CS 32 Week 5 Discussion 1B

this week's topics:
inheritance, polymorphism, recursion

TA: **Yiyou Chen** / LA: **Ian Galvez**

a couple things before we start...

- ***First, please find a partner / form a group!*** (Yes, I mean right now.) We'll be doing practice problems throughout the class.
- After that, please fill out the LA feedback form: https://tinyurl.com/S22LAFeedback
  - You can also use the QR code aha >>
  - It really helps me improve as an LA, and improve our section! I want to provide resources that y'all need/want
- Just a reminder: ***I highly recommend going to TA/LA office hours*** if you're ever having any questions about projects/homeworks!
- Discussion section is more for reviewing and practicing concepts learned in class.

Topics

- Inheritance and Polymorphism:
    1. Inheritance and overriding
    2. Polymorphism and (pure) Virtual functions
    3. Construction and Destruction
- Recursions and some complexity analysis:
    1. Finding maximal value of a array
    2. Merge sort
    3. Depth-first search(DFS) using recursion
    4. Recursion for generating all permutations

Inheritance

Goal: to have classifications of classes based on shared features.

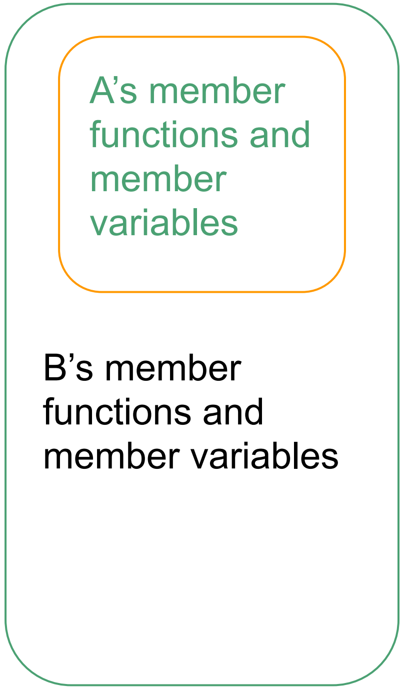
- B is inherited from A. (B shares some characteristics of A).
- A is called the base class, and B is called a derived class.

Example:

Base class: Characters

Derived classes: Players, Zombies, Enemies ...

Derived class can only access **public member variables** of its base class, but **not private member variables** of its base class.

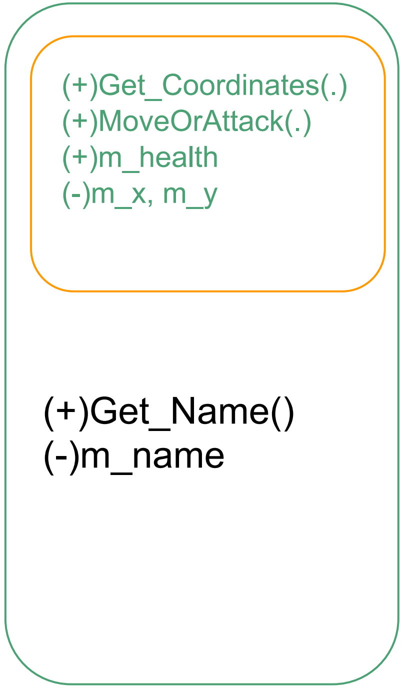

A's member functions and member variables
B's member functions and member variables

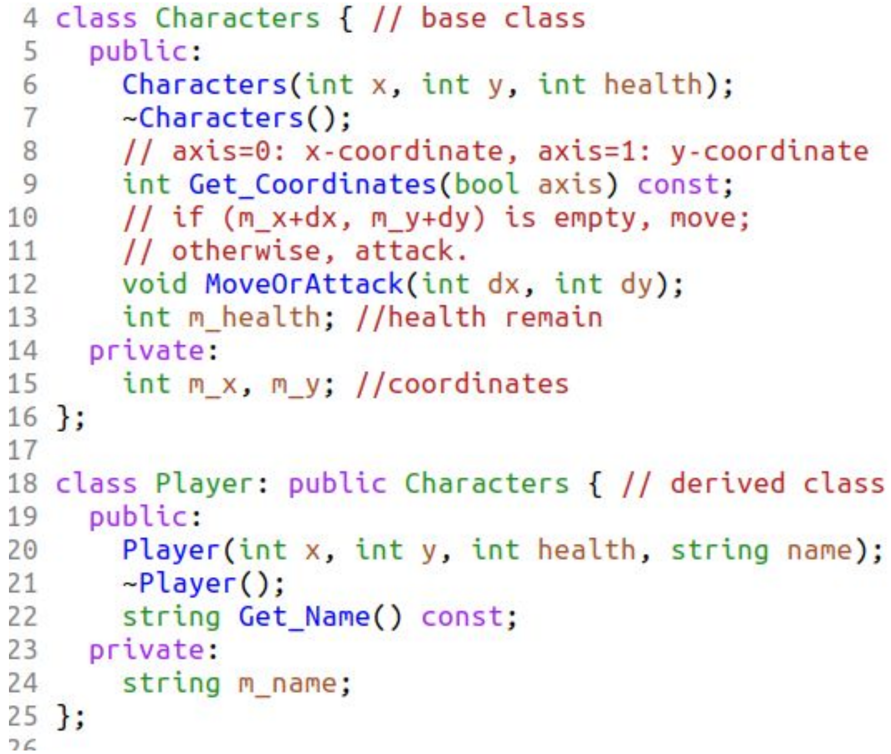
Inheritance

- Player is inherited from Characters.

Can Player directly access m_health?

Can Player directly access m_x, m_y?




```
class Characters { // base class
  public:
    Characters(int x, int y, int health);
    ~Characters();
    // axis=0: x-coordinate, axis=1: y-coordinate
    int Get_Coordinates(bool axis) const;
    // if (m_x+dx, m_y+dy) is empty, move;
    // otherwise, attack.
    void MoveOrAttack(int dx, int dy);
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    Player(int x, int y, int health, string name);
    ~Player();
    string Get_Name() const;
  private:
    string m_name;
};
```

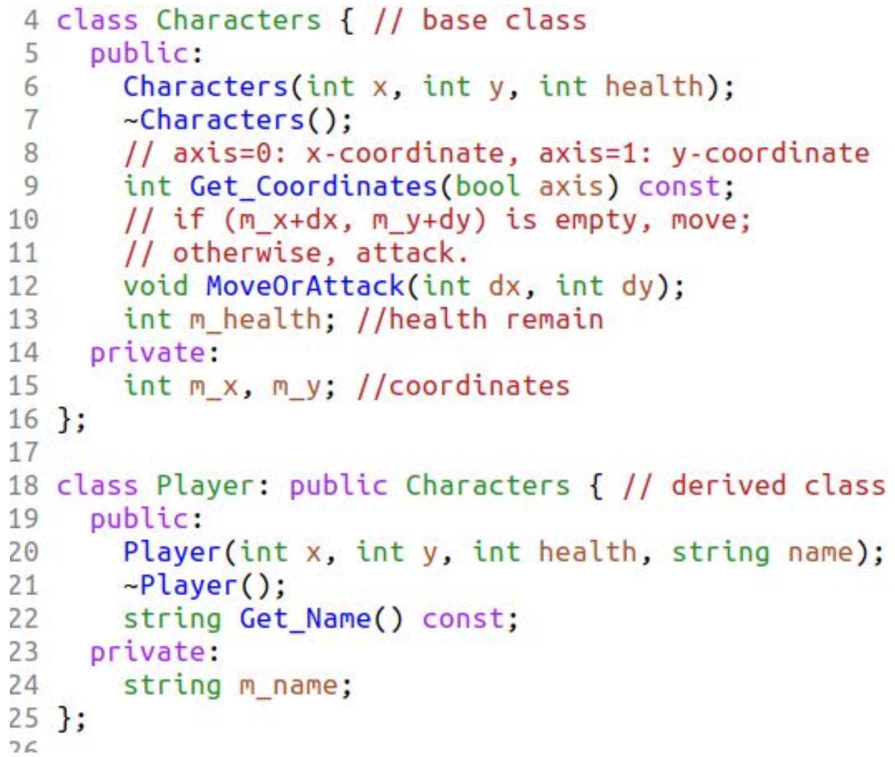
Inheritance (create objects)

Create base class objects:

1. Characters* c = new Characters(..);
2. Characters c(..);

Create derived class objects:

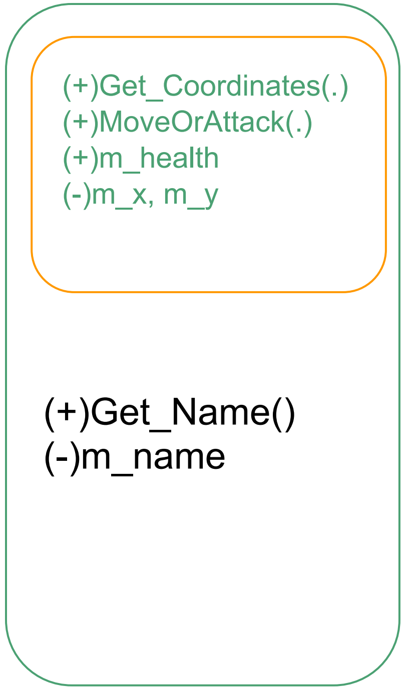
1. Characters* c = new Player(..);
2. Player c(..);



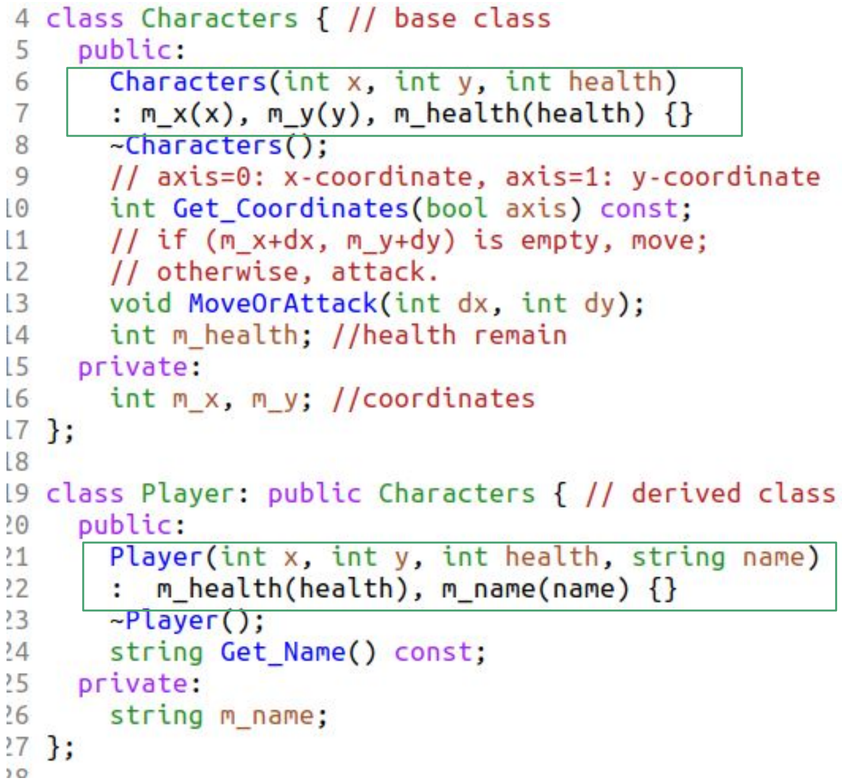
```
class Characters { // base class
  public:
    Characters(int x, int y, int health);
    ~Characters();
    // axis=0: x-coordinate, axis=1: y-coordinate
    int Get_Coordinates(bool axis) const;
    // if (m_x+dx, m_y+dy) is empty, move;
    // otherwise, attack.
    void MoveOrAttack(int dx, int dy);
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    Player(int x, int y, int health, string name);
    ~Player();
    string Get_Name() const;
  private:
    string m_name;
};
```

Order of Construction

1. Construct the Base Class (**default constructor if not specified in initialization list**)
2. Member variables' default constructor (use initialization list when there's no default constructor)
3. Body of the constructor for the derived class



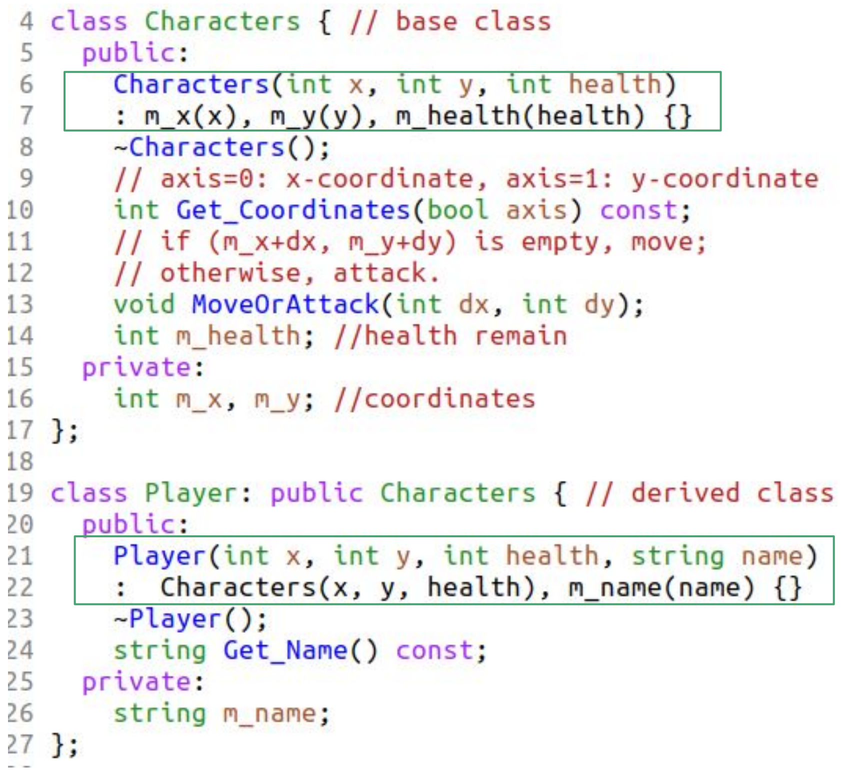
Order of Construction



```cpp
class Characters { // base class
  public:
    Characters(int x, int y, int health)
    : m_x(x), m_y(y), m_health(health) {}
    ~Characters();
    // axis=0: x-coordinate, axis=1: y-coordinate
    int Get_Coordinates(bool axis) const;
    // if (m_x+dx, m_y+dy) is empty, move;
    // otherwise, attack.
    void MoveOrAttack(int dx, int dy);
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    Player(int x, int y, int health, string name)
    :  m_health(health), m_name(name) {}
    ~Player();
    string Get_Name() const;
  private:
    string m_name;
};
```

Wrong!
No default constructor for base class Characters.

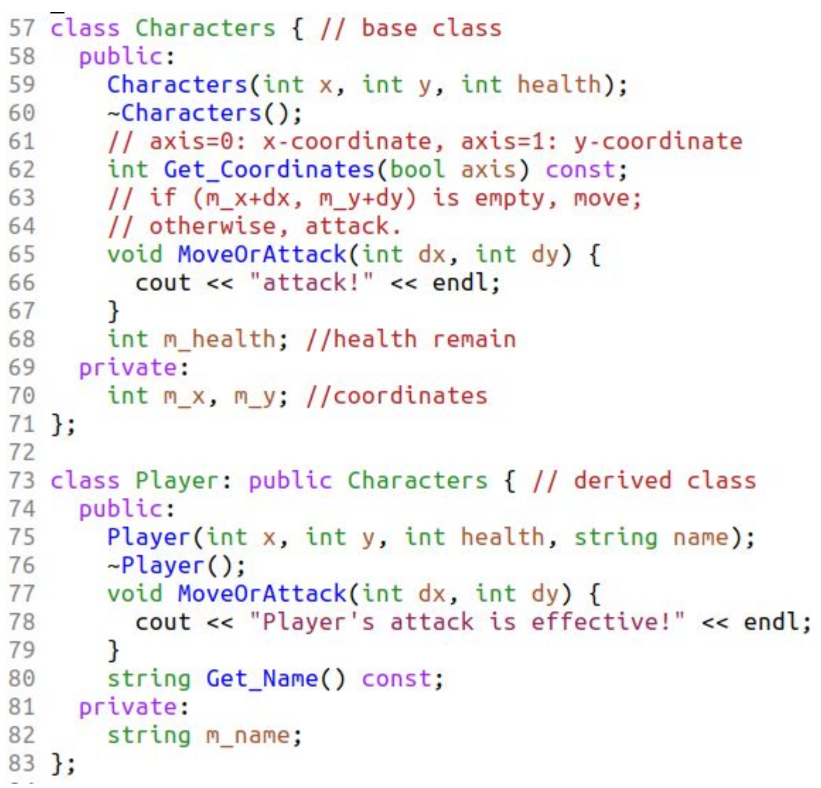
Order of Construction



```
class Characters { // base class
  public:
    Characters(int x, int y, int health)
    : m_x(x), m_y(y), m_health(health) {}
    ~Characters();
    // axis=0: x-coordinate, axis=1: y-coordinate
    int Get_Coordinates(bool axis) const;
    // if (m_x+dx, m_y+dy) is empty, move;
    // otherwise, attack.
    void MoveOrAttack(int dx, int dy);
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    Player(int x, int y, int health, string name)
    :  Characters(x, y, health), m_name(name) {}
    ~Player();
    string Get_Name() const;
  private:
    string m_name;
};
```

Correct!

