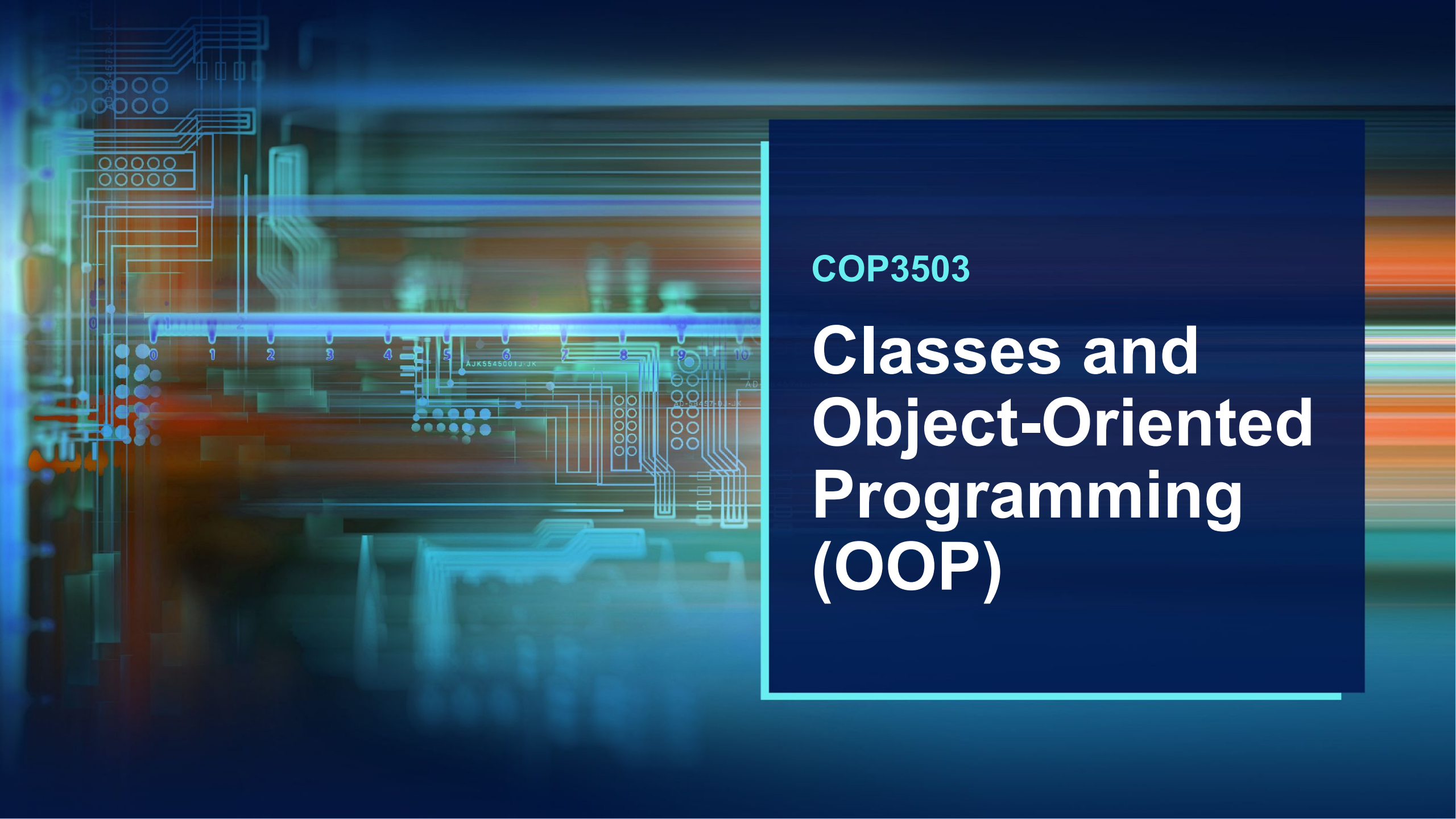
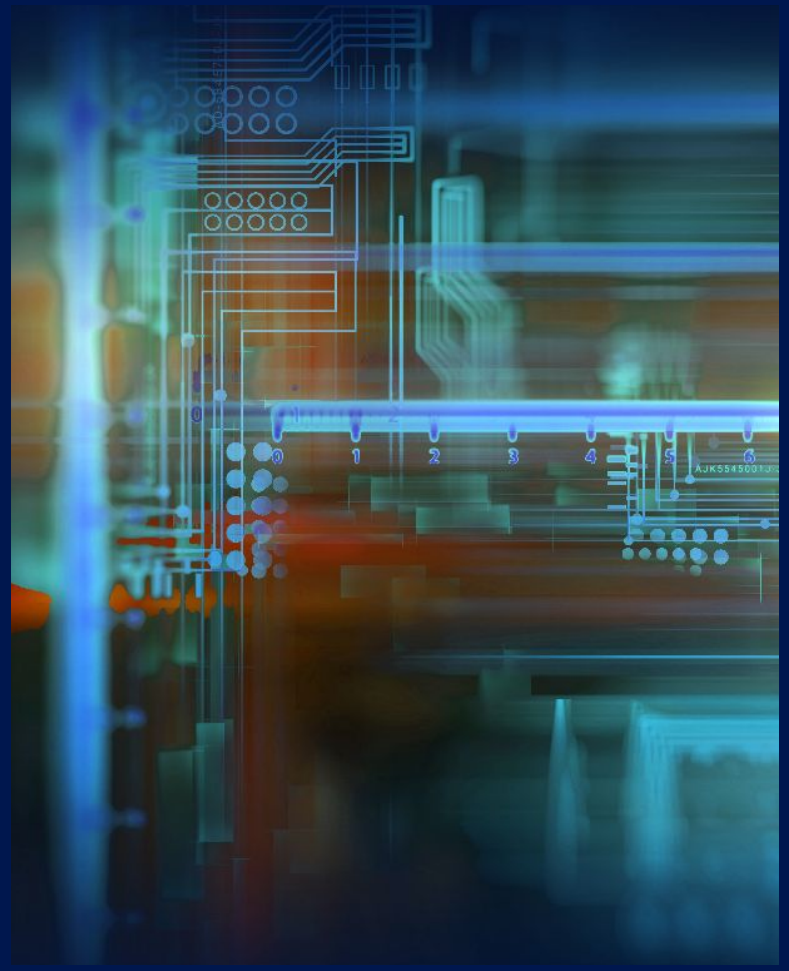


COP3503

Classes and Object-Oriented Programming (OOP)

Welcome!



