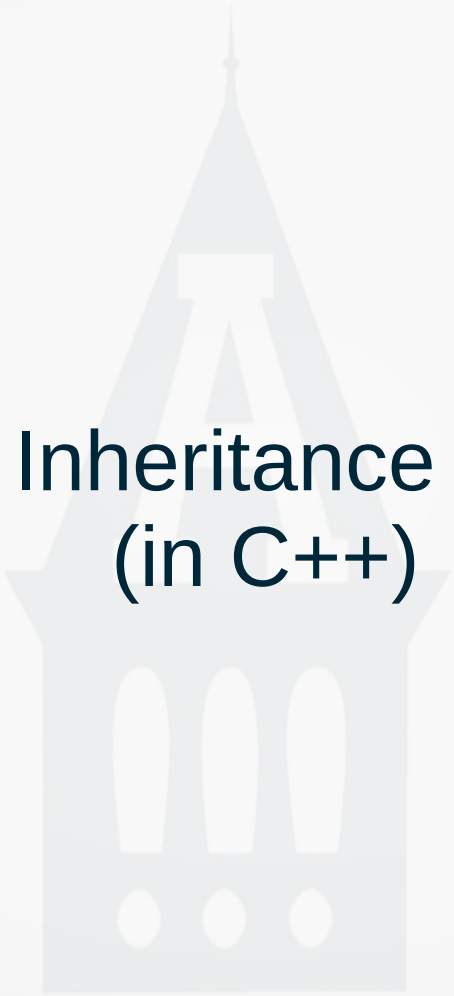
CS 3460



Introduction to Inheritance & Polymorphism (in C++)

