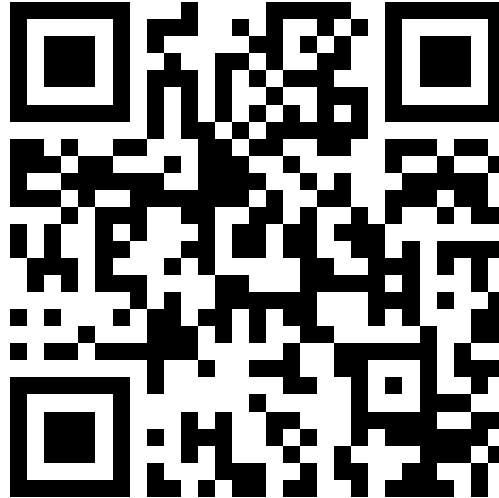


# Lecture 12: Fantastic Friends and Where to Declare Them

Bart Iver van Blokland

# PSA 1 / 2: Reminder: halfway evaluation

- BIG thank you to everyone who has responded already!!



<https://forms.office.com/e/nFrKFB8xG3>

# PSA 2 / 2: Lecture plan

Date	Topic
11.03	Inheritance, friend, static
18.03	<b>GUEST LECTURE</b> Exceptions, testing / git
25.03	C++ libraries, standard library
01.04	Summary
08.04	[tentative] bonus lecture: introduction to parallel computing
15.04	PÅSKE
22.04	PÅSKE
27.04	Looking at projects, project prizes

# Do you remember?

- What is the most important thing you must do when trying to make code run faster?
- What is a way in which you can best utilise the CPU cache?
- What is the difference between building your program in debug and release mode?
- What is an example of a template data type that we have used in the course?
- What is the capacity() of a `std::vector`?



# Black Magic

# C++ Programmer

BRUNDELWALD THREAT:  
IS FLAMBOYANT  
FAWLEY DOING  
ENOUGH?

EDITORIAL

SPEAKING NEWS



MINISTRY OF MAGIC  
TO RESPOND TO GROWING  
PUBLIC FEAR

SPEAKING NEWS

# Trondheim SCHOOL INCREASES SECURITY

E

HEADMASTER CALLS FOR  
EMERGENCY  
MEETING WITH  
PARENTS

F

STUDENTS TO BE  
SENT HOME EARLY



# We are going to..

- Define functions without writing them
- Call functions that do not exist
- Make functions remember the last time they were called
- Make friends (hopefully)

# We are going to..

- Define functions without writing them
- Call functions that do not exist
- Make functions remember the last time they were called
- Make friends (hopefully)

# .. by using:

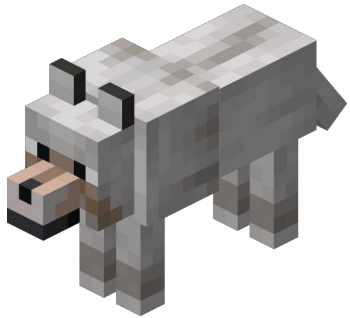
- Inheritance
- Virtual methods
- The static keyword
- The friend keyword

# This week: Inheritance

# What do these mobs have in common?

(in terms of their behaviour or functionality)

Note: many correct answers here!



