COMP130: Game Architecture

3: Advanced OOP Design

Sprint reviews and retrospectives

- Documentation is now available on the COMP130 LearningSpace
- If your Product Owner has not already gone through these with you, they should soon!

Access control

Access control

- For **encapsulation**, it is a good idea to restrict access to certain attributes and methods from outside the class
- **Private** members are only accessible from the class's **own** methods
- **Protected** members are accessible from the class's own methods, **and** methods defined in **subclasses**
- **Public** members are accessible from **outside** the class

Access control in C++

```
class MyClass
{
public:
    void thisMethodIsPublic();
    int thisFieldIsPublicToo;

protected:
    void thisMethodIsProtected();
    int thisFieldIsProtectedToo;
    float soIsThisOne;

private:
    void thisMethodIsPrivate();
    int thisFieldIsPrivateToo;
    std::string andThisOne;
};
```

Rules of thumb for access control

- The **public interface** of an object is how it interacts with other objects and the rest of the program
- The **protected interface** of an object is what allows subclasses to change the way the base class behaves
- The **private members** of an object are implementation details, hidden from the outside world
- For maintainable and reusable classes, minimise the **surface area**
  - Make as much as possible private
  - If it **needs** to be accessible from subclasses, make it protected
  - If it **needs** to be accessible from outside, make it public
- Avoid making fields public
  - Unless outside code **needs** unrestricted read/write access to your data? (If it does then you've probably designed it wrong...)

Getters and setters

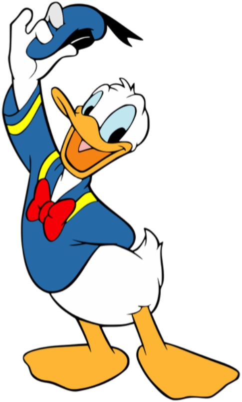
```
class MyClass
{
private:
    float speed;

public:
    float getSpeed() { return speed; }
    void setSpeed(float value) { speed = value; }
};
```

- Allows different access control for getting and setting, e.g. public getter, protected setter
- Allows extra logic upon getting or setting values
- Maybe a slight performance penalty, but compiler can often inline them (if they're not virtual)

Inheritance

Relationships

- OOP models three types of real-world relationships: **is a**, **has a** and **is a type of**



- Donald **is a** duck
- A duck **has a** bill
- A duck **is a type of** bird

Is-a $\rightarrow$ Instantiation

- "X is a Y" means "the specific object X is an object of the type Y"
- Is-a is modelled by **classes** and **instances**:
  - "Donald is a duck" $\rightarrow$ "`donald` is an **instance** of the **class** `Duck`"

```
class Duck
{
};

Duck donald;
```

Has-a $\rightarrow$ Composition

- "X has a Y" means "an object of type X **possesses** an object of type Y"
- OOP models this by having a **field** on X which holds an instance of Y
  - "A duck has a bill" $\rightarrow$ "The class `Duck` has a **field** which contains an instance of the class `Bill`"

Composition in C++

"A duck has a bill"

- "Each instance of class Duck contains **an instance** of class Bill"

```
class Bill { ... };

class Duck
{
private:
    Bill bill;
};
```

- **Or** "Each instance of class Duck contains **a pointer** to an instance of class Bill"

```
class Bill { ... };

class Duck
{
private:
    Bill* bill;
};
```

Composition in C++

- The contained instance of `Bill` is stored **inside** the instance of `Duck` (literally, in memory)
- It is constructed when the `Duck` instance is constructed, and destroyed when it is destroyed

- The contained instance of `Bill` is stored **outside** the instance of `Duck`, which only stores a **pointer**
- It is usually constructed manually using **`new`**, and so must be destroyed manually using **`delete`**

When to use each?

- Pointers are more versatile
  - Allow several pointers to the same instance (e.g. several ducks might **have-a** single pond)
  - Allow **circular references** (e.g. a duck **has-a** bill, and a bill **has-a** duck)
  - Pointers allow **polymorphism** (e.g. a pointer to a "duck" might actually be a pointer to a mallard)
- **But** stored instances are easier to work with
  - Destruction is handled automatically
- They model slightly different types of **has-a** relationship
  - Instance: **has-a** in the sense of "contains"
  - Pointer: **has-a** in the sense of "is associated with"

Is-a-type-of $\rightarrow$ Inheritance

- "X is a type of Y" means "If an object is of type X, then it is **also** of type Y"
  - "A duck is a type of bird" $\rightarrow$ "If something is a duck, then it is also a bird"
  - "Every duck is a bird"
  - "If something is true for all birds, then it must be true for ducks"
- In OOP terms, this is called **inheritance**

Inheritance

- Recall: an **object** is a collection of **fields** (data) and **methods** (code)
- Recall: the **class** defines which fields and methods an object possesses
- "X is a type of Y" $\rightarrow$ class `X` inherits from class `Y`
- Class X inherits all of the fields and methods from class Y, as well as any fields and methods of its own

When to inherit?

