COP3503

Polymorphism

Welcome!

Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Polymorphism

Three pillars of OOP:
1. Encapsulation
2. Inheritance
3. Polymorphsim

- The **third pillar** of object-oriented programming—it provides access to different types through a single interface.
- We often write programs to **branch** (change behavior) based on data values.

```
if (someVariable >= 10 || someBoolean == false)
    // Do something
else
    // Do something else
```

- Sometimes we want programs to branch based on **data types of an object**.

```
if (someObject is Car)
    // Do something Car related
else if (someObject is Airplane)
    // Do something Airplane related
else if (someObject is FlyingCar)
    // Do something FlyingCar related
// etc for all data types
```

```
// This would be much better to write:
someObject.DoTypeAppropriateThing();
```

Polymorphism makes that possible!

We have to write **other** code to enable this behavior... keep watching to see how!

Multiple, Similar Data Types Can Be a Pain

```
class Car
{ /*insert code here*/ };

class Limousine   : public Car {};
class Convertible : public Car {};
class SUV         : public Car {};
class Pickup      : public Car {};
```

- If we needed to store collections of these in a program?

```
class Garage
{
    // Everyday storage, nothing new here
    vector<Limousine>    _limos;
    vector<Convertible>  _convertibles;
    vector<SUV>          _utilityVehicles;
    vector<Pickup>       _pickups;
};
```

Multiple Data Types Can Be a Pain

```
// Calculate the total value of all stored vehicles
double Garage::GetInventoryValue()
{
    double total = 0;

    for (unsigned int i = 0; i < _limos.size(); i++)
        total += _limos[i].GetValue();

    for (unsigned int i = 0; i < _convertibles.size(); i++)
        total += _convertibles[i].GetValue();

    for (unsigned int i = 0; i < _utilityVehicles.size(); i++)
        total += _utilityVehicles[i].GetValue();

    for (unsigned int i = 0; i < _pickups.size(); i++)
        total += _pickups[i].GetValue();

    return total;
}
```

What if we…
…want to store motorcycles?
…no longer want to store limos?

A lot of code may have to change…

Polymorphism can bypass all of this!

A Simplified Version

```
// Calculate the total value of all stored vehicles
double Garage::GetInventoryValue()
{
    double total = 0;

    // One container to rule them all…
    for (unsigned int i = 0; i < ALL_OUR_VEHICLES.size(); i++)
        total += ALL_OUR_VEHICLES[i].GetValue();

    return total;
}
```

For polymorphism to work, we need two things:

1. Base class pointers
2. Virtual functions

```
// Instead of storing all of these...
vector<Limousine>    _limos;
vector<Convertible>  _convertibles;
vector<SUV>          _utilityVehicles;
vector<Pickup>       _pickups;
```

```
// ...we can store just base class pointers
vector<Car*> _allCars;
```

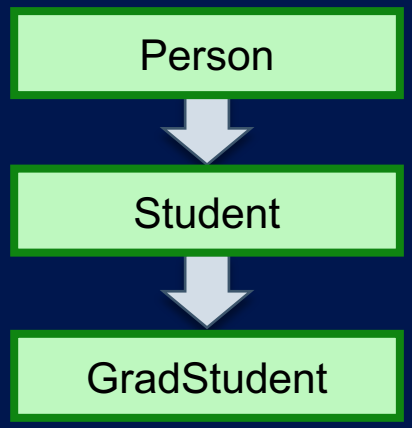
What Can Base Class Pointers Point to?

- **Base class pointers** can point to an instance of **any** class derived from it.

```
class Person
{ /*insert code here*/ };

class Student : public Person
{ /*insert code here*/ };

class GradStudent : public Student
{ /*insert code here*/ };
```



Converting a derived class pointer to a base class pointer is called **upcasting**.