Wolf



Cat



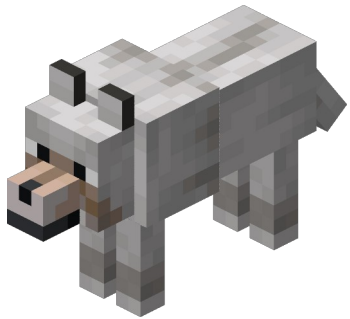
Cave spider



Spider

# What do these mobs have in common?

Can be tamed



Wolf



Cat

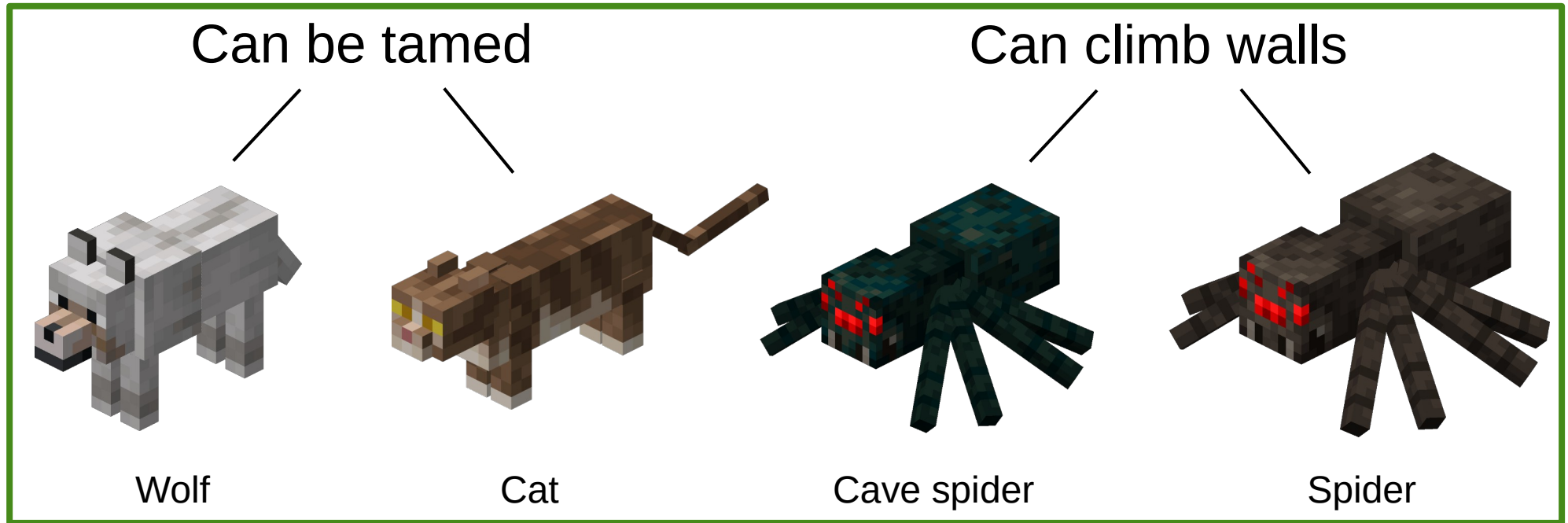


Cave spider



Spider

# What do these mobs have in common?



# What do these mobs have in common?

Follow the player

Can be tamed

Can climb walls



Wolf



Cat



Cave spider



Spider



# Task: build a tree of shared functionality for these



Zombie



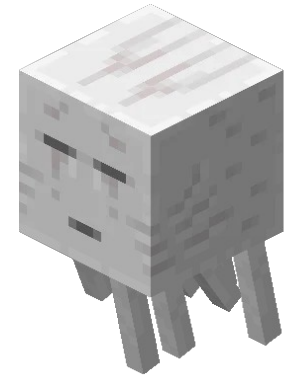
Llama



Cow



Mooshroom



Ghast



Donkey



Skeleton



Horse



