WEEK 7

Inheritance – Derived Classes, Functions in a Hierarchy

AGENDA

- Week 7-1
  - Hierarchy
  - Relationship
  - Derived Classes & Base Classes
  - Access Levels
- Week 7-2
  - Functions in a Hierarchy
    - Shadowing
  - Constructors and Destructors

WEEK 7-1

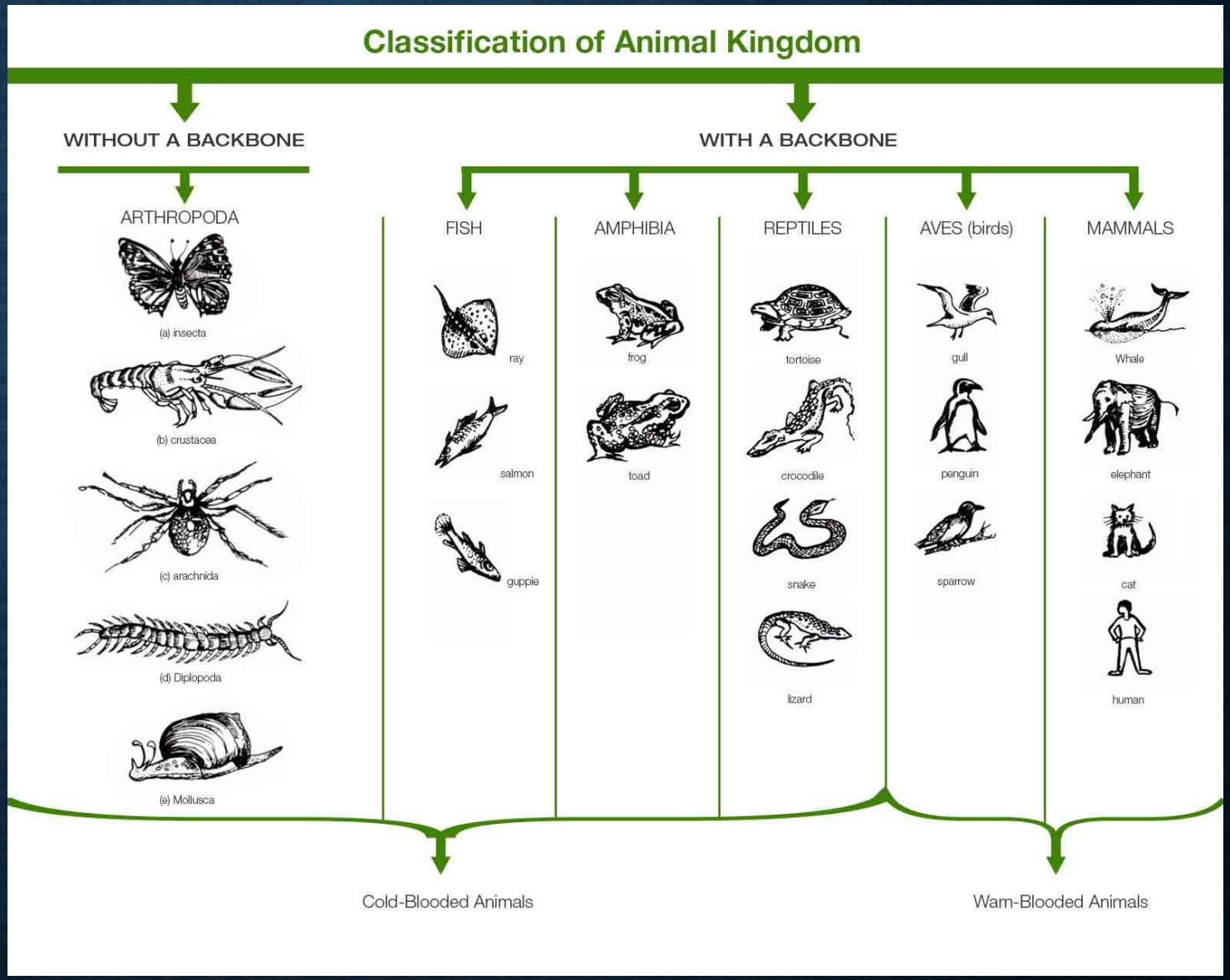
HIERARCHY

- Inheritance is the 2nd pillar of OOP and it forms a set of relationships between related classes
- A hierarchy will help us describe the concept of inheritance in the Object Oriented style of programming