Upcasting "just works" and happens automatically.

```
// Three simple objects
Person      person;
Student     student;
GradStudent gradStudent;
```

```
// A Person pointer can point to any of them
Person* ptr = &person;
ptr = &student;     // Okay, derived from Person
ptr = &gradStudent; // Ditto
```

```
// A Student pointer can't point to Persons
Student* ptr  = &person;      // Error, Person is "above" Student in the hierarchy
Student* ptr2 = &student;     // Okay, same data type
Student* ptr3 = &gradStudent; // Okay, GradStudent is "below" (derives from) Student
```

Upcasting Works With Objects Too

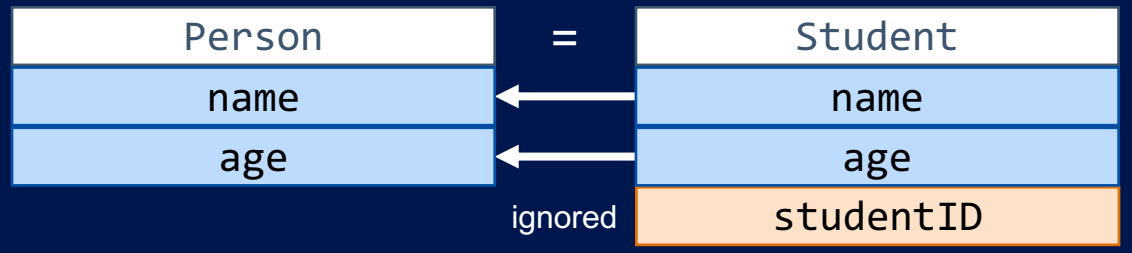
```
class Person
{
    string name;
    int age;
};
```

```
class Student : public Person
{
    string studentID;
};
```

```
Person person;
Student student;

person = student; // Upcasting
```

Derived to base → copy **only** base data

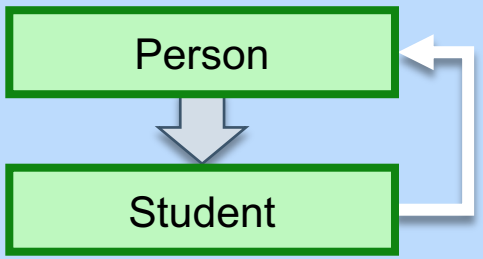


Because `Student` IS A `Person`, it can be **implicitly converted** to a `Person`.

We're assigning data **to** a Person object, any data beyond a `Person` is just ignored.

Upcasting Works With Objects Too

Upcasting →
Convert derived to base

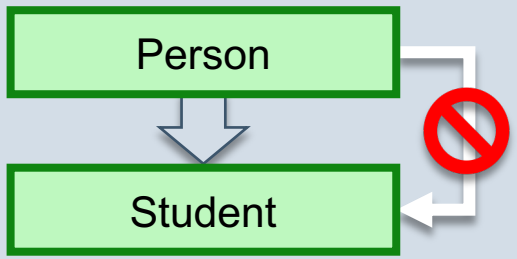



```
Person person;
Student student;

person = student;
```

Upcasting works implicitly, because student IS A `Person`.

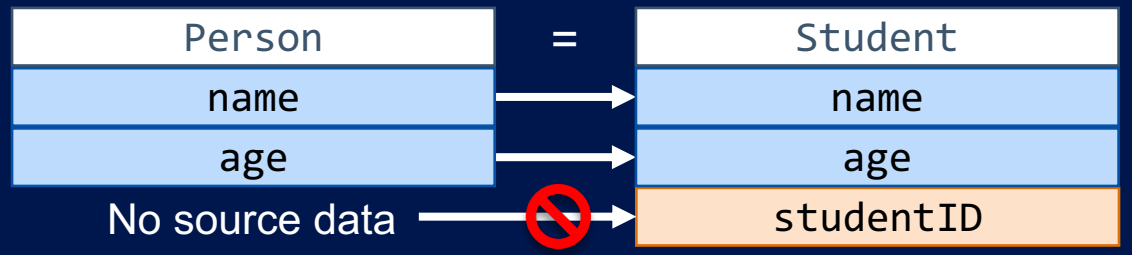
Upcasting →
Convert derived to base




```
student = person;
```

Downcasting doesn't work. `Person` IS NOT A `Student`, and is missing data

Derived to base → copy **only** base data



Downcasting **can** work with some additional code but is something you should avoid.

If you ever find yourself needing to do this, you may want to rethink your approach.