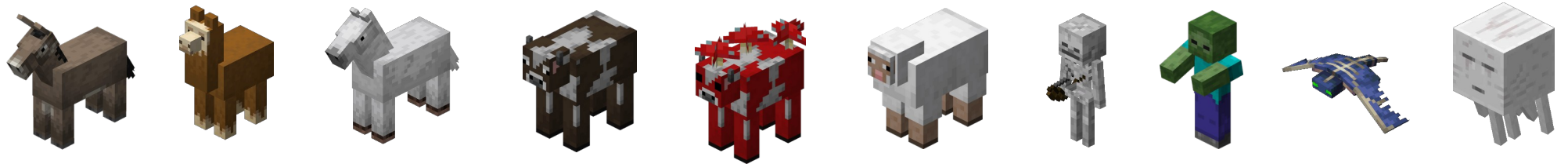
Phantom



Sheep

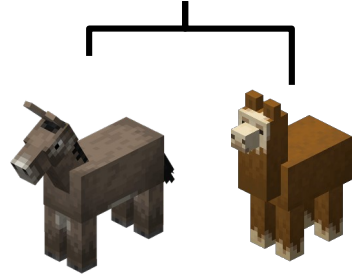


# Structure used by the game

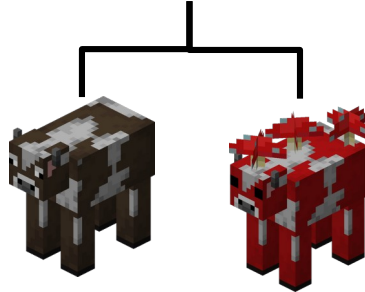


# Structure used by the game

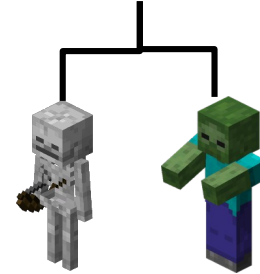
Can carry chest



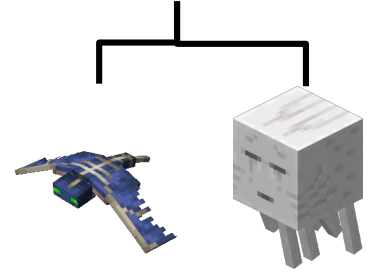
Can be milked



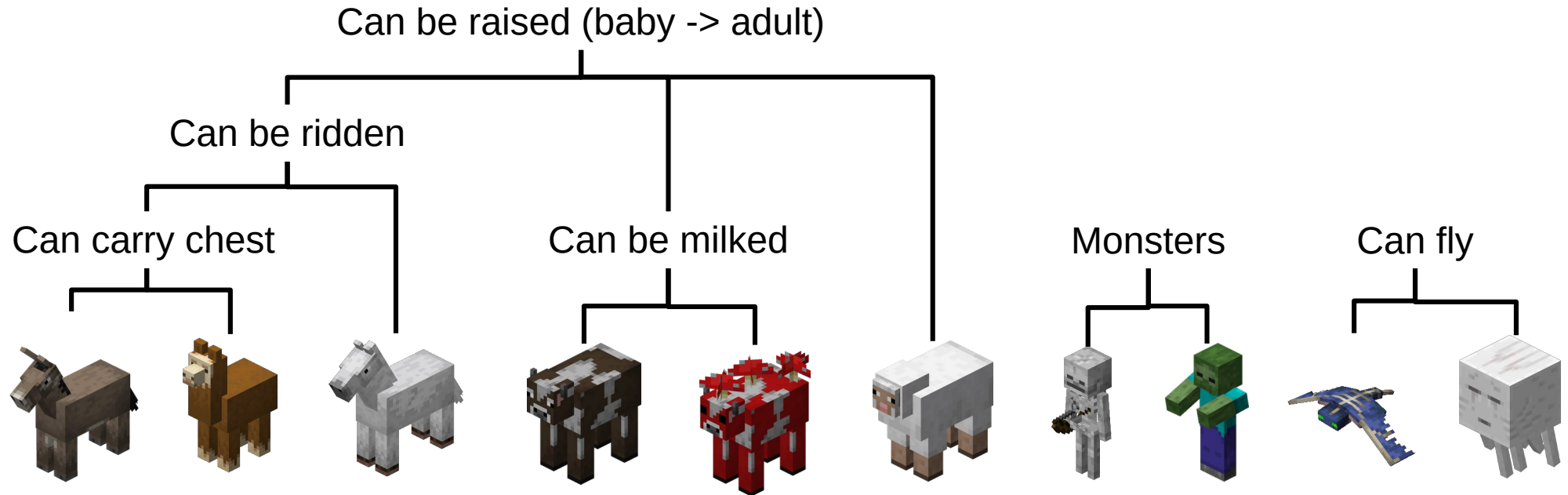
Monsters



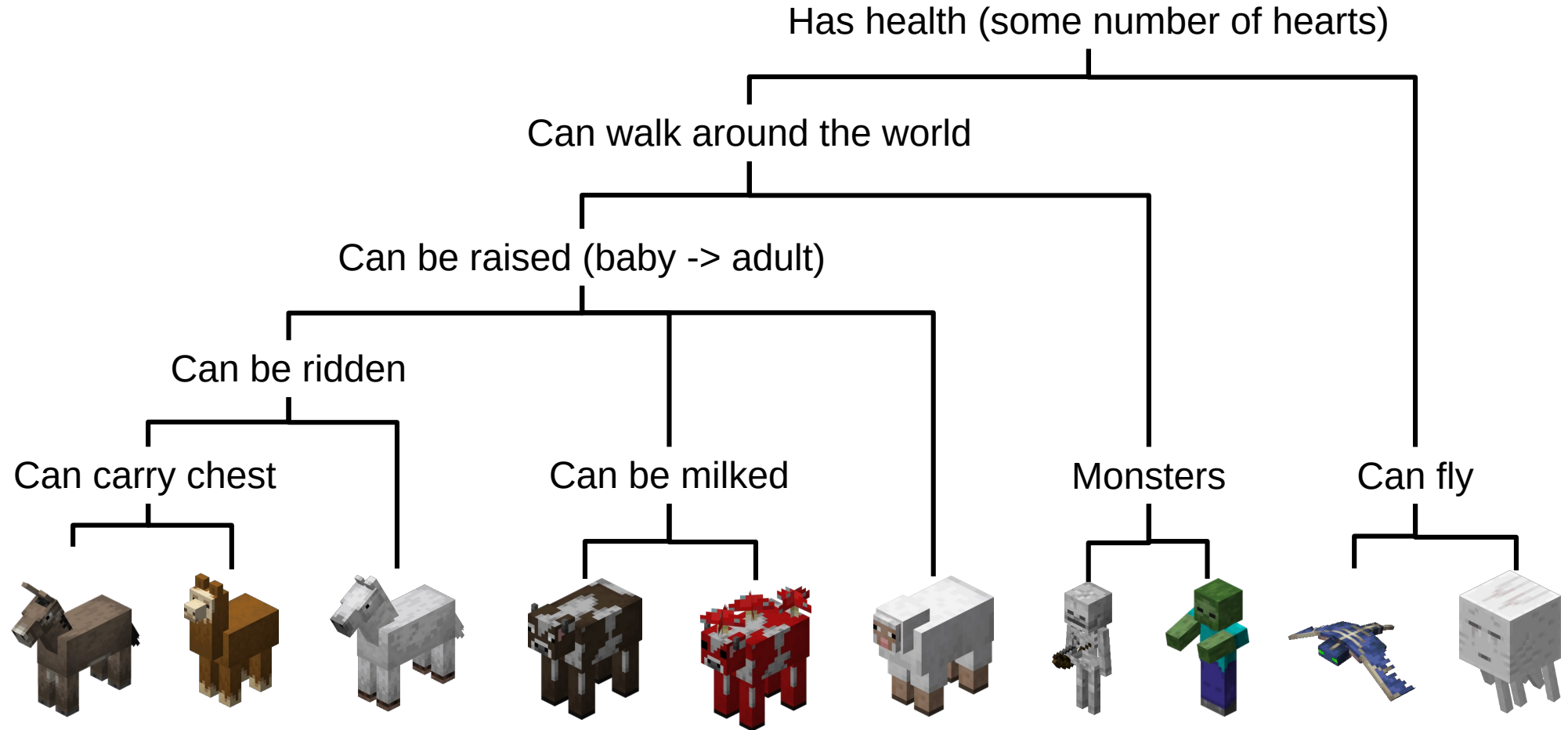
Can fly



# Structure used by the game



# Structure used by the game



# Inheritance

Main idea: if we build up functionality incrementally, it can be reused elsewhere



