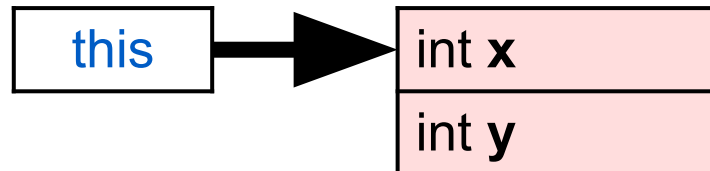
```
p = Point(1, 2)
px = Point.getX(p)
```

Passing **self** as parameter

this in C++

this is a pointer to the current class

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



The importance of **this**

Are these two snippets of code the same?

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

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

 These are the same

The importance of **this**

Are these two snippets of code the same?

```
void Point::setX(int x)
{
    x = x;
}
```

```
void Point::setX(int x)
{
    this->x = x;
}
```

✗ Not the same

What is **this**?

```
int Point::setX(int x)
{
    this->x = x;
}
```

Point* this

->

Mwahahaha pointer dereference

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

What questions do you have?



bjarne_about_to_raise_hand

Inheritance

**A mechanism for one class to inherit properties
from another**

Many kinds of cars

Car



Many kinds of cars, but they all inherit

- An engine
- Wheels
- A steering wheel

Toyota Camry



Honda Civic



Ford Mustang

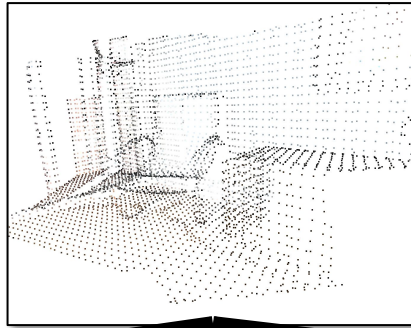


Jeep Gladiator



Many kinds of shapes

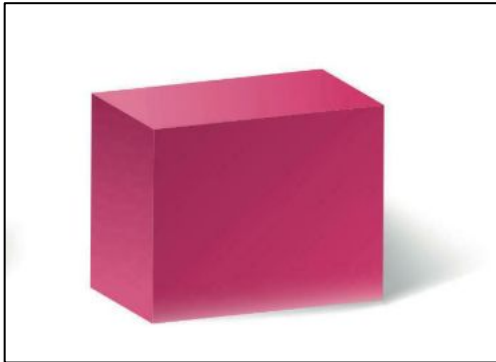
Shape



Every shape has a

- Volume
- Surface area

Box



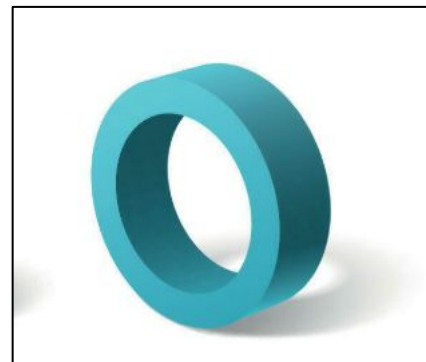
Sphere



Cone



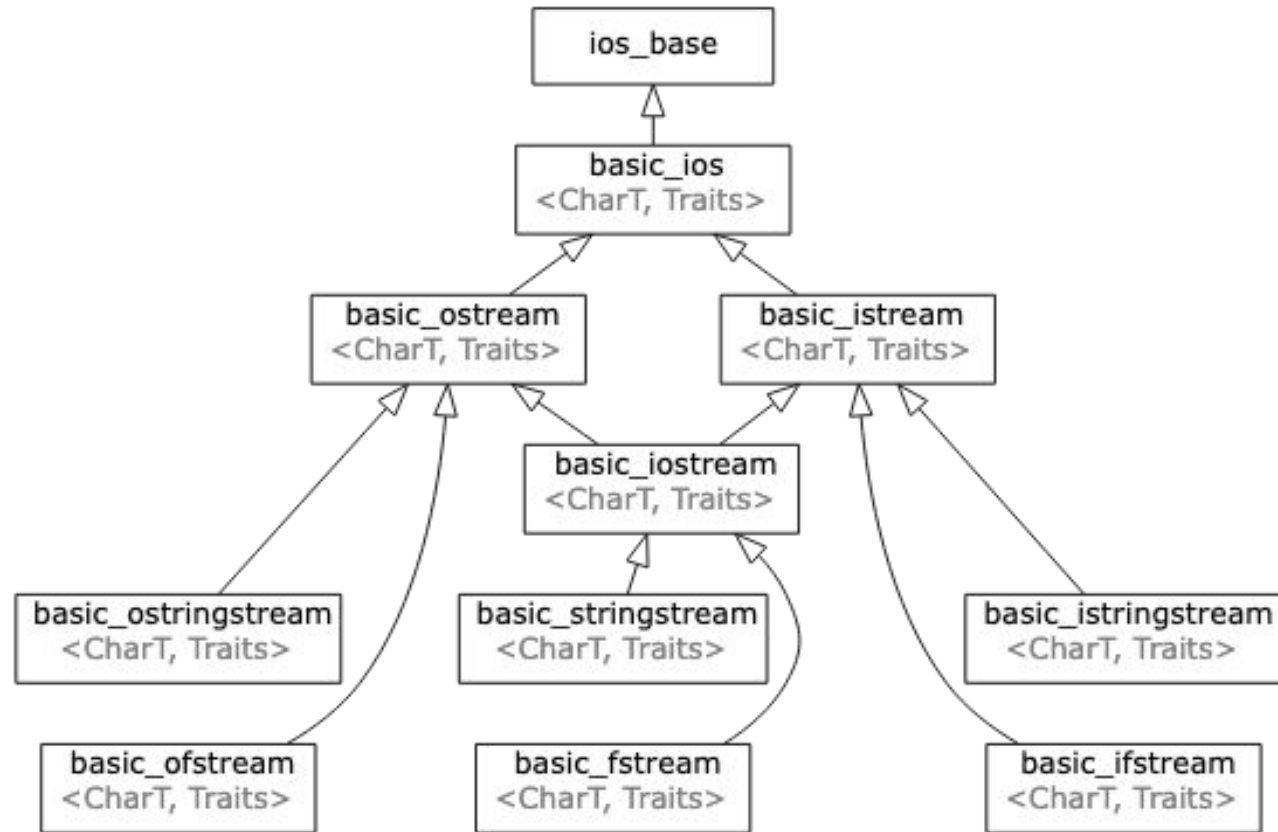
Ring



Buckyball

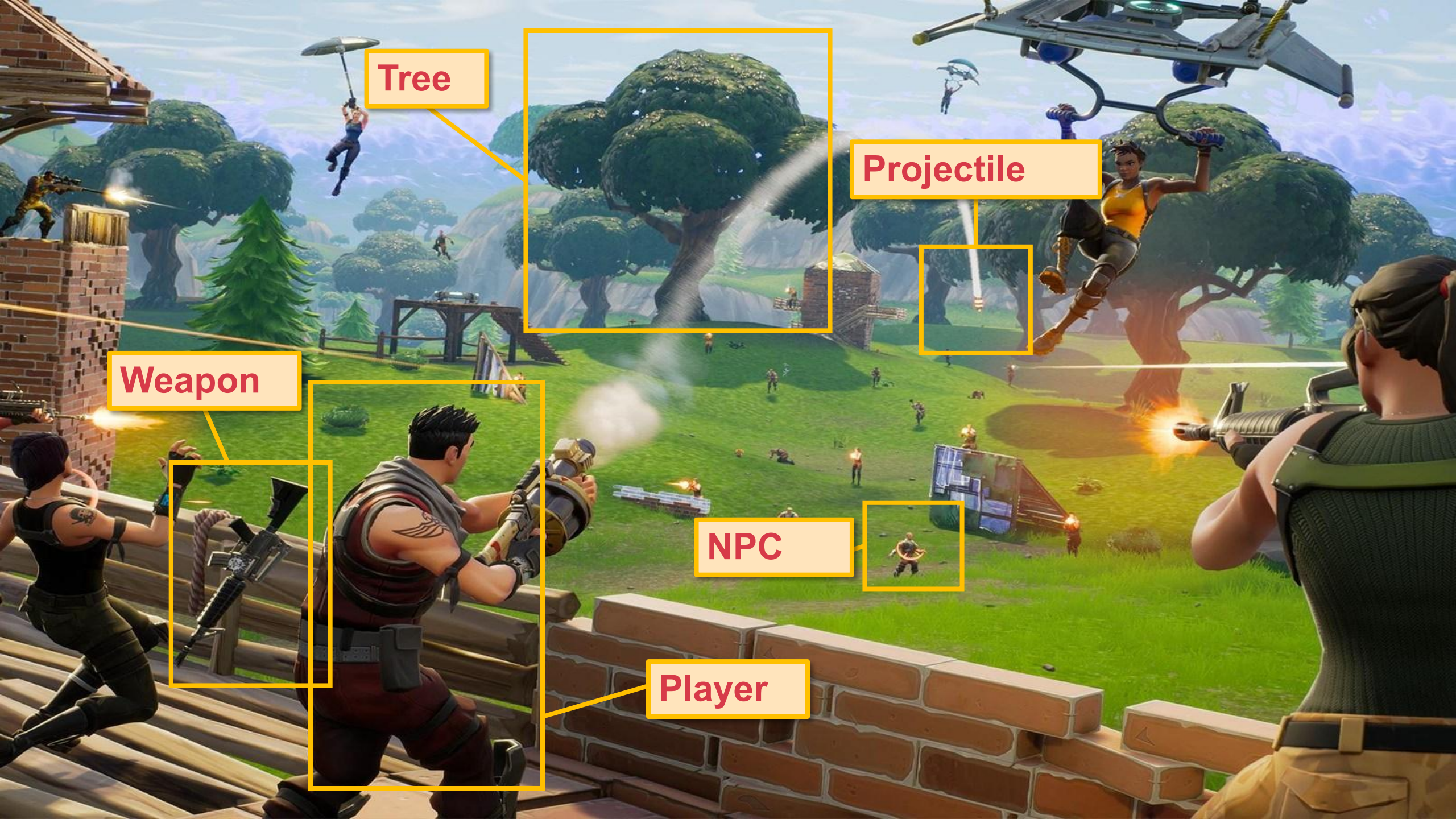


We've seen this before: streams!



Is-A relationship: An `std::ifstream` is a `std::istream` is a `std::ios`

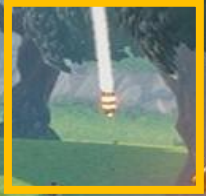
How do we model inheritance in C++?



Tree



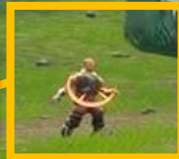
Projectile



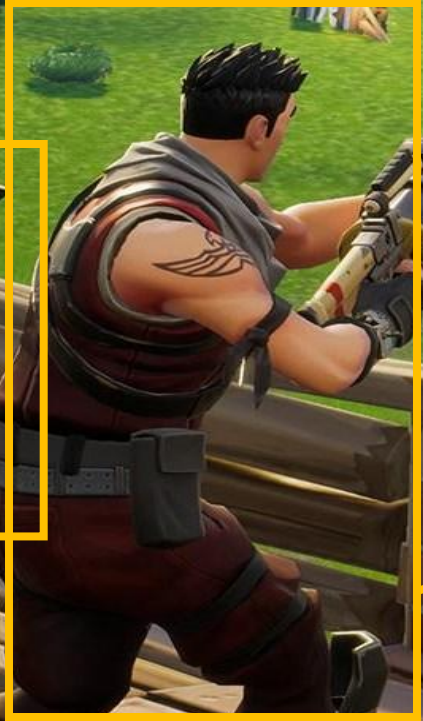
Weapon



NPC



Player

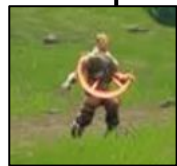
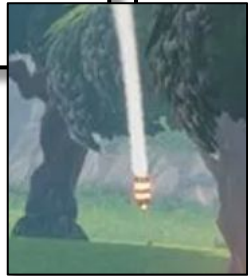


Fortnite as classes

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

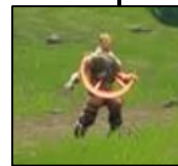
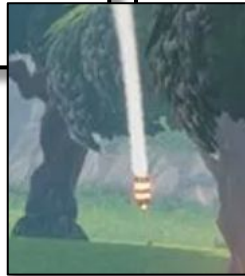


There's a lot of redundancy here!

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



This model is also a pain to modify