Upcasting With Dynamic Memory

```
class Car {};
class Limousine : public Car {};
class Convertible : public Car {};
class SUV : public Car {};
class Pickup : public Car {};
```

```
vector<Car*> allOurCars; // Each pointer can point to ANY class derived from Car
allOurCars.push_back(new Limousine);
allOurCars.push_back(new SUV);
allOurCars.push_back(new Car);
allOurCars.push_back(new Convertible);
allOurCars.push_back(new Pickup);
// Remember: At SOME point, you have to deallocate these with delete!
```

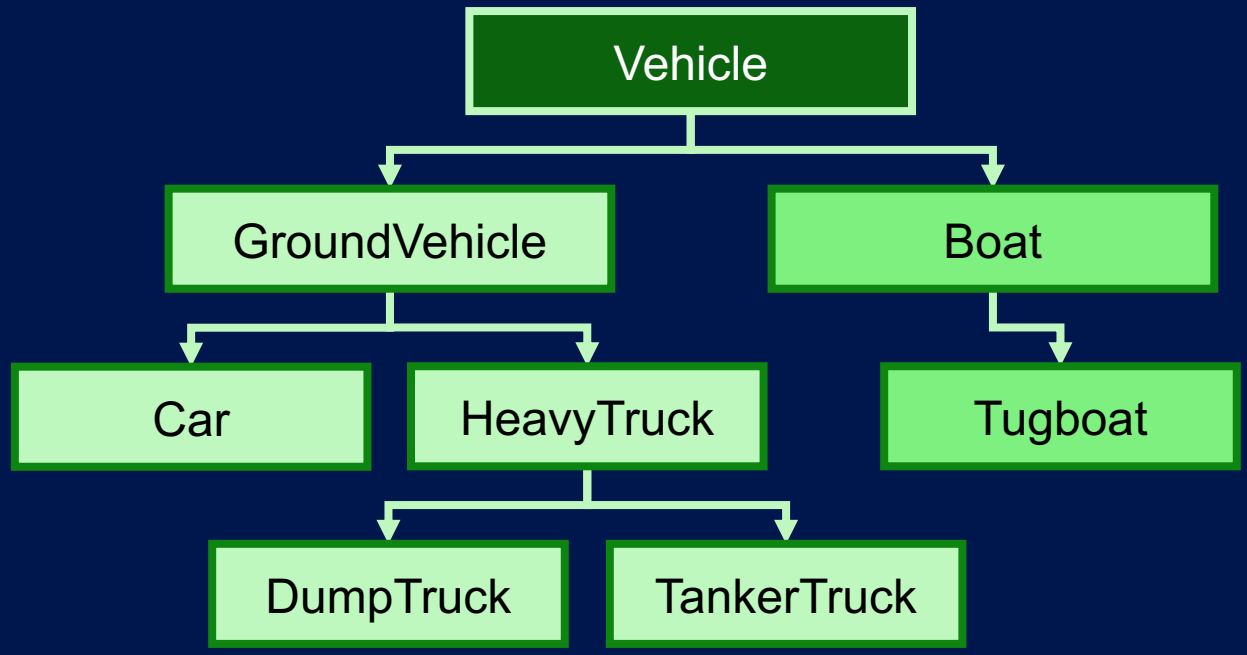
- One container stores 5 pointers to objects that are “Car-like” or **AT LEAST A** car.
- On the surface, we don’t know **exactly** what types of objects we have pointers to.
- With polymorphism, we don’t need to know. (We’re getting to this part, keep watching!)

Upcasting With Dynamic Memory

```
class Vehicle {};
class GroundVehicle : public Vehicle{};
class Car : public GroundVehicle {};
class HeavyTruck : public GroundVehicle {};
class TankerTruck : public HeavyTruck {};
class DumpTruck : public HeavyTruck {};
class Boat : public Vehicle {};
class Tugboat : public Boat {};

class IndustrialShop
{
    vector<HeavyTruck*> _trucks;
    Car _companyCar;
};

class Marina
{
    vector<Boat*> _customerBoats;
    vector<Tugboat*> _workBoats;
};
```



Can store (pointers to) HeavyTrucks, TankerTrucks, DumpTrucks…

But no Cars (except the company car!)

Marina can store any type of Boat or Tugboat, but no Cars, Trucks, etc.

Upcasting and storing base-class pointers gives us a lot of flexibility!