Overriding



```
class Characters { // base class
  public:
    Characters(int x, int y, int health);
    ~Characters();
    // axis=0: x-coordinate, axis=1: y-coordinate
    int Get_Coordinates(bool axis) const;
    // if (m_x+dx, m_y+dy) is empty, move;
    // otherwise, attack.
    void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    Player(int x, int y, int health, string name);
    ~Player();
    void MoveOrAttack(int dx, int dy) {
      cout << "Player's attack is effective!" << endl;
    }
    string Get_Name() const;
  private:
    string m_name;
};
```

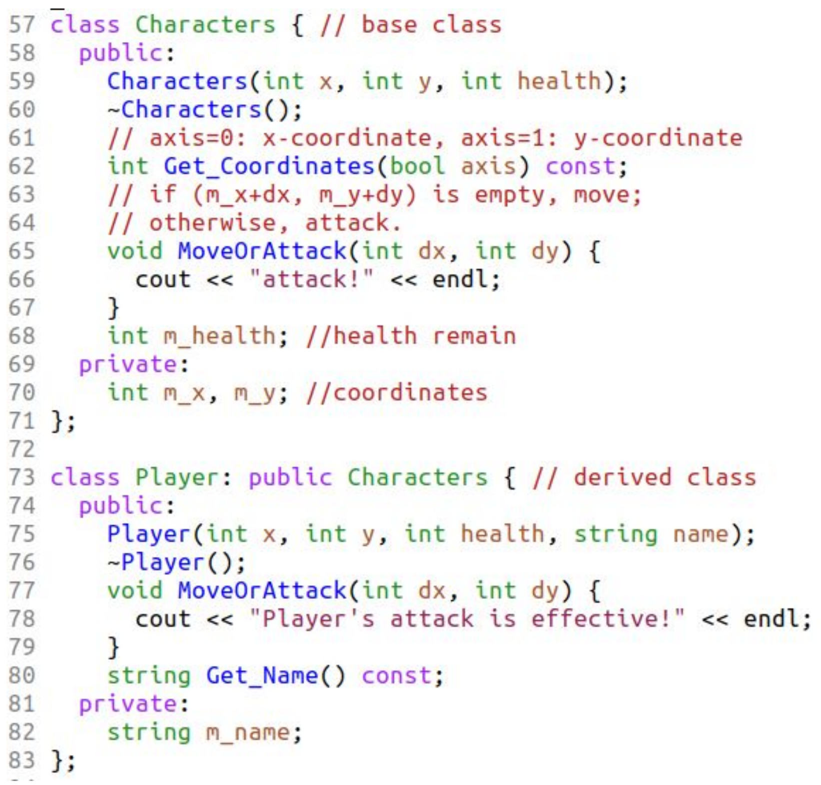
What are the outputs?

Characters p1(1,1,1);
p1.MoveOrAttack(1,0);

Player p2(1,1,1, “CS32”);
p2.MoveOrAttack(1,0);

Take a second to talk about it in your groups!

Overriding



```cpp
class Characters { // base class
  public:
    Characters(int x, int y, int health);
    ~Characters();
    // axis=0: x-coordinate, axis=1: y-coordinate
    int Get_Coordinates(bool axis) const;
    // if (m_x+dx, m_y+dy) is empty, move;
    // otherwise, attack.
    void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    Player(int x, int y, int health, string name);
    ~Player();
    void MoveOrAttack(int dx, int dy) {
      cout << "Player's attack is effective!" << endl;
    }
    string Get_Name() const;
  private:
    string m_name;
};
```

What are the outputs?

Characters p1(1,1,1);
p1.MoveOrAttack(1,0);

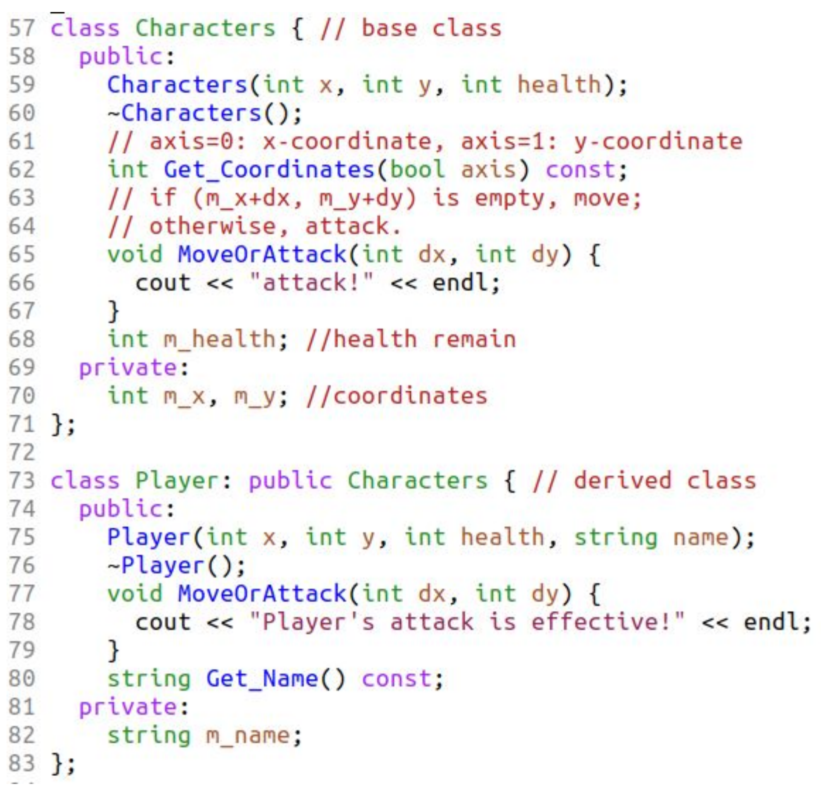
Player p2(1,1,1, “CS32”);
p2.MoveOrAttack(1,0);

“attack!”
“Player’s attack is effective!”

What if we want p2 to output “attack!”?

We can override a member function of a derived class.

Overriding



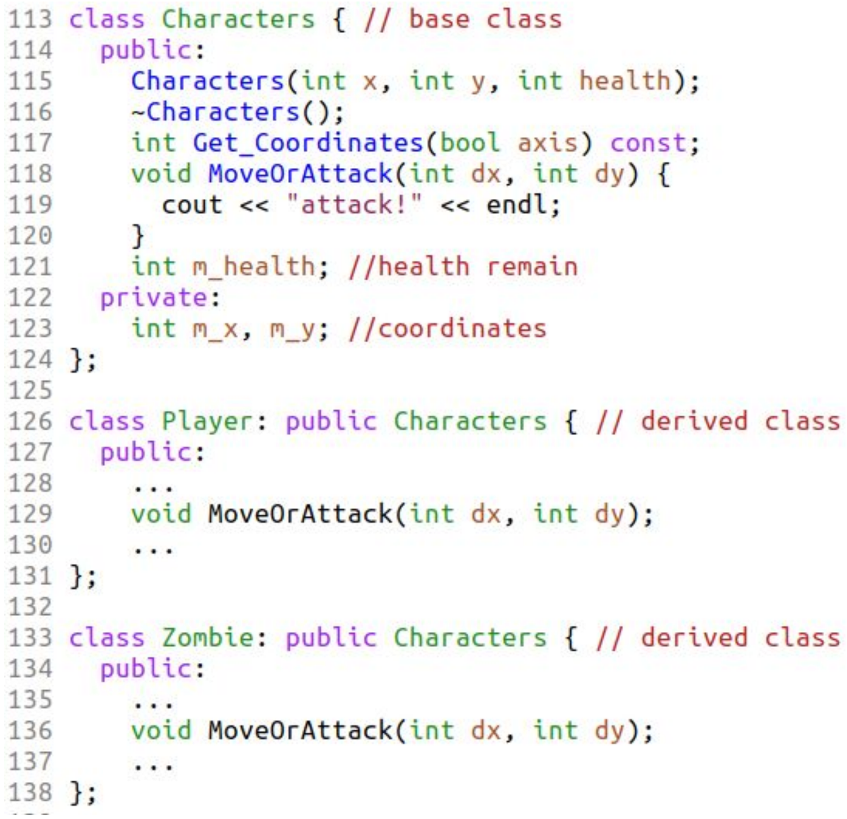
```
class Characters { // base class
  public:
    Characters(int x, int y, int health);
    ~Characters();
    // axis=0: x-coordinate, axis=1: y-coordinate
    int Get_Coordinates(bool axis) const;
    // if (m_x+dx, m_y+dy) is empty, move;
    // otherwise, attack.
    void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    Player(int x, int y, int health, string name);
    ~Player();
    void MoveOrAttack(int dx, int dy) {
      cout << "Player's attack is effective!" << endl;
    }
    string Get_Name() const;
  private:
    string m_name;
};
```

What if we want p2 to output “attack!”?

We can override a member function of a derived class.

Player p2(1,1,1, “CS32”);
p2.Characters::MoveOrAttack(1,0);

More Derived Classes



```cpp
class Characters { // base class
  public:
    Characters(int x, int y, int health);
    ~Characters();
    int Get_Coordinates(bool axis) const;
    void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    int m_health; //health remain
  private:
    int m_x, m_y; //coordinates
};

class Player: public Characters { // derived class
  public:
    ...
    void MoveOrAttack(int dx, int dy);
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    void MoveOrAttack(int dx, int dy);
    ...
};
```

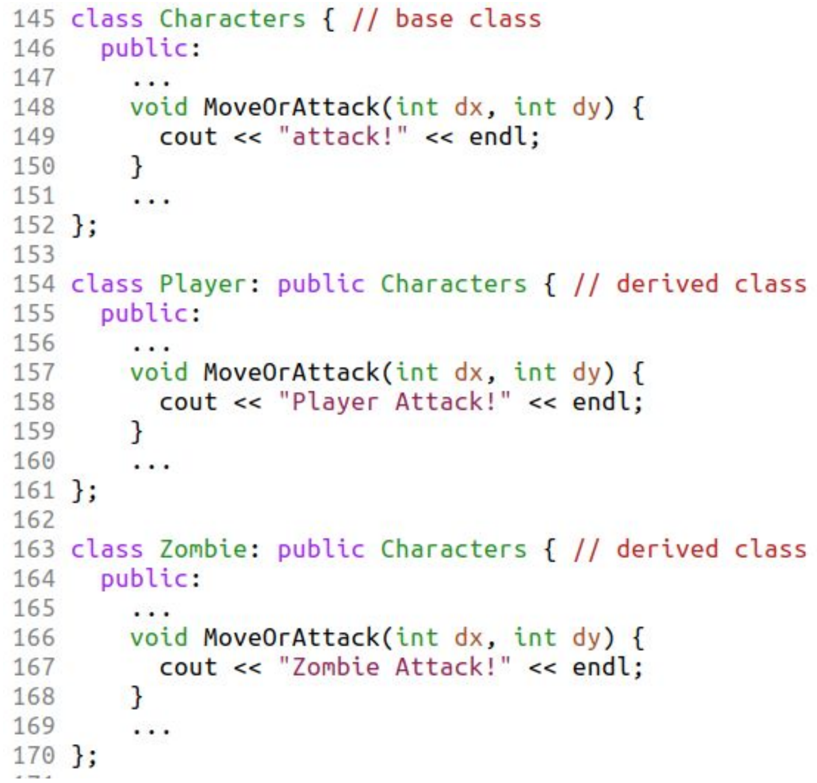
Our goal is to create Characters objects that can be one of those derived classes.

During the compile time, we don't really need to know what derived classes they belong to.

For example, we may just create an array of Characters pointers:
Characters* character[100];

During runtime, we may specify
character[0] = new Player(...);
character[1] = new Zombie(...);
…

More Derived Classes



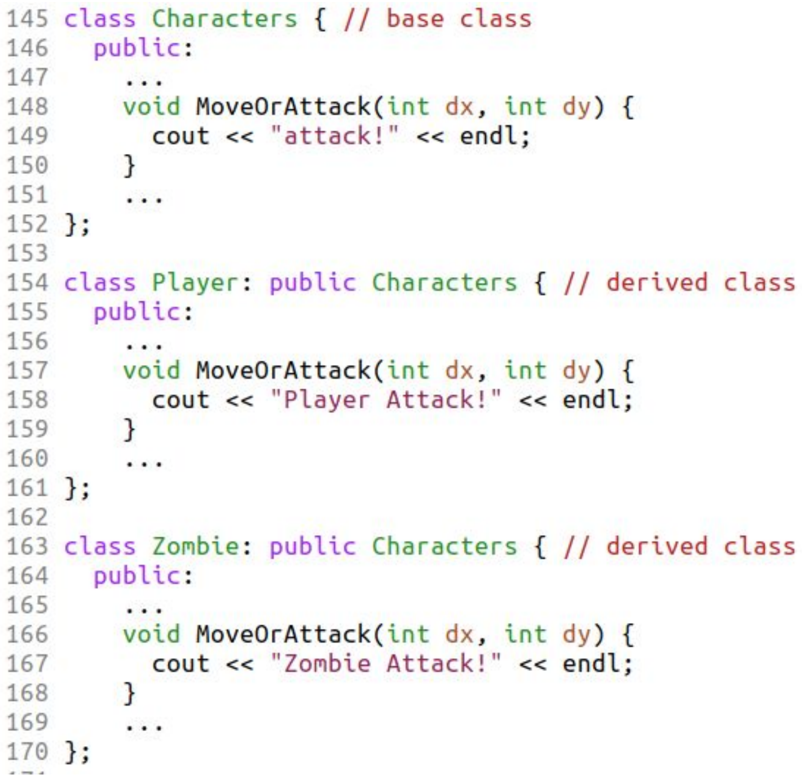
```
class Characters { // base class
  public:
    ...
    void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    void MoveOrAttack(int dx, int dy) {
      cout << "Player Attack!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    void MoveOrAttack(int dx, int dy) {
      cout << "Zombie Attack!" << endl;
    }
    ...
};
```

During runtime, we may specify
Characters* character[100];
character[0] = new Player(...);
character[1] = new Zombie(...);
…

If we use the code as shown, what are the outputs of the following?

character[0]->MoveOrAttack(0, 1);
character[1]->MoveOrAttack(0, 1);

More Derived Classes



```
class Characters { // base class
  public:
    ...
    void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    void MoveOrAttack(int dx, int dy) {
      cout << "Player Attack!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    void MoveOrAttack(int dx, int dy) {
      cout << "Zombie Attack!" << endl;
    }
    ...
};
```

During runtime, we may specify
character[0] = new Player(...);
character[1] = new Zombie(...);
…

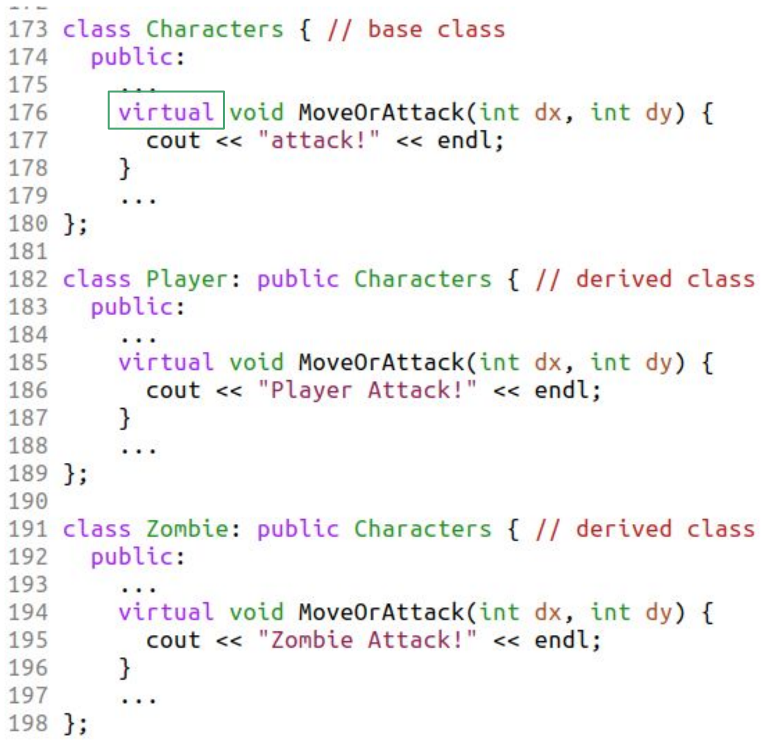
If we use the code as shown, what are the outputs of the following?

character[0]->MoveOrAttack(0, 1);
character[1]->MoveOrAttack(0, 1);

“Attack!”
“Attack!”

This is not what we expect!

Polymorphisms (virtual functions)



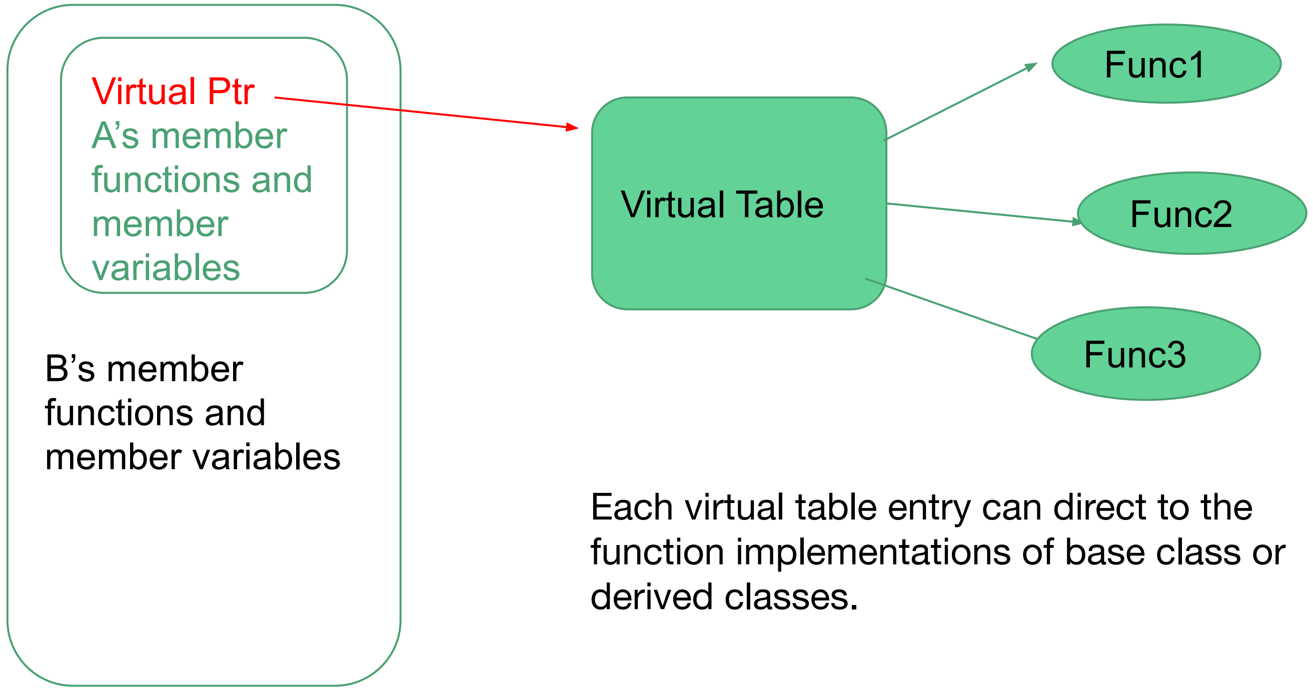
```
class Characters { // base class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      cout << "Player Attack!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      cout << "Zombie Attack!" << endl;
    }
    ...
};
```

Virtual functions tells us that its derived classes can implement their own version (MoveOrAttack in this case).

If the base classes don’t implement the virtual functions, they will by default use the base class’ implementations.

