COMP3016

OOP with C++ Final

Immersive Games Technologies

Dr Ji-Jian Chin

University of Plymouth 2024

Recap of Last Week

- Understand C++ concepts in file processing functions of:
  - File Open
  - File Read
  - File Write
  - File Close
- Apply the aforementioned C++ File processing functions
- Define inheritance and identify the different types available
- Create code for classes that show inheritance properties
- Differentiate between access specifiers for class members and inheritance specifiers class objects
- Define multiple inheritance and make use in program code

This week

- Polymorphism
- Game programming patterns

Topics for this lecture segment C

- Define virtual base classes and how they are used in controlling access to identifiers in class hierarchies
- Make use of virtual methods and pure virtual methods
- Make use of abstract base classes

Pointing/referring to a base class

- However, because Cat and Dog are derived from Animal, Cat and Dog have an Animal part. Therefore, it makes sense to do the following:

```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() <<
endl;
}
```

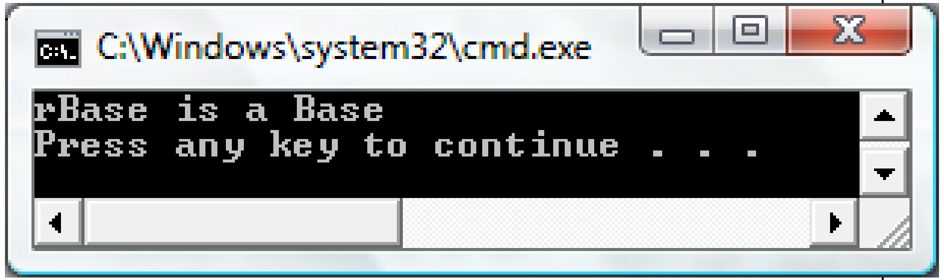
- This would allow passing in any class derived from **Animal**
- Instead of one function per animal, we get one function that works with all classes derived from **Animal**!
- There is one problem here as **rAnimal** is an **Animal** reference, **rAnimal.Speak()** will call **Animal::Speak()** instead of the derived version of **Speak().**
- To solve this issue, make the base pointers call the derived version of a function instead of the base version, **virtual functions or methods** are introduced.

Virtual methods

- A virtual function is a special type of function that resolves to the most-derived version of the function with the same signature or identifier.
- To declare a method as virtual, just add a keyword `virtual` in front of the declaration.

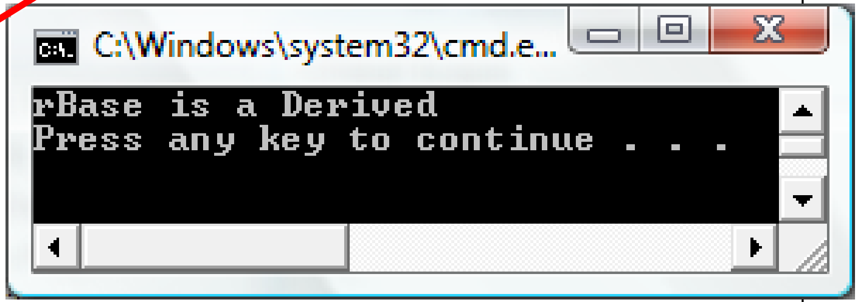
```
class Base {
public:
    const string GetName() {
   return "Base";
    }
};

class Derived: public Base {
public:
    const string GetName() {
   return "Derived";
    }
};

int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

Because **rBase** is a reference to the **Base** portion of a **Derived** object, when **rBase.GetName()** is evaluated, it would be resolved to **Base::GetName().**