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

What Is Object-Oriented Programming?

- A programming paradigm that emphasizes the use of **objects**.

- Objects represent the **data** for aspect of a program

A button in a user interface

A simple list of numbers

A complex character in a game

What Is Object-Oriented Programming?

- A programming paradigm that emphasizes the use of **objects**.
- Objects represent the **data** for aspect of a program.
- Objects may also contain **functions** to operate on that data.
- To create these objects in code, we write **classes**.

Why Use Objects?

- **Encapsulation**:
  Group data and functionality together

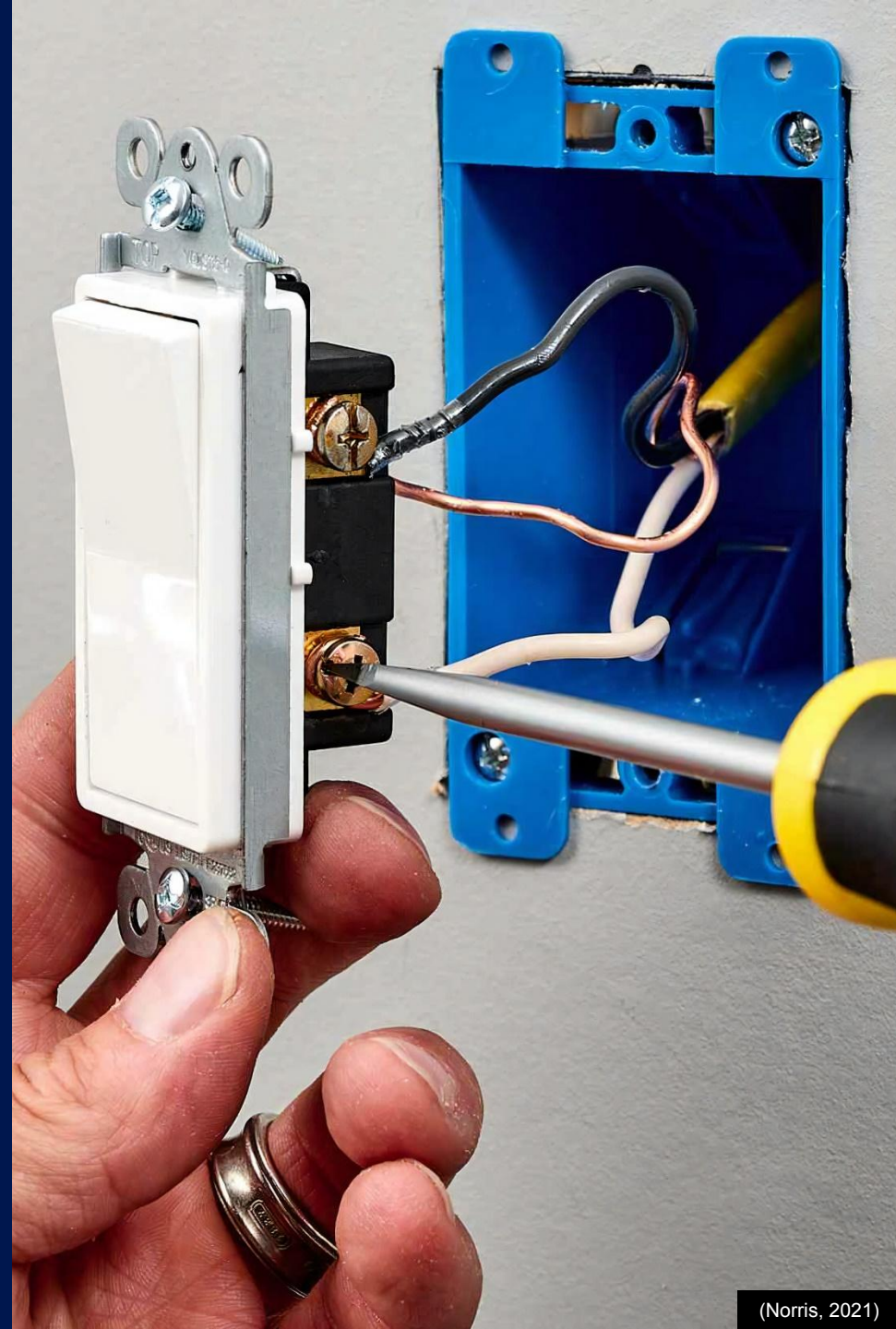
- Create a public **interface**, do the “real work” in private
  - Hide the details and control access to data

```
class Hero
{
    string _name;
    int _strength;
    int _hitpoints;
    int _experience;
public:
    Hero(string name, int str, int hp);
    string GetName();
    int GetStrength();
    int GetHitpoints();
    void TakeDamage(int amount);
    void GainExperience(int exp);
    void PrintStats();
};


int main()
{
    Hero ourHero("Thor", 70, 500);
    ourHero.GainExperience(50);
    ourHero.PrintStats();
    return 0;
}
```

Why Use Objects?

- **Encapsulation**:
  Group data and functionality together

- Create a public **interface**, do the “real work” in private
  - Hide the details and control access to data
  - **Analogy**:
    A light switch (interface) versus the wiring in the wall (behind-the-scenes implementation)



(Norris, 2021)