Back to the Garage

```
class Garage
{
    vector<Car*> _cars;
public:
    void AddVehicle(Car* newCar);
    double GetInventoryValue();
};

void Garage::AddVehicle(Car* newCar)
{
    _cars.push_back(newCar);
}

double Garage::GetInventoryValue()
{
    double total = 0;
    for (unsigned int i = 0; i < _cars.size(); i++)
        total += _cars[i]->GetValue();

    return total;
}
```

Upcasting lets us store Cars, or **any other (or future) class** derived from Car!

For polymorphism to work, we need two things:

1. ~~Base class pointers~~
2. Virtual functions

Virtual Functions

- Inheritance lets us reuse code (hooray!).
- Sometimes we want to **override** some base class functionality.
- For example:
  - A car's value is based on mileage (a `GetValue()` function).
  - A pickup's value is based on mileage **plus** how big of a carrying bed it has.
  - Each class does the same thing—they just do it differently.
- A **virtual function in the base class** enables this behavior.

```
class Car
{
    // Virtual means derived class
    // MAY implement their own version
    virtual double GetValue();
}
```

Virtual functions plus base class pointers enable **polymorphism**, or **polymorphic behavior**.

Virtual Functions == Different Behavior

```
class Car
{
public:
     // How much is this thing worth?
     // Virtual keyword on base class function
     virtual double GetValue();
protected:
     // A basic calculation shared by
     // all cars (and derived objects)
     double MileageValueCalculator();
};
```

```
// Calculate value only based on mileage
double Car::GetValue()
{
     return MileageValueCalculator();
}
```

A more complex function, but outside of this class, it's still just `GetValue()`.

```
class Pickup : public Car
{
public:
     // Implement our own version of this
     double GetValue();
protected:
     // Unique functionality for this class
     double BedValueCalculator();
     double TowCapacityCalculator();
};
```

```
// Calculate value based on mileage plus towing and
// hauling capabilities
double Pickup::GetValue()
{
     double mileValue   = MileageValueCalculator();
     double bedValue    = BedValueCalculator();
     double towValue    = TowCapacityCalculator();

     return mileValue + bedValue + towingValue;
}
```

Virtual Functions == Different Behavior

```
vector<Car*> vehicles;
vehicles.push_back(new Car);
vehicles.push_back(new Pickup);

// Car, Pickup, whatever... this works with all types derived from Car
// Each vehicle can BEHAVE differently now
double total = 0;
for (unsigned int i = 0; i < vehicles.size(); i+
    total += vehicles[i]->GetValue();
```

When a virtual function is called from a base class pointer…

…your program will call the version of GetValue() that matches the type of the object that the pointer points to (i.e., the pointee)

Pointing to a `Car`? Call `Car::GetValue()`

Pointing to a `Pickup`? Call `Pickup::GetValue()`

The best part? You don't have to reference those types anywhere!

You set things up "behind the scenes" so you can write cleaner code elsewhere.

Virtual Functions

- Non-virtual functions are “resolved” at **compile time**.
  - The compiler checks the type of the **invoking object** and calls the appropriate version of the function.
- Virtual functions are resolved at **run-time**.
  - The compiler adds additional data to a class with virtual functions (generally a pointer called a **virtual pointer** or **v-pointer**).
  - This pointer points to something in memory called a **virtual table**, or **v-table**, that tracks the “varieties” of virtual functions.
  - When a function is called, the v-pointer is used, and data from the v-table is retrieved (i.e. the correct function is found).
- All of this happens for you! (if you used the virtual keyword…)

You don’t have to memorize this (or learn it for a test).

Every language does a lot of stuff “automatically” for you.

Trying to do this manually can make for very messy code…

Virtual Alternative

```
vector<Car*> vehicles;
/* Assume initialization with various Car/Pickup/etc pointers */
double total = 0;
for (unsigned int i = 0; i < vehicles.size(); i++)
{
    // dynamic_cast<> attempts to DOWNCAST the pointer
    // If the result is nullptr, our original pointer (vehicles[i])
    // is NOT pointing to that type of object
    Car*       car    = dynamic_cast<Car*>(vehicles[i]);
    Pickup*    pickup = dynamic_cast<Pickup*>(vehicles[i]);
    SUV*       suv    = dynamic_cast<SUV*>(vehicles[i]);
    Limousine* limo   = dynamic_cast<Limousine*>(vehicles[i]);

    if (car != nullptr) // It's a Car! Call Car::GetValue()
        total += car->GetValue();
    else if (pickup != nullptr) // It's a Pickup, call Pickup::GetValue()
        total += pickup->GetValue();
    else if (suv != nullptr) // Use the SUV version
        total += suv->GetValue();
    else if (limo != nullptr) // Use the Limousine version
        total += limo->GetValue();
}
```

This is an example of what you **shouldn't** do.