```
class Base {
public:
     virtual const string GetName() {
    return "Base";
     }
};

class Derived: public Base {
public:
     virtual const string GetName() {
    return "Derived";
     }
};

int main()
{
     Derived cDerived;
     Base &rBase = cDerived;
     cout << "rBase is a " << rBase.GetName() << endl;

     return 0;
}
```

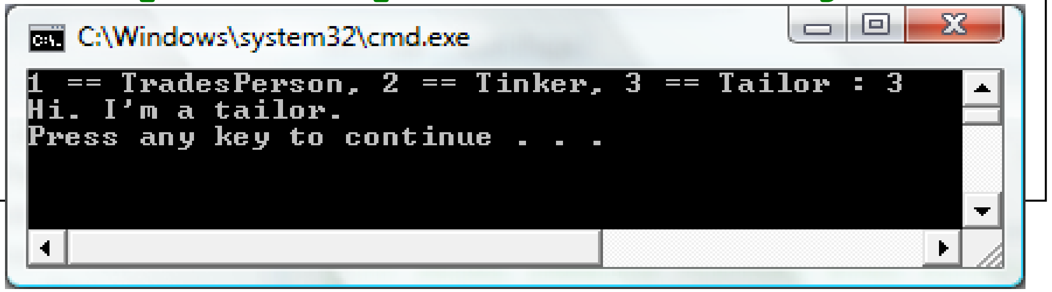
When made virtual, the compiler looks through all inherited classes looking for the most derived version of the same function and use that instead.



Another example of virtual function usage

```
#include <iostream>
using namespace std;
class TradesPerson { // base class
 public:
       virtual void sayHi( ) { cout << "Just hi." << endl; }
};

class Tinker : public TradesPerson { // derived class 1
 public:
 virtual void sayHi( ) { cout << "Hi, I'm a tinker." << endl; }
// sayHi remains virtual even without explicitly declared virtual
// however, it is advisable to explicitly make it virtual
// Tinker::sayHi( ) overrides TradesPerson::sayHi( )
};

class Tailor : public TradesPerson { // derived class 2
 public:
 virtual void sayHi( ) { cout << "Hi. I'm a tailor." << endl; }
// sayHi remains virtual even without explicitly declared virtual
// however, it is advisable to explicitly make it virtual
// Tailor::sayHi( ) overrides TradesPerson::sayHi( )
};
```

```
int main( ) {
  TradesPerson* p; // pointer to base class object
  int selection;
  cout << "1 == TradesPerson, 2 == Tinker, 3 == Tailor";
  cin >> selection;
  switch(selection ) {
        case 1: p = new TradesPerson(); break;
        case 2: p = new Tinker(); break;
        case 3: p = new Tailor(); break;
// A base class pointer may point to any base or derived class
  object.
// No cast is needed.
  }
// invoke the sayHi method via the pointer

  p->sayHi( ); // run-time binding in effect
               // if p points to a TradesPerson, then
                                          TradesPerson::sayHi( )
               // if p points to a Tinker, then Tinker::sayHi( )
               // if p points to a Tailor, then Tailor::sayHi( )
  delete       p;      // free the dynamically allocated storage
  return 0;
}
```



Animal Class with Virtual methods

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;

class Animal
{
private:
    std::string m_strName;

protected:
    // We're making this constructor protected because
    // we don't want users to create Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
    {
    m_strName = strName;
    }

public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};
```

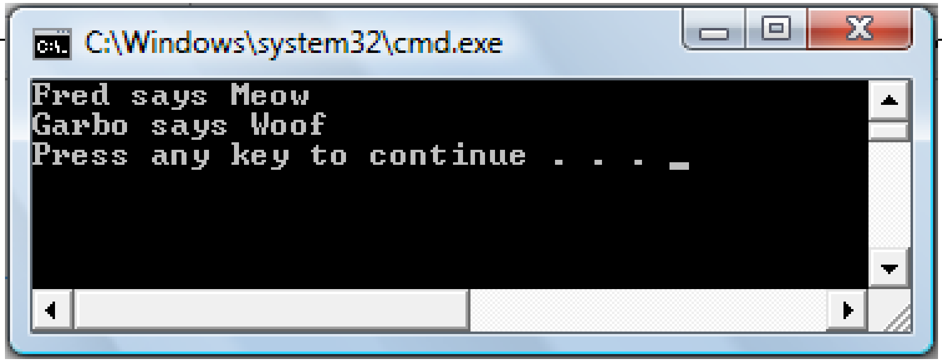
Note **Animal::GetName()** was **NOT** made virtual. This is because **GetName()** is never overridden in any of the derived classes, therefore there is no need.