Wolf

Something that has hearts



Something that has hearts,  
and can walk around the world



Something that has hearts,  
can walk around the world,  
and can be raised



Something that has hearts,  
can walk around the world,  
can be raised,  
and can be tamed



```
struct Mob {};
```

```
struct PathfinderMob : public Mob {};
```

```
struct AgeableMob : public PathfinderMob {};
```

```
struct TameableAnimal : public AgeableMob {};
```

```
struct Wolf : public TameableAnimal {};
```

Something that has hearts



Something that has hearts,  
and can walk around the world



Something that has hearts,  
can walk around the world,  
and can be raised



Something that has hearts,  
can walk around the world,  
can be raised,  
and can be tamed



```
struct Mob {  
    int healthInHearts = 8;  
};
```

healthInHearts is **only** declared here

```
struct PathfinderMob : public Mob {  
    int healthInHearts = 8;  
};
```

```
struct AgeableMob : public PathfinderMob {  
    int healthInHearts = 8;  
};
```

```
struct TameableAnimal : public AgeableMob {  
    int healthInHearts = 8;  
};
```

```
struct Wolf : public TameableAnimal {  
    int healthInHearts = 8;  
};
```

But **all** inheriting classes will have the healthInHearts variable

Something that has hearts



Something that has hearts,  
and can walk around the world



Something that has hearts,  
can walk around the world,  
and can be raised



Something that has hearts,  
can walk around the world,  
can be raised,  
and can be tamed



```
struct Mob {  
    int healthInHearts = 8;  
};
```

Parent class does not  
have wanderAround()

Something that has hearts

```
struct PathfinderMob : public Mob {  
    void wanderAround();  
    int healthInHearts = 8;  
};
```

wanderAround() is  
**only** declared here

Something that has hearts,  
and can walk around the world

```
struct AgeableMob : public PathfinderMob {  
    void wanderAround();  
    int healthInHearts = 8;  
};
```

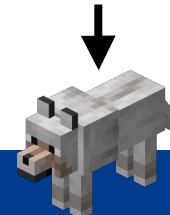
Something that has hearts,  
can walk around the world,  
and can be raised

```
struct TameableAnimal : public AgeableMob {  
    void wanderAround();  
    int healthInHearts = 8;  
};
```

Something that has hearts,  
can walk around the world,  
can be raised,  
and can be tamed

```
struct Wolf : public TameableAnimal {  
    void wanderAround();  
    int healthInHearts = 8;  
};
```

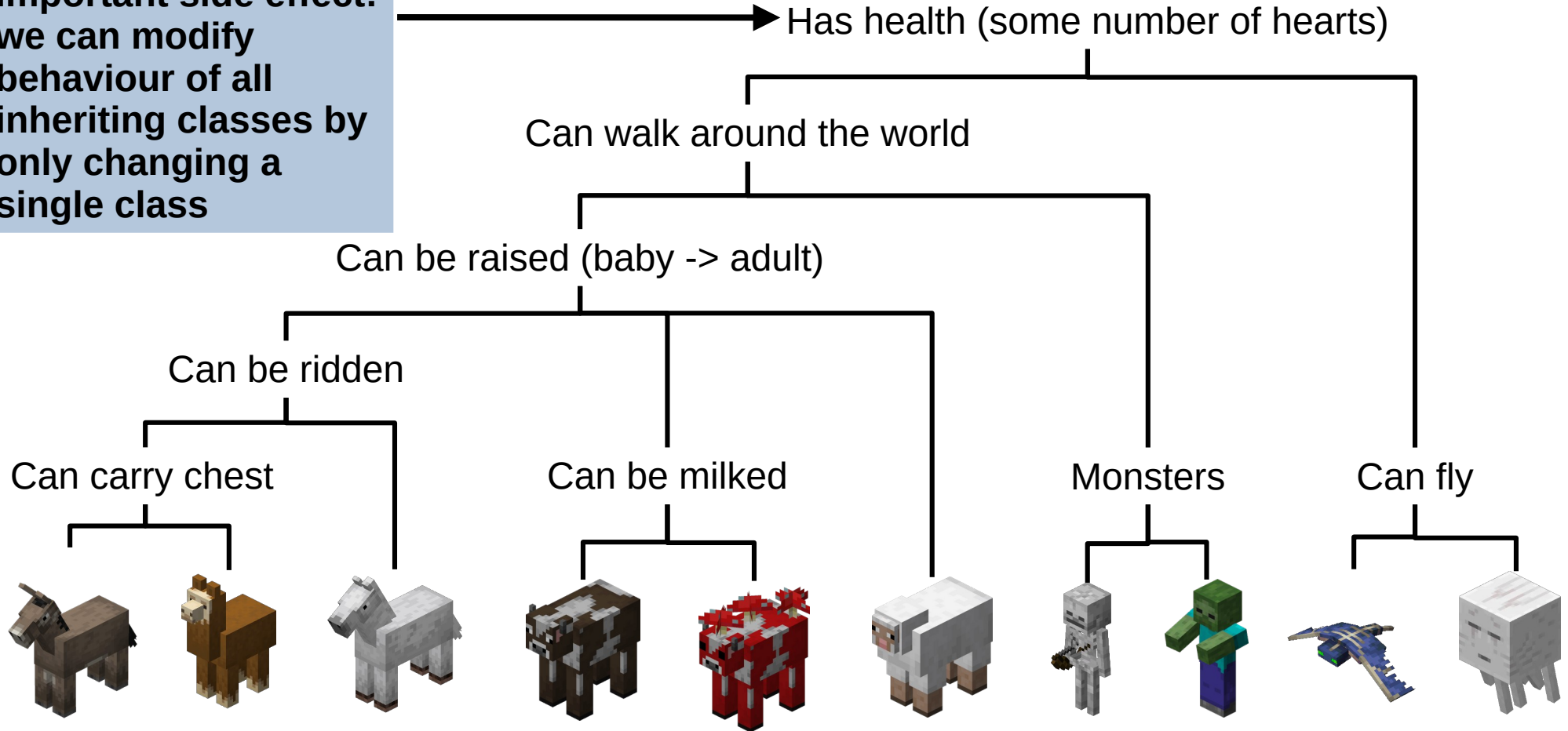
Methods are inherited too,  
**including their definitions**