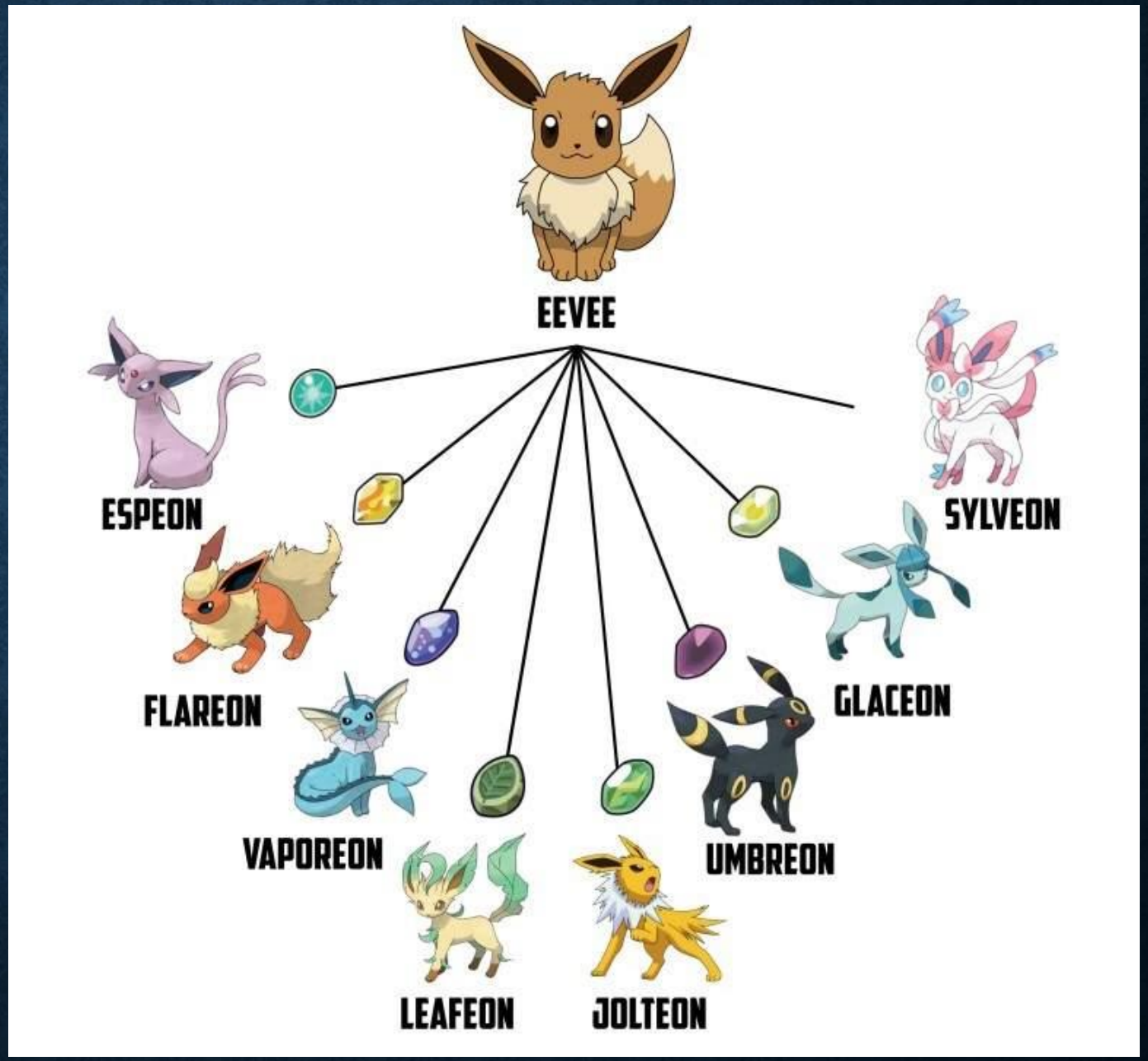
HIERARCHY

- Hierarchies exist in our world in a variety of shapes and sizes
  - The categorization of the animal kingdom where a species is related to another
  - The chain of command in an army
  - The genealogy of a family
  - The evolution of Pokémon
- In essence this sort of connecting relationship of related items is a visual representation of inheritance. It looks like a tree of sorts.