Polymorphisms (virtual functions)

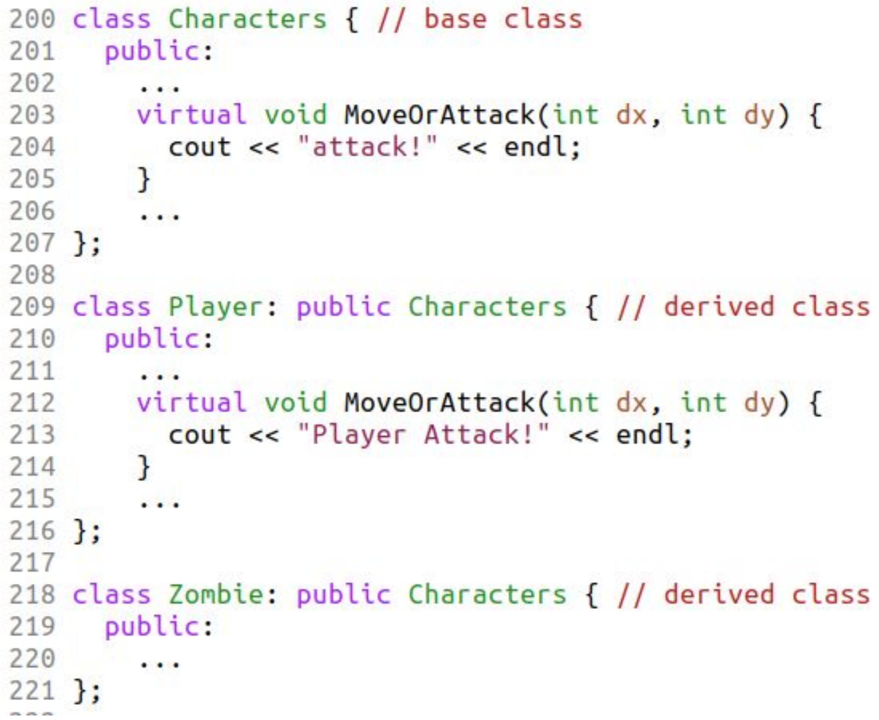
B is a derived class of A.



Virtual Ptr
A's member functions and member variables
B's member functions and member variables
Virtual Table
Func1
Func2
Func3

Each virtual table entry can direct to the function implementations of base class or derived classes.

Polymorphisms (virtual functions)

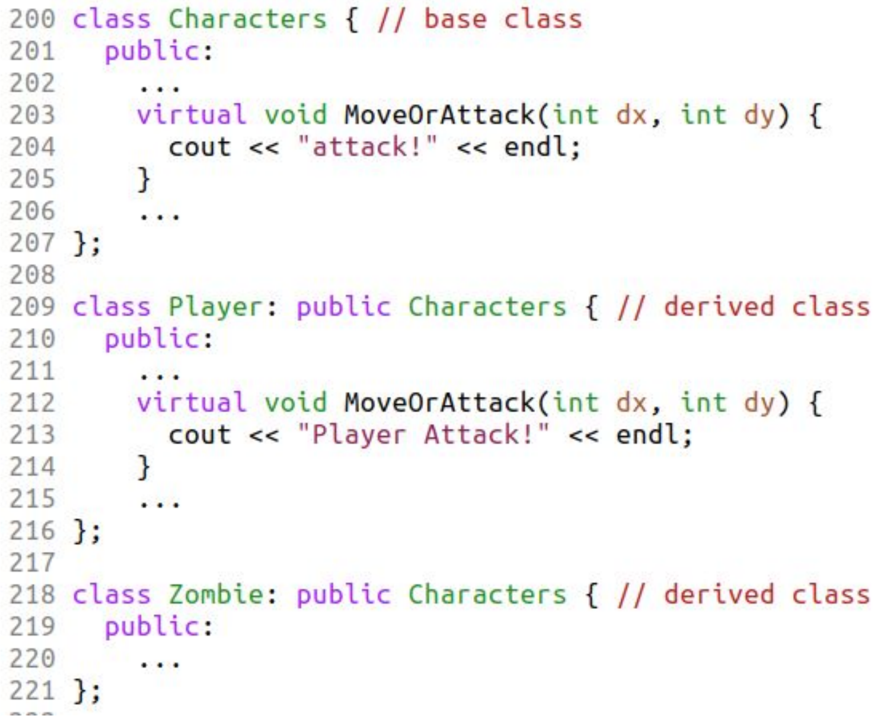


```cpp
class Characters { // base class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      cout << "Player Attack!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
};
```

```
Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
  characters[i]->MoveOrAttack(0, 1);
```

What’s the output?

Polymorphisms (virtual functions)

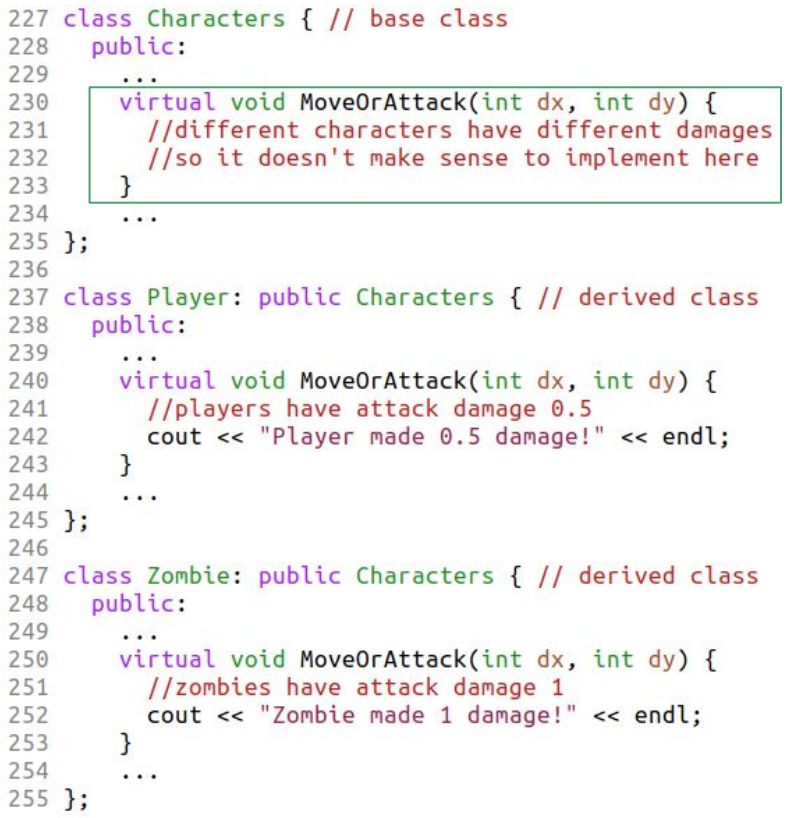


```
class Characters { // base class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      cout << "attack!" << endl;
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      cout << "Player Attack!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
};
```

```
Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
   characters[i]->MoveOrAttack(0, 1);
```

What's the output?
"attack!"
"Player Attack!"
"attack!"

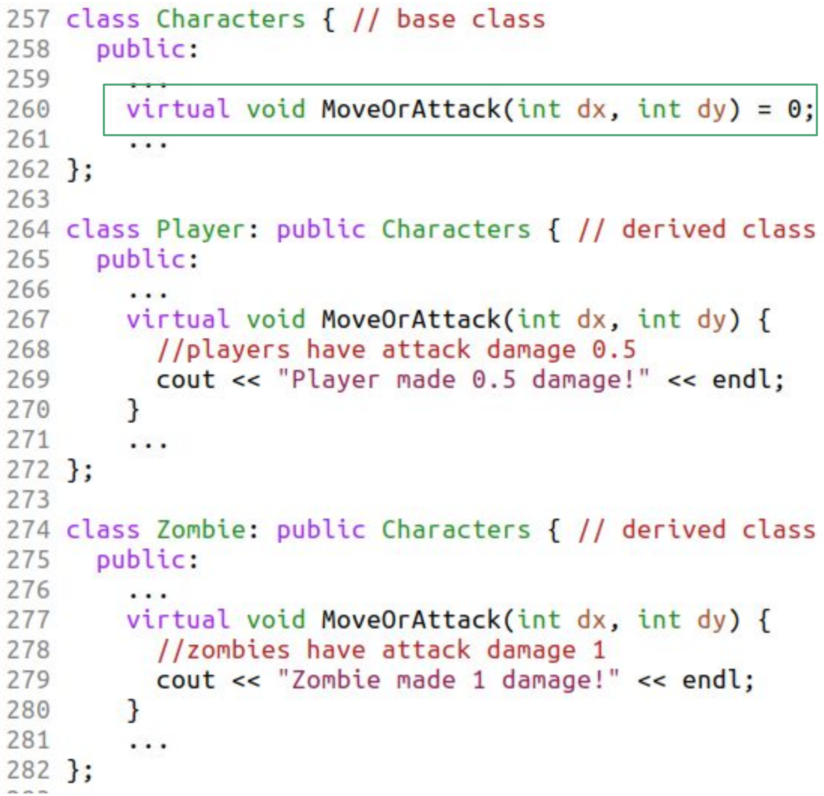
Polymorphisms (virtual functions)



```cpp
class Characters { // base class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      //different characters have different damages
      //so it doesn't make sense to implement here
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      //players have attack damage 0.5
      cout << "Player made 0.5 damage!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      //zombies have attack damage 1
      cout << "Zombie made 1 damage!" << endl;
    }
    ...
};
```

Sometimes, we don't want a base class to implement a virtual function when it doesn't make sense.

Polymorphisms (pure virtual functions)

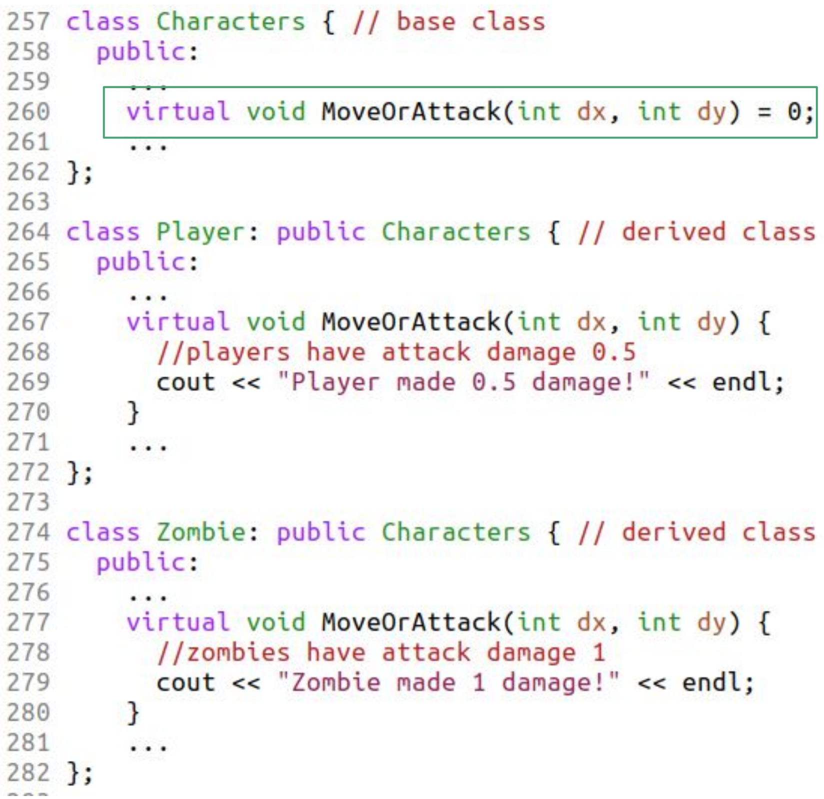


```
class Characters { // base class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) = 0;
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      //players have attack damage 0.5
      cout << "Player made 0.5 damage!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      //zombies have attack damage 1
      cout << "Zombie made 1 damage!" << endl;
    }
    ...
};
```

Pure Virtual Functions allow us to omit the implementation of a virtual function in the base class. However, all its derived classes are **required** to implement the pure virtual function.

A base class that contains at least one pure virtual function is called an **abstract base class** (ABC).

Polymorphisms (pure virtual functions)



```
class Characters { // base class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) = 0;
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      //players have attack damage 0.5
      cout << "Player made 0.5 damage!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    virtual void MoveOrAttack(int dx, int dy) {
      //zombies have attack damage 1
      cout << "Zombie made 1 damage!" << endl;
    }
    ...
};
```

We **cannot** declare or allocate space for an object of abstract base class, since it doesn't implement the pure virtual functions as its derived classes do, which will "confuse" the compiler.

Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);

Characters c(.);

Both won't compile.

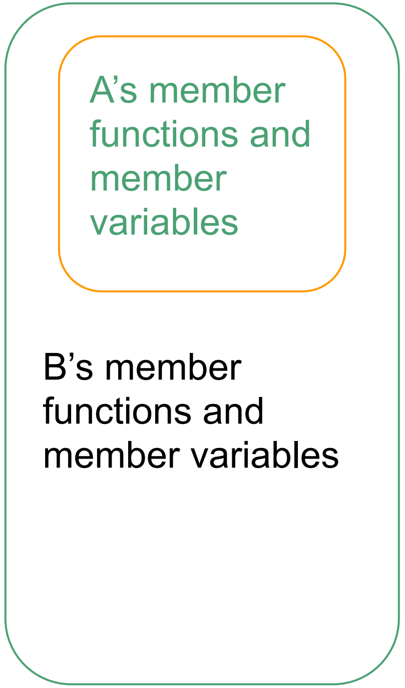
Inheritance (Order of Destruction)

Order of Construction:

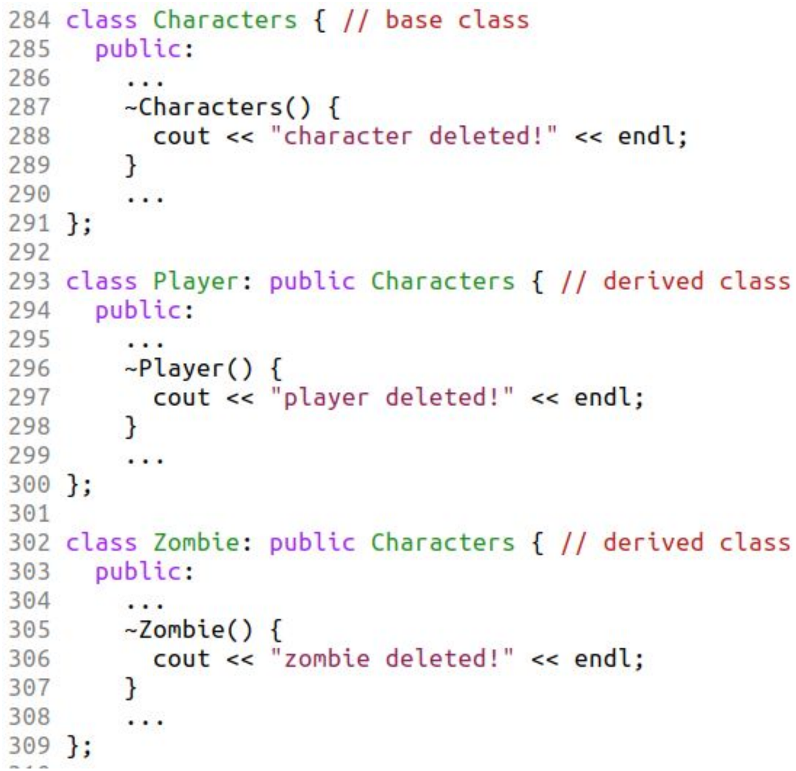
1. Construct the base Class (initialization list)
2. Member variables' default constructor (or initialization list) for derived class.
3. Body of the constructor for the derived class

Order of Destruction:

1. Body of the destructor for the derived class
2. Member variables' default destructor for the derived class
3. Base Class' destructor


A's member functions and member variables
B's member functions and member variables

Inheritance (Destruction)

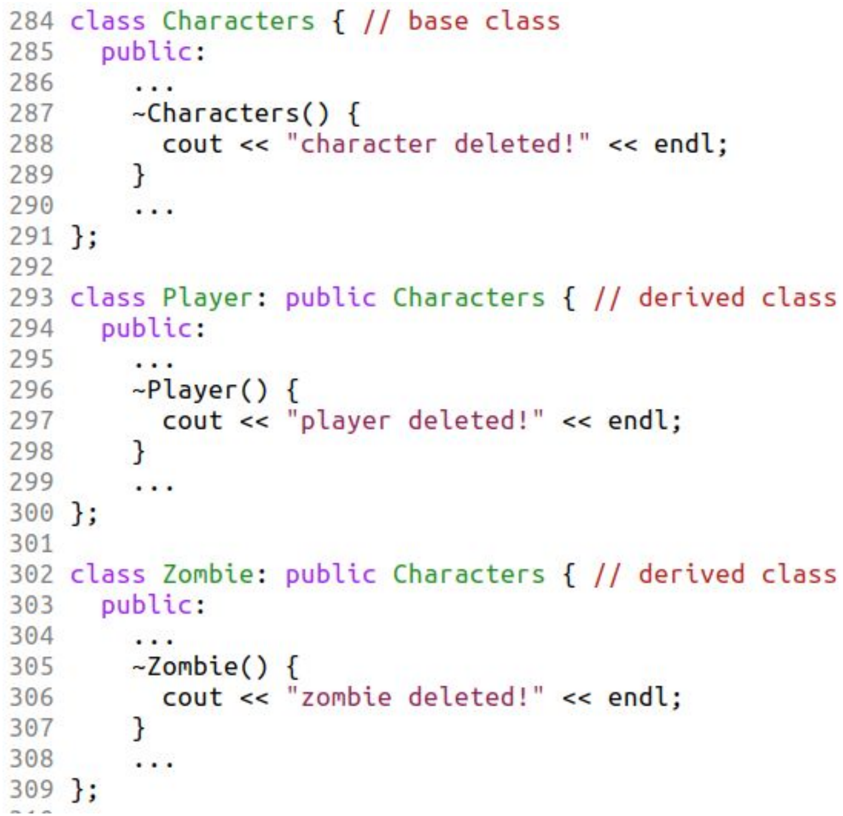


```
class Characters { // base class
  public:
    ...
    ~Characters() {
      cout << "character deleted!" << endl;
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    ~Player() {
      cout << "player deleted!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    ~Zombie() {
      cout << "zombie deleted!" << endl;
    }
    ...
};
```

```
Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
  delete characters[i];
return 0;
```

What’s the output when the program exits?