Animal Class with Virtual methods (cont.)

```
class Cat: public Animal
{
public:
    Cat(std::string strName)
    : Animal(strName)
    {
    }
    virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
    : Animal(strName)
    {
    }
    virtual const char* Speak() { return "Woof"; }
};
```

Animal Class with Virtual methods (cont.)

```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() <<
endl;
}

int main()
{
    Cat cCat("Fred");
    Dog cDog("Garbo");

    Report(cCat);
    Report(cDog);

    return 0;
}
```



Pure virtual methods

- All examples before this have shown virtual methods with the body definition (method operations fully defined).
- However, there is a special method in C++ called pure virtual method or abstract method with no body.
- A pure virtual method is a method created solely as a placeholder and to be overwritten/redefined in other classes - i.e. a task similar to a function prototype declaration.
- A pure virtual method has no function body.
- To create a pure virtual method, instead of defining the body of the method, just equate the method to 0 (zero).
- Only virtual methods can be defined as `pure`.
- This tells the compiler that the method will be further defined by another class (derived)

```
class Base
{
public:
    const char* SayHi() { return "Hi"; }
       // a normal non-virtual function
    virtual const char* GetName() { return "Base"; }
       // a normal virtual function
    virtual int GetValue() = 0;
       // a pure virtual function
};
```

- Two consequences of pure virtual classes:
  - Any class with one or more **pure virtual functions** becomes an **Abstract Base Class (ABC)**, which means that **it cannot be instantiated**.
  - Any **derived** class must define a body for this virtual function, or that derived class will be considered an **Abstract Base Class** as well.

Abstract Base Classes

- Any class with a pure virtual function automatically makes it an Abstract Base Class (ABC) - exists to be inherited by other classes and itself cannot be used to create object instances.
- Because the existence of the pure virtual function, the class is said to be incomplete - when the compiler encounters the pure virtual function statement, compilation fails because the function is not defined.
- A class may be derived from an ABC but MUST define body for at least the pure virtual function(s) - other functions need not be defined because they can inherit the base class's versions.
- Once all pure virtual methods in the abstract base class are overridden in a derived class, the later will become a concrete class, i.e., class that can be used to instantiate objects

Back to our Animal Class example

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;

class Animal
{
private:
    std::string m_strName;

protected:
    // We're making this constructor protected because
    // we don't want users to create Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
    {
    m_strName = strName;
    }

public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};
```

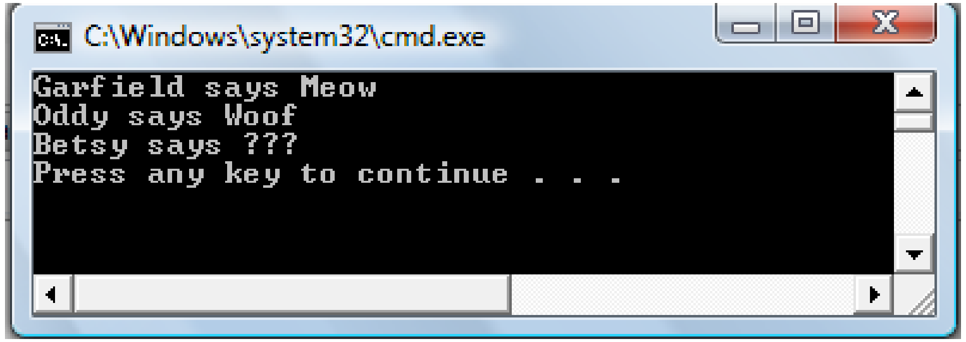
Back to our Animal Class example (cont.)

```
class Cat: public Animal
{
public:
    Cat(std::string strName)
    : Animal(strName)
    {
    }
    virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
    : Animal(strName)
    {
    }
    virtual const char* Speak() { return "Woof"; }
};
```

Back to our Animal Class example (cont.)

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
    : Animal(strName)
    {
    }

    // Oops! Forgot to redefine Speak virtual function
};

void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}

int main()
{
    Cat cCat("Garfield");
    Dog cDog("Oddy");
    Cow cCow("Betsy");

    Report(cCat);
    Report(cDog);
    Report(cCow);

    return 0;
}
```

Forgot to redefine **Speak**, so cCow.Speak() resolved to Animal.Speak(), which isn't what we wanted