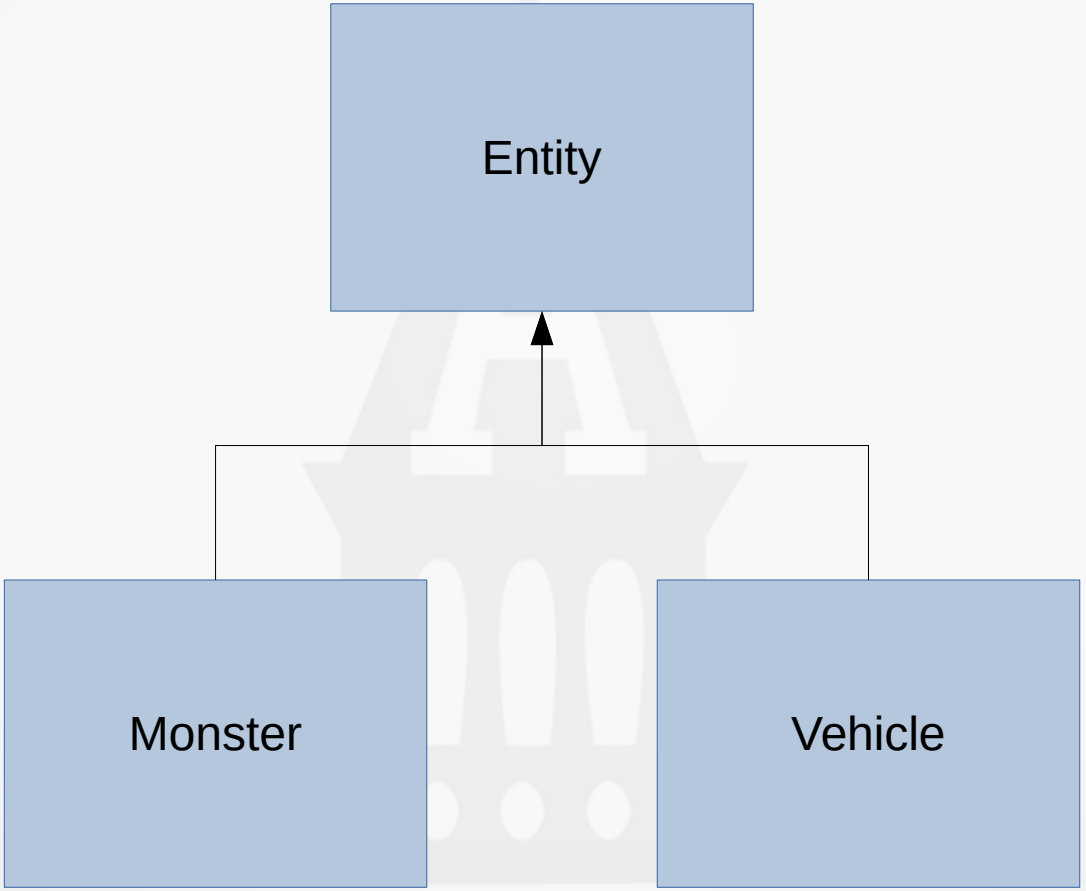
Inheritance & Polymorphism

- C++ and Java share a lot in common
  - class-based inheritance
  - virtual/overridable methods
  - abstract methods and classes
- As expected, some difference in terminology and capabilities, but fundamentally the same concepts in both languages

Terminology

- ***Base*** or ***parent*** class : Java uses term ***super*** class
- ***Derived*** or ***child*** class : Java uses term ***sub-class***
- ***Abstract method***
  - Java uses the `abstract` keyword
  - C++ uses the term ***pure virtual method***; different syntax
- ***Abstract class***
  - Java uses the `abstract` keyword
  - No specific syntax; any class with a pure virtual method
- ***Virtual method***
  - In Java all methods are virtual
  - C++ requires the keyword `virtual`

Example Topic for Demonstration



Entity Class – Code Tour

```
class Entity
{
  public:
    Entity(double facing, double posX, double posY);

    virtual void update(double elapsedTime);
    void report();

    std::uint32_t getId() { return m_id; }
    double getFacing() { return m_facing; }
    double getSpeed() { return m_speed; }
    double getPosX() { return m_posX; }
    double getPosY() { return m_posY; }

  protected:
    double m_facing;
    double m_speed;
    double m_posX;
    double m_posY;

    virtual std::string getType() = 0;
    virtual void reportUnique() = 0;
    void move(double elapsedTime);

  private:
    static std::uint32_t nextId;
    std::uint32_t m_id;
};
```

Entity Class – Code Tour

```
class Entity
{
  public:

    virtual void update(double elapsedTime);

  protected:

    virtual std::string getType() = 0;
    virtual void reportUnique() = 0;

  private:
    static std::uint32_t nextId;
    std::uint32_t m_id;
};
```

- Notice `virtual` on the update method
- Notice `virtual` and `= 0` syntax on `getType`/`reportUnique`
- Notice `static` on `nextId`

Entity Class – Code Tour

```
std::uint32_t Entity::nextId = 0;


Entity::Entity(double facing, double posX, double posY) :
    m_id(nextId++),
    m_speed(0),
    m_facing(facing),
    m_posX(posX),
    m_posY(posY)
{
}
```

- Static variables (in a class) require an implementation in the `.cpp` file
- Overloaded constructor, using member initialization list
  - Note use of static `nextId` to initialize `m_id`

Entity Class – Code Tour

```
void Entity::update(double elapsedTime)
{
    move(elapsedTime);
}

void Entity::move(double elapsedTime)
{
    auto vectorX = std::cos(m_facing);
    auto vectorY = std::sin(m_facing);

    m_posX += (vectorX * elapsedTime * m_speed);
    m_posY += (vectorY * elapsedTime * m_speed);
}
```

- Nothing remarkable about the `update` implementation
  - But note it is marked as `virtual;` it can be overridden
- Nothing remarkable about the `move` implementation

Entity Class – Code Tour

```
void Entity::report()
{
    std::cout << std::format("--- {} Report ---\n", getType());
    std::cout << std::format("id       : {}\n", m_id);
    std::cout << std::format("Position : ({:.3f}, {:.3f})\n", m_posX, m_posY);
    std::cout << std::format("Facing   : {}\n", m_facing);
    std::cout << std::format("Speed    : {}\n", m_speed);

    reportUnique();

    std::cout << std::endl;
}
```

- Note the calls to the `getType` and `reportUnique` methods
  - They are both pure virtual methods
  - This is *run-time polymorphism*