Inheritance (Destruction)



```
class Characters { // base class
  public:
    ...
    ~Characters() {
      cout << "character deleted!" << endl;
    }
    ...
};

class Player: public Characters { // derived class
  public:
    ...
    ~Player() {
      cout << "player deleted!" << endl;
    }
    ...
};

class Zombie: public Characters { // derived class
  public:
    ...
    ~Zombie() {
      cout << "zombie deleted!" << endl;
    }
    ...
};
```

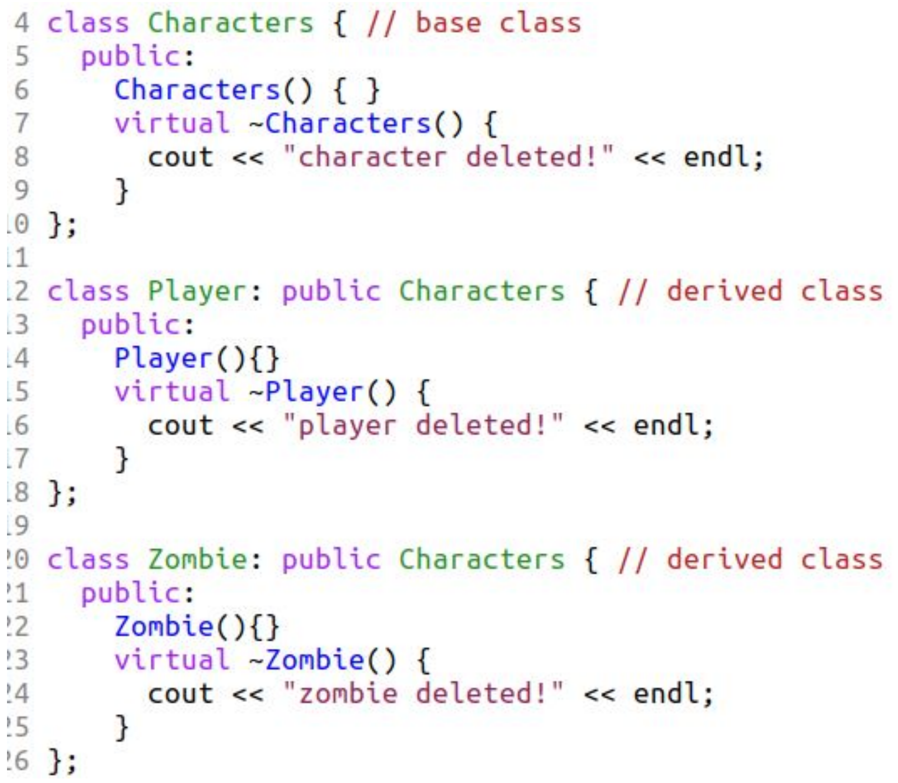
```
Characters* characters[3];
characters[0] = new Characters(.);
characters[1] = new Player(.);
characters[2] = new Zombie(.);
for (int i = 0; i < 3; ++i)
  delete characters[i];
return 0;
```

What’s the output when the program exits?

“character deleted!”
“character deleted!”
“character deleted!”

The destructors must be virtual!

Inheritance (Destruction)



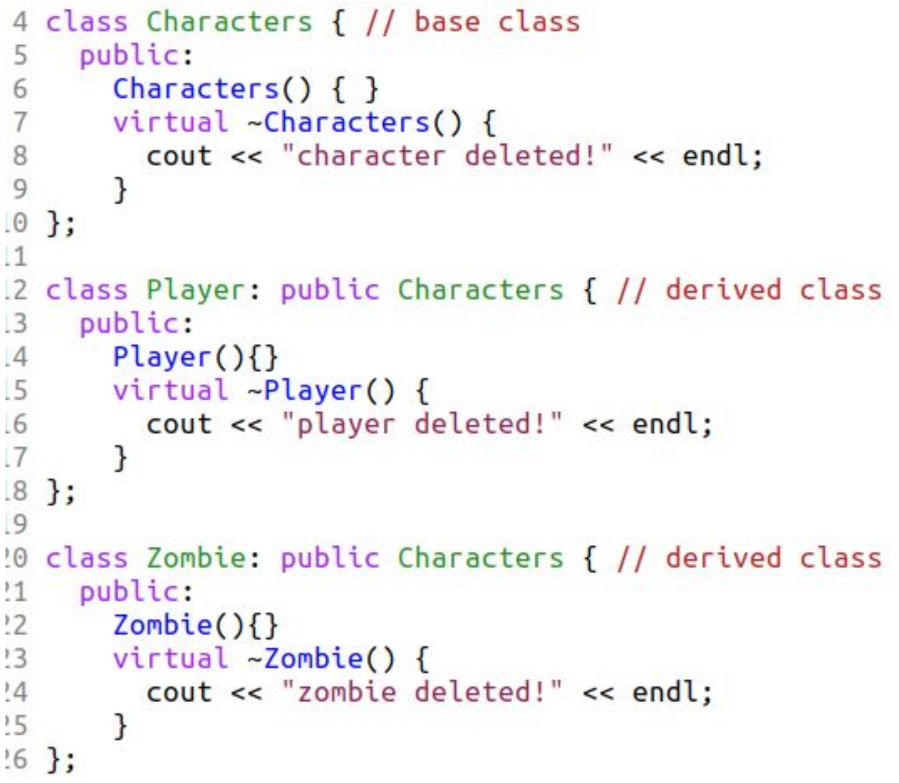
```
class Characters { // base class
  public:
    Characters() { }
    virtual ~Characters() {
      cout << "character deleted!" << endl;
    }
};

class Player: public Characters { // derived class
  public:
    Player(){}
    virtual ~Player() {
      cout << "player deleted!" << endl;
    }
};

class Zombie: public Characters { // derived class
  public:
    Zombie(){}
    virtual ~Zombie() {
      cout << "zombie deleted!" << endl;
    }
};
```

The destructors must be virtual!

Characters characters1;
Player characters2;
Zombie characters3;
return 0;

What's the output when the program exits?

Inheritance (Destruction)



```cpp
class Characters { // base class
  public:
    Characters() { }
    virtual ~Characters() {
      cout << "character deleted!" << endl;
    }
};

class Player: public Characters { // derived class
  public:
    Player(){}
    virtual ~Player() {
      cout << "player deleted!" << endl;
    }
};

class Zombie: public Characters { // derived class
  public:
    Zombie(){}
    virtual ~Zombie() {
      cout << "zombie deleted!" << endl;
    }
};
```

The destructors must be virtual!

```
Characters characters1;
Player characters2;
Zombie characters3;
return 0;
```

What's the output when the program exits?

```
zombie deleted!
character deleted!
player deleted!
character deleted!
character deleted!
```

Summary of Key Concepts: Inheritance and Polymorphism

1. Inheritance (internal structure)
2. Override.
3. Virtual functions
4. Pure virtual functions and abstract base class
5. Construction and Destruction.

Inheritance and Polymorphism
Practice Time!

Problem-Solving Tips!

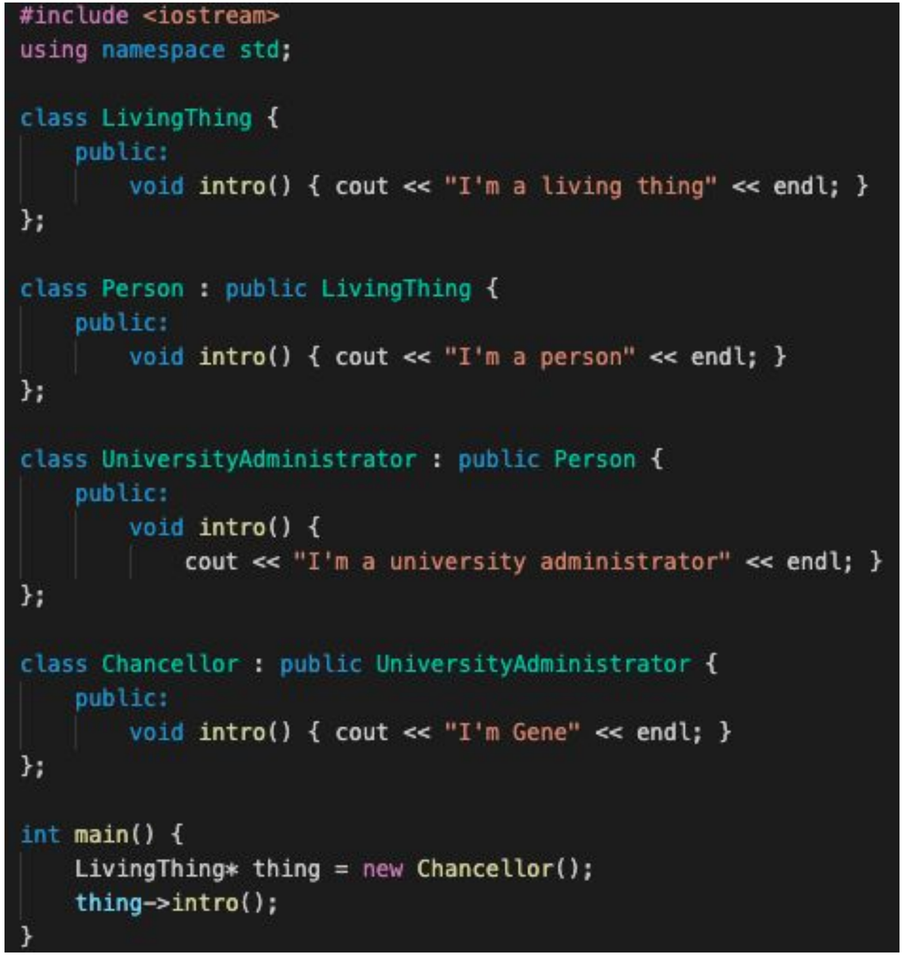
- Check your intuition! Ask yourself the following questions:
    - What is our input? Is it an int? An array? A stack? A pointer?
    - What is our output? Do we write anything to cout?
    - Before you write any code, think about ***how you'd intuitively solve the problem*** step-by-step.
    - Write that down into pseudocode!
    - Try to think through other test cases - if you gave your function this input, what would the expected output be?
    - Using your pseudocode, turn that into valid C++ code
    - Check test cases to make sure your code is sound!

Into A Virtual World

(taken from Wk 5 LA Worksheet, Section 1, Problem 1)

What does the following code output, and what changes do you have to make to it to have it output “I’m Gene”?

(Hint, hint: we *virtually* already did this problem earlier)

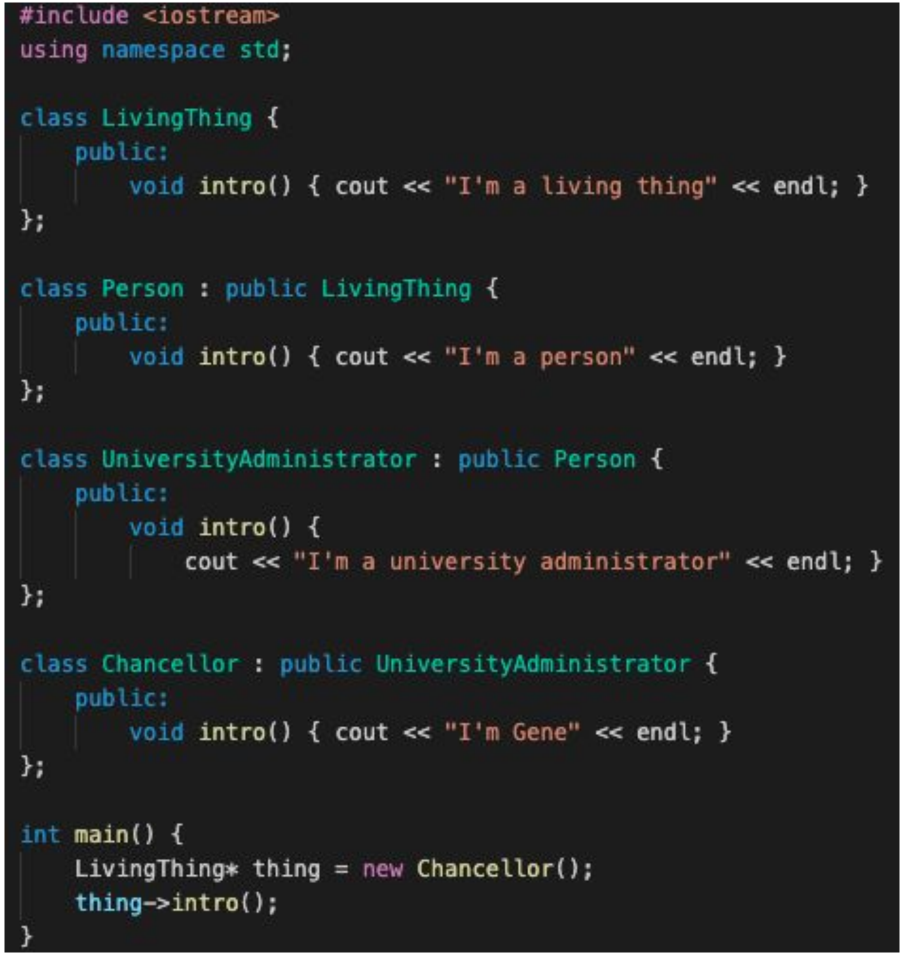


```cpp
#include <iostream>
using namespace std;

class LivingThing {
    public:
        void intro() { cout << "I'm a living thing" << endl; }
};

class Person : public LivingThing {
    public:
        void intro() { cout << "I'm a person" << endl; }
};

class UniversityAdministrator : public Person {
    public:
        void intro() {
            cout << "I'm a university administrator" << endl; }
};

class Chancellor : public UniversityAdministrator {
    public:
        void intro() { cout << "I'm Gene" << endl; }
};

int main() {
    LivingThing* thing = new Chancellor();
    thing->intro();
}
```

Solution

(to Into A Virtual World)

- Here, we're using ***polymorphism***!
- We have a pointer of type LivingThing which points to a Chancellor object
- However, when we call ***thing->intro();*** the compiler cannot guarantee at *compile-time* that the ***thing*** variable is of type Chancellor! All it can guarantee is that the pointer points to an instance of LivingThing.
- Since intro() is not virtual, the program will just call the base class function, i.e. ***LivingThing::intro()***.
- Our output is then: ***"I'm a living thing"***

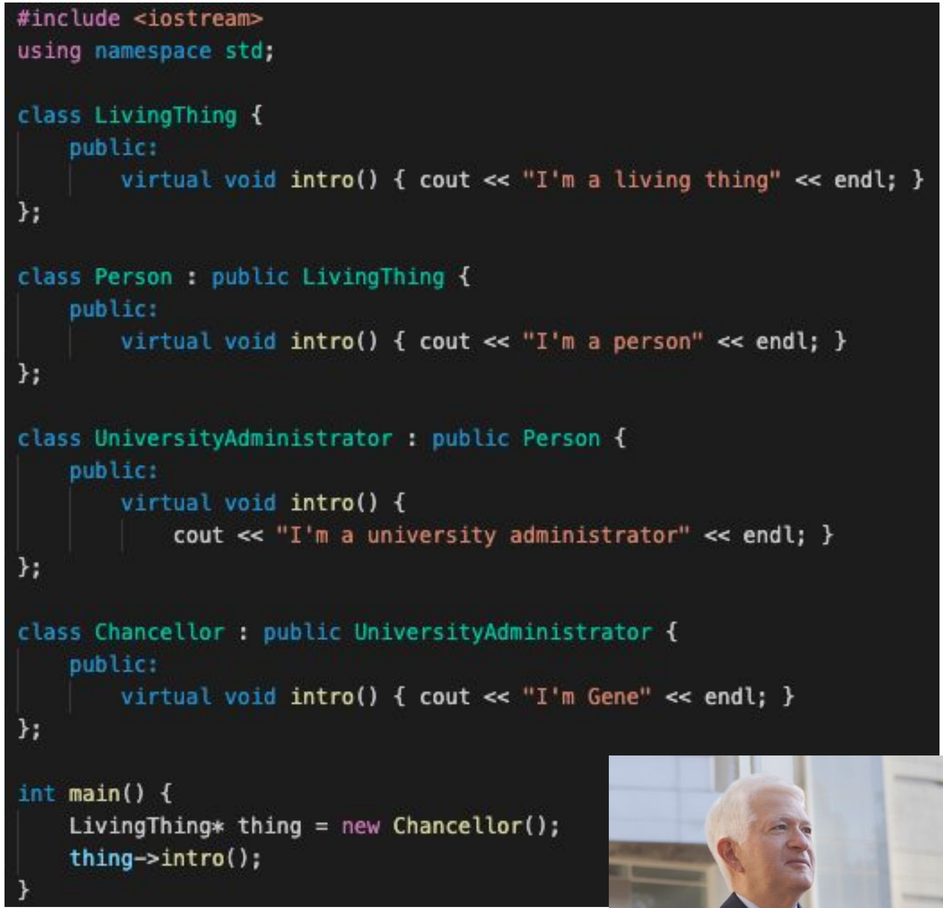


```
#include <iostream>
using namespace std;

class LivingThing {
    public:
        void intro() { cout << "I'm a living thing" << endl; }
};

class Person : public LivingThing {
    public:
        void intro() { cout << "I'm a person" << endl; }
};

class UniversityAdministrator : public Person {
    public:
        void intro() {
            cout << "I'm a university administrator" << endl; }
};

class Chancellor : public UniversityAdministrator {
    public:
        void intro() { cout << "I'm Gene" << endl; }
};

int main() {
    LivingThing* thing = new Chancellor();
    thing->intro();
}
```

Solution, cont.

(to Into A Virtual World)

- In order for the program to output “I’m Gene”, it must call Chancellor’s ***intro()*** function.
- Remember that the ***virtual*** keyword means that every time the function is called, the program checks *at run-time* which version of intro() to call using the v-table!
- So now when we call ***thing->intro();*** we’ll call ***Chancellor::intro()***, and thus print “I’m Gene”.
- (You only need the ***virtual*** keyword in LivingThing, but for clarity you should include it in the derived classes too)



```
#include <iostream>
using namespace std;

class LivingThing {
    public:
        virtual void intro() { cout << "I'm a living thing" << endl; }
};

class Person : public LivingThing {
    public:
        virtual void intro() { cout << "I'm a person" << endl; }
};

class UniversityAdministrator : public Person {
    public:
        virtual void intro() {
            cout << "I'm a university administrator" << endl; }
};

class Chancellor : public UniversityAdministrator {
    public:
        virtual void intro() { cout << "I'm Gene" << endl; }
};

int main() {
    LivingThing* thing = new Chancellor();
    thing->intro();
}
```

Classes All The Way Down

(taken from Wk 5 LA Worksheet, Section 1, Problem 4)

Would the following work in C++? Why or why not?

```
class B;

class A : public B { … code for A … };

class B : public A { … code for B … };
```

Solution

(to Classes All The Way Down)

```
class B;

class A : public B { … code for A … };

class B : public A { … code for B … };
```

If you said ***no***, you’re 100% correct!