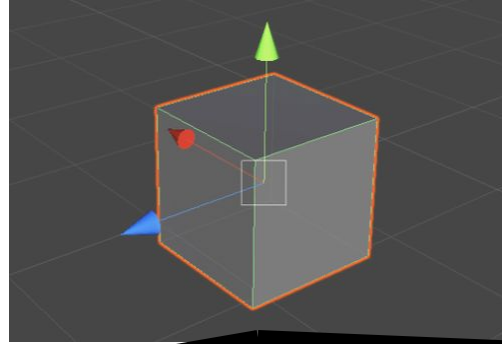
Imagine we wanted to add an **overlapsWith** method to each object that checks if it overlaps in space with another object

```
class Player {  
    /* ... */  
public:  
    bool overlapsWith(const Player& other);  
    bool overlapsWith(const NPC& other);  
    bool overlapsWith(const Tree& other);  
    bool overlapsWith(const Projectile& other);  
    bool overlapsWith(const Weapon& other);  
};  
  
// And we'd do the same for NPC, Tree, Projectile, Weapon!
```

This doesn't scale!
What if we add more object
types!

Taking a step back

Entity



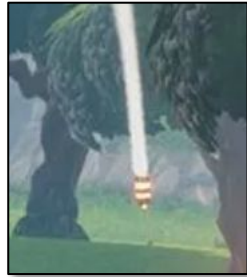
Many different objects, but they all share:

- A position in space
 - A hitbox
- An update and render method!

Player



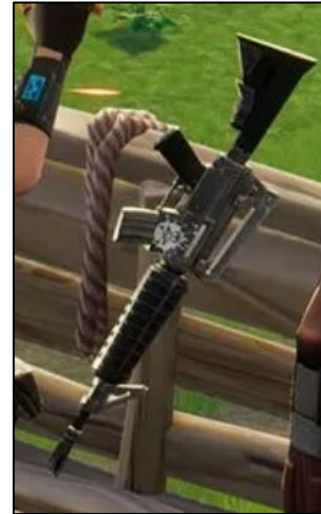
Projectile



NPC



Weapon

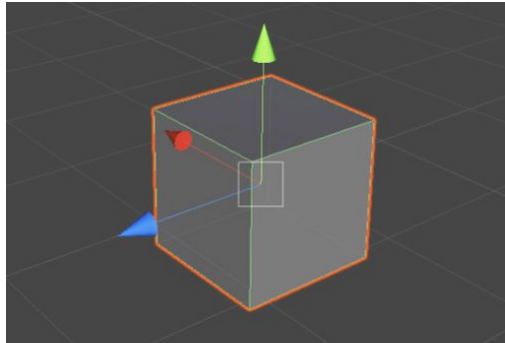


Tree



Introducing a common base class: **Entity**

Entity



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

Introducing a common base class: **Entity**

```
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 Entity {  
    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();  
};
```

Now we inherit!

```
class Player : Entity {  
    double hitpoints;  
public:  
    void damage(double hp);  
};
```

```
class Projectile : Entity  
{  
    double vx, vy, vz;  
};
```

```
class Weapon : Entity  
{  
    size_t ammo;  
public:  
    void fire();  
};
```

```
class Tree : Entity  
{  
};
```

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

```
class NPC : Entity {  
    double hitpoints;  
public:  
    void damage(double hp);  
};
```



Now we inherit!

```
class Player : Entity {  
    double hitpoints;  
public:  
    void damage(double hp);  
};
```

```
class Projectile : Entity  
{  
    double vx, vy, vz;  
};
```

```
class Weapon : Entity  
{  
    size_t ammo;  
public:  
    void fire();  
};
```

```
class Tree : Entity  
{  
};
```

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

```
class NPC : Entity {  
    double hitpoints;  
public:  
    void damage(double hp);  
};
```

Notice: there is still some redundancy!