Inheritance – Visibility Modifiers

Access Location	`public`	`protected`	`private`
Within the class	yes	yes	yes
Derived class	yes	yes	no
Use of object	yes	no	no

- Key difference is the `protected` keyword
  - No concept of packages like Java

Derived Classes

- General form for declaring a derived class

```
class <derived> : <visibility modifier> <parent>
{
    ... rest of class declaration ...
};
```

- Specific example…

```
class Vehicle : public Entity
{
    ... rest of class declaration ...
};
```

- Note use of `public` before the parent class
  - Can be `public`, `protected`, or `private`; next slide...

Inheritance – More Visibility Modifiers

Inherited/Existing	`public`	`protected`	`private`
`public`	public	protected	private
`protected`	protected	protected	private
`private`	not visible	not visible	not visible

- For example: If parent has a `public` member, and derived class inherits as `protected`…
  - `public` member from parent is `protected` in derived
- Usually inheritance is `public`, but you'll see others used

Inheritance – Constructors

- Constructor Delegation
  - From Java you remember constructor chaining
  - Same concept/rules in C++
- Inheriting Constructors
  - When deriving a class, constructors are not inherited
    - They exist at the type, but aren't inherited/renamed
  - C++ allows constructors to be inherited!

```
class Monster : public Entity
{
  public:
    using Entity::Entity; // inheriting the Entity constructors
```

Polymorphism

- Same kinds of polymorphism you are used to from Java
  - Overloading functions/methods
  - Functional – functors, lambdas
    - Java has lambdas; we don’t teach it in CS 1410
  - Run-time – dynamic binding
  - Compile time – templates

Polymorphism – Virtual Methods

- In Java all methods are virtual; polymorphically overridden
- Not true in C++
  - Must be marked as `virtual` in parent class
  - Optionally mark as `virtual` in derived, but recommended
- Annotating polymorphic methods
  - In Java use the `@Override` annotation
  - In C++ use the `override` keyword

Polymorphism – Non-Virtual Methods

- Do **NOT** do this!! You will confuse people
- C++ allows a non-virtual method from a parent class to be overridden
  - Behavior is much different from virtual methods
  - The method invoked depends on what type is being used to access the object
- I’ve never needed or used this

Polymorphism – Abstract Methods & Classes

- In C++ an abstract method is called a pure virtual method

```
virtual void myAbstractMethod() = 0;
```

- Remember, also have regular virtual methods

```
virtual void myVirtualMethod() {…}
```

- Any class with a pure virtual method is abstract
  - No additional syntax needed
  - Can not instantiate an abstract class

Vehicle Class – Code Tour

```
class Vehicle : public Entity
{
  public:
    enum class Color
    {
        Red,
        Blue,
        Silver,
        White,
    };

    Vehicle(Color color, double facing, double posX, double posY);

    virtual void update(double elapsedTime) override;

  protected:
    virtual std::string getType() override { return "Vehicle"; }
    virtual void reportUnique() override;

  private:
    Color m_color;
};
```

Vehicle Class – Code Tour

```
class Vehicle : public Entity
{
  public:
    enum class Color
    {
        Red,
        Blue,
        Silver,
        White,
    };

    Vehicle(Color color, double facing, double posX, double posY);

    virtual void update(double elapsedTime) override;

  protected:
    virtual std::string getType() override { return "Vehicle"; }
    virtual void reportUnique() override;

  private:
    Color m_color;
};
```

Vehicle Class – Code Tour

```
class Vehicle : public Entity
{
  public:
    enum class Color
    {
        Red,
        Blue,
        Silver,
        White,
    };

    Vehicle(Color color, double facing, double posX, double posY);

    virtual void update(double elapsedTime) override;

  protected:
    virtual std::string getType() override { return "Vehicle"; }
    virtual void reportUnique() override;

  private:
    Color m_color;
};
```