Can anyone explain why?

Solution, cont.

(to Classes All The Way Down)

```
class B;

class A : public B { … code for A … };

class B : public A { … code for B … };
```

If you said ***no***, you’re 100% correct!

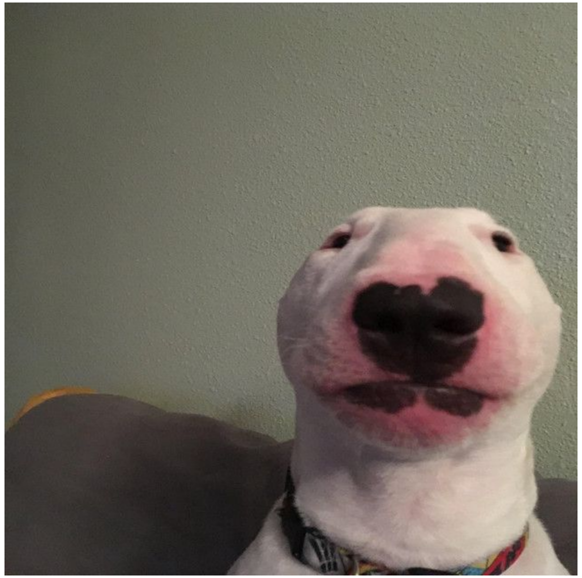
Conceptually, this code is saying:
“A is a proper subset of B, and B is a proper subset of A”, which is nonsense.

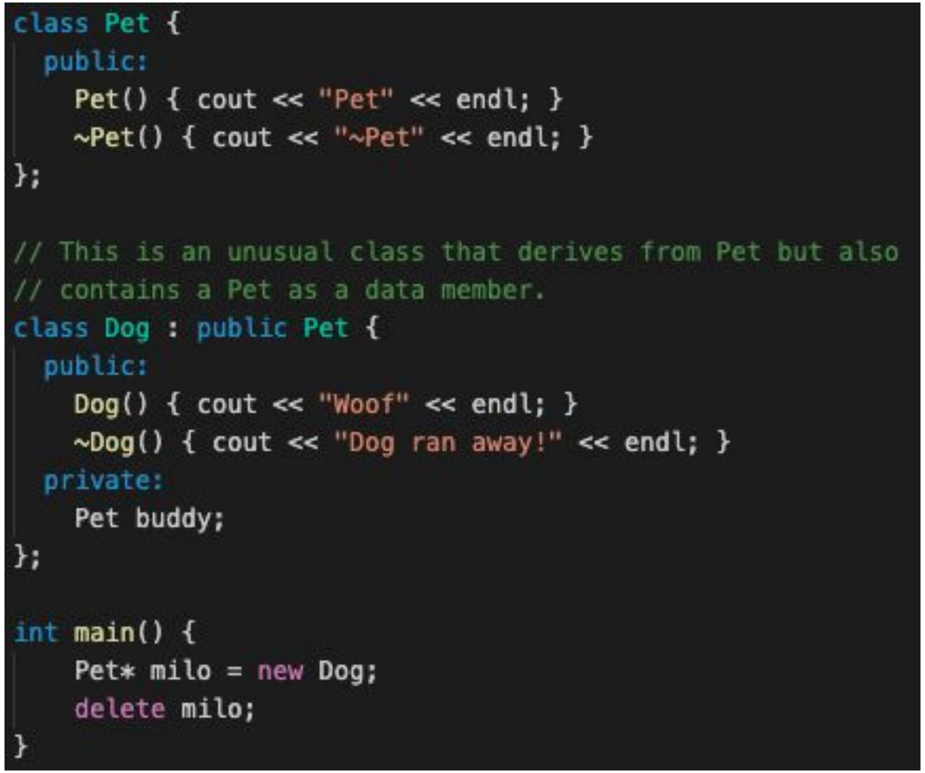
Practically, every object of a derived class contains an instance of the base class. If the code above were legal, a B object would contain an A object that contains a B object that contains an A object, ad infinitum.

What The Dog Doin?

(taken from Wk 5 LA Worksheet, Section 1, Problems 2 and 3)

What does this code output?





```cpp
class Pet {
  public:
    Pet() { cout << "Pet" << endl; }
    ~Pet() { cout << "~Pet" << endl; }
};

// This is an unusual class that derives from Pet but also
// contains a Pet as a data member.
class Dog : public Pet {
  public:
    Dog() { cout << "Woof" << endl; }
    ~Dog() { cout << "Dog ran away!" << endl; }
  private:
    Pet buddy;
};

int main() {
    Pet* milo = new Dog;
    delete milo;
}
```

Solution

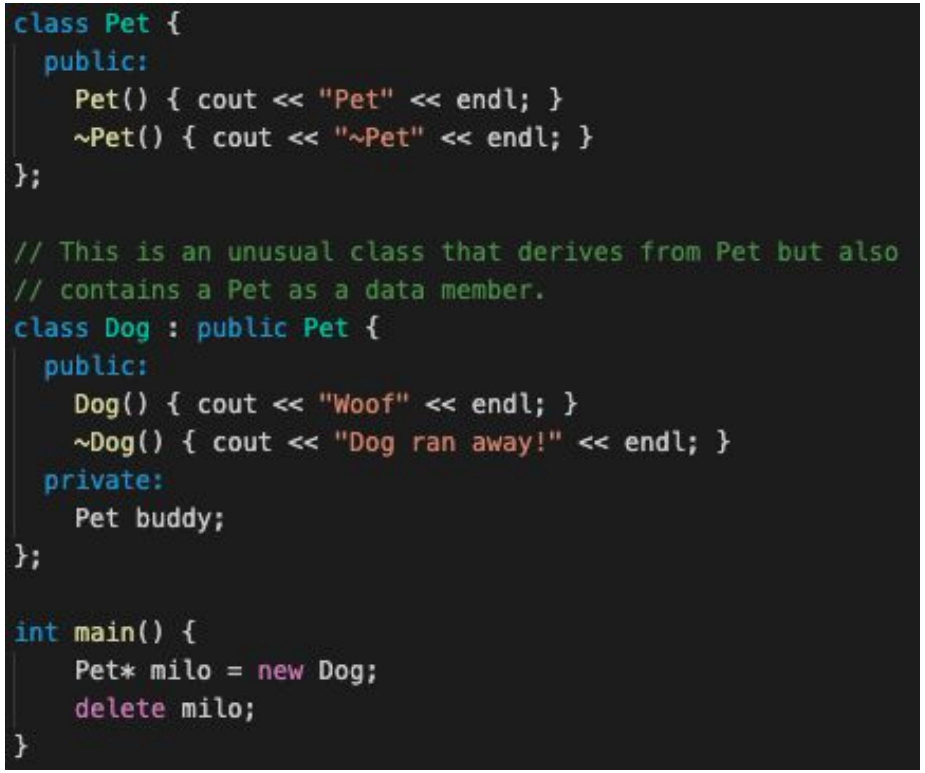
(to What The Dog Doin?)

Remember how construction and destruction works with derived objects!

Constructing a derived class:

- Call base class constructor
- Call constructors of member variables (if needed)
- Run body of derived class constructor

The order is reversed for destruction!



```cpp
class Pet {
  public:
    Pet() { cout << "Pet" << endl; }
    ~Pet() { cout << "~Pet" << endl; }
};

// This is an unusual class that derives from Pet but also
// contains a Pet as a data member.
class Dog : public Pet {
  public:
    Dog() { cout << "Woof" << endl; }
    ~Dog() { cout << "Dog ran away!" << endl; }
  private:
    Pet buddy;
};

int main() {
    Pet* milo = new Dog;
    delete milo;
}
```

Solution, cont.

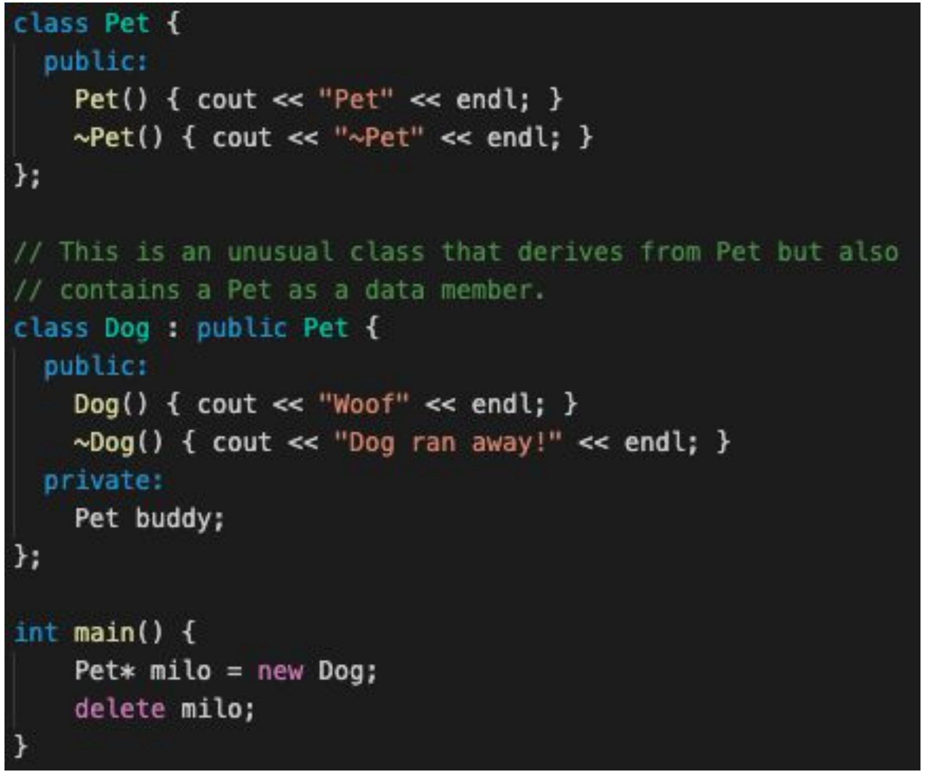
(to What The Dog Doin?)

Some of you may have gotten this:

```
Pet
Pet
Woof
Dog ran away!
~Pet
~Pet
```

This is actually ***incorrect***!

Can anyone explain why?



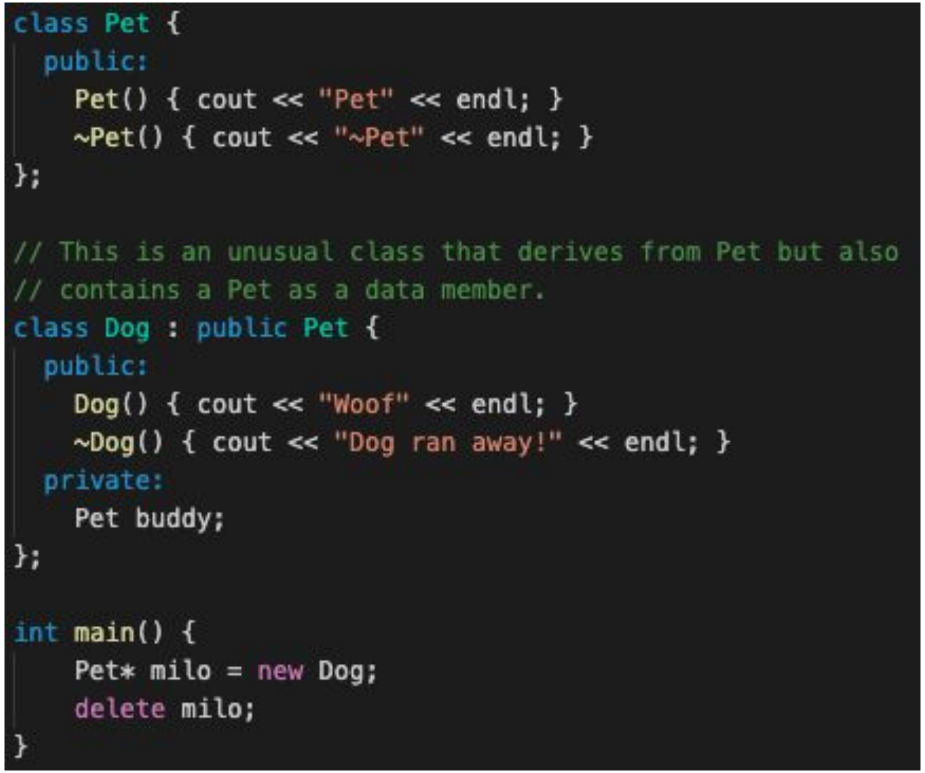
```
class Pet {
  public:
    Pet() { cout << "Pet" << endl; }
    ~Pet() { cout << "~Pet" << endl; }
};

// This is an unusual class that derives from Pet but also
// contains a Pet as a data member.
class Dog : public Pet {
  public:
    Dog() { cout << "Woof" << endl; }
    ~Dog() { cout << "Dog ran away!" << endl; }
  private:
    Pet buddy;
};

int main() {
    Pet* milo = new Dog;
    delete milo;
}
```

Solution, cont.

(to What The Dog Doin?)

Let's trace through the code.

- When we call ***Pet* milo = new Dog;***
  - We call the base class (Pet) constructor
  - We call the constructor of member variables (in this case, just Pet)
  - We run the body of Dog's constructor
- When we call ***delete milo;***
  - The program ***doesn't know*** that milo points to a Dog - it can only guarantee that it points to a Pet object
  - Since Pet's destructor is **not virtual**, the program ***only calls Pet's destructor***.
- This is also undefined behavior!



```
class Pet {
  public:
    Pet() { cout << "Pet" << endl; }
    ~Pet() { cout << "~Pet" << endl; }
};

// This is an unusual class that derives from Pet but also
// contains a Pet as a data member.
class Dog : public Pet {
  public:
    Dog() { cout << "Woof" << endl; }
    ~Dog() { cout << "Dog ran away!" << endl; }
  private:
    Pet buddy;
};

int main() {
    Pet* milo = new Dog;
    delete milo;
}
```

Solution, cont.

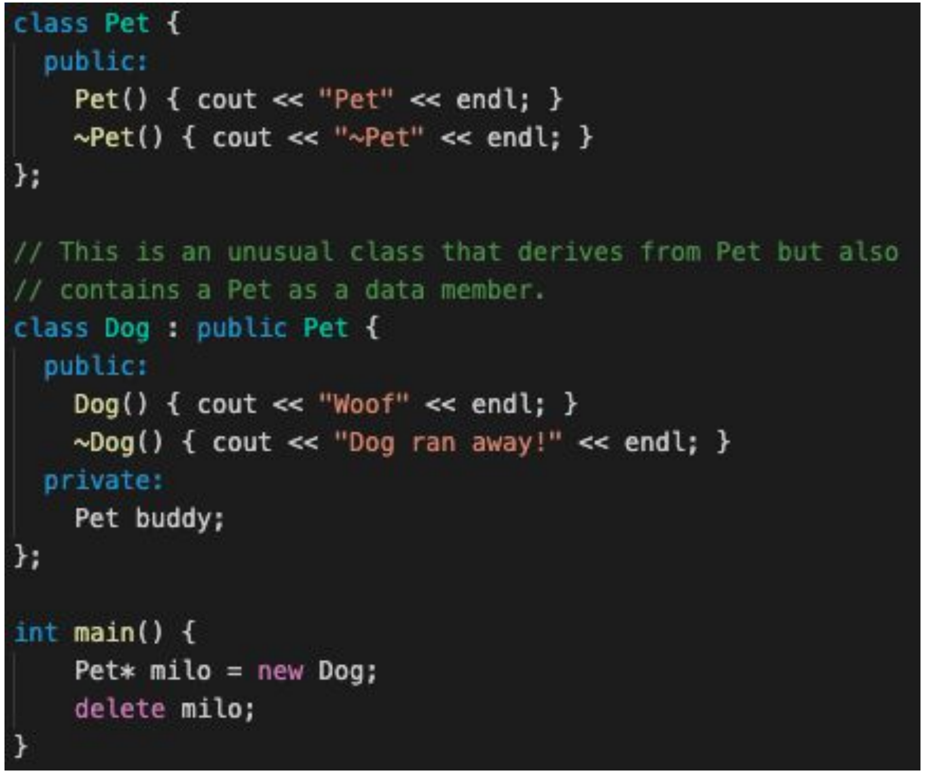
(to What The Dog Doin?)

The actual output would be:

```
Pet
Pet
Woof
~Pet
```

After we output the last line, we would have **undefined behavior**.

To get the output from earlier, we would need to ***make Pet's destructor virtual***!

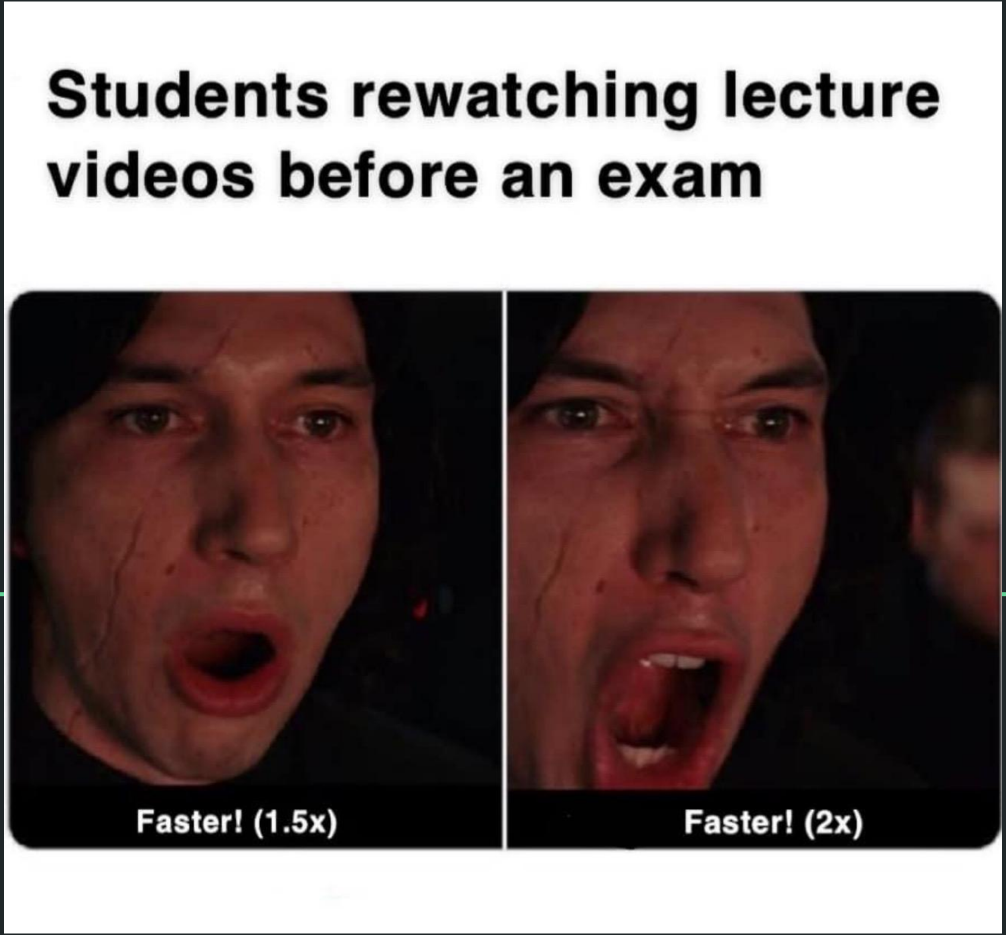


```
class Pet {
  public:
    Pet() { cout << "Pet" << endl; }
    ~Pet() { cout << "~Pet" << endl; }
};

// This is an unusual class that derives from Pet but also
// contains a Pet as a data member.
class Dog : public Pet {
  public:
    Dog() { cout << "Woof" << endl; }
    ~Dog() { cout << "Dog ran away!" << endl; }
  private:
    Pet buddy;
};

int main() {
    Pet* milo = new Dog;
    delete milo;
}
```

Another reminder to fill out the feedback form :)

congrats, you've made it to Break Time

enjoy a Complimentary Meme on the house



Recursion

The core idea aligns with mathematical induction.