More layers of inheritance...

```
class Player : Actor {};
```

```
class Projectile : Entity  
{  
    double vx, vy, vz;  
};
```

```
class Weapon : Entity  
{  
    size_t ammo;  
public:  
    void fire();
```

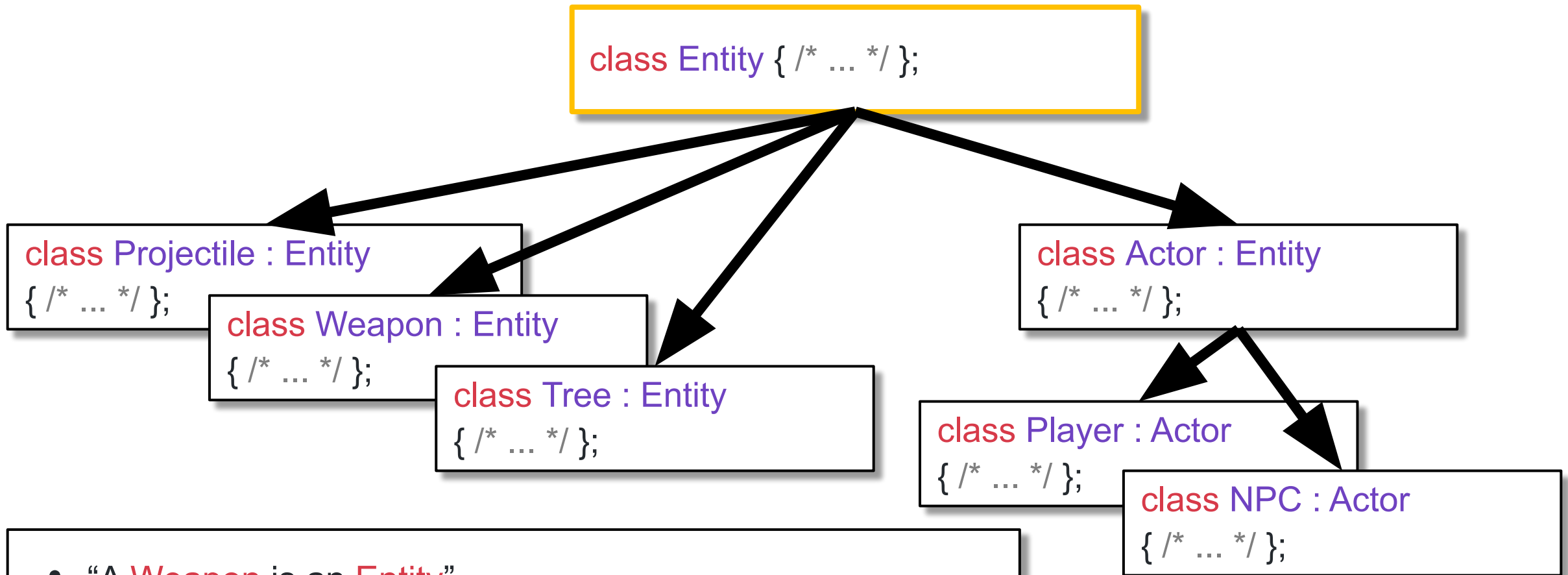
```
class Tree : Entity  
{
```

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

```
class Actor : Entity  
{  
    double hitpoints;  
public:  
    void damage(double hp);  
};
```

```
class NPC : Actor {};
```

An inheritance tree defines is-a relationships



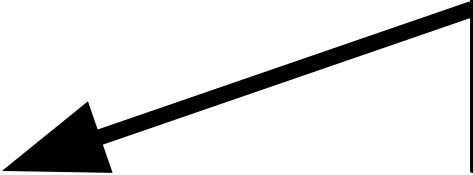
- “A **Weapon** is an **Entity**”
- “An **NPC** is an **Actor**, and is also an **Entity**”

Defining common functionality is trivial!

Now we can check if two **Entity**'s overlap, no matter what kind it is!

```
class Entity {  
    double x, y, z;  
    HitBox hitbox;  
  
public:  
    void update();  
    void render();  
    bool overlapsWith(const Entity& other);  
};
```

```
Player player { /* ... */ };  
Projectile bullet { /* ... */ };  
bool isHit = player.overlapsWith(bullet);
```



To implement, check **x,y,z** and **hitbox** to see if there was an overlap

What questions do you have?



bjarne_about_to_raise_hand

Note: access modifiers

That last line won't actually work due to access modifiers!

```
Player player { /* ... */ };  
Projectile bullet { /* ... */ };  
bool isHit = player.overlapsWith(bullet);
```



Compiler: overlapsWith is
inaccessible

Note: access modifiers

By default, classes are inherited privately.

```
class Entity {  
public:  
    bool overlapsWith(const Entity& other);  
};  
  
class Player : /* private */ Entity {  
    // Private inheritance:  
    // - private members of Entity are inaccessible to all  
    // - public members become private (inaccessible to outside)  
};
```

Note: access modifiers

We can fix this issue by inheriting `Entity` publicly!

```
class Entity {  
public:  
    bool overlapsWith(const Entity& other);  
};  
  
class Player : public Entity {  
    // Public inheritance:  
    // - private members of Entity are still inaccessible  
    // - public members become public (accessible to outside)  
};
```

Note: access modifiers

private inheritance (default)

```
class Child : private Parent
```

private in parent

public in parent

inaccessible in child

private in child

public inheritance

```
class Child : public Parent
```

private in parent

public in parent

inaccessible in child

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

Note: **protected** access modifier

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

- Remember, class members are *private* by default

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

We need to mark them
protected inside **Entity**

private by default

```
class Projectile  
    Entity {  
private:  
    double vx, vy, vz;  
public:  
    void move() {  
        x += vx;  
        y += vy;  
        z += vz;  
    }  
};
```

In order to access **x**, **y**, and **z**
inside **Projectile**

What questions do you have?




bjarne_about_to_raise_hand

Let's build a game!

Let's build a game!

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



We'll implement the logic for our game by overriding **update** and **render** for each kind of **Entity**.

Let's build a game!

Let's **override** the **update** and **render** function for each **Entity** type!

```
void Player::update() {  
    // Handle controller input  
}
```

```
void Projectile::update() {  
    // Move the projectile  
}
```

⋮

```
// By default, do nothing!  
void Entity::update() {}
```

```
void Player::render() {  
    // Draw the player!  
}
```

```
void Projectile::render() {  
    // Cool particle effects!  
}
```

⋮

```
// By default, do nothing!  
void Entity::render() {}
```

Let's build a game!

A game is basically a collection of entities updated and rendered every frame!

```
int main() {  
    std::vector<Entity> entities { Player(), Tree(), Projectile() };  
    while (true) {  
        for (auto& entity : entities) {  
            entity.update();  
            entity.render();  
        }  
    }  
}
```

**Game event
loop (runs every
frame)**

Let's try it out!

It didn't work! What's going on!?