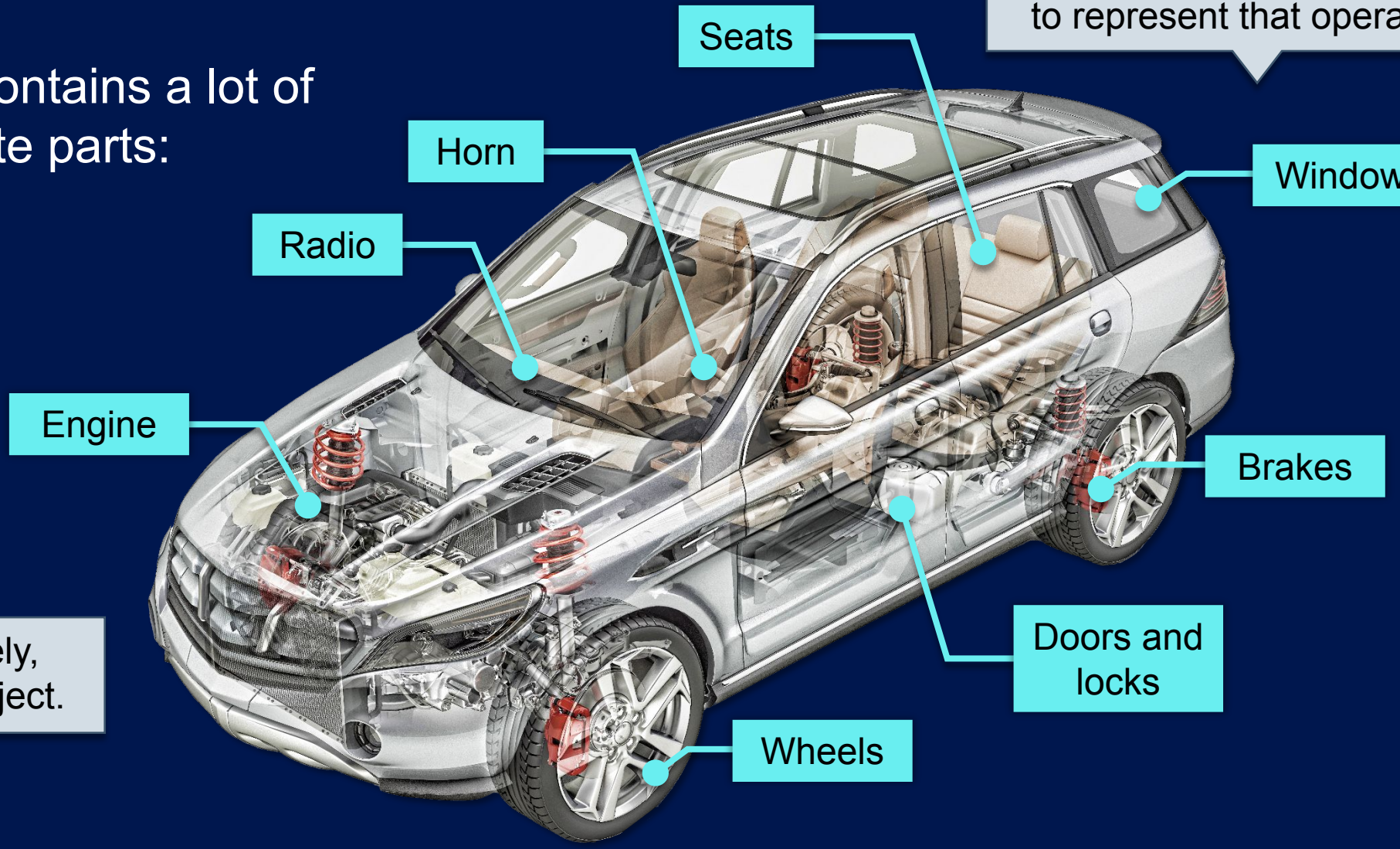
Why Use Objects?

- Code reuse
  - Once a class is created, you can **instantiate** that class.
  - Instantiation creates an **instance** of the class, a copy of the data and functionality.
  - You can create as many instances as you want, and never have to redefine the class again.

```
class Hero
{
    string _name;
    int _strength;
    int _hitpoints;
    int _experience;
public:
    Hero(string name, int str, int hp);
    string GetName();
    int GetStrength();
    int GetHitpoints();
    void TakeDamage(int amount);
    void GainExperience(int exp);
    void PrintStats();
};


int main()
{
    Hero ourHero("Thor", 70, 500);
    ourHero.GainExperience(50);
    ourHero.PrintStats();
    return 0;
}
```

A Car Is a Complex Object

- A car contains a lot of separate parts:



When we need to access some specific detail of the car, we can write some code to represent that operation.

Collectively, it is one object.

Encapsulation Example

```
// Data to describe aspects of a person
string name;
int age;
float salary;
```

```
// Function to print out that data
void PrintPerson(int age, float sal, string name)
{
    cout << "Name: " name << endl;
    cout << "Salary: " sal << endl;
    if (age < 18)
        cout << "Minor";
    else if (age < 25)
      cout << "Young adult";
    else
      cout << "Age is just a number...";
}
```

The Person class **encapsulates** the data and functionality into a single unit.

```
class Person
{
    string name;
    int age;
    float salary;

public:
    void Print();
};
```

```
Person erica, lee;

erica.Print();
lee.Print();
```

Each of these **Person** objects has the same grouping of data and functionality.

What Should Be a Class?

- What are the **separate parts** of your program?
- What **data** needs to be stored for those parts to work properly?

What Should Be a Class?

- Imagine a program that needs to store books (for a library or a store).
- Create a public **interface**, do the “real work” in private.
  - Title
  - Author
  - Page Count
  - $ Retail price
  - # ISBN
- All of those variables could be stored together in a class.

What Should Be a Class?

- Imagine a program that needs to store books (for a library or a store).
- Create a public **interface**, do the “real work” in private

- All of those variables could be stored together in a class.

```
class Book
{
    string title;
    string author;
    int pageCount;
    float price;
    string isbn;
};
```

```
class Store
{
    string name;
    Book aBook;
    Book books[10];
};
```

What If We Didn't Use Classes?

If we don't use classes, then it can be cumbersome to keep track of the data for complex objects.

```
// One Person
string name;
int age;
float salary;
```

Adding a variable?

```
// One Person
string name;
int age;
float salary;
float height;
```

Add 2 more variables

```
// One Person
string name;
int age;
float salary;
float height;
float weight;
float shoeSize;
```

```
// Two persons
string names[2];
int ages[2];
float salaries[2];
```

```
// Two persons
string names[2];
int ages[2];
float
salaries[2];
float heights[2];
```

```
// Two persons
string names[2];
int ages[2];
float salaries[2];
float heights[2];
float weight[2];
float shoeSize[2];
```

Same Process With Classes

With classes, no **external** changes are required!