To show a statement P(N) is true for all N>=n, we only need to show:

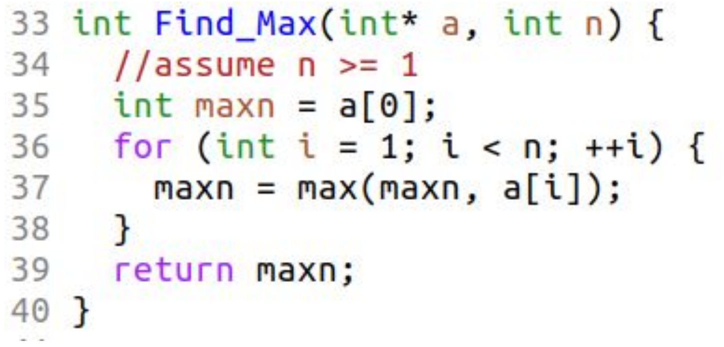
- Base case: P(n) is true.
- Inductive Hypothesis: P(k) is true implies P(k+1) is true for all k>=n.

In computer science language, to solve a problem with size N, we may divide it into smaller subproblems so that combining the results of the subproblems produces the result for the larger problem.

Fun fact: there are some functional programming languages such as LISP and Ocaml that do not often support loops, so one has to implement recursions to traverse a list/array.

Recursion: finding maximal value in an array

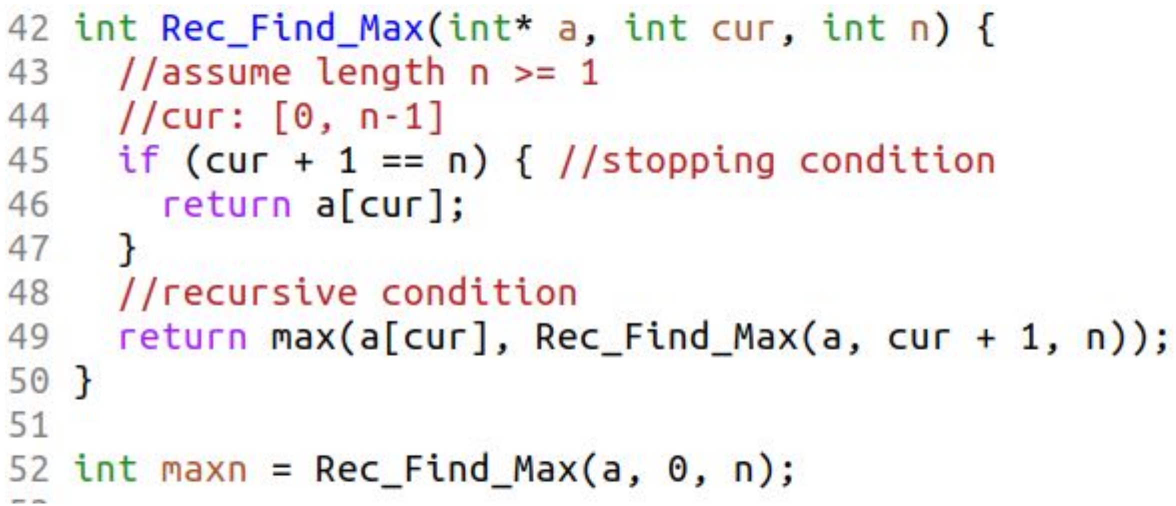
Using For loop: search one by one.



```
int Find_Max(int* a, int n) {
  //assume n >= 1
  int maxn = a[0];
  for (int i = 1; i < n; ++i) {
    maxn = max(maxn, a[i]);
  }
  return maxn;
}
```

Recursion: finding maximal value in an array

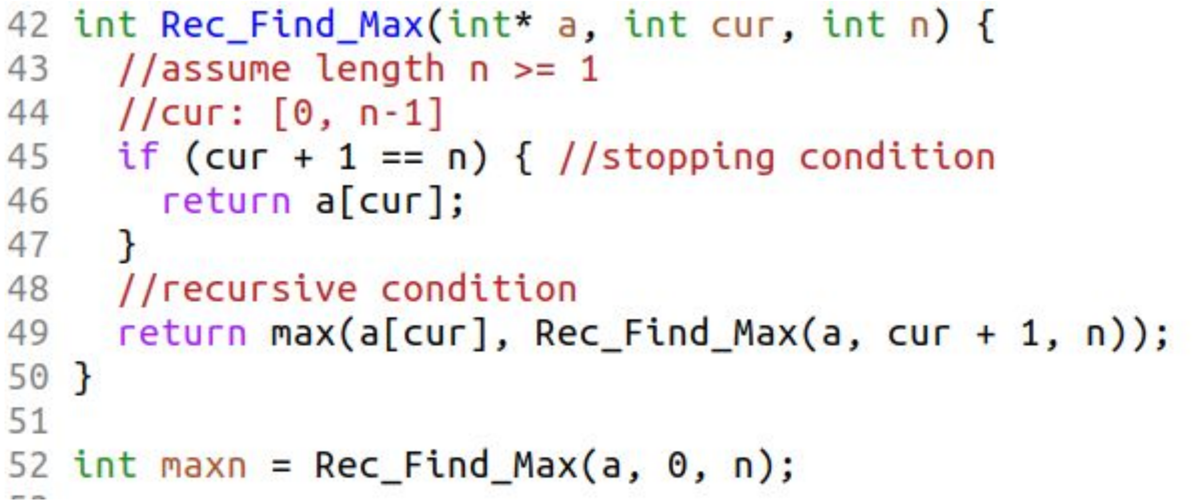
Recursion inspired by For loop.



```
int Rec_Find_Max(int* a, int cur, int n) {
  //assume length n >= 1
  //cur: [0, n-1]
  if (cur + 1 == n) { //stopping condition
    return a[cur];
  }
  //recursive condition
  return max(a[cur], Rec_Find_Max(a, cur + 1, n));
}

int maxn = Rec_Find_Max(a, 0, n);
```

Recursion: finding maximal value in an array

Recursion inspired by For loop.

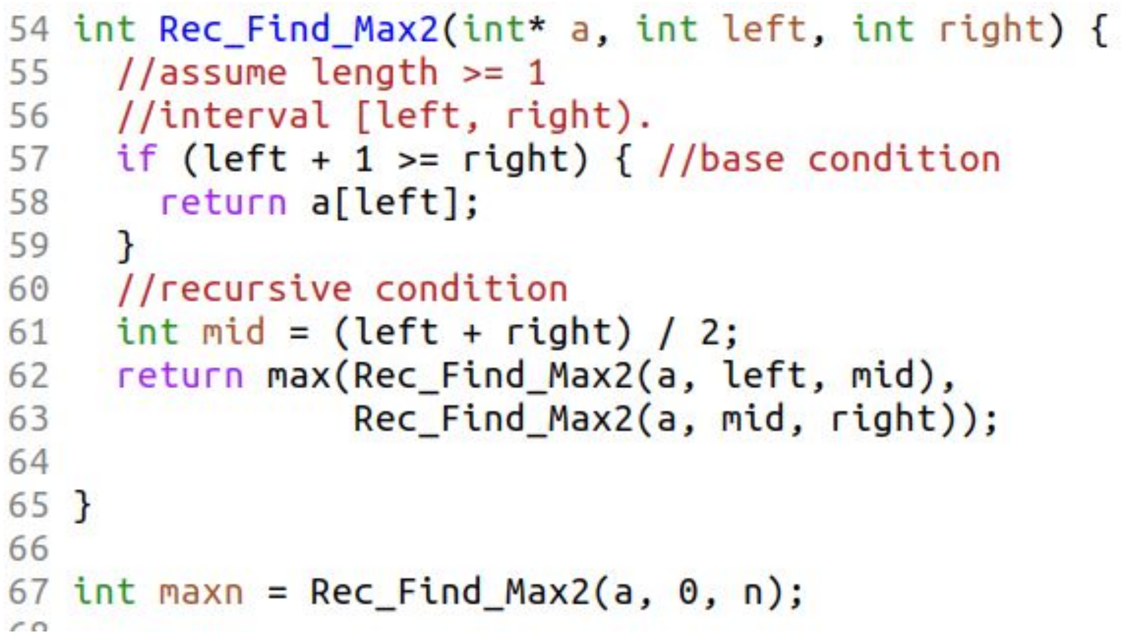


```
int Rec_Find_Max(int* a, int cur, int n) {
  //assume length n >= 1
  //cur: [0, n-1]
  if (cur + 1 == n) { //stopping condition
    return a[cur];
  }
  //recursive condition
  return max(a[cur], Rec_Find_Max(a, cur + 1, n));
}

int maxn = Rec_Find_Max(a, 0, n);
```

Q: Does it have to be one by one? How about divide in half?

Recursion: finding maximal value in an array

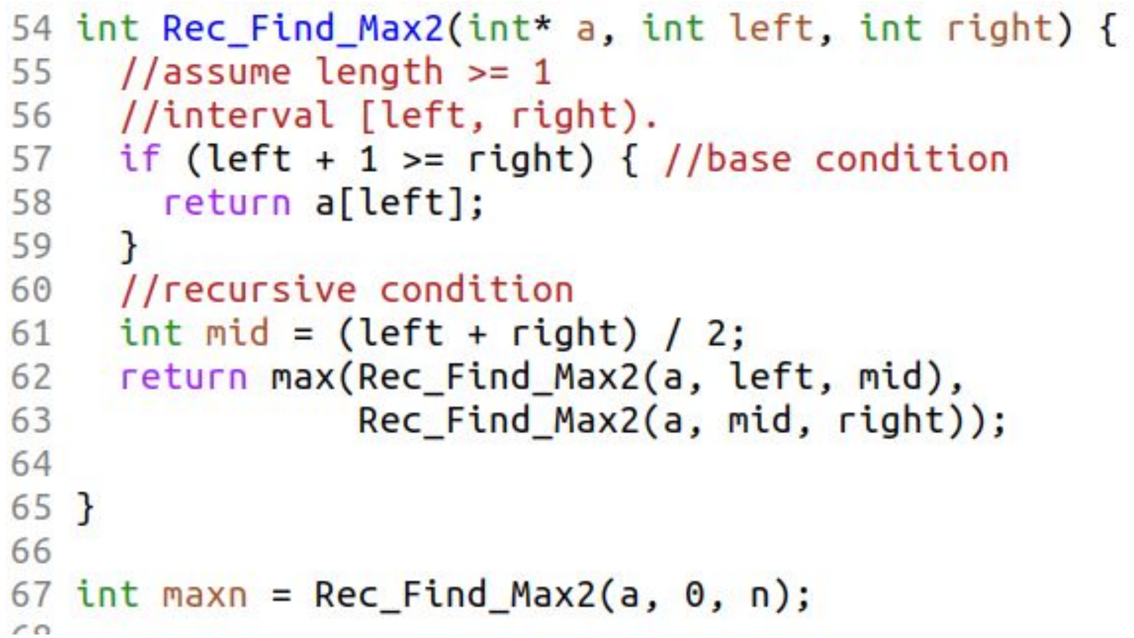
Recursion inspired by binary division.



```c
int Rec_Find_Max2(int* a, int left, int right) {
  //assume length >= 1
  //interval [left, right).
  if (left + 1 >= right) { //base condition
    return a[left];
  }
  //recursive condition
  int mid = (left + right) / 2;
  return max(Rec_Find_Max2(a, left, mid),
             Rec_Find_Max2(a, mid, right));

}

int maxn = Rec_Find_Max2(a, 0, n);
```

Recursion: finding maximal value in an array

Recursion inspired by binary division.



```
int Rec_Find_Max2(int* a, int left, int right) {
  //assume length >= 1
  //interval [left, right).
  if (left + 1 >= right) { //base condition
    return a[left];
  }
  //recursive condition
  int mid = (left + right) / 2;
  return max(Rec_Find_Max2(a, left, mid),
             Rec_Find_Max2(a, mid, right));

}

int maxn = Rec_Find_Max2(a, 0, n);
```

Q: How to measure the efficiency of the recursion algorithms?

Recursion: time efficiency measure

Time complexity: the number of operations to be performed $T(n)$, with respect to input length n.

For loop: $T(n) \sim n$

Recursion inspired by For loop: $T(n) = T(n-1) + 1$

Recursion inspired by binary division: $T(n) = 2T(n/2) + 1$

To compare, we need to find the closed form expressions of them.

Recursion: efficiency measure

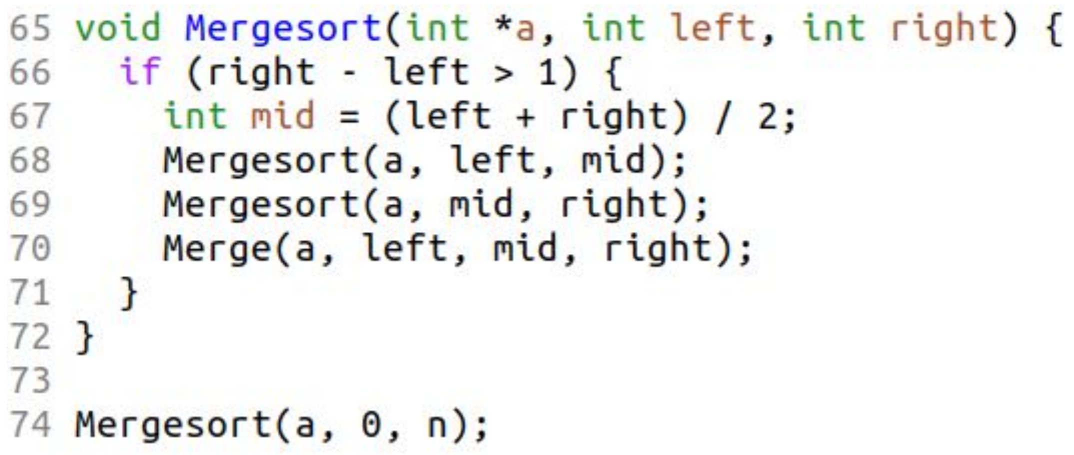
For loop: $T(n) \sim n$

Recursion inspired by For loop: $T(n) = T(n-1) + 1$ ----> $T(n) \sim n$

Recursion inspired by binary division: $T(n) = 2T(n/2) + 1$ ----> $T(n) \sim n$

In this particular case, all have same the order of time complexity up to constant.

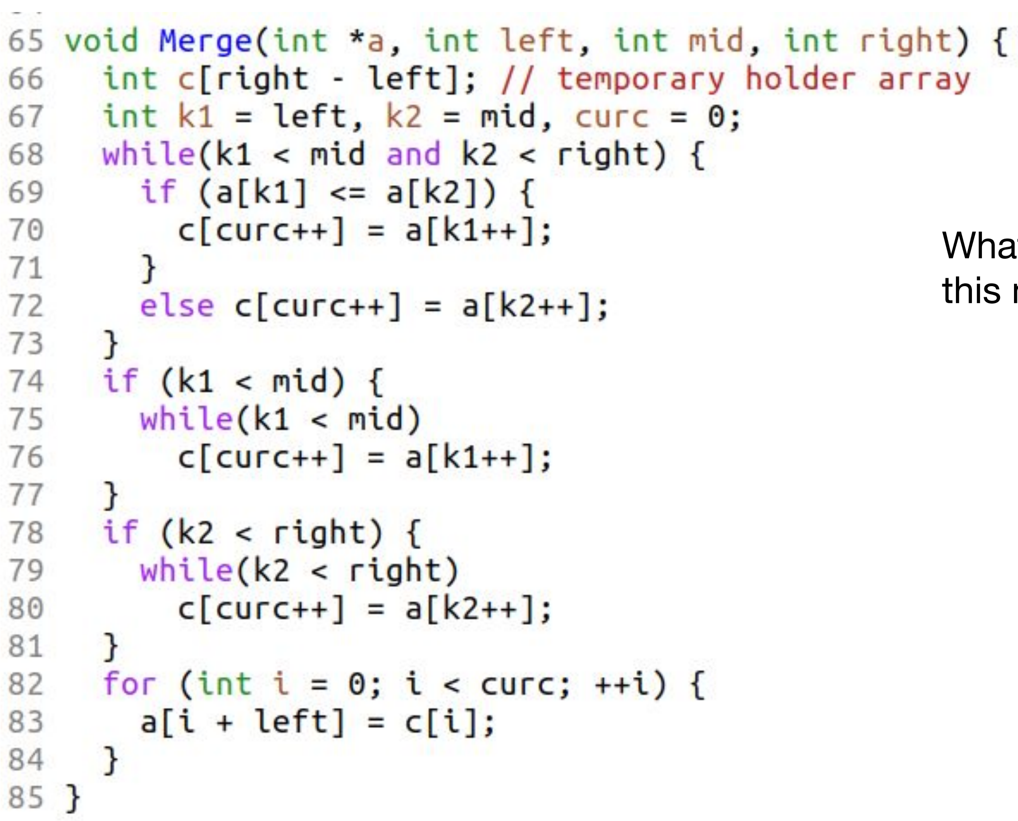
Recursion: mergesort



```c
void Mergesort(int *a, int left, int right) {
  if (right - left > 1) {
    int mid = (left + right) / 2;
    Mergesort(a, left, mid);
    Mergesort(a, mid, right);
    Merge(a, left, mid, right);
  }
}

Mergesort(a, 0, n);
```

How to merge?