# Structure used by the game

Important side effect:  
we can modify  
behaviour of all  
inheriting classes by  
only changing a  
single class



# Inheritance: takeaways

- Inheritance is one of the main pillars of object-oriented programming
- Main advantage:  
    Functionality (e.g. tracking the number of hearts) is located in a single place.

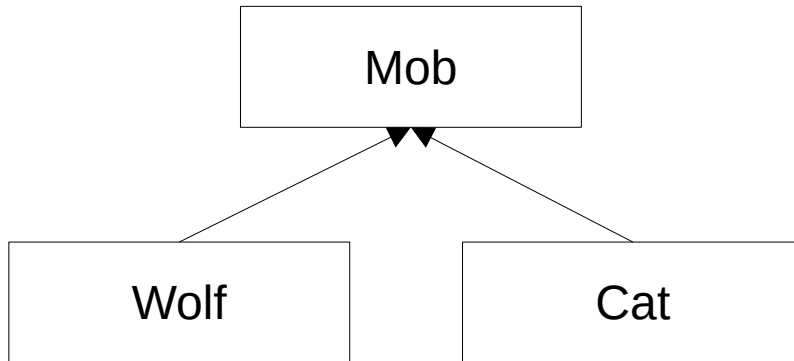
If we make changes to that one place, we modify that functionality for all classes making use of it

# Inheritance: nitty gritty details

- Type conversion
- Access modifiers
- Calling constructors with arguments
  
- We'll do a short task about these and inheritance in general afterwards!

# Inheritance: type conversion

- **Allowed, but may only convert to the type of a parent class.**
  - Example: wolves and cats are mobs, but a mob is ***not necessarily*** a wolf or cat
  - Also: a Wolf cannot be converted into a Cat



# Inheritance: type conversion

- Only variables that are in the parent class are copied. The rest is not.

```
struct Mob {  
    int healthInHearts = 8;  
};  
struct Wolf : public Mob {  
    bool isTamed = false;  
};  
void doStuff() {  
    Wolf wolf;  
    wolf.isTamed = true;  
    wolf.healthInHearts = 10;  
    Mob mobValue = wolf;  
    wolf = mobValue;  
}
```

We use an inherited variable here

Allowed: Mob is parent of Wolf.  
Only healthInHearts is copied  
because Mob does not have isTamed

Not allowed: Wolf is not a parent of Mob

# Inheritance: access modifiers

- There is a third access specifier: protected

```
class Mob {  
    int healthInHearts = 8;  
};  
  
class PathfinderMob : public Mob {  
protected:  
    void wanderAround();  
};  
  
class Wolf : public PathfinderMob {  
    void takeDamage() {  
        healthInHearts--;  
        wanderAround();  
    }  
};
```

Access level	What can access it?
Public	Everything
Protected	Methods in this class and child classes
Private	Only this class

Not allowed: Wolf has healthInHearts member, but only methods in Mob are allowed to modify it

Allowed: wanderAround() is protected, and is called from Wolf

# Inheritance: calling constructors

- Constructors of parent classes must be called explicitly if they have parameters
  - Mandatory to use the : syntax for this

```
struct Mob {  
    int healthInHearts = 8;  
    Mob(int heartCount) : healthInHearts{heartCount} {}  
};
```

```
struct Wolf : public Mob {  
    bool isTamed;  
    Wolf(int initialHearts) : Mob(initialHearts), isTamed{false} {}  
};
```



Constructor of parent class must be called before initialising any other member variables