Behind the scenes

Recall that C++ lays out the fields of an object sequentially

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

Entity

double x
double y
double z
HitBox hitbox

Behind the scenes

C++ stacks the subclass's members below the inherited ones!

```
class Projectile
    : public Entity {
private:
    double vx, vy, vz;
public:
    void move();
}
```

Projectile

double x
double y
double z
HitBox hitbox
double vx
double vy
double vz

**Members
inherited
from Entity**

Behind the scenes

Be careful: when you assign a derived class to a base class, it gets sliced!

Projectile

double x
double y
double z
HitBox hitbox
double vx
double vy
double vz

std::vector<Entity>

double x	double x	double x
double y	double y	double y
double z	double z	double z
HitBox hitbox	HitBox hitbox	HitBox hitbox

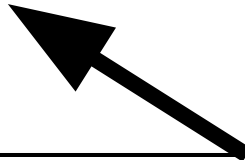
Issue: every element in the vector is an **Entity**, so the compiler calls **Entity::update()** (which does nothing) instead of **Player::update()**, **Tree::update()**, **Projectile::update**, etc.

A Projectile doesn't "fit" into an Entity

Solution: Use an **Entity*** instead

Object slicing only happens when a copy is made!

```
int main() {  
    Player p; Tree t; Projectile b;  
  
    std::vector<Entity*> entities { &p, &t, &b };  
    while (true) {  
        for (auto& ent : entities) {  
            ent->update();  
            ent->render();  
        }  
    }  
}
```

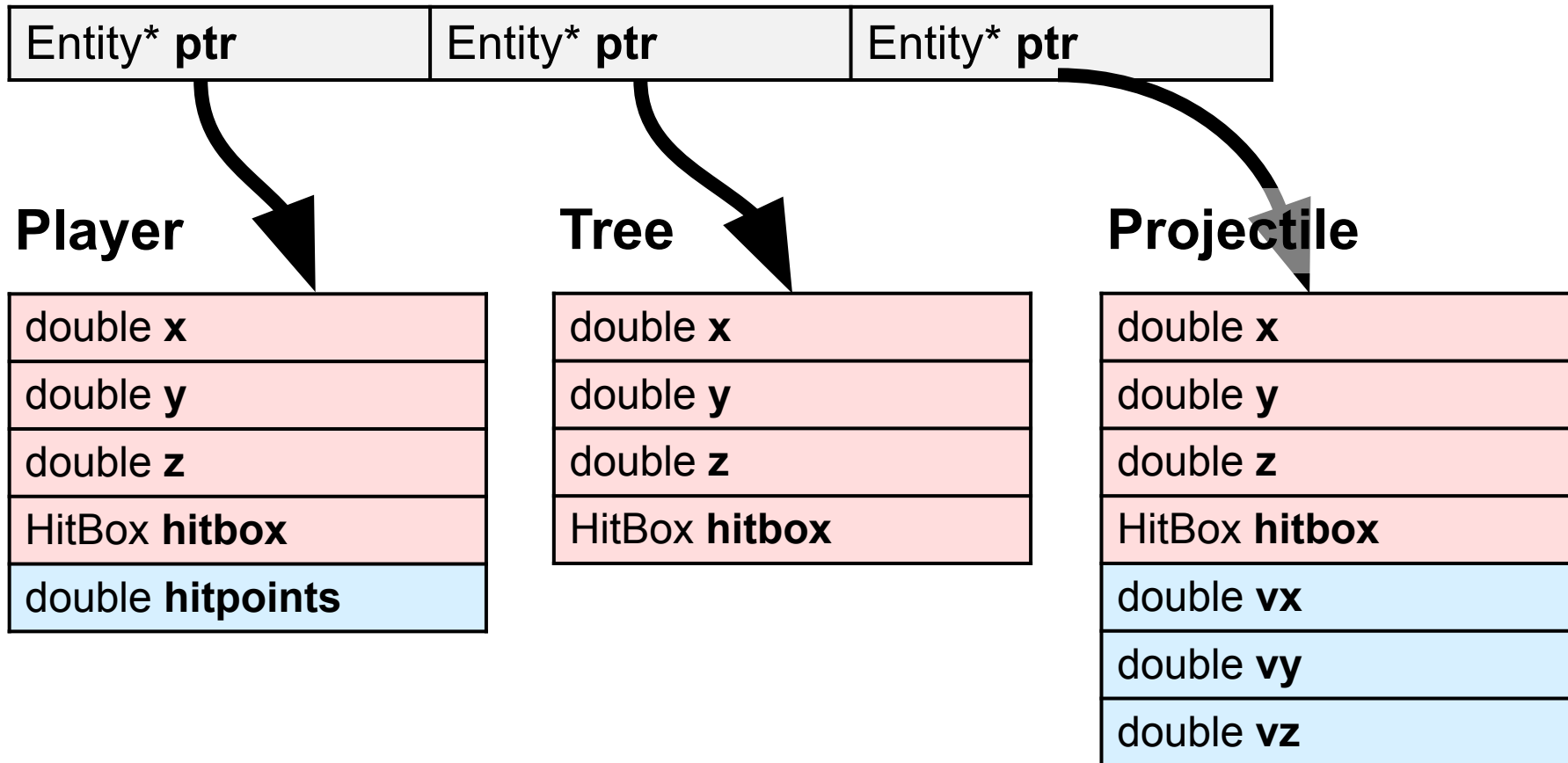


Storing pointers to entities here,
not the entities themselves!

Solution: Use an **Entity*** instead

Pointers retain the details of the subclass by avoiding copies

`std::vector<Entity*>`



What questions do you have?



bjarne_about_to_raise_hand

Let's try it again!

It still didn't work..   

Announcements

- Assignment 1 due this Friday
- A note on assignment workload!
 - We want you guys to spend ~1 hour on each assignment!
 - If you consistently spend more time than this, come to OH!
- OH is 9:30-10:30am Mon, 3-4pm Friday... we are so lonely, please come!!

Virtual Functions

Problem: Which one is called?

- We have many different **update** methods

```
void Projectile::update();
```

```
void NPC::update();
```

```
void Player::update();
```

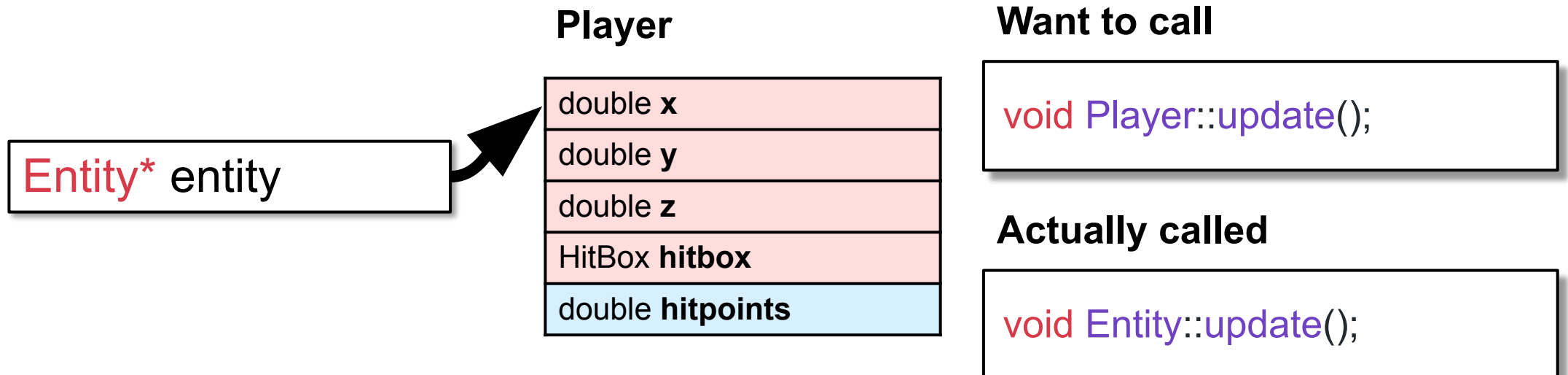
```
void Entity::update();
```

- Given a pointer to an **Entity**, how does the compiler know which method to call?

```
Entity* entity = entities[0];  
entity->update();
```