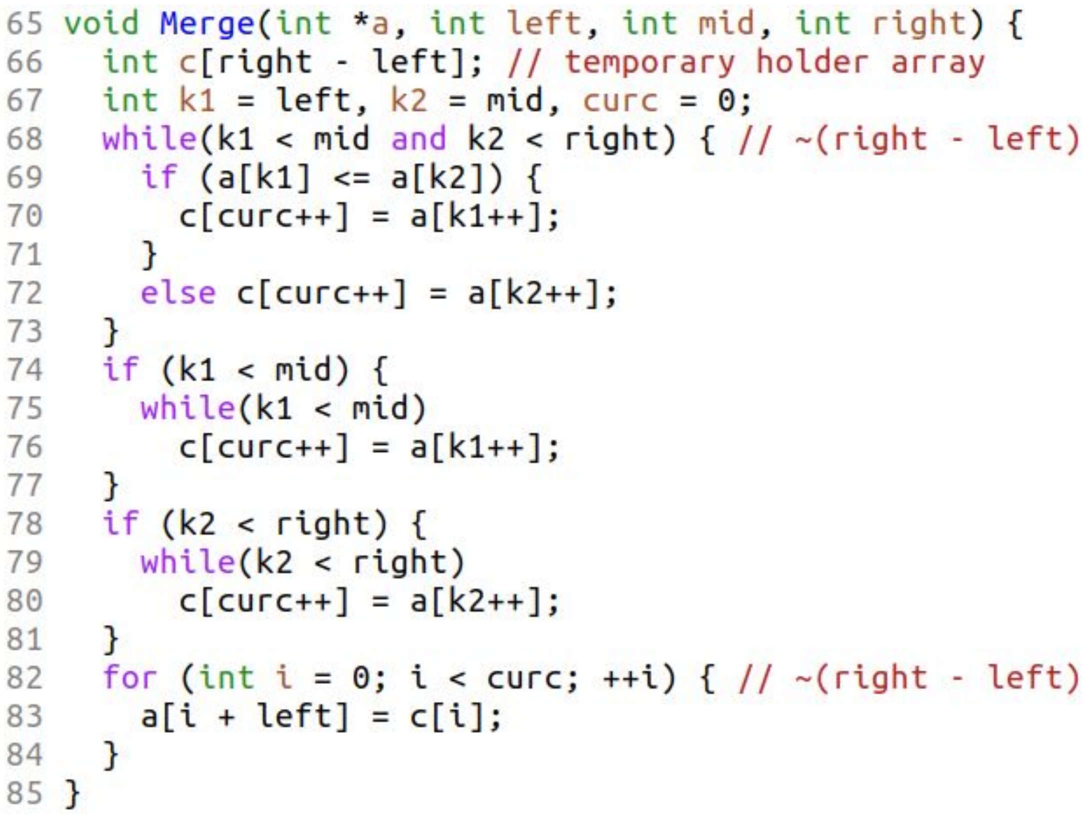
Recursion: mergesort’s merge



```
void Merge(int *a, int left, int mid, int right) {
  int c[right - left]; // temporary holder array
  int k1 = left, k2 = mid, curc = 0;
  while(k1 < mid and k2 < right) {
    if (a[k1] <= a[k2]) {
      c[curc++] = a[k1++];
    }
    else c[curc++] = a[k2++];
  }
  if (k1 < mid) {
    while(k1 < mid)
      c[curc++] = a[k1++];
  }
  if (k2 < right) {
    while(k2 < right)
      c[curc++] = a[k2++];
  }
  for (int i = 0; i < curc; ++i) {
    a[i + left] = c[i];
  }
}
```

What’s the time complexity for this merge implementation?

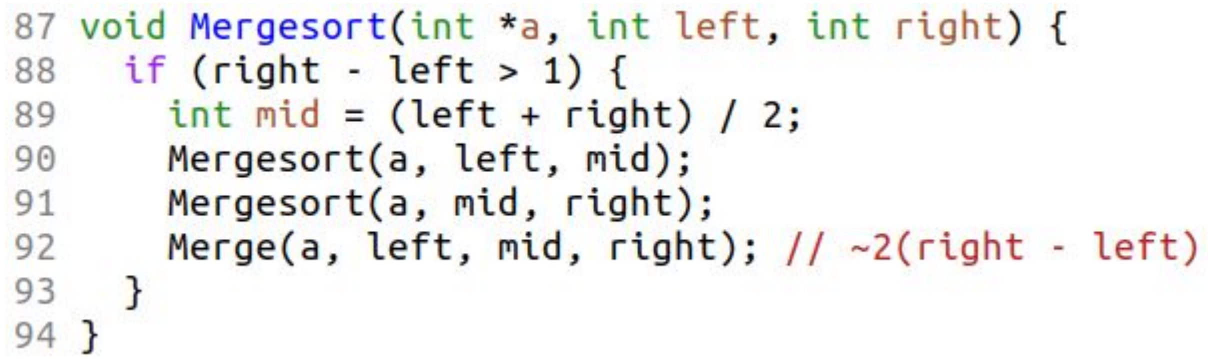
Recursion: mergesort's merge



```
void Merge(int *a, int left, int mid, int right) {
  int c[right - left]; // temporary holder array
  int k1 = left, k2 = mid, curc = 0;
  while(k1 < mid and k2 < right) { // ~(right - left)
    if (a[k1] <= a[k2]) {
      c[curc++] = a[k1++];
    }
    else c[curc++] = a[k2++];
  }
  if (k1 < mid) {
    while(k1 < mid)
      c[curc++] = a[k1++];
  }
  if (k2 < right) {
    while(k2 < right)
      c[curc++] = a[k2++];
  }
  for (int i = 0; i < curc; ++i) { // ~(right - left)
    a[i + left] = c[i];
  }
}
```

For length n interval, ~2n.

Recursion: mergesort complexity

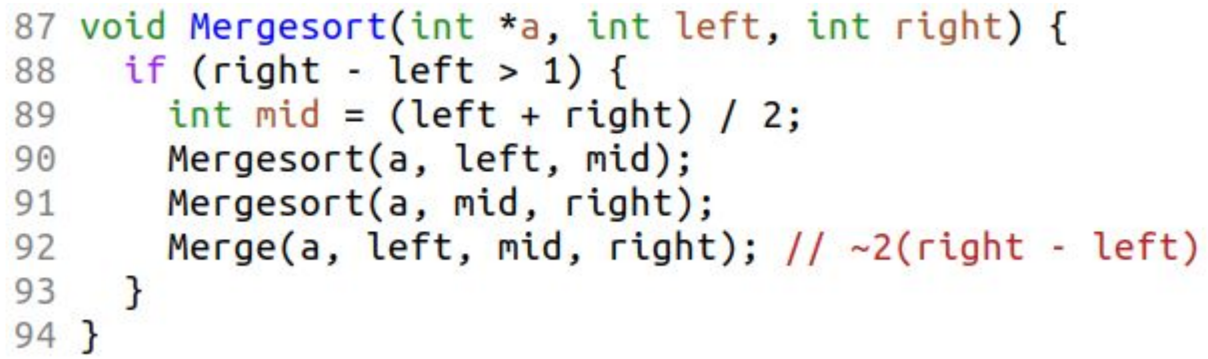


```
void Mergesort(int *a, int left, int right) {
  if (right - left > 1) {
    int mid = (left + right) / 2;
    Mergesort(a, left, mid);
    Mergesort(a, mid, right);
    Merge(a, left, mid, right); // ~2(right - left)
  }
}
```

$T(n) = 2T(n/2) + 2n$.

Can you find a closed form of the complexity $T(n)$?

Recursion: mergesort complexity

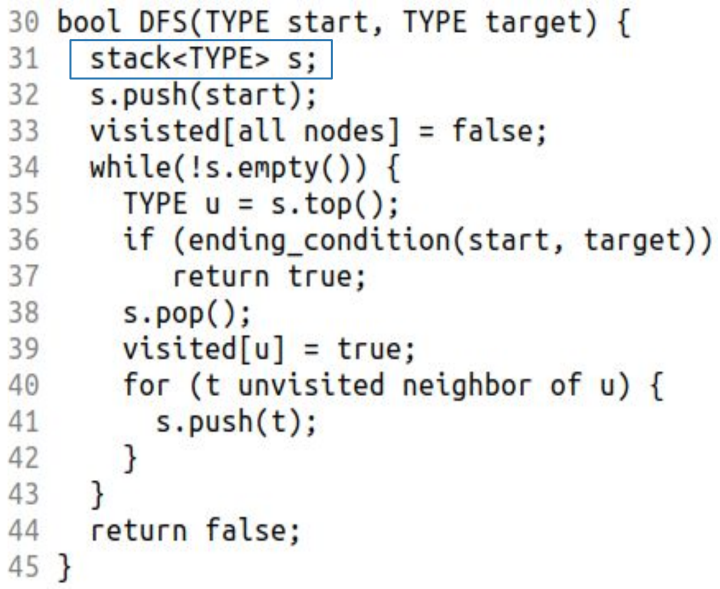


```
void Mergesort(int *a, int left, int right) {
  if (right - left > 1) {
    int mid = (left + right) / 2;
    Mergesort(a, left, mid);
    Mergesort(a, mid, right);
    Merge(a, left, mid, right); // ~2(right - left)
  }
}
```

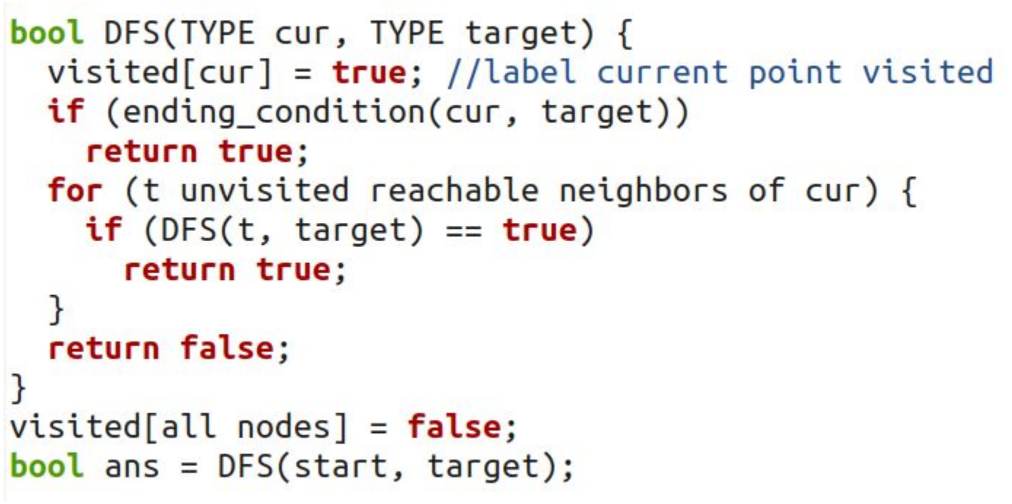
$T(n) = 2T(n/2) + 2n$.
Can you find a closed form of the complexity $T(n)$?

$T(n) \sim 2n\log n \sim n\log n$ if we ignore the constant multiplier

Recursion: Depth First Search (DFS)



```
bool DFS(TYPE start, TYPE target) {
  stack<TYPE> s;
  s.push(start);
  visited[all nodes] = false;
  while(!s.empty()) {
    TYPE u = s.top();
    if (ending_condition(start, target))
       return true;
    s.pop();
    visited[u] = true;
    for (t unvisited neighbor of u) {
      s.push(t);
    }
  }
  return false;
}
```



```
bool DFS(TYPE cur, TYPE target) {
  visited[cur] = true; //label current point visited
  if (ending_condition(cur, target))
    return true;
  for (t unvisited reachable neighbors of cur) {
    if (DFS(t, target) == true)
      return true;
  }
  return false;
}
visited[all nodes] = false;
bool ans = DFS(start, target);
```

Summary of key concepts: Recursion

1. Base case.
2. Recursive step.
3. Time complexity analysis.

recursion

recursion

recursion

recursion

Practice Time!

Do Geese See God?

(taken from Wk 5 LA Worksheet, Section 2, Problem 2)

Implement the function ***isPalindrome*** recursively. The function should return whether the given string is a palindrome. A palindrome is described as a word, phrase or sequence of characters that reads the same forward and backwards.

interface and expected output:

```
bool isPalindrome(string foo);

isPalindrome("kayak"); // true

isPalindrome("stanley yelnats"); // true

isPalindrome("LAs rock"); // false (but the sentiment is true :))
```

Solution

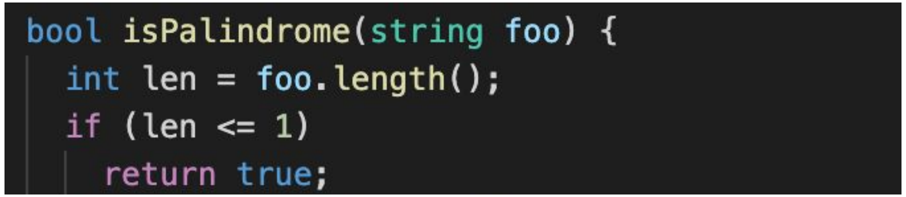
(to Do Geese See God?)

What's our base case?

All one-letter words are palindromes! Then let's return true if our string's length is 1 or less.

What about the recursive step?

Think about how you'd do this intuitively. For a word to be a palindrome, the first and last letters have to match. Then, we keep going inwards to the center until we run out of letters or we find a mismatch.



```
bool isPalindrome(string foo) {
  int len = foo.length();
  if (len <= 1)
    return true;
```

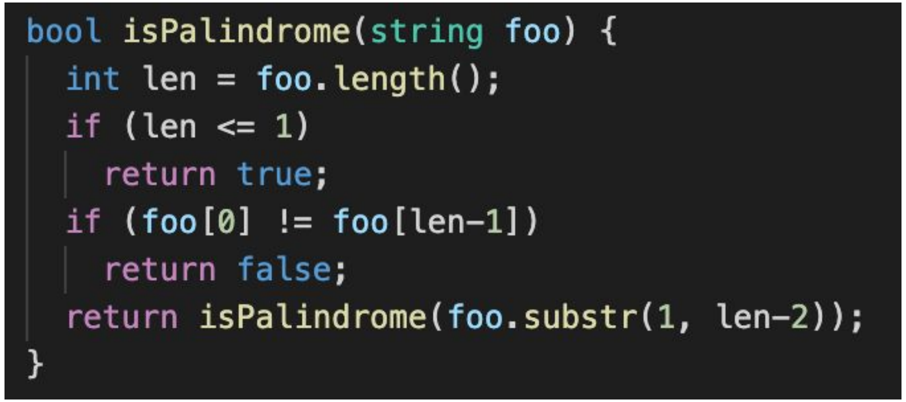
Solution, cont.

(to Do Geese See God?)

How do we implement this recursively?

Once we do the comparison of the first and last letters, knowing that we have a working recursive function, we can just pass the rest of the string (i.e. the middle bits) into the recursive function and let that deal with it.

Either we find a mismatch on the way inward (so we return false) or we don't and reach the base case (so we return true).



```
bool isPalindrome(string foo) {
    int len = foo.length();
    if (len <= 1)
        return true;
    if (foo[0] != foo[len-1])
        return false;
    return isPalindrome(foo.substr(1, len-2));
}
```

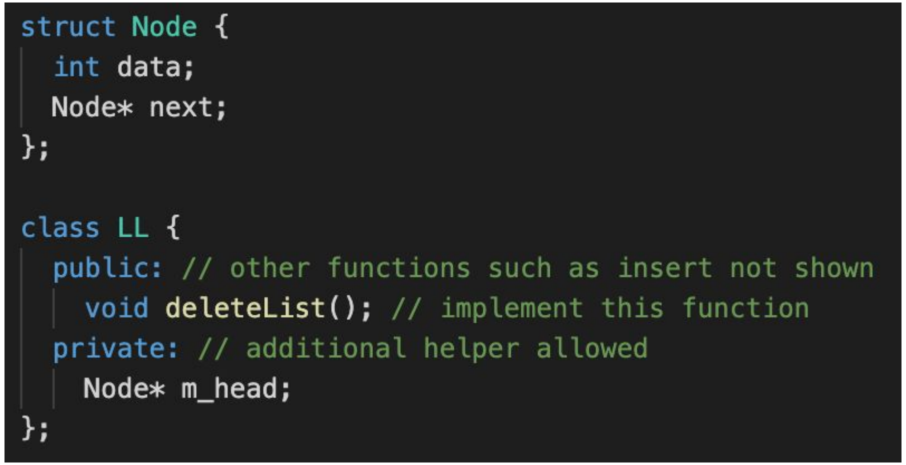
So No Head?

(taken from Wk 5 LA Worksheet, Section 2, Problem 1)

Given a singly-linked list class ***LL*** with a member variable ***head*** that points to the first ***Node*** struct in the list, write a function ***void LL::deleteList()*** that recursively deletes the whole list.

Assume each ***Node*** object has a ***next*** pointer.

(Hint: ***deleteList()*** takes no parameters, but to use recursion, you need to pass parameters - perhaps a helper function can do the job?)



```
struct Node {
  int data;
  Node* next;
};

class LL {
  public: // other functions such as insert not shown
    void deleteList(); // implement this function
  private: // additional helper allowed
    Node* m_head;
};
```

Solution

(to So No Head?)

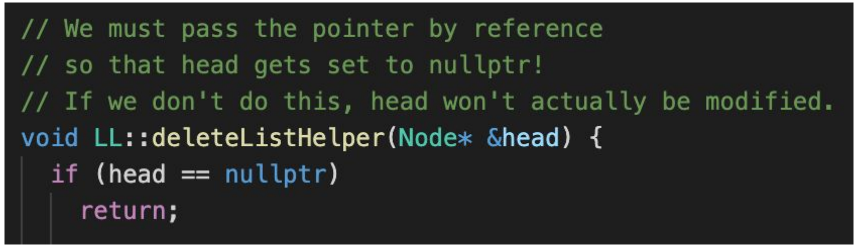
What's our base case?

The list is empty, i.e. ***head == nullptr***

So do nothing!

What about the recursive step?