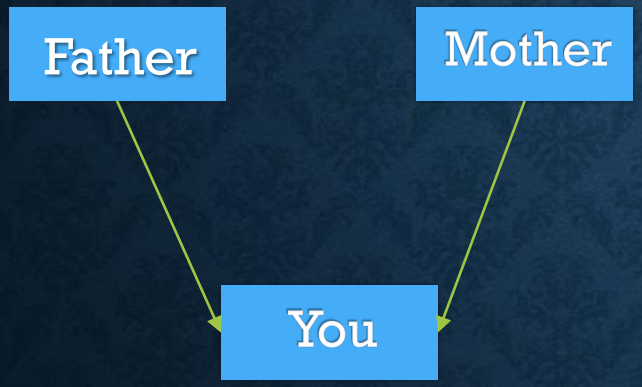
HIERARCHY


Classification of Animal Kingdom
WITHOUT A BACKBONE
WITH A BACKBONE
ARTHROPODA
(a) insecta
(b) crustacea
(c) arachnida
(d) Diplopoda
(e) Mollusca
FISH
ray
salmon
guppie
AMPHIBIA
frog
toad
REPTILES
tortoise
crocodile
snake
lizard
AVES (birds)
gull
penguin
sparrow
MAMMALS
Whale
elephant
cat
human
Cold-Blooded Animals
Wam-Blooded Animals

HIERARCHY

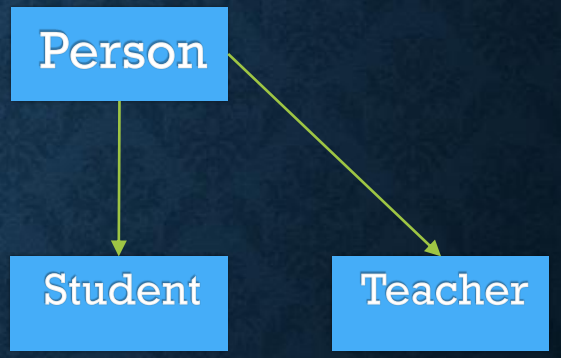

EEVEE
ESPEON
SYLVEON
FLAREON
GLACEON
VAPOREON
UMBREON
LEAFEON
JOLTEON

HIERARCHY



A common hierarchy

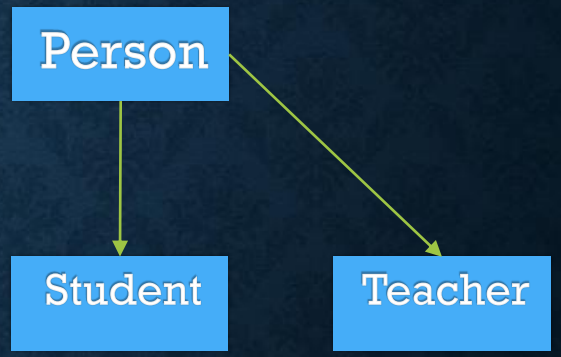
Each of these are C++ classes



A common hierarchy for inheritance in OOP

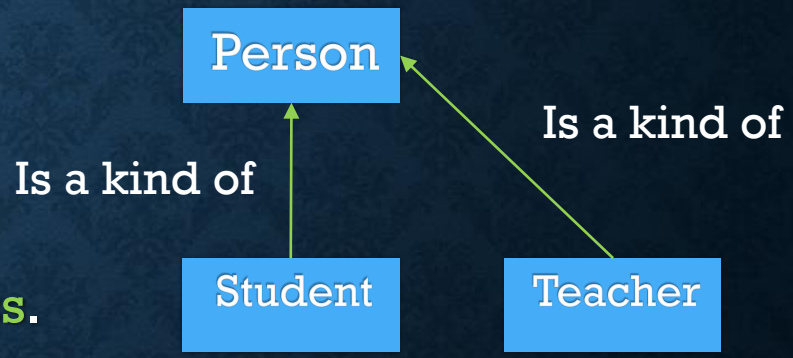
RELATIONSHIP

- The relationship of inheritance is said to be a transitive one
- Considering the previous example of a person class:
  - A student is a person
  - A teacher is person
  - A person could be a student or a teacher but isn't necessarily one