# Task: Do you see any mistakes?

```
class Mob {
    int healthInHearts = 8;
    Mob(int heartCount) : healthInHearts{heartCount} {}
};

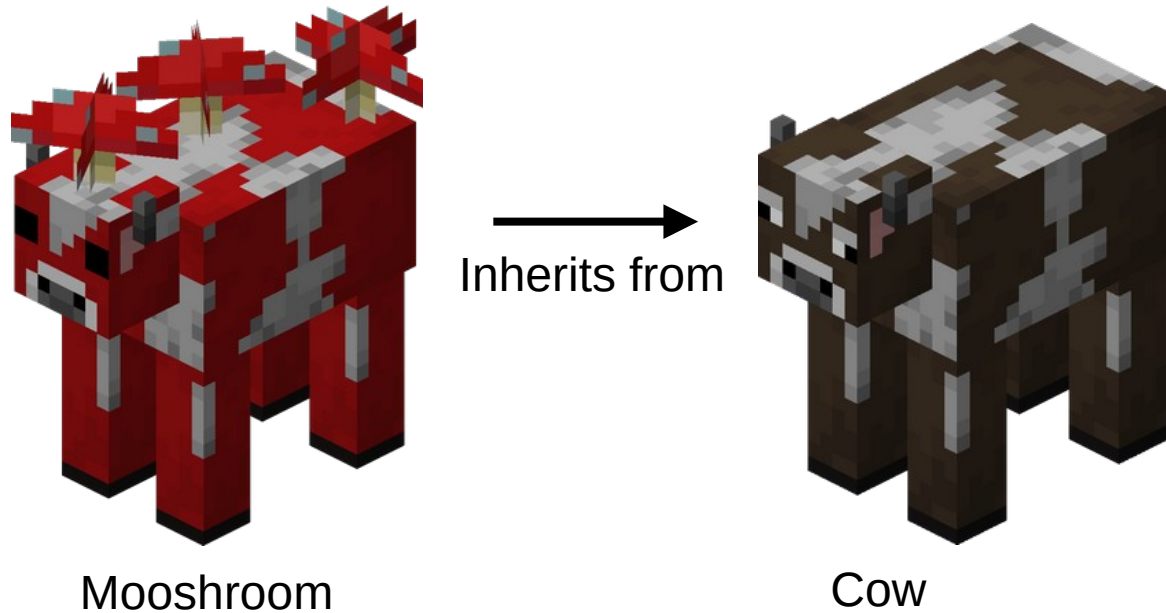
class Zombie : public Mob {
    Zombie() : Mob(10) {}
    void makeNoise() { std::cout << "BuuUuUUuUuHhHhh" << std::endl; }
};

struct BabyZombie : public Zombie {};

int main() {
    BabyZombie zomb(5);
    Mob mob = zomb;
    mob.makeNoise();
}
```



# But that is not all..



```
class Cow : public Animal {};  
class Mooshroom : public Cow {};
```

But that is not all..



Mooshroom



Also a mooshroom

# But that is not all..



But that is not all..



+



=



+



=



# But that is not all..



Cow

← Inherits from



Mooshroom

```
class Cow {  
    void hitByLightning() {  
        // BBQ  
    }  
};
```

```
class Mooshroom : public Cow {  
    void hitByLightning() {  
        setColour(BROWN);  
    }  
};
```

# Inheritance: overriding methods

- When a child object defines the same function as its parent, it *replaces* the version defined by its parent.
  - This mechanism is called «overriding»
  - Can be used to change the behaviour of portions of the object, but not everything

```
class Cow {  
    void hitByLightning() {  
        // BBQ  
    }  
};  
  
class Mooshroom : public Cow {  
    void hitByLightning() {  
        setColour(BROWN);  
    }  
};
```

# Inheritance: overriding methods

- The best part:
  - We define a method (this example: `attack()`) in a parent class that its children override
  - Any monster can now `attack()`, and we don't need to know what it does

```
class Monster {  
    void attack() {  
        // do nothing  
    }  
};
```

```
class Spider : Monster {  
    void attack() {  
        bother(player);  
    }  
};
```



```
class Creeper : Monster {  
    void attack() {  
        hug(player);  
    }  
};
```



```
class Skeleton : Monster {  
    void attack() {  
        shoot(player);  
    }  
};
```




# Inheritance: overriding methods

- The best part:
  - We can now even create a vector of all monsters and iterate over them!

```
struct World {  
    std::vector<Monster> monsters;  
  
    void spawn(Monster monster) {  
        monsters.push_back(monster);  
    }
```

```
    void update() {  
        for(int i = 0; i < monsters.size(); i++) {  
            monsters.at(i).attack();  
        }  
    }  
};
```

```
int main() {  
    World world;  
    Creeper creeper;  
    world.spawn(creeper);  
    return 0;  
}
```





# Inheritance: overriding methods

- The best part:
  - We can now even create a vector of all monsters and iterate over them!

```
struct World {  
    std::vector<Monster> monsters;  
  
    void spawn(Monster monster) {  
        monsters.push_back(monster);  
    }
```

```
    void update() {  
        for(int i = 0; i < monsters.size(); i++) {  
            monsters.at(i).attack();  
        }  
    }  
};
```

```
int main() {  
    World world;  
    Creeper creeper;  
    world.spawn(creeper);  
    return 0;  
}
```


**Does not work!**

# Inheritance: overriding methods

- The best part:
  - We can now even create a vector of all monsters and iterate over them!

```
struct World {  
    std::vector<Monster> monsters;  
  
    void spawn(Monster monster) {  
        monsters.push_back(monster);  
    }  
  
    void update() {  
        for(int i = 0; i < monsters.size(); i++) {  
            monsters.at(i).attack();  
        }  
    }  
};
```

If you call spawn() with an instance of a child class (ex: Creeper), it is automatically converted into a Monster. It is no longer an instance of the child class



# Inheritance: overriding methods

- Solution (step 1 of 2):
  - We make the vector contain references (pointers) instead of instances
  - This is not enough. We need one more thing..

```
struct World {  
    std::vector<std::unique_ptr<Monster>> monsters;  
  
    void spawn(std::unique_ptr<Monster> monster) {  
        monsters.push_back(std::move(monster));  
    }  
  
    void update() {  
        for(int i = 0; i < monsters.size(); i++) {  
            monsters.at(i)->attack();  
        }  
    }  
};
```