Animal Class example with pure virtual function

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

public:
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

    std::string GetName() { return m_strName; }
    virtual const char* Speak() = 0; // pure virtual function
};
```

- **Speak()** is now a pure virtual function. As a pure virtual function, **Speak()** in **Animal** class has no function body.
- This makes **Animal** class an abstract class and cannot be instantiated.
- Since **Cow** class is derived from **Animal** class and did not define the **Speak()** function, **Cow** class is then considered an abstract class as well and will yield compilation error if instantiated.

Animal Class example with pure virtual function (cont.)

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
    : Animal(strName)
    {
    }

    // Oops! Forgot to redefine Speak virtual function
};

void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() <<
endl;
}

int main()
{
    Animal("Animal");
    Cow cCow("Betsy");
    Report(cCow);
    return 0;
}
```

Speak() not defined although the base class has set this function as a pure virtual function.

Attempt to instantiate Animal class will fail as this class contains a pure virtual function

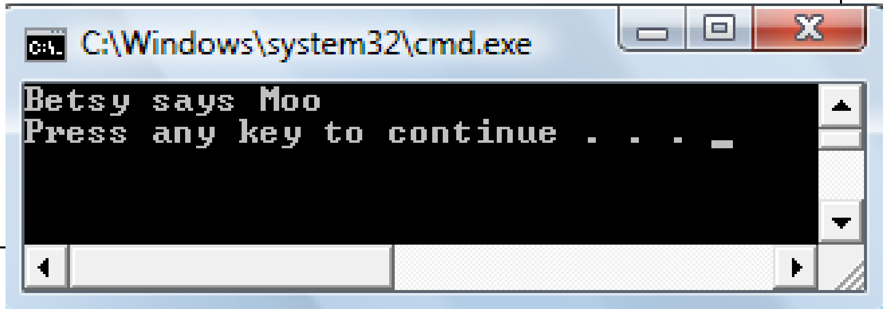
Attempt to instantiate Cow class will result in compilation error

Animal Class example with pure virtual function (cont.)

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
    : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Moo"; }
};

void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() <<
endl;
}

int main()
{
    Cow cCow("Betsy");
    Report(cCow);
    return 0;
}
```

To fix the compilation error of the previous slide, a function body is defined for Speak()

Attempt to instantiate Cow class with an object is now successful



Another example of abstract class usage

```
class ABC { // Abstract Base Class
public:
    virtual void open( ) = 0; // pure virtual function
};
class X : public ABC { // 1st derived class
public:
    virtual void open( ) {
      cout<<"overridden open()";
    }  // override ABC::open( )
};
class Y : public ABC { // 2nd derived class
    // *** open is not overridden
};
int main( ) {
    ABC a1;  // ERROR: ABC is abstract, cannot instantiate
    X x1;     // OK: X overrides ABC::open( )
          // and is no more abstract
    Y y1;     // ERROR: Y does not override ABC::open( )
          // and is still abstract
    return 0;
}
```

Interface Base Classes

- Similar to ABCs, Interface Base Classes (IBC) also have pure virtual functions that must be redefined in derived classes.
- However, unlike ABCs, Interface Base Classes <u>do not have any other members - no data members, no non-virtual methods, nothing else! These classes contain on pure virtual functions.</u>
- Interface Base Classes are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class - i.e. define functions that all derived classes MUST have.
- Interface classes are often named beginning with an “I”. For example: class IEncoder or class IProcessor

Polygon Example based on Interface class

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;

// Interface class
class IPolygon
{
// Operations
public:
    virtual void SetValues(int value1, int value2) = 0;
    virtual int CalcArea() = 0;
};
```

Interface class, **IPolygon**

Pure virtual functions **SetValues** & **CalcArea**

- Take note that the Interface class **IPolygon** only has **pure virtual member function** declarations.
- There are no data members or non pure virtual function declarations.