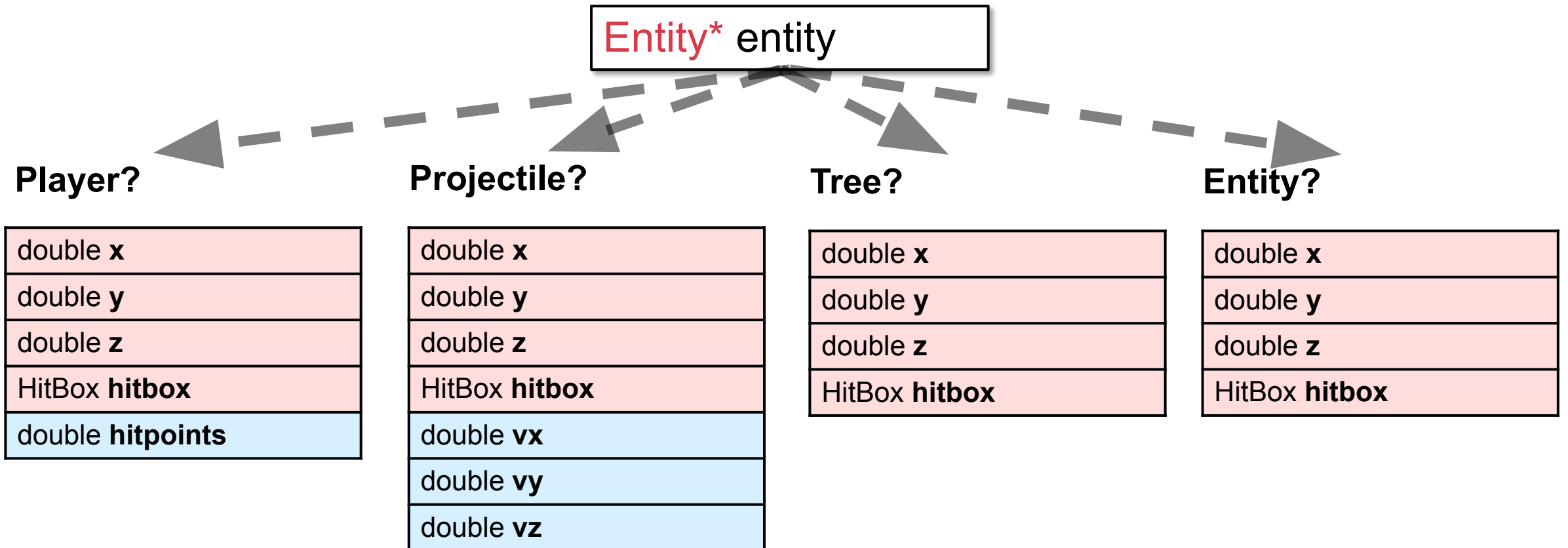
Problem: Which one is called?

- 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!



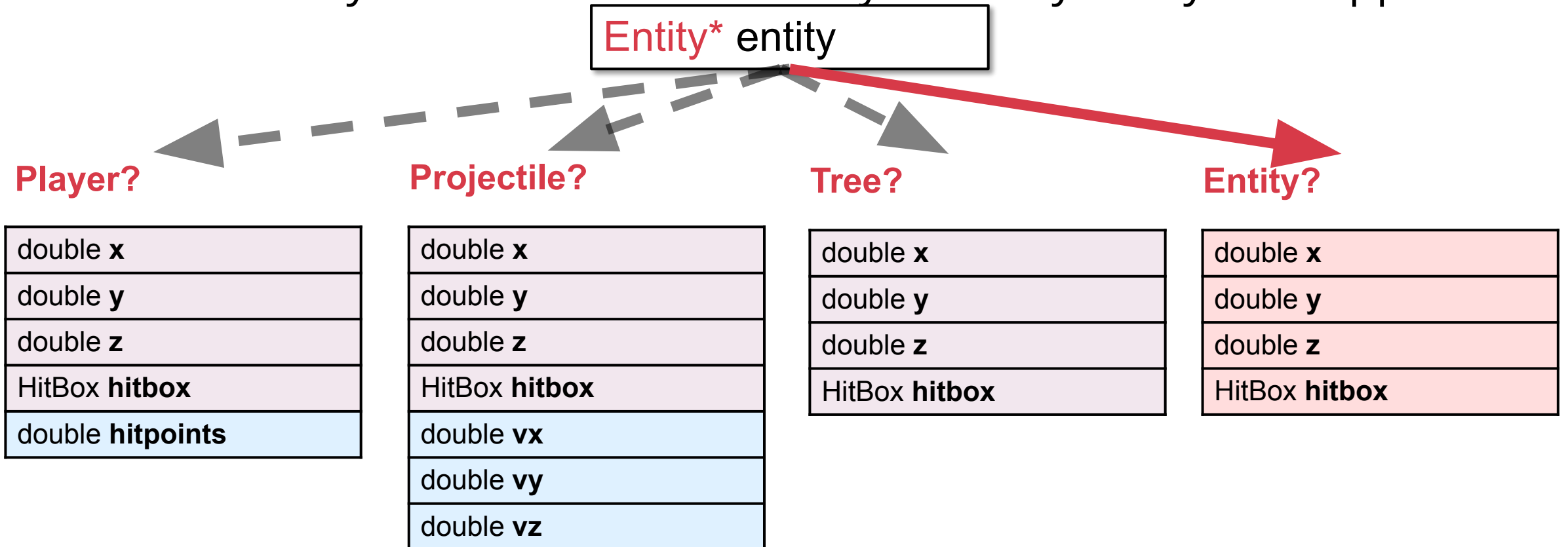
Problem: Which one is called?

- Given an **Entity***, does it point to:



Problem: Which one is called?

- The compiler defaults to assuming `entity` points to an `Entity`
- This is the only one it can be absolutely sure any entity will support



Using **Entity*** comes at a cost:

We **“forget”** which type the object actually is

This is not what we wanted!

- Notice: there is a difference between the **compile-time** vs. **runtime** type of the object!
 - At compile time, it is treated as an **Entity**
 - At runtime, it could be an **Entity** or any subclass, e.g. **Projectile**, **Player**, etc.
- What we need is **dynamic dispatch**
- Depending on the runtime (dynamic) type of the object, a different method should be called (dispatched)!

What questions do you have?



bjarne_about_to_raise_hand

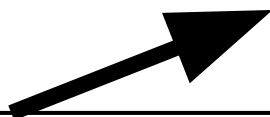
Introducing **virtual** functions

Virtual functions

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

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

```
class Projectile : public Entity {  
public:  
    void update() override {};  
};
```



override isn't required but is good for readability! It will check that you are overriding a **virtual** method instead of creating a new one.

Does it work?

YES!!! 😄

What questions do you have?