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

Adding a variable?

```
class Person
{
    string name;
    int age;
    float salary;
    float height;
};
```

Add 2 more variables

```
class Person
{
    string name;
    int age;
    float salary;
    float height;
    float weight;
    float shoeSize;
};
```

```
// One Person
Person solo;

// Two persons
Person group[2];
```

```
// One Person
Person solo;

// Two persons
Person group[2];
```

```
// One Person
Person solo;

// Two persons
Person group[2];
```

Programs Have Lots of Classes

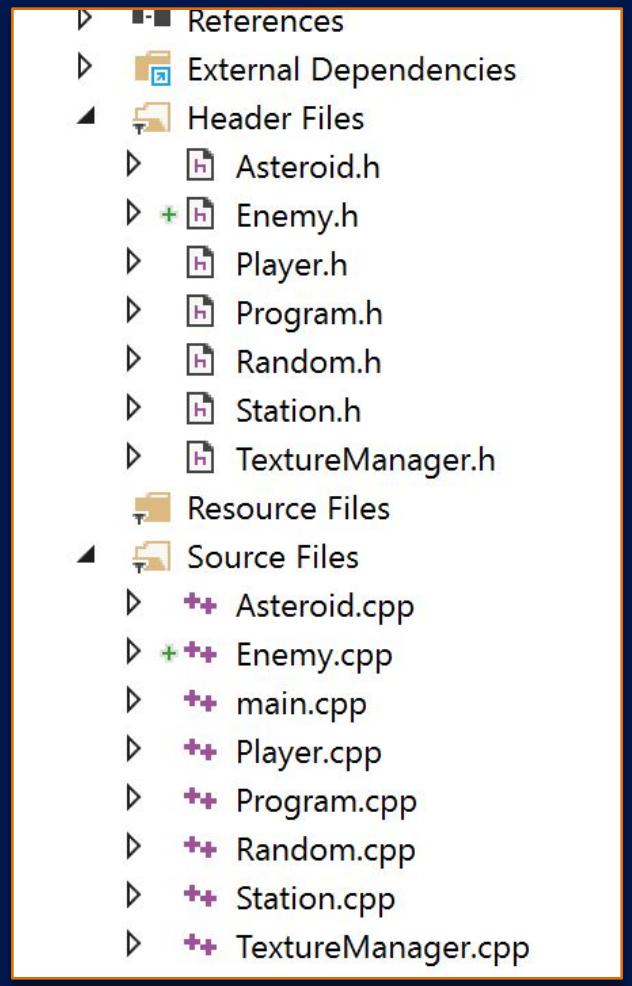
- Imagine a game with the following characteristics

- Different “things” in the game:
  - Station
  - Cannon
  - Projectiles
  - Asteroids
  - Ships
  - Score

- Possible classes

```
class Station
class Cannon
class Projectile
class Asteroid
class VerticalShip
class HorizontalShip
class ScoreDisplay
```

Even Simple Programs Can Have a Lot of Classes



Class Components

```
class Person
{
    string name;
    int age;
    float salary;

public:
    Person();
    Person(int a, float s, string n);
    int GetAge();
    float GetSalary();
    void GiveRaise(float percent);
};
```

Member variables:
The data your class is storing

Accessibility Keyword:
Defines a region of public, private, or protected accessibility (more on this soon)

Member functions:
These allow you to operate on the data

Some languages may call these "methods" or "procedures".

Key Functions in a Class

Accessors:
"Get" functions, retrieve something from the class

```
someBankAccount.GetBalance();
```

Mutators:
"Set" functions, change something about the class

```
somePerson.SetAge(10);
someDoor.Unlock();
someAircraft.IncreaseVelocity(2.15f);
```

Constructors:
Build the object in a variety of ways

- A constructor is only ever called once, when an object is first created.
- You can define multiple constructors for options/flexibility.

Destructor:
Destroy the object (only one per class)

- Called automatically when an object falls out of scope
- Used to "finalize" the object, perform any last steps
- Not always needed! (More on these later)

Key Functions in a Class

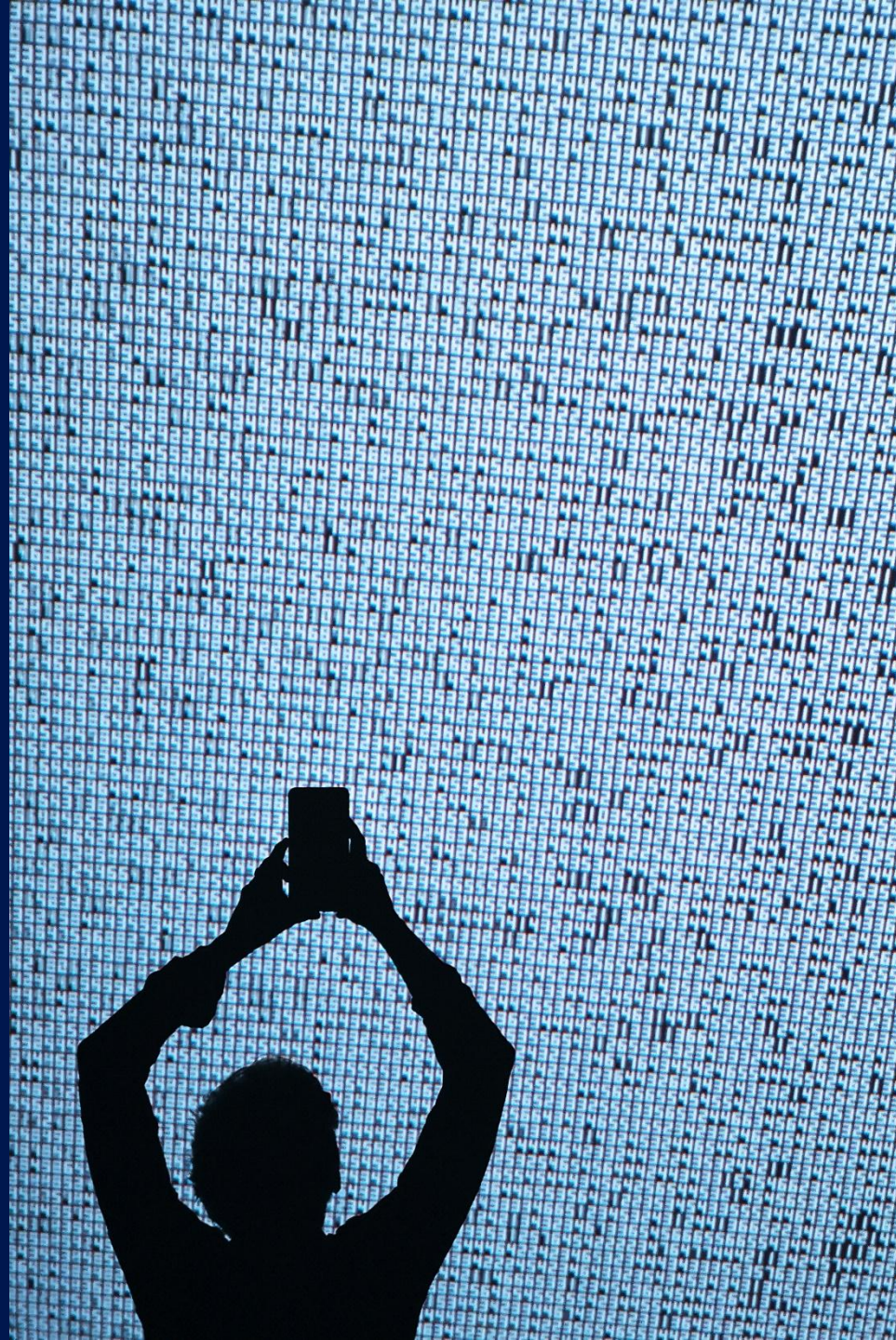
```
class Person
{
    string name;
    int age;
    float salary;

public:
    Person();                                   // Constructor
    Person(int a, float s, string n);           // Constructor
    int GetAge();                               // Accessor
    float GetSalary();                          // Accessor
    void GiveRaise(float percent);              // Mutator
};
```