- When modelling an **is-a-type-of** relationship from the real world
- When several classes can **share** some fields and/or methods
  - I.e. to minimise **code duplication**
- When several classes should have methods with the **same names**, but which do **different things**
  - This is called **polymorphism** — more on this later

Inheritance in C++

```
class Shape
{
public:
    float centreX, centreY;
    Shape(float cx, float cy)
        : centreX(cx), centreY(cy) { }
};

class Circle : public Shape
{
public:
    float radius;
    Circle(float cx, float cy, float r)
        : Shape(cx, cy), radius(r) { }
    float getArea()
    {
        return 3.14159f * radius * radius;
    }
};
```

Chains of inheritance

- "A mallard is a type of duck, which is a type of bird, which is a type of vertebrate, which is a type of animal..."
- Is-a-type-of is **transitive**
  - If A is-a-type-of B and B is-a-type-of C, then A is-a-type-of C
- Likewise: class A inherits from class B, which inherits from class C, ...
  - "Inherits from" is also transitive

```cpp
class A {
public:
    int x;
    A(int x) : x(x) {}
    int foo() { return x*x; }
};

class B: public A {
public:
    int y;
    B(int x, int y) : A(x), y(y) {}
};

class C: public B {
public:
    int z;
    C(int x, int y, int z)
        : B(x, y), z(z) {}
};

class D: public A {
public:
    int y;
    D(int y) : A(20), y(y) {}
    int bar() { return x*x*x; }
};

class E {
public:
    int x;
    E(int x) : x(x) {}
};
```

Socrative FALCOMPED

```cpp
void first() {
    C c(10, 20, 30);
    cout << c.z << endl;
}

void second() {
    B b(10, 20);
    cout << b.z << endl;
}

void third() {
    B b(10, 20);
    cout << b.foo() << endl;
}

void fourth() {
    B b(10, 20);
    cout << b.bar() << endl;
}

void fifth() {
    D d(10);
    cout << d.foo() << endl;
}
```

Polymorphism

Polymorphism

- From Greek: “many-shape-ism”
- Different classes can have the **same public interface**
- Thus we can write code that **uses** this interface, but doesn’t need to worry about the **implementation** behind it

Method overriding

- A class can **override** methods defined in the class from which it inherits
- The overridden method can call the method from the base class, but it doesn't have to

Without polymorphism

```
class Shape { ... };
class Circle : public Shape { ... };
class Square : public Shape { ... };
class Triangle : public Shape { ... };

std::vector<Shape*> shapes;
// Populate shapes with circles, squares, triangles...

for (Shape* shape : shapes)
{
    if (shape->isCircle)
        drawCircle(shape->centre, shape->radius);
    else if (shape->isSquare)
        drawSquare(shape->centre, shape->size);
    ...
}
```

Polymorphism to the rescue!

```
class Shape {
    public: virtual void draw() {}
};
class Circle : public Shape {
    public: void draw() override {
        drawCircle(centre, radius);
    }
};
class Square : public Shape {
    public: void draw() override {
        drawSquare(centre, size);
    }
};

for (Shape* shape : shapes)
    shape->draw();
```

Polymorphism to the rescue!

- All subclasses of `Shape` implement `draw`
- We can call `shape->draw()` without worrying which type of shape it is
- The **virtual method table** takes care of calling the correct `draw` function depending on the type of `shape`, no extra code required

Instance types

```
Shape myShape;
```

- `myShape` is an instance of `Shape`, allocated on the **stack**

```
Shape* myShapePtr;
```

- `myShapePtr` points to an instance of `Shape` or of a subclass of `Shape`, allocated on the **heap**
- Polymorphism works for pointers, but not for instances on the stack

Abstract classes

- Some classes should never be instantiated directly, as they only exist to be inherited from
  - `Shape` is an example
- Such classes are called **abstract**
- Abstract classes generally have one or more **pure virtual methods** — methods which are left unimplemented so **must** be implemented in subclasses

Abstract class example

```
class Shape
{
public:
    virtual void draw() = 0;
};
```

- Here `= 0` marks the method as **pure virtual**
- In C++, having at least one pure virtual method implicitly marks the class as **abstract**
- Now you will get a compile error if you try to instantiate `Shape` directly
- To become not abstract, subclasses of `Shape` must override `draw`
- Subclasses of `Shape` which **do** override `draw` can be instantiated
- Trying to instantiate a subclass of `Shape` which **does not** override `draw` will also give a compile error

Interfaces

- An abstract class in which **all** methods are pure virtual is called an **interface**
- Interfaces do not contain any implementation, but specify a set of methods which subclasses must implement
- NB: some languages (e.g. C#, Java) make a distinction between classes and interfaces; C++ does not

OOP design

- What **classes** might be defined in a Mario-style platform game?
- What classes might **inherit** from one another?

Worksheet B: Mandelbrot