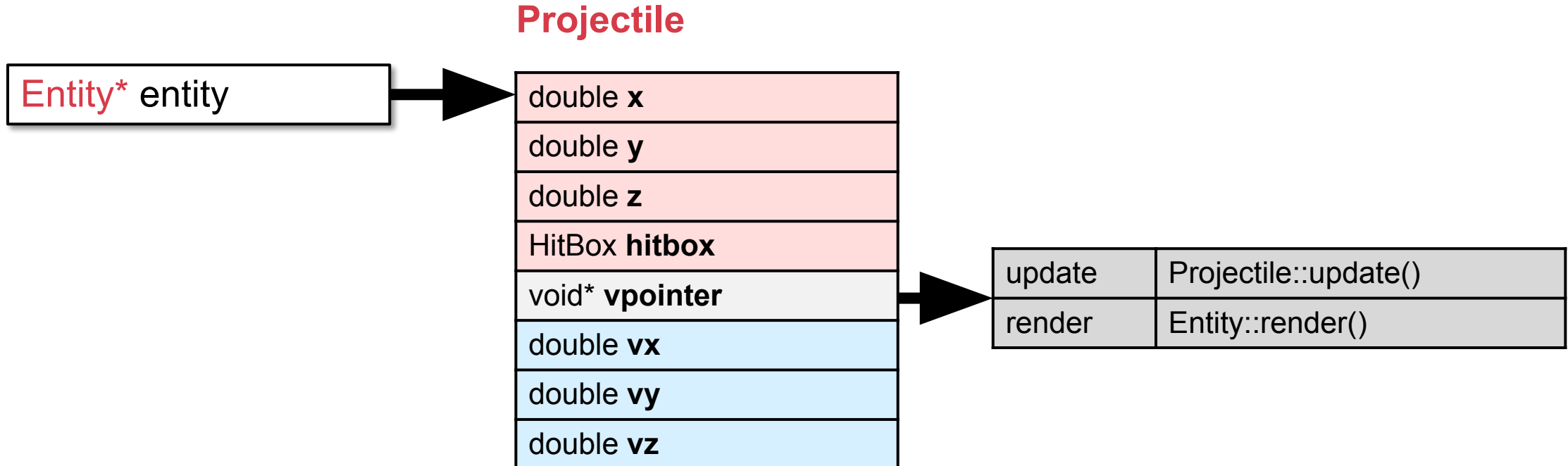


bjarne_about_to_raise_hand

How does it work?

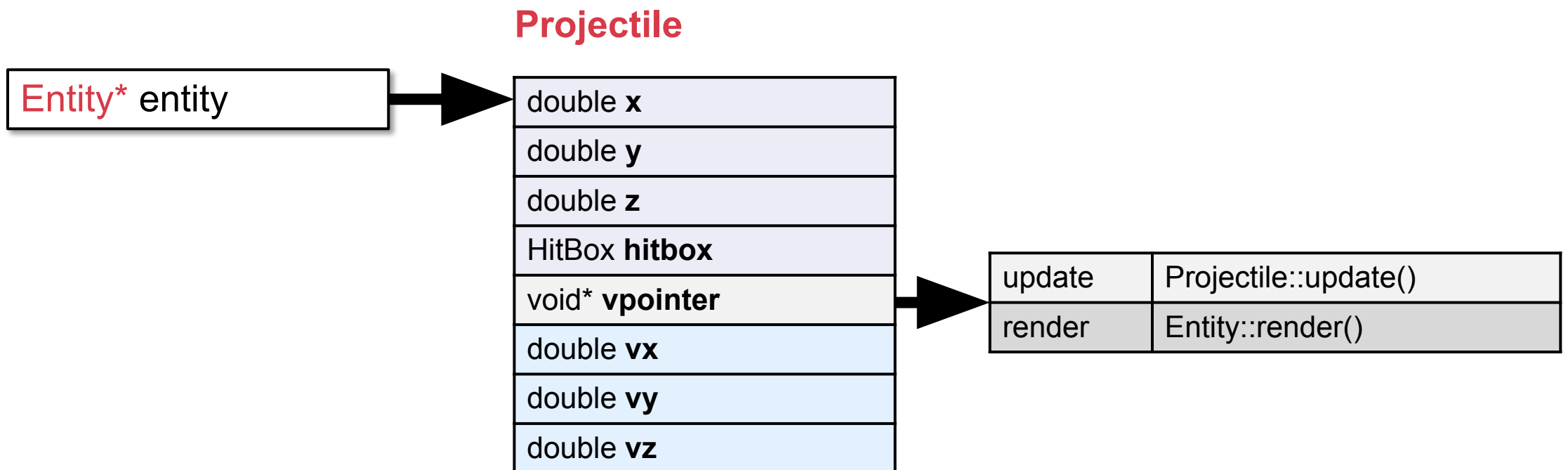
Behind the Scenes

- Adding **virtual** to a function adds some metadata to each object
- Specifically, it adds a pointer (called a **vpointer**) to a table (called a **vtable**) that says, for each **virtual** method, which function should be called for that object



Behind the Scenes

```
Entity* p = new Projectile { /* ... */ };  
p->update();
```

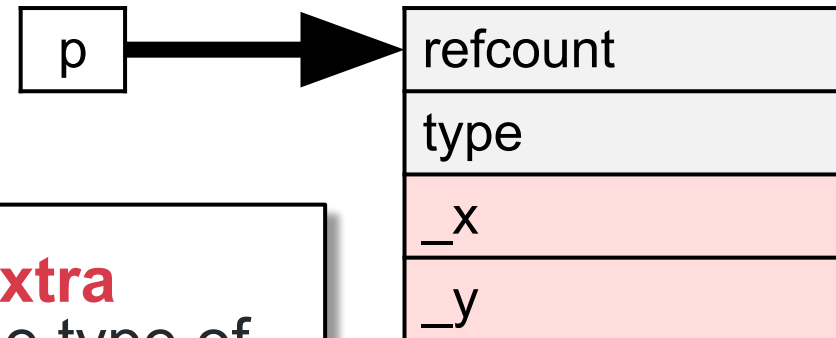


Recall: Classes in Python

```
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
```