Every class can be different and have any number of member functions and variables.

Access Levels

Public

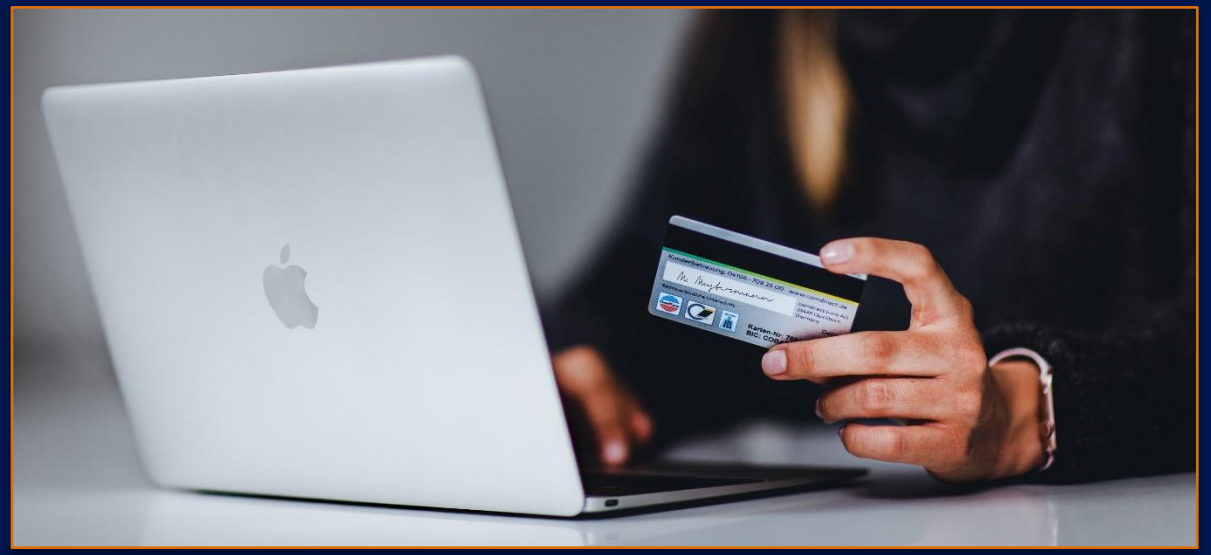
- Public members are accessible everywhere.
- Use this sparingly, especially for member variables!
- In general, we want to hide data as much as we can.



Access Levels

Public

```
class BankAccount
{
public:
    float balance;
    void Deposit(float amount);
    void Withdraw(float amount);
    float GetBalance();
};
```



```
BankAccount account;
account.Deposit(50);            // Use a function to change class data
account.balance = 1000000;      // Unrestricted access can lead to problems!
```

Access Levels

Private

- The default accessibility, only code within class functions (i.e., the class itself) can access.
- Trying to access private data outside of a class results in a compiler error.
- To avoid undesired changes, as much data as possible should be private, to restrict changes.
- Generally speaking: Use **private variables**, **public functions**.

Access Levels

Private

```
class Person
{
    string name;
    int age;
    float salary;
public:
    string nickname;

    Person();
    int GetAge();
    float GetSalary();
};
```

```
Person student;
cout << "Age: " << student.GetAge() << endl;  // Public function, okay
cout << "Nickname: " << student.nickname << endl; // Public variable, okay
cout << "Name: " << student.name << endl;     // Private variable, error
```

Access Levels

Protected

- This only applies when using **inheritance**, which we'll cover later in the course
- Until then, you can ignore this keyword!
- We'll only worry about public and private for now.

Accessibility Style

Neither approach is right or wrong, it just comes down to personal preference.

- Class members are private by default, so no accessibility indicator is necessary

```
class Person
{
    /* Private members first */

public:
    /* Public members second */
};
```

- Once you create a public region, everything that follows is public. A private keyword is required in this case to indicate private members.

```
class Person
{
public:
    /* Public members first */
private:
    /* Private members second */
};
```

How Do We Access Class Members?

- We use the **membership operator** "." (the period) to access a variable or invoke a member function
- This indicates the variable or function being accessed "belongs" to the object

How we access the variables and functions changes.
How we **use** those things hasn't changed.

```
int age = 20;    // Change a variable

Person maria;
maria.age = 20; // Change a variable that belongs to the object "maria"
```

```
SomeFunction(); // Call a "global" function
maria.Print();  // Call the Print() function that belongs to the object
```

Structures, a Similar Concept to Classes

- C++ has the `struct` keyword, from the C language.
  - Other languages may have this as well.
  - Typically, it is similar to a class, but many have minor differences.
- In C++ a structure is **exactly** like a class, except for one, and only **one** difference:
  - Everything in a structure defaults to **public** accessibility, instead of **private**, in classes.
- For **all** (yes, **all**) other purposes, structures and classes are the same.
- All “class” concepts (except private by default) also apply to structures.