Virtual functions eliminate all of this.

Virtual and Non-Virtual Functions Can Co-Exist

```
class Character
{
    int _maxHitpoints;
    int _hitpoints;
    string _name;
public:
    Character(string name = "Bob", int hp = 10);

    // Non-virtual functions: Derived classes use these in th
    const string& GetName();
    int GetHP();
    int GetMaximumHP();

    // Future classes CAN override some or any of these, IF you want/need to
    virtual bool IsAlive();
    virtual void TakeDamage(int amount);
    virtual void Heal(int amount);
    virtual int Attack(Character* target);
}
```

Things like simple accessors may not need to be overridden.

If you want to, that is… just because a function is marked virtual doesn't mean it **must** be overridden.

Examples

```
// Take damage in a "standard" way
void Character::TakeDamage(int amount)
{
     // Reduce our hitpoints, make sure we
     // don't end up with a negative value
     _hitpoints -= amount;
     if (_hitpoints < 0)
          _hitpoints = 0;
}
```

```
class ArmoredKnight : public Character {
     int _armor;
};

void ArmoredKnight::TakeDamage(int amount)
{
     // Knights are tough, and take less damage
     amount -= _armor;

     // Use existing functionality in the base
     // class. Also _hitpoints might be private!
     Character::TakeDamage(amount);
}
```

```
class Wizard : public Character {
     bool _magicBarrier;
};
void Wizard::TakeDamage(int amount)
{
     // If we have a barrier, negate all damage
     if ( _magicBarrier)
     {
          _magicBarrier = false;
          amount = 0;
     }
     Character::TakeDamage(amount);
}
```

```
vector<Character*> party;
party.push_back(new Character);
party.push_back(new ArmoredKnight);
party.push_back(new Wizard);

// Deal damage to the party... but each
// character calculates their own result
for (int i = 0; i < 3; i++)
     party[i]->TakeDamage(25);
```

One line, 3 different results.

Virtual Destructors

- When using inheritance/polymorphism, you should create virtual destructors in the base class.
- Virtual lets your program determine which function to call, based on data type.
- **Destructors are functions**, and follow the same rules when it comes to data types and virtual (or non-virtual) functions.

```
Character* hero = new Wizard;
hero->TakeDamage(5); // Call the Wizard version of the function

delete hero; // Which version of the destructor to call?
//hero->~Character(); // delete invokes the (Character) destructor
```

The Wizard destructor is never called, which will likely cause problems.

Simple solution:
Virtual destructor in the Character class

```
class Character
{
    virtual ~Character() // Only need virtual on base class
    {
        cout << "Character Destructor called!" << endl;
    }
};

class Wizard : public Character
{
    ~Wizard()
    {
        cout << "Wizard Destructor called!" << endl;
    }
};

Character *hero = new Wizard;
hero->TakeDamage(5);
delete hero;
```

The derived **destructor** is called first—this is the opposite of constructor order.

Whether you use the destructor for some "real" purpose or not, the base class needs one, and it needs to be virtual.

For derived classes? You can skip the destructor. (Remember they get a "do-nothing" version implicitly)

Without virtual destructor

```
Character Destructor called!
Press any key to continue...
```

With virtual destructor

```
Wizard Destructor called!
Character Destructor called!
Press any key to continue...
```

Abstract Base Classes (ABCs)

- Sometimes you want to **inherit** a base class…
- …but you never want to **use** the base class by itself (i.e., **never instantiate the base class**).

```
class Person
{
    string _name;
    int _age;
public:
    Person(const char* name = "Bob", int age = 10);
    const string& GetName() const;
}
```

We always want to **use** the Person class…

…but only as part of a more complex class

```
Person generic; // not enough info
```

We can make Person an abstract base class (ABC) and **prevent instantiation**.

```
// The "real" classes
class Student : public Person
{ /*cool stuff here*/ };

class Faculty : public Person
{ /*cool stuff here*/ };
```

```
// The "real" classes
class School
{
    vector<Student> _students;
    vector<Faculty> _teachers;
};
```

Creating Abstract Base Classes

- To create an abstract class, we need to create at least one **pure virtual function** (in C++ - other languages may differ).

```
class AbstractBaseClass
{
public:
    virtual void PureVirtualFunction() = 0;
};
```

`= 0`
Indicates this function is pure virtual; it will **not** have a body.