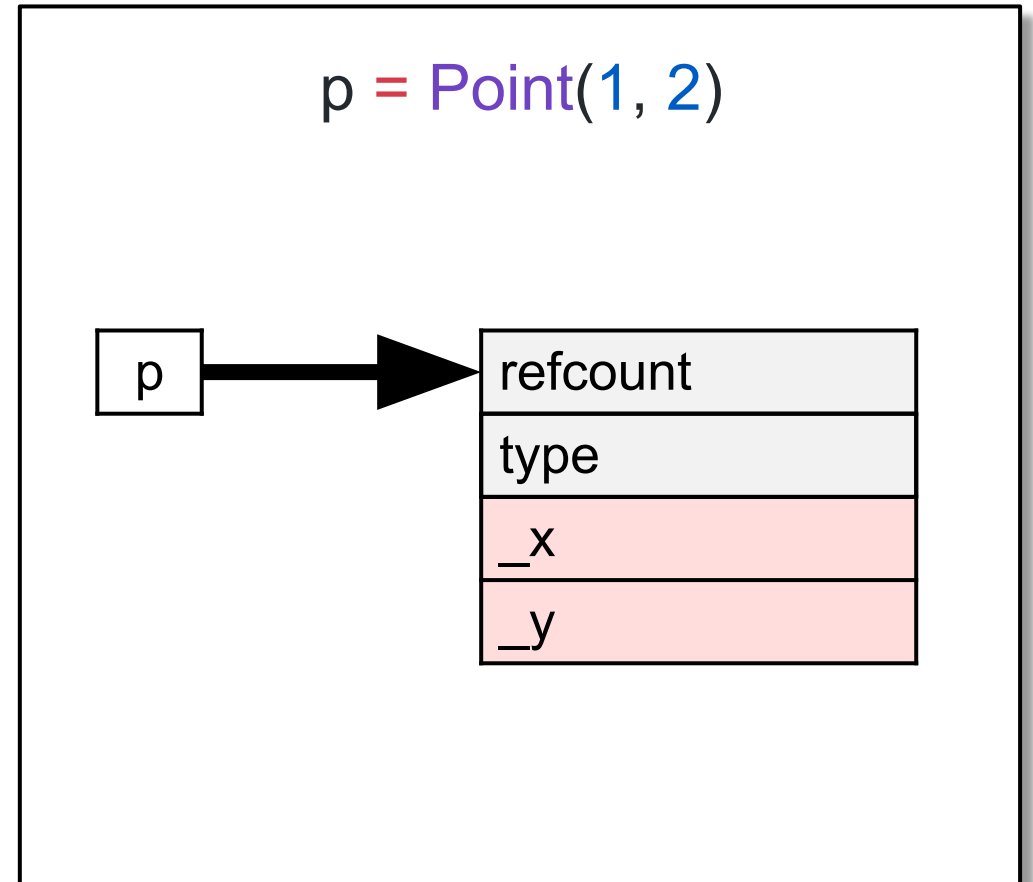
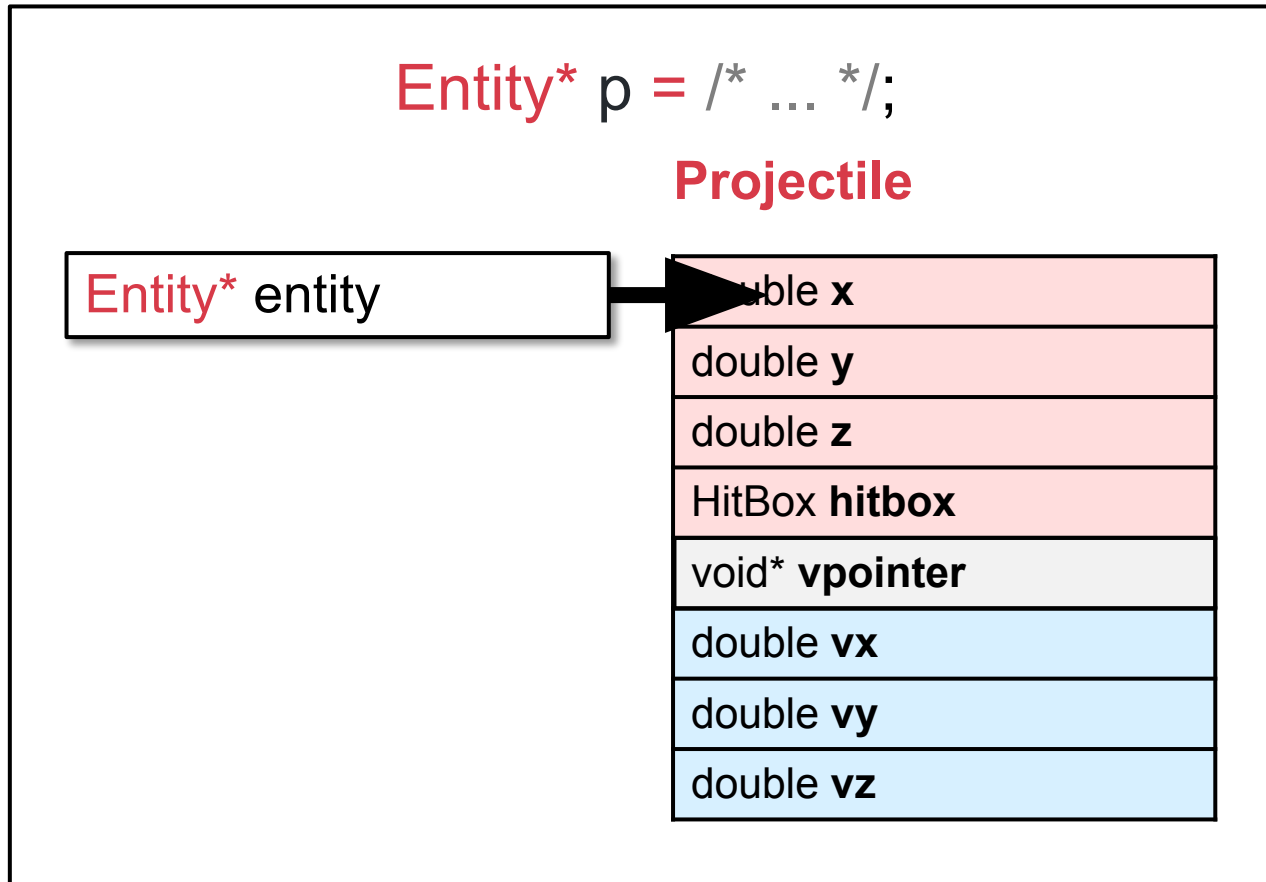
p = Point(1, 2)



Python **stores extra information** about the type of the object in its memory footprint! This enables runtime type checking.

virtual is kind of like Python

Both Python and C++ **virtual** functions store type-specific information



Quick check: pros/cons of **virtual** functions?

- Turn to a partner and talk about:
 - When might you want to use a virtual function?
 - When might you **not** want to use a virtual function?

Quick check: pros/cons of **virtual** functions?

- Turn to a partner and talk about:
 - When might you want to use a virtual function?
 - When might you **not** want to use a virtual function?
- In many other languages, class functions are virtual by default
- **Key idea:** In C++, you have to opt in because they are **more expensive**
 - Increased **size** of memory layout of the class
 - Takes **longer** to look up **vtable** and call the method
- In quant finance and industries where nanoseconds count, virtual functions are not used!

What questions do you have?



bjarne_about_to_raise_hand

Pure virtual functions

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

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!

Pure virtual functions

- A class with one or more pure virtual functions is an **abstract** class, it can't be instantiated!
- Overriding all of the pure virtual functions makes the class **concrete**!

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

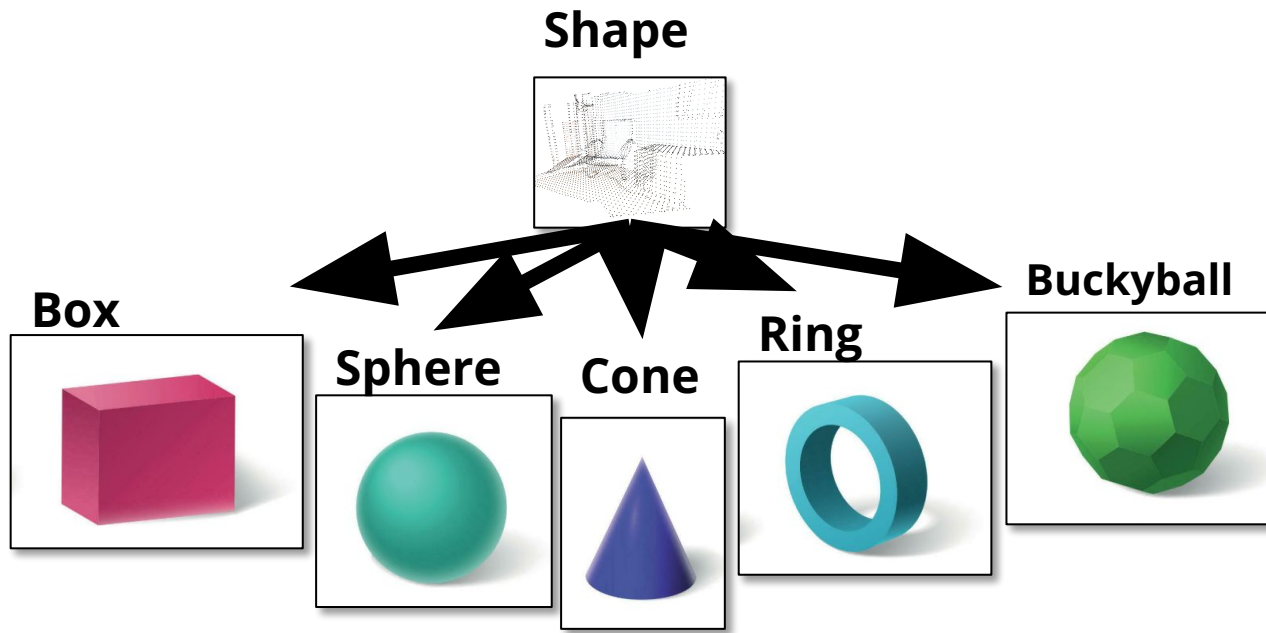
```
Entity e;  
// ❌ Entity is abstract!
```

```
class Projectile  
    : public Entity {  
public:  
    void update() override {};  
    void render() override {};  
};
```

```
Projectile p;  
// ✅ Projectile is concrete
```

Pure virtual functions

Pure virtual functions are useful when there's **no clear default implementation!**



```
class Shape {  
public:  
    virtual double volume() = 0;  
};
```

What's the default volume of a **Shape**? Let's mark it pure virtual and let subclass decide!