Class vs. Struct

class

```
class Person
{
public:
    Person();
    Person(const Person &);
    ~Person();

private:
    string name;
    int age;
    float salary;
};
```

If you remove these…

These become private.

struct

```
struct Person
{
public:
    Person();
    Person(const Person &);
    ~Person();

private:
    string name;
    int age;
    float salary;
};
```

These stay public.

Why Even Use Structures?

Not all languages have the `struct` keyword. Some languages may have it and treat structures differently than classes.

- Structures can be good for “simple” objects.
- Objects that are **only** a collection of data, with no complex functionality

```
// Just 3 floats, nothing else
struct PointIn3DSpace
{
    float x, y, z;
};
```

```
// No access concerns, everything is public
PointIn3DSpace point;
point.x = 2;
point.y = -5;
point.z = 42;
```

Similarities With Other Languages

- The core idea of a class object is more or less universal.
  - Encapsulate data and functionality
- Syntactical differences are always a thing across languages.
- When dealing with **dynamic memory**, things get much more complicated (more on this later).
- Not all languages use dynamic memory.

Writing Classes in C++

- Class files are typically split into **two** separate files:

Header file (.h or .hpp)
The initial definition of the **class itself**

Source file (.cpp)
Where **class member functions** are defined

```
// File: example.h
class Example
{
    // Private variables
public:
    // Function PROTOTYPES
    void Foo(int x);
};
```

Writing Classes in C++

- Class files are typically split into **two** separate files:

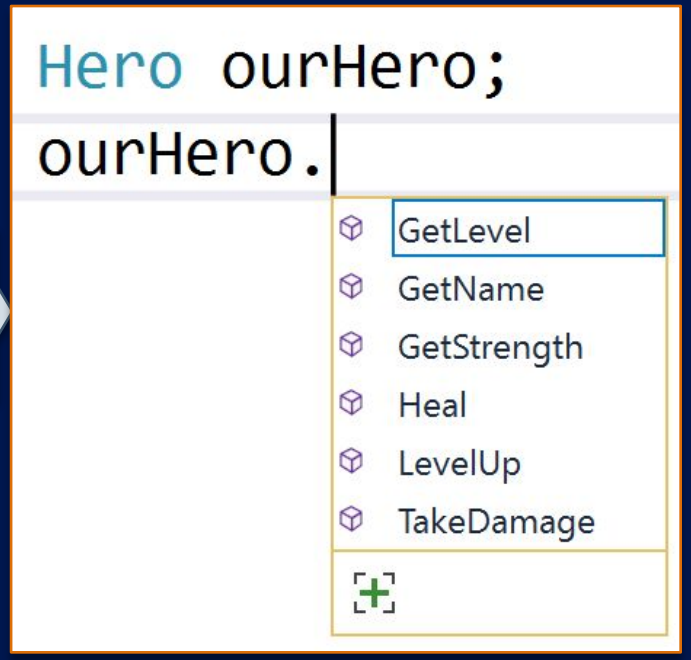
Header file (.h or .hpp)
The initial definition of the **class itself**

Source file (.cpp)
Where **class member functions** are defined

```
// File: hero.h
class Hero
{
    int _strength;
    string _name;
public:
    string GetName();
    int GetStrength();
    void TakeDamage(int damageAmount);
    void Heal();
    void LevelUp();
};
```

Prototypes serve as a “table of contents”.

When we use this class:



Writing Classes in C++

Source Files

You **must** write the name of the class and the Scope-Resolution operator before **all** class member function names.

It indicates “ownership” of the function.

This variable doesn’t exist anywhere and generates an error.

The variable belongs to the **invoking object**.

```
// file: hero.cpp
#include "Hero.h"

// Member function definition
string Hero::GetName()
{
    return _name;
}

int GetStrength()
{
    return _strength;
}

int Hero::GetStrength()
{
    return _strength;
}
```

#include a header file that contains class definition.

This function has no connection to any class.

This is a member function of the Hero class.