Vehicle Class – Code Tour

```
class Vehicle : public Entity
{
  public:
    enum class Color
    {
        Red,
        Blue,
        Silver,
        White,
    };

    Vehicle(Color color, double facing, double posX, double posY);

    virtual void update(double elapsedTime) override;

  protected:
    virtual std::string getType() override { return "Vehicle"; }
    virtual void reportUnique() override;

  private:
    Color m_color;
};
```

Vehicle Class – Code Tour

```
Vehicle::Vehicle(Color color, double facing, double posX, double posY) :
    m_color(color),
    Entity(facing, posX, posY)
{
}
```

- Uses member initialization for the color
- Delegates remaining parameters to `Entity` constructor

Vehicle Class – Code Tour

```
void Vehicle::update(double elapsedTime)
{
    if (m_speed == 0)
    {
        m_speed = 0.75;
    }

    Entity::update(elapsedTime);
}
```

- Provides a new `update` method
- But note how it invokes the `Entity::update` method

Vehicle Class – Code Tour

```
void Entity::report()
{
    std::cout << std::format("--- {} Report ---\n", getType());
    std::cout << std::format("id       : {}\n", m_id);
    std::cout << std::format("Position : ({:.3f}, {:.3f})\n", m_posX, m_posY);
    std::cout << std::format("Facing   : {}\n", m_facing);
    std::cout << std::format("Speed    : {}\n", m_speed);

    // Let each derived class report on its unique properties
    reportUnique(); // Runtime polymorphism, dynamic binding

    std::cout << std::endl;
}
```

- `Entity` has a non-virtual `report` method. It makes a call to the pure virtual `reportUnique` method

Vehicle Class – Code Tour

```
void Vehicle::reportUnique()
{
    std::cout << "Color: ";
    switch (m_color)
    {
        case Color::Red:
            std::cout << "Red";
            break;
        case Color::Blue:
            std::cout << "Blue";
            break;
        case Color::Silver:
            std::cout << "Silver";
            break;
        case Color::White:
            std::cout << "White";
            break;
    }

    std::cout << std::endl;
}
```

- Here we have the `Vehicle::reportUnique` implementation

Monster Class – Code Tour

```
class Monster : public Entity
{
  public:
    using Entity::Entity;
    Monster(std::string name, double facing, double posX, double posY);

    virtual void update(double elapsedTime) override;

    std::string getName() { return m_name; }

  protected:
    virtual std::string getType() override { return "Monster"; }
    virtual void reportUnique() override;

  private:
    std::string m_name{ "anonymous" };
};
```

Monster Class – Code Tour

```
class Monster : public Entity
{
  public:
    using Entity::Entity;
    Monster(std::string name, double facing, double posX, double posY);

    virtual void update(double elapsedTime) override;

    std::string getName() { return m_name; }

  protected:
    virtual std::string getType() override { return "Monster"; }
    virtual void reportUnique() override;

  private:
    std::string m_name{ "anonymous" };
};
```

- If there were more than one `Entity` constructor, all would be inherited; can't pick and choose
- Sets `m_name` to `anonymous`; for initialization

Code Demo – Use of Entity, Vehicle, & Monster

Inheritance – Destructors

- Automatically invoked when object goes out of scope
  - code scope (stack)
  - deleted from heap
- Invoked in reverse order of constructors, derived to parent
- If intended to be used polymorphically, declare destructors as virtual
  - Otherwise not all in class hierarchy are invoked!

Code Demo – non-Virtual & Virtual Destructors

Default Methods

- If not defined, C++ writes some default implementations…
  - default constructor
  - copy constructor
  - assignment operator
- If you provide any overloaded constructor, the default constructor is not automatically provided
  - But can provide it like

```
class MyClass
{
    public:
        MyClass() = default;
        MyClass(int param);
}
```

Deleted Methods

- Think about `std::unique_ptr`, the compiler guarantees a copy of it can't be made.
  - Default copy constructor
  - Default assignment operator
- How can that be done: deleted methods!
- General form: `[whatever the method] = delete;`

Code Demo – Default/Deleted Methods

Final Classes & Methods

- Can prevent a class from being used in inheritance by marking it as final

```
class ImTheLastOfMyType final {}
```

- Can do the same for virtual methods

```
virtual void ImTheBestPossible final {}
```