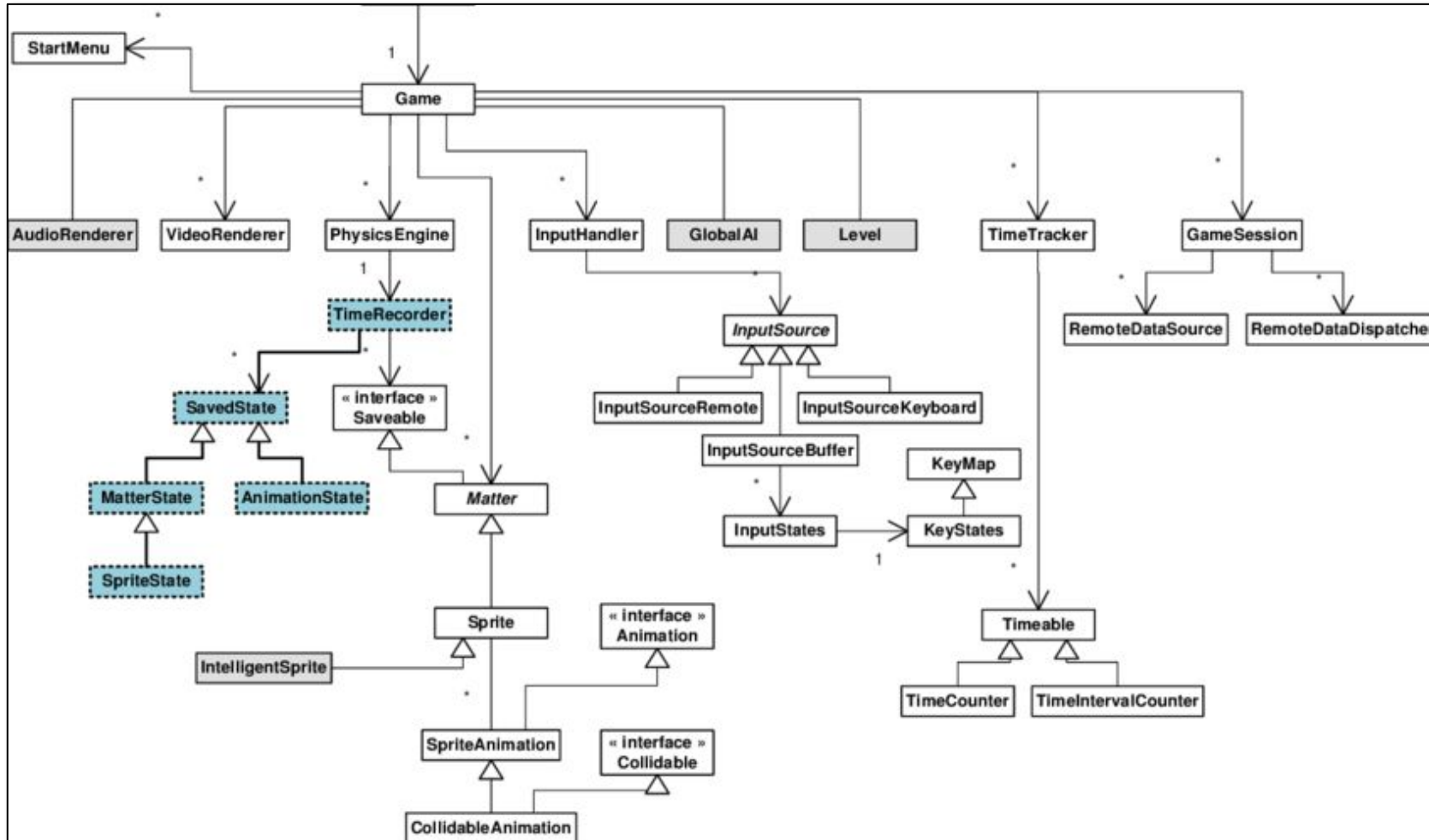
What questions do you have?



bjarne_about_to_raise_hand

Closing Thoughts

Sometimes inheritance can get out of hand

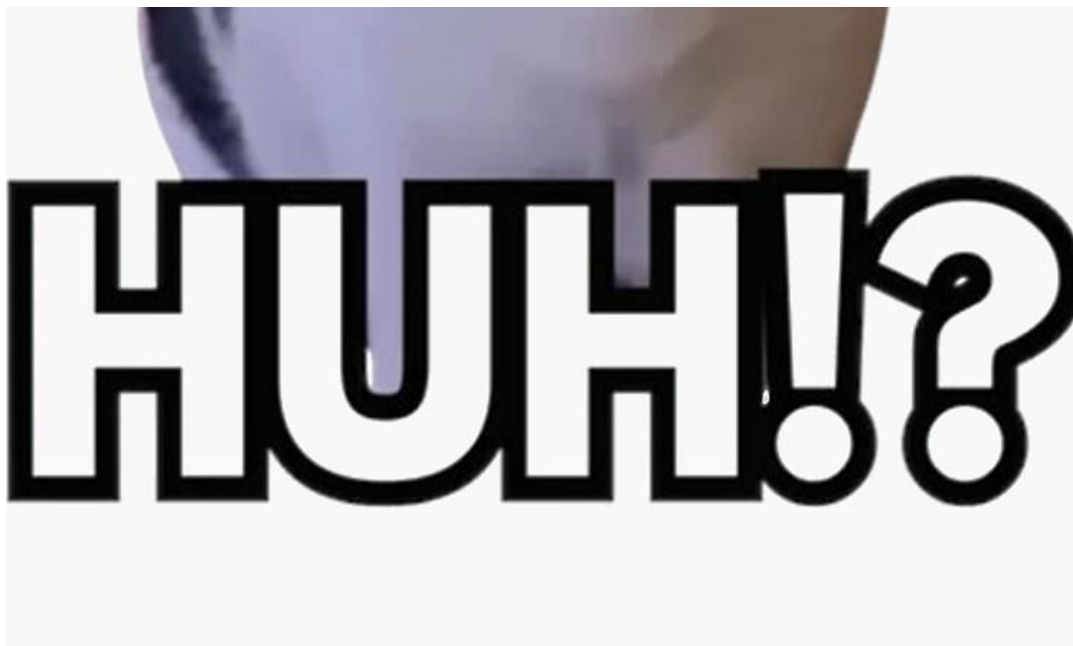


Sometimes inheritance can get out of hand

- 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



“A car is an engine”





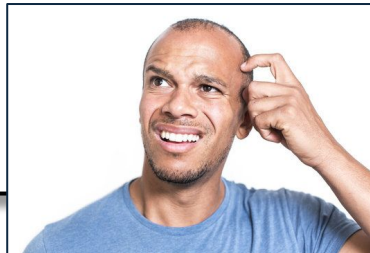
“A car ~~is~~ has an engine”



Prefer composition over inheritance

Inheritance is a powerful tool, but sometimes, composition just makes more sense!

```
class Car
  : public Engine
  , public SteeringWheel
  , public Brakes
{
  /* Hmm... this doesn't seem
  quite right */
};
```



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

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!

What questions do you have?



bjarne_about_to_raise_hand

have a
good
weekend
:D