# Inheritance: overriding methods

- Solution (step 2 of 2):
  - We add the «virtual» keyword to the method in the parent class

```
class Monster {  
    virtual void attack() {  
        // do nothing  
    }  
};
```

```
class Spider : Monster {  
    void attack() {  
        bother(player);  
    }  
};
```



```
class Creeper : Monster {  
    void attack() {  
        hug(player);  
    }  
};
```



```
class Skeleton : Monster {  
    void attack() {  
        shoot(player);  
    }  
};
```



# Inheritance: overriding methods

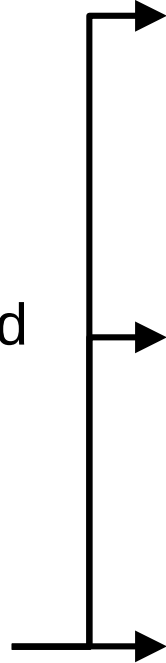
- Best practice:
  - Also declare child methods overriding a virtual method as virtual
  - Use the override keyword to indicate a virtual method is overridden

```
class Monster {  
    virtual void attack() {  
        // do nothing  
    }  
};
```

```
class Spider : Monster {  
    virtual void attack() override {  
        bother(player);  
    }  
};
```

```
class Creeper : Monster {  
    virtual void attack() override {  
        hug(player);  
    }  
};
```

```
class Skeleton : Monster {  
    virtual void attack() override {  
        shoot(player);  
    }  
};
```



# Inheritance: overriding methods

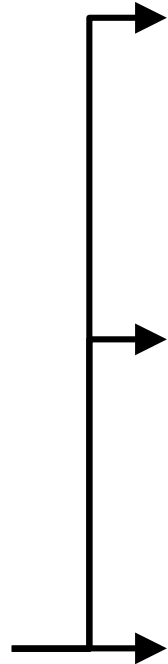
- Best practice:
  - If a class contains at least one virtual method, define a virtual destructor as well.

```
class Monster {  
    virtual void attack() {  
        // do nothing  
    }  
    virtual ~Monster() {}  
};
```

```
class Spider : Monster {  
    virtual void attack() override {  
        bother(player);  
    }  
};
```

```
class Creeper : Monster {  
    virtual void attack() override {  
        hug(player);  
    }  
};
```

```
class Skeleton : Monster {  
    virtual void attack() override {  
        shoot(player);  
    }  
};
```



# Virtual methods

- Allows creating vectors/arrays of objects of different types, where each object implements that method differently
  - One of the BIG advantages of object-oriented programming
- For this to work:
  - Vector **must** contain **pointers** to object instances
    - Objects are often allocated on the heap
    - Should be smart pointers
    - References also work in the case of function variables and parameters
  - Method in parent class **must** be declared virtual

# What have we created?

- What even **IS** a Monster?
- What does it look like?
- What sounds does it make?
- What does it like for dinner?

```
class Monster {  
    virtual void attack() {  
        // do nothing  
    }  
};
```

It does not always make sense to  
allow instantiation of every class!



# Abstract classes

- Solution:
  - We can declare and define any functions that all monsters should have
  - Any method that must be defined by a child class can be declared «pure virtual» by writing = 0 after a declaration of a virtual method.
  - A class with at least one pure virtual function is called an «abstract class» and cannot be instantiated.

```
class Monster {  
    virtual void attack() = 0;  
};
```

# Abstract classes

- Only child classes that define all pure virtual functions can be instantiated.

```
class Monster {  
    virtual void attack() = 0;  
};  
  
// Does not override attack()  
// Thus is abstract itself  
class FlyingMonster : Monster {  
  
};  
  
// Overrides attack(), and can  
// be instantiated  
class Creeper : Monster {  
    virtual void attack() override {  
  
    }  
};
```

# Abstract classes: Interface

- An interface is an abstract class that only contains pure virtual methods.
- Not something the language recognises explicitly, just a common way of using abstract classes.
- Classes in C++ can inherit from multiple parent classes, allowing you to cast to the interface type
- You should otherwise avoid multiple inheritance.

```
class Shearable {  
    virtual Item shear() = 0;  
    virtual bool readyToShear() = 0;  
};
```

```
// An interface guarantees a class  
// has certain methods available  
// These are virtual functions,  
// and can as such be called the  
// same way we've seen before
```

```
class Mooshroom : Animal, Shearable {  
  
};
```