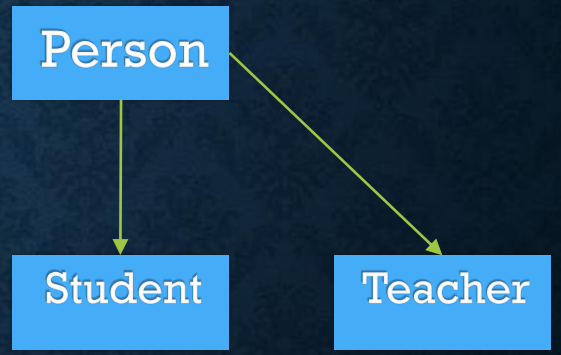
RELATIONSHIP

- Effectively we can use those arrows to denote a "is a kind of" relationship between the classes.
  - A Student is a type of Person
  - A Teacher is a type of Person
- In other words, the student class derives from person class.



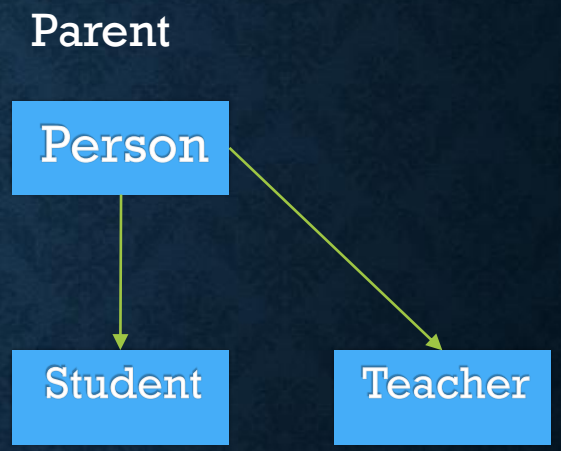
DERIVED CLASSES & BASE CLASSES

- Going into how these relationships will be represented in our code let's place down the terms derived and base for our classes.



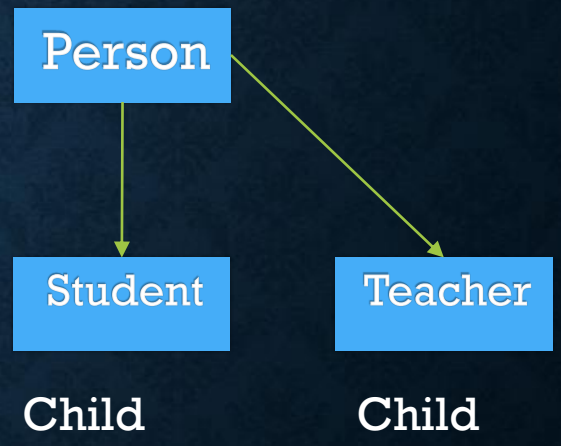
DERIVED CLASSES & BASE CLASSES

- A base class can be said to be a parent class or a super class
  - The base will act like our origin from which other classes can spring from
  - Much like the birth of a child requires an origin which is their parent
  - It is essentially just a regular class as we have experienced so far



DERIVED CLASSES & BASE CLASSES

- A derived class can be said to be a child class or a sub class
  - The derived will use the base class as a starting point to build themselves
  - In other words anything in the base class will be inherited by the child (much like DNA, or physical traits)



DERIVED CLASSES

- As mentioned previously, a derived class takes in all the attributes of its parent class and builds on top that. When speaking of all the attributes, this means all the data members and all the member functions (NOT including the specials ones ie constructors, destructors…)

DERIVED CLASSES

- The syntax to define a class as a derived class is as follows:

```
class Derived : access Base { … }
class playdoh : public compound { … }
```

- In this case the Derived is the name of the derived class, the access is the access level permitted to the derived class from the base (eg can the derived class access the base's private members?). Typically access is stated to be public but by default it is private. Lastly the Base is the name of the base class.

COMPOUND + PLAYDOH EXAMPLE

ACCESS LEVELS

- There are three access levels present for use in C++, you already know two of them:
  - Public – Essentially everyone can access this from anywhere.
  - Private – Can only access from within a class or its members.