Assume the list is not empty, i.e. head is not nullptr. We essentially only have access to the ***head*** pointer and the ***next*** pointer in head. What can we do?



```
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;
```

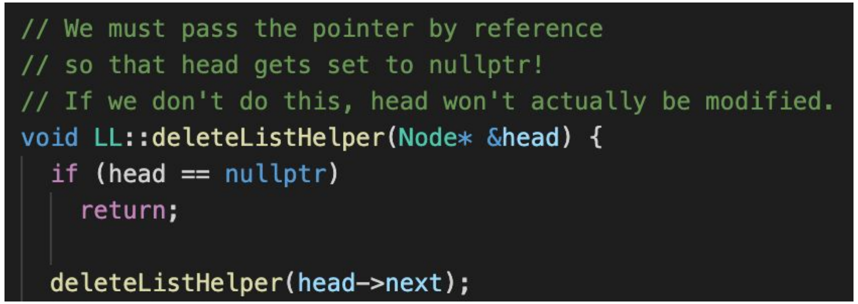
Solution, cont.

(to So No Head?)

What do we do?

We can break the linked list into two sublists: the first element (which ***head*** gives us access to) and the rest of the list (which ***next*** gives us access to).

The ***next*** pointer points to what is essentially another linked list, just an element shorter. So assuming our helper function already works for size n-1, we can just recursively call it!



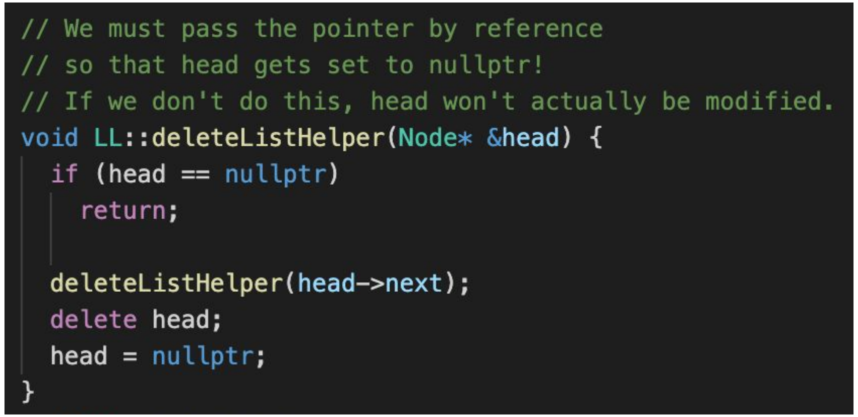
```
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;

  deleteListHelper(head->next);
```

Solution, cont.
(to So No Head?)

What about the first element?
Now that we know we've handled the rest of the list, we can think about what to do with the first element.

We just have to deallocate the Node object by using ***delete head;*** and then set ***head = nullptr*** for pointer safety.

Note that we have to pass the ***head*** pointer by reference so we can set it to nullptr (what would happen if we instead passed the pointer by value?)



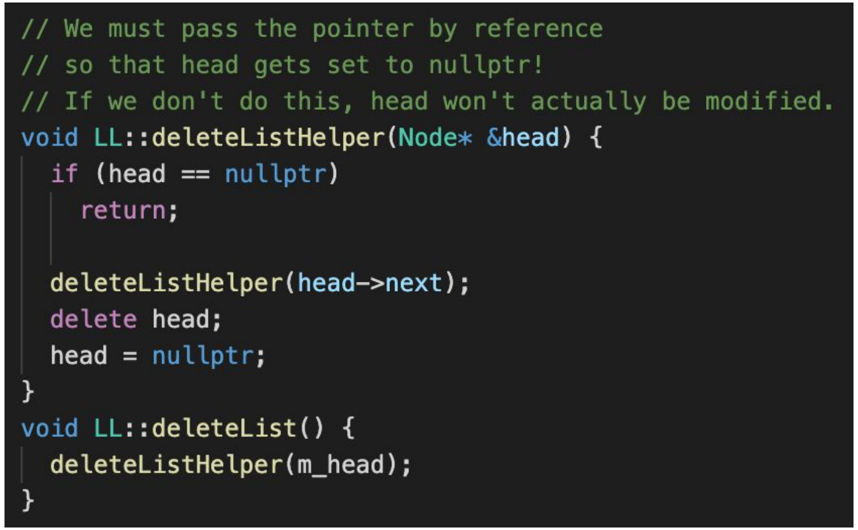
```
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;

  deleteListHelper(head->next);
  delete head;
  head = nullptr;
}
```

Solution, cont.

(to So No Head?)

One more final touch…

This is not the actual function we're trying to implement - this is the ***helper function!*** The actual interface takes no parameters (and even if we changed it, we can't even pass ***m_head*** in because it's private). So we do a quick little workaround to get our final code.



```cpp
// We must pass the pointer by reference
// so that head gets set to nullptr!
// If we don't do this, head won't actually be modified.
void LL::deleteListHelper(Node* &head) {
  if (head == nullptr)
    return;

  deleteListHelper(head->next);
  delete head;
  head = nullptr;
}
void LL::deleteList() {
  deleteListHelper(m_head);
}
```

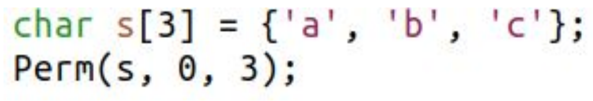
The Possibilities Are Endless

(not from the worksheet, credit to Yiyou for this one)

Given a character array without possible repetitions, print all permutations.

Hint: Think about induction! If we are given all the permutations of n-1 letters from the array, what's an easy way to construct permutations of n letters?

interface and expected output:



```
char s[3] = {'a', 'b', 'c'};
Perm(s, 0, 3);
```

Output:

```
abc
acb
bac
bca
cba
cab
```

Solution, cont.

(to The Possibilities Are Endless)

Time to rely on our friend combinatorics!
If we have ***n*** objects, we have ***n!*** permutations. Why??

Let's say we have 3 objects, A, B, and C.
There are 3 ways to pick the first object: either A, B, or C can go first.

Once we pick the first object, we have 2 objects left.
So now there are 2 ways to pick the next object.

Finally, after we've picked the first two objects, we only have 1 object left.
So there's only 1 possibility for the last object.

That means the number of orderings, or ***permutations***, of 3 objects are 3 * 2 * 1 = ***3!***
In general, the number of permutations of ***n*** objects is n * (n-1) * ... * 2 * 1 = ***n!***
Also, note that n! = n * (n-1)! This will be very useful for doing this recursively.

Solution, cont.

(to The Possibilities Are Endless)

Okay, so what does combinatorics have to do with recursion?
Similar to the linked list problem, we can split the array into two parts: the first element, and the rest of the list.

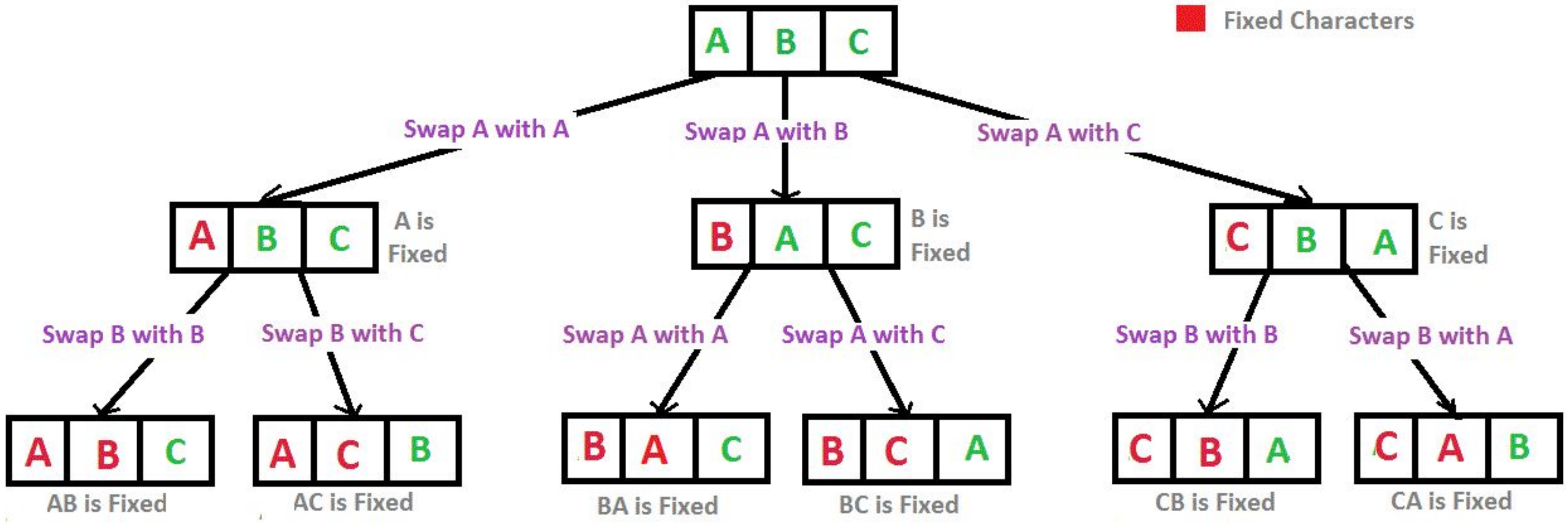
We basically need to choose which character is gonna be the first one in our specific ordering. For a list of size n, there are n ways to do this.
(In practice, we can do this by swapping each character to the front of the array.)

Then, we can realize that once we've chosen the first character, we can let our recursive function operate on the rest of the remaining characters that are left! Essentially, we "fix" characters from the array one-by-one until we run out!

Solution, cont.

(to The Possibilities Are Endless)

One visualization of how the recursion might work:



Recursion Tree for Permutations of String "ABC"

Solution, cont.

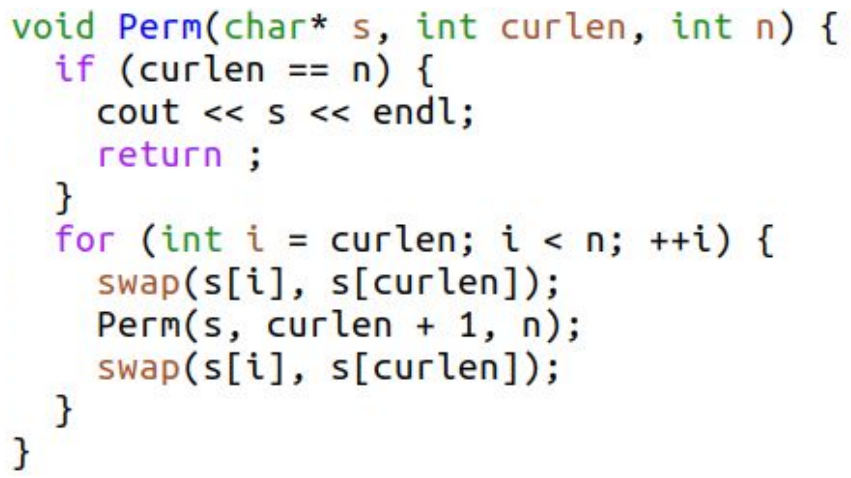
(to The Possibilities Are Endless)

What's our base case?

Here, ***curlen*** represents the number of characters we've fixed, and ***n*** represents the total length of the string. So if ***curlen == n***, we're done creating this permutation and we can output it!

What about the recursive step?

We use a for loop to swap all the remaining characters to the front of the list, and call ***Perm*** on each iteration. Doing it this way means we can operate on the string in place without creating new variables.

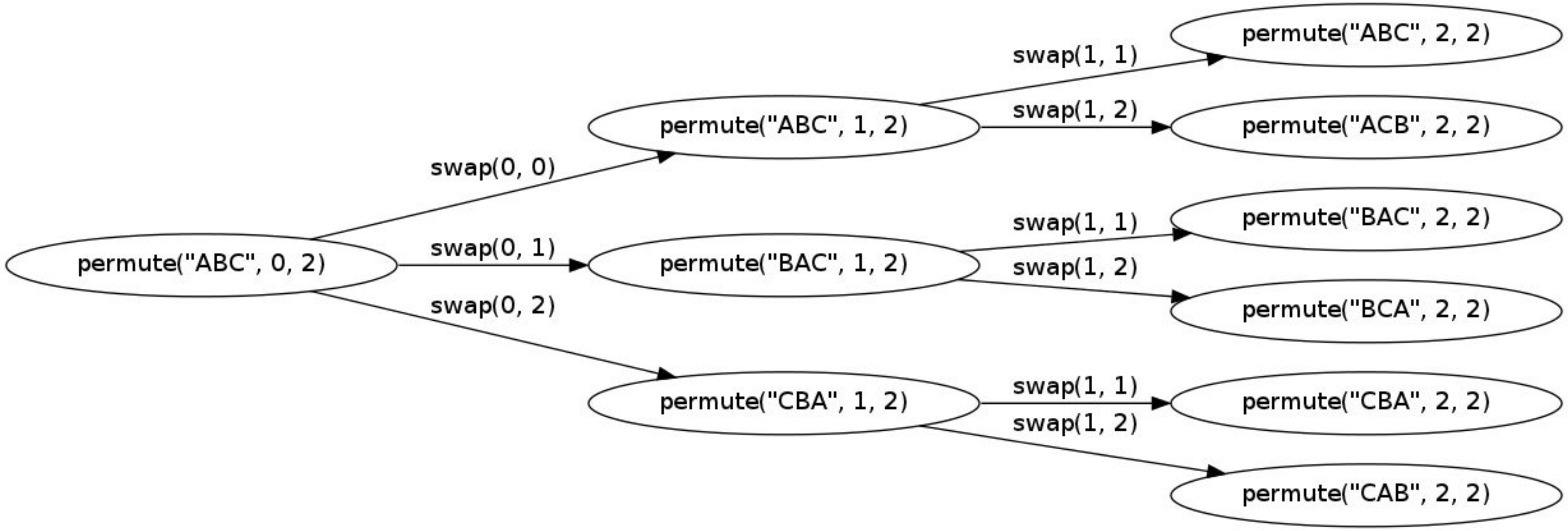


```
void Perm(char* s, int curlen, int n) {
  if (curlen == n) {
    cout << s << endl;
    return ;
  }
  for (int i = curlen; i < n; ++i) {
    swap(s[i], s[curlen]);
    Perm(s, curlen + 1, n);
    swap(s[i], s[curlen]);
  }
}
```

Solution, cont.

(to The Possibilities Are Endless)

One visualization of how the recursion function calls might work:



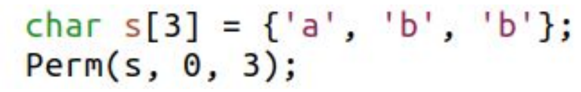
The Possibilities Are Not, In Fact, Endless

(not from the worksheet, credit to Yiyou for this one)

Given a character array without possible repetitions,
print all ***distinct*** permutations.

(the old code doesn't work anymore, otherwise we wouldn't be asking you this)

interface and expected output:



```
char s[3] = {'a', 'b', 'b'};
Perm(s, 0, 3);
```

Output (using old code):

```
abb
abb
bab
bba
bba
bab
```

Same code doesn't work!

Actual desired output:

```
abb
bab
bba
```

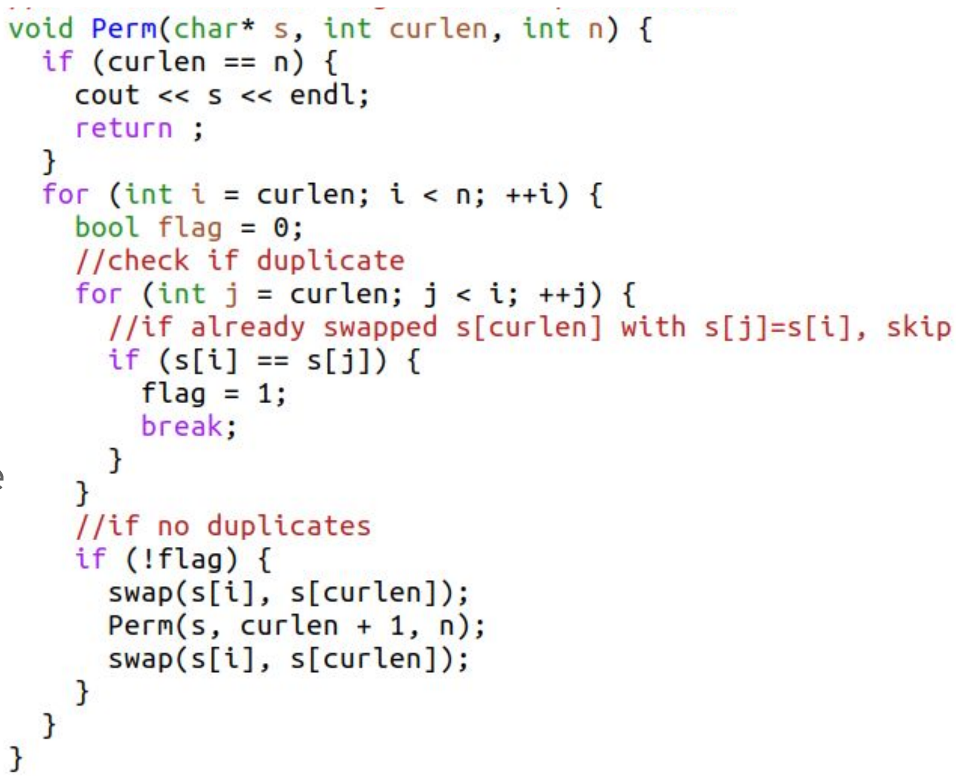
Solution, cont.

(to The Possibilities Are Not, In Fact, Endless)

How do we know if we have a duplicate?
Let’s say we have the string “ABABC”
We know that A and B are duplicates, so we want to act ***as if*** the string was “ABC”

One way to do this without modifying the string: let’s say we want to check if the second “B” is a duplicate. Walk through the string. When we see the first “B”, we can conclude that “B” appears twice, so we don’t perform a swap with the second “B”.

This code is just one of many ways to implement the solution!



```
void Perm(char* s, int curlen, int n) {
  if (curlen == n) {
    cout << s << endl;
    return ;
  }
  for (int i = curlen; i < n; ++i) {
    bool flag = 0;
    //check if duplicate
    for (int j = curlen; j < i; ++j) {
      //if already swapped s[curlen] with s[j]=s[i], skip
      if (s[i] == s[j]) {
        flag = 1;
        break;
      }
    }
    //if no duplicates
    if (!flag) {
      swap(s[i], s[curlen]);
      Perm(s, curlen + 1, n);
      swap(s[i], s[curlen]);
    }
  }
}
```

thank you all for coming!

good luck on your midterms and projects :)

solutions to the rest of the worksheet problems will be posted in a couple days!