# Today

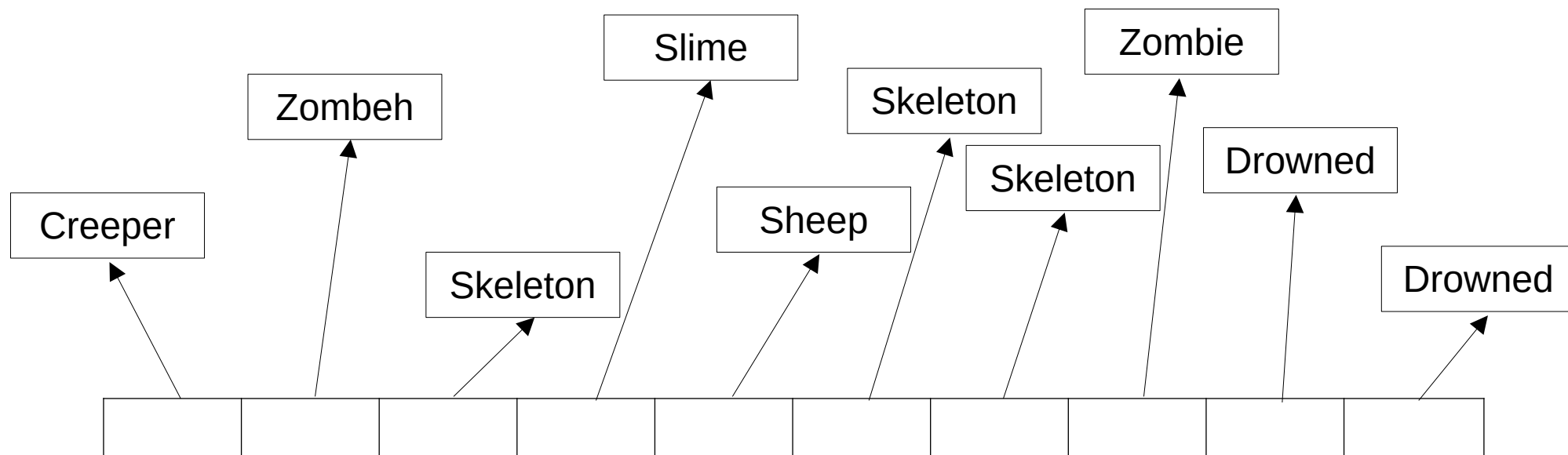
- Inheritance
- Virtual methods
- **The cost of virtual methods**
- static
- friend keyword

# The price tag

- Object-Oriented Programming has a bit of a reputation for being slow. This is partially true, but is also a little nuanced.

# The price tag

- Individual monsters are often allocated on the heap using the **new** operator. This causes them to be located in very different places in memory



```
std::vector<std::unique_ptr<Monster>> monsters;
```

# The price tag

- Individual monsters are often allocated on the heap using the **new** operator. This causes them to be located in very different places in memory
  - Bad for memory bandwidth: tend to only use a few variables when iterating over them, which means bad utilisation of loaded cache lines
  - Bad for cache: objects are not located near each other, so the cache is mostly useless

# The price tag

- One possibility: create vectors of each type of object (e.g. vector of Creepers, Zombies, ..), and store raw pointers (Monster\*) in our list of monsters. That at least ensures that all monsters of the same type are located in each other's vicinity.



# Today

- Inheritance
- Virtual methods
- The cost of virtual methods
- **static**
- friend keyword

# Static keyword

- The static keyword roughly means «there is only one»
- A static variable stays around between function calls, and the next time you call that function the value will be what you left it the last time around
- The value of a static field is the same in all instances of that class. Useful for defining constants used within the class.

```
int counter() {  
    static int next = 0;  
    next++;  
    return next;  
}
```

# Static keyword

- Can also be used for methods. These are functions that happen to be located in a class, and if an instance of that class is passed in as a parameter, can access private members
- Since they are functions that happen to be located inside that class, they can only modify static member variables.

```
class Counter {  
    int count = 0;  
public:  
    static void increment(Counter counter) {  
        counter.count++;  
    }  
};
```

Can only modify count because an instance of Counter is passed in



# Today

- Inheritance
- Virtual methods
- Operator overloading
- **Friend keyword**


# Friend

- Allows an object to give another object or function access to its private members.
- Primary use case: printing classes using `std::cout` or `std::ofstream`
- Friends can only be declared while declaring an object. They cannot be added after the fact.

# Using friend: Functions

- Case 1 of 3: Giving access to a specific function

```
class House {  
    bool plantsHaveBeenWatered = false;  
    void openFrontDoor();  
    friend void useHouseKey(House &house);  
};  
  
void useHouseKey(House &house) {  
    house.openFrontDoor();  
    house.plantsHaveBeenWatered = true;  
}
```




Simply copy and paste the function declaration after the friend keyword

The function can now use private fields and methods!

# Using friend: Classes

- Case 2 of 3: Giving access to all methods in another class

```
class House {  
    bool plantsHaveBeenWatered = false;  
    void openFrontDoor();  
  
    friend class Buddy;  
};  
  
class Buddy {  
public:  
    void openHouseDoor(House &house) {  
        house.openFrontDoor();  
        house.plantsHaveBeenWatered = true;  
    }  
};
```



Simply copy and paste the class declaration after the friend keyword

All methods in Buddy can now use the private members of House

# Using friend: Methods

- Case 3 of 3: Giving access to a specific method in another class

```
class Buddy;
```



You may need to forward declare the class!

```
class House {  
    bool plantsHaveBeenWatered = false;  
    void openFrontDoor();  
};
```

Copy the method, and add the class name in front (here: Buddy::)

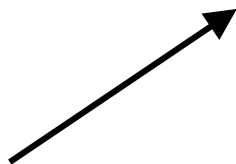
```
friend void Buddy::openHouseDoor(House &house);
```

```
};
```

```
class Buddy {  
public:
```

```
    void openHouseDoor(House &house) {  
        house.openFrontDoor();  
        house.plantsHaveBeenWatered = true;  
    }
```

```
};
```



The openHouseDoor() method in Buddy can now access private House members. Other methods can't.



# Today

- Inheritance
- protected keyword
- virtual methods
- static
- friend

# Next week

- **Guest lecture!**
- Error handling
- Testing