Two Files

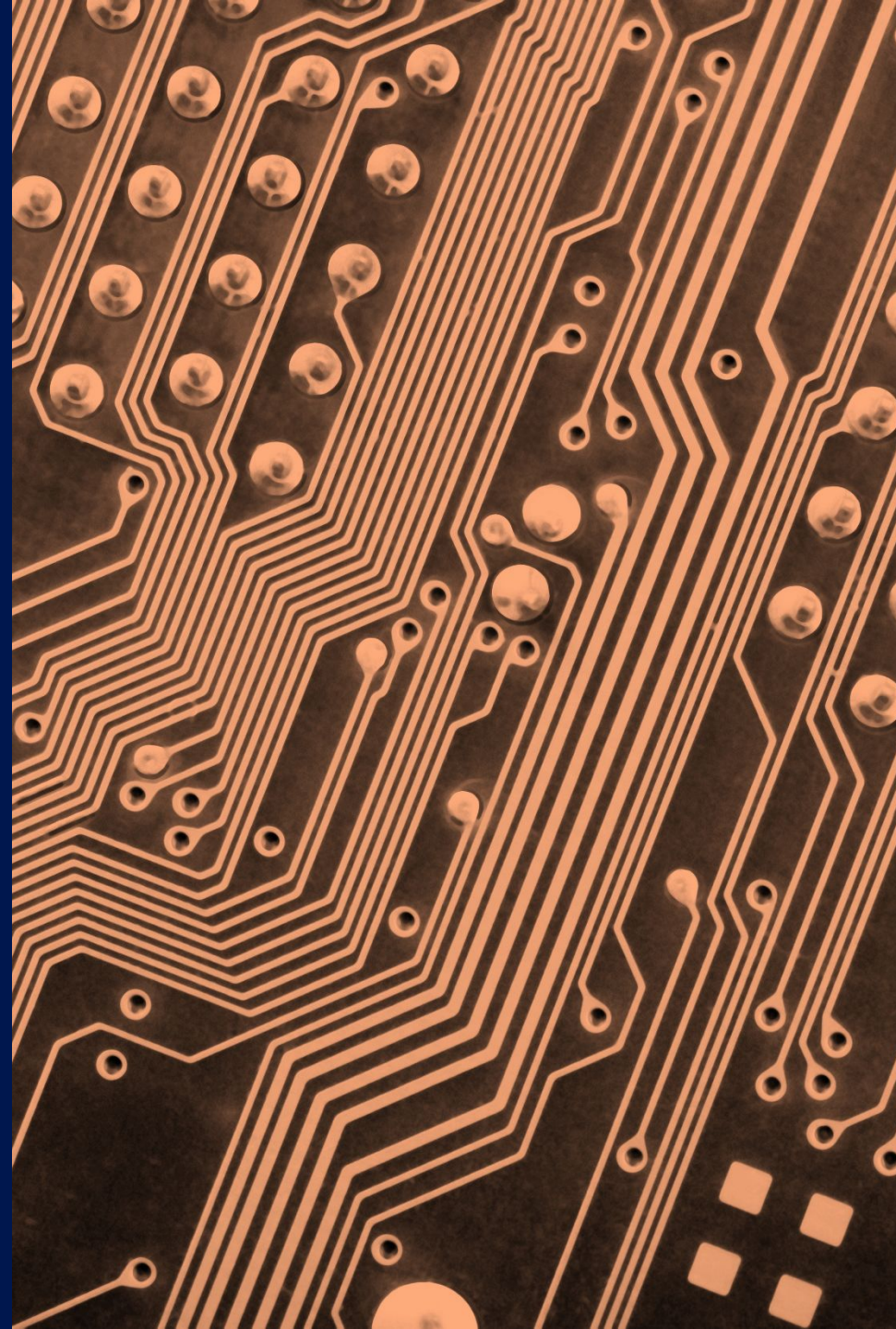
Good Practice

- Separating classes into two files helps with organization.
  - Most C++ code follows this convention.
- You **can** write a class entirely in the header file.
  - In some cases, we have to (we'll look at **templates** later).
- Get into the habit of breaking all classes into two separate files.
  - **Header (.h) file**:
    Class definition, member function prototypes
  - **Source (.cpp) file**:
    Member function definitions
- Develop good habits now (in any language) and it will pay off later.

I may break this rule in presentations to save space!

Classes and the `this` Variable

- All classes have a variable they can access called "`this`".
  - It is implicitly passed to all class member functions.
- **`this`** represents the **invoking object**—the object that called the function.
  - Technically it's a **pointer** to the invoking object (more on pointers later)
- It lets you reuse code in a class member function.
  - The same code can **operate on different instances** of class variables.
- It can resolve ambiguity between member variables and local variables.



Using this

```
class Person
{
    string name;
    int    age;
public:
    Person(string name, int age)
    {
        this->name = name;
        this->age = age;
    }
    void ShowInformation()
    {
        cout << this->name << endl;
        cout << this->age << endl;
    }
};
```

this is pointing to the "Sarah" object.

```
// Code in main()
Person bob("Bob", 25);
Person sarah("Sarah", 28);

bob.ShowInformation();
sarah.ShowInformation();
```

The same is true in this function. Each time it's called, a different object is referenced by this.

Where Does `this` Come From?

- It comes from the object that called the function (i.e. the **invoking object**).
- A pointer to the object is passed to the function, automatically.

```
// What we have
void Person::ShowInformation()
{
    cout << this->name << endl;
    cout << this->age << endl;
}
```

```
bob.ShowInformation();
sarah.ShowInformation();
```

```
// What we effectively have
void Person::ShowInformation(Person* this)
{
    cout << this->name << endl;
    cout << this->age << endl;
}
```

```
ShowInformation(&bob);
ShowInformation(&sarah);
```

The compiler handles this part automatically.

Using this to Resolve Ambiguity

```
class Person
{
    string name;
    int age;
public:
    Person(string name, int age);
};
```

```
Person::Person(string name, int age)
{
    name = name;
    age = age;
}
```

C++ looks at the most "local" variables first.

```
Person::Person(string name, int age)
{
    this->name = name;
    this->age = age;
}
```

this->name belongs to the invoking object.

Avoiding Ambiguity in the First Place

```
class Person
{
    string _name;
    int    _age;
public:
    Person(string name, int age);
};
```

Use leading underscores to indicate class variables.

```
// Option 1
Person::Person(string name, int age)
{
    _name = name;    // Assign the arguments to a private class variable
    _age = age;
}
```

```
// Option 2
Person::Person(string name, int age)
{
    this->name = name;
    this->age = age;
}
```

The `this` Keyword Is Optional

```
class Example
{
    int a, b, c, d, e, f;
public:
    Example();
};
```

```
Example::Example()
{
    this->a = 0;
    this->b = 0;
    this->c = 0;
    this->d = 0;
    this->e = 0;
    this->f = 0;
}
```

If you like the first option, that's okay! Just try to be consistent.

Versus

```
Example::Example()
{
    a = 0;
    b = 0;
    c = 0;
    d = 0;
    e = 0;
    f = 0;
}
```

Personally, I prefer the option without the repetition of the **`this`** keyword.