Pure virtual functions in a class:

- Mark a class as an Abstract Base Class.
- Do not have a definition (i.e., no body)—they don't DO anything.
- All classes that derive from this class must provide a definition (or else they will be ABCs as well).
- They **provide an interface for future classes**—derived classes will have this behavior…
- …but the details of the behavior are not specified.
- They are a **promise of future functionality**.

Abstract vs. Concrete Classes

```
class ABC
{
public:
    // Derived classes must implement this
    virtual void DoSomething() = 0;

    // Abstract classes CAN contain regular
    // functions with definitions.
    void Foo() { cout << "Hello world!"; }
};

class ConcreteClass   public ABC
{
public:
    void DoSomething()
    {
        cout << "This function does something!";
    }
};
```

```
// Compiler error, can't create
// instance of an abstract base class
ABC object;
```

```
// No problems, just regular object creation
ConcreteClass object;
object.Foo(); // Inherited from ABC
object.SomeFunction();
```

A **concrete class** is an ordinary class that can be instantiated – it contains **no pure virtual functions**.

ABC Example

```
class UIButton
{
protected:
     int _xPos, _yPos;
public:
     // Check to see if the x/y position of a mouse click hit this control
     virtual bool WasClicked(int clickX, int clickY) = 0;
};

class RectangularButton : public UIButton
{
     int _width, _height;
public:
     bool WasClicked(int clickX, int clickY);
};

class CircularButton : public UIButton
{
     int _radius;
public:
     bool WasClicked(int clickX, int clickY);
};
```

A **pure virtual** function says this class won't define this function… but future classes **must**.

Different functions to find a point in a rectangle vs. a circle, but it's the same goal:

*****Was this thing clicked?*****

The **overall** behavior of the classes is the same—it's the specific implementation that differs.

One Pure Virtual Function, Two Concrete Implementations

```
bool CircularButton::WasClicked(int clickX, int clickY)
{
    // 1. Get distance from center to click
    // 2. return distance <= radius
    float distance = sqrt(pow(clickX - _xPos, 2) + pow(clickY - _yPos, 2));
    return distance <= _radius;
}

bool RectangularButton::WasClicked(int clickX, int clickY)
{
    // Is a point inside a rectangle? See if...
    // x is within left-right boundaries
    // y is within top-bottom boundaries
    return (clickX >= _xPos && clickX <= _xPos + _width &&
            clickY >= _yPos && clickY <= _yPos + _height);
}
```

Different **implementations**, hidden behind a function with a standardized **interface**.

Pure Virtual Destructors

- Need an abstract base class, but all of your functions have definitions?

```
class Character
{
    virtual bool IsAlive()                  { /*definition*/ }
    virtual void TakeDamage(int amount)     { /*definition*/ }
    virtual void Heal(int amount)           { /*definition*/ }
    virtual int Attack(Character* target) { /*definition*/ }

    // Solution: Mark the destructor as pure virtual!
    virtual ~Character() = 0;
}
```

Problem: Can't make these pure virtual (i.e., can't make an abstract class!)

- Destructors **cannot**, ever, be left without a definition – C++ demands it.
- Pure virtual destructors will have a body, as an exception to the rule.

```
Character::~Character()
{ // Must have a body, even if it's empty one }
```

Every language has its own quirks, and this is one of them.

Recap

- **Polymorphism** is a way to change program behavior based on object types instead of values.
- This allows programmers to write “generic” code that gets different behavior from identical code,
  - This reduces (possibly eliminates!) type-dependent code in a program.
- Polymorphism requires **base class pointers** and **virtual functions**.
- **Virtual functions can be overridden** by derived classes to provide different behaviors.
- An **Abstract Base Class** is a class that cannot be instantiated
  - It contains important data/functions that will be part of some future class.
  - They are often used to define interfaces for future classes.

Conclusion

Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.