ACCESS LEVELS

- The third is protected.
  - This level of access specifies that only members of a derived class (and the class itself) can access this member
  - Protected access should be limited to member functions only (a query perhaps)

ACCESS LEVELS

- Giving protected access to a data member is not advised as this can allow for unwarranted access to what would be our private data from the base class. We should continue to access private members through member functions (perhaps protected ones)

WEEK 7-2

Functions in a Hierarchy

FUNCTIONS IN A HIERARCHY

- Perhaps already noted from the compound base class + playdoh derived class example, we encountered functions in both our base class and the derived class
- The functions present in our base class were inherited to the derived class which could make use of them

FUNCTIONS IN A HIERARCHY

- Note that we had a display function in both classes with the exact same identifier.
- When we called p.display(); which display function was called?
  - Was it the playdoh display or the compound display?
  - Was it the base display or the derived display function?

FUNCTIONS IN A HIERARCHY - SHADOWING

- The answer to that previous question was the derived class' display function was called.
- This act wherein the derived function masks the base function of the same name is called shadowing.

FUNCTIONS IN A HIERARCHY - SHADOWING

- Eventually we'll learn what controls this mechanism in the weeks to come but for now consider that when a class in a hierarchy makes a call to a function that exists throughout that hierarchy the 'most derived one' that matches will be called.
- Eg a playdoh object calling display will default to the playdoh definition of display rather than the compound version

FUNCTIONS IN A HIERARCHY - SHADOWING

- We can choose to manually access a particular member function from our hierarchy with the use of the scope resolution operator :: (double colon)
- The syntax looks similar to when we are specifying the scope/domain of a member function of class's definition:
  - Base::identifier(arguments)
  - Eg. compound::display() – This will explicitly call the compound version of display

CONSTRUCTORS

- A derived class doesn't inherit the base class constructor by default
- If we don't define a constructor in our derived class as per normal the compiler inserts an empty default for us

CONSTRUCTORS

- So then what occurs for our playdoh object when it goes through construction? One part of its data is derived from the base compound class (the weight)
- The sequence will be reminiscent to our previous stint with the creation of objects

CONSTRUCTORS

- Broken down into to phases and using our compound and playdoh example:
  - 1. The base class portion of the derived class is created first
    - Allocate memory for the compound portion of the playdoh object
    - Execute the base class' (compound) constructor
  - 2. The derived portion is then constructed
    - Allocate memory for the playdoh portion of the playdoh object
    - Execute the derived class' (playdoh) constructor

CONSTRUCTORS

Allocate memory for compound portion

Compound

Execute the compound default constr

Compound

weight = 0

Allocate memory for playdoh portion

Compound

weight = 0

Playdoh

Execute the playdoh 2 arg constr

Compound

weight = 55

Playdoh

colour = ‘r’

Playdoh p (‘r’, 55); // Goes through the above process

CONSTRUCTORS

- Notice that in our previous attempt we saw that the compiler seemed to automatically call the default constructor when creating the playdoh derived object.
- In regards to derived classes the following is the default behavior:
  - A derived class' constructor automatically calls the base class' default constructor.
  - A derived class' destructor automatically calls the base class' destructor.
  - A derived class' copy assignment operator automatically calls the base class' copy assignment operator.

CONSTRUCTORS

- Rather than default to the compiler's automatic calls for the default constructor when we're creating a playdoh object, we can reuse the compound's constructors
- While we don't inherit those special functions in our derived class we can pass values into the base constructor by adding the following to our derived constructors:
  - Derived (parameters) : Base (parameters) { … }
  - playdoh(char c, int w) : compound (w) { … }

DESTRUCTORS

- As mentioned already destructors in a derived class automatically call the destructors of its base class
- Because of this nature, clean up of derived classes will act in the way we desire. Ie everything will be cleaned up in sequence in a domino like fashion
- This will be similar to what we learned about destruction order previously (First in Last out, first created last destroyed – FILO)
- The base portion of a derived class is created first thus it is destructed last and the derived portions are destructed first

DESTRUCTORS

Deallocate memory for compound portion

Compound

Execute the compound destructor

Compound

weight = 0

Deallocate memory for playdoh portion

Compound

weight = 0

Playdoh

Execute the playdoh destructor

Compound

weight = 55

Playdoh

colour = ‘r’

Playdoh p (‘r’, 55); // Goes through the above process for destruction