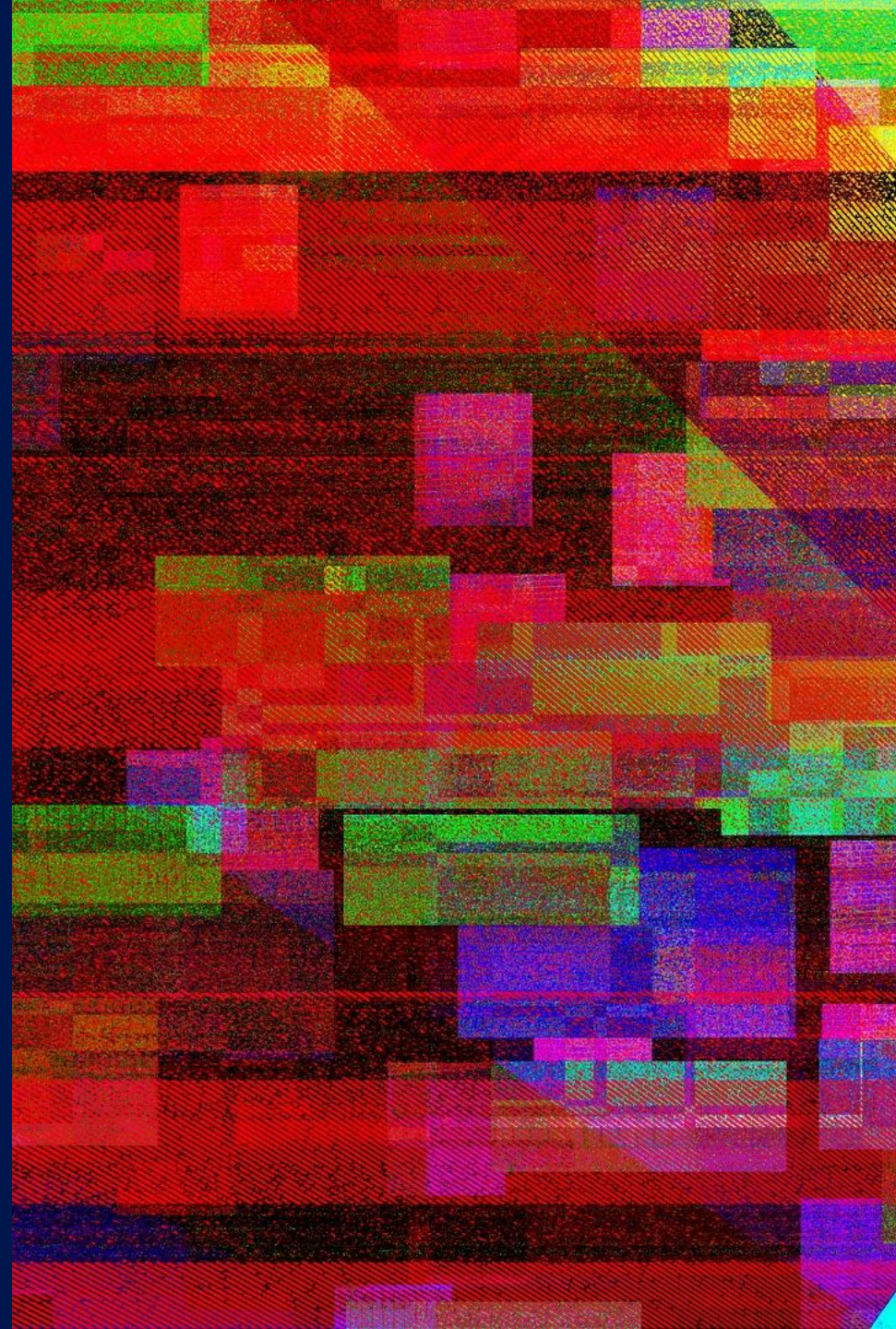
Why `this->` and not `this.`?

(AKA, Why is this C++ different than Java?)

- In C++, `this` is a **pointer** to the invoking object (we haven't gotten to pointers yet).
- Java (and other languages) use the period for accessing members of classes.
- C++ does as well, but for **objects**, not pointers:

```
personOBJECT.ShowInformation();
personPOINTER->ShowInformation();
```

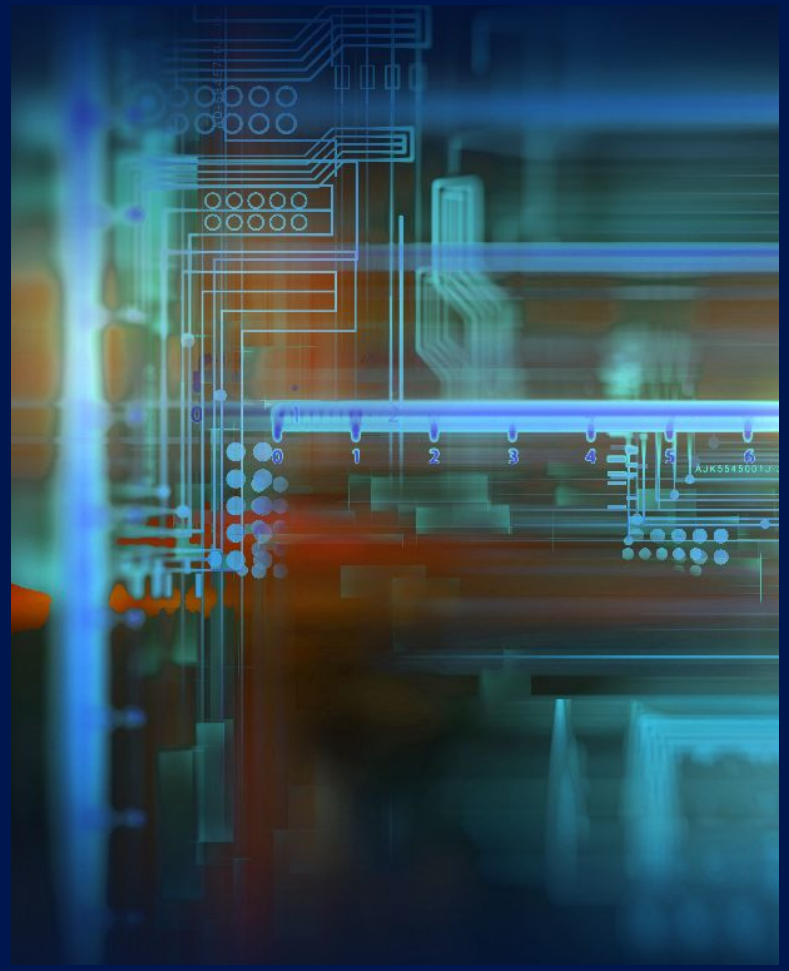
- We'll look at the distinction a bit more later on, but for now, inside a class, it's **always** `this->` in C++.



Recap

- Object-oriented programming utilizes **classes** to **encapsulate** data and functionality.
- Classes are reusable components that can be **instantiated** as often as we need to.
- Classes can **hide data** and present public **interfaces** to prevent unwanted changes to our data.
- Every language has its own quirks, but the concepts behind classes are more or less universal.

Conclusion



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

Thank you for watching.

References

Norris, K. (2021). Image of light switch [Online Image]. The Spruce. https://www.thespruce.com/how-to-wire-and-install-single-pole-switches-1152330

Playstation. (2018, September 11). Screenshot of Spider-man video game [Online Image]. CNET. https://www.cnet.com/tech/gaming/spider-man-is-the-same-video-game-weve-been-playing-for-a-decade/

Simplilearn. (2022, February 21). Screenshot of C++ code [Online Image]. Simplilearn. https://www.simplilearn.com/tutorials/cpp-tutorial/oops-concepts-in-cpp