Polygon Example based on Interface class (cont.)

```
// Implementation class (Rectangle)
class Rectangle : public IPolygon
{
// Attributes
private:
    int m_Width;
    int m_Height;

// Operations
public:
    Rectangle(){};
    ~Rectangle(){};

    // Implement IPolygon
    virtual void SetValues(int width, int height)
    {
        m_Width = width;
        m_Height = height;
    }

    virtual int CalcArea()
    {
        return (m_Width * m_Height);
    }
};
```

Rectangle class is derived from the **IPolygon** interface class

Implement the pure virtual functions of **IPolygon** class

Polygon Example based on Interface class (cont.)

```
// Implementation class (Triangle)
class Triangle : public IPolygon
{
// Attributes
private:
    int m_Width;
    int m_Height;

// Operations
public:
    Triangle(){};
    ~Triangle(){};

    // Implement IPolygon
    virtual void SetValues(int width, int height)
    {
        m_Width = width;
        m_Height = height;
    }

    virtual int CalcArea()
    {
        return (m_Width * m_Height / 2);
    }
};
```

Rectangle class is derived from the **IPolygon** interface class

Implement the pure virtual functions of **IPolygon** class

Polygon Example based on Interface class (cont.)

Instantiate a pointer **pPolygon** of type **IPolygon** and dynamically assign **Rectangle** as the derived class

```
int main()
{
    IPolygon *pPolygon = new Rectangle();
    pPolygon->SetValues(10, 5);
    cout << "Polygon Area: " << pPolygon->CalcArea() << endl;

    delete pPolygon;
    pPolygon = new Triangle();
    pPolygon->SetValues(10, 5);
    cout << "Polygon Area: " << pPolygon->CalcArea() << endl;

    return 0;
}
```

Instantiate a pointer **pPolygon** of type **IPolygon** and dynamically assign **Triangle** as the derived class

Break time

Topics for this lecture segment D

- Design Pattern Categories
  - From the original Design Patterns: Elements of Reusable Object-Oriented Software
  - Sequencing Patterns
  - Behavioural Patterns
  - Decoupling Patterns
  - Optimisation Patterns
- Top 5 design patterns for games

Game Programming Patterns

https://gameprogrammingpatterns.com/contents.html

Who Am I?

I'm Bob Nystrom. I started writing this book while working at Electronic Arts. In my eight years there, I saw a lot of beautiful code, and a lot of not-so-beautiful code. My hope was that I could take what I learned from the good stuff, write it down here, and then teach it to the people writing the awful stuff.

If you want to get in touch with me, you can email `bob` at this site or just ask me (`@munificentbob`) on twitter. If you just can't get enough of my writing, I also have a blog. If you like the book, you'll probably like it too.

Design Patterns: Elements of Reusable Object-Oriented Software

- Command
- Flyweight
- Observer
- Prototype
- Singleton
- State

Sequencing Patterns

- Double buffer
- Game loop
- Update method

Behavioral Patterns

- Bytecode
- Subclass Sandbox
- Type object

Decoupling Patterns

- Component
- Event Queue
- Service Locator

Optimisation Patterns

- Data locality
- Dirty Flag
- Object Pool
- Spatial Partition

Top 5 Design Patterns for Games

- Component – build flexible and reusable game objects
- State – handles state transitions of characters or systems
- Object Pool – crucial in performance critical games
- Event Queue – asynchronous event management where timing is outside main game loop
- Command – useful in games with complex input systems, undo or replay (RTS and turn-based)

Component Pattern

Problem: Traditional inheritance-based systems (e.g., class hierarchies) make game objects rigid and difficult to extend, leading to complexity and code duplication as the game grows. A class Enemy might inherit from a base class like GameObject, but adding new features to different enemy types becomes cumbersome.

Solution: The **Component Pattern** solves this by **decomposing** game entities into smaller, reusable behaviors (components). Instead of rigid hierarchies, each game entity is composed of independent components, allowing for flexibility, easy modification, and reusability.

- Composition over Inheritance: build objects combining modular components to avoid deep hierarchies.
- Extensibility: add, remove or swap components dynamically during runtime
- Decoupling of behaviour: each component is responsible for a single, well-defined behaviour, reducing dependencies and improving maintainability

```cpp
#include <iostream>
#include <unordered_map>
#include <string>
#include <memory>
// Base class for all components
class Component {
public:
    virtual void Update() = 0;  // Pure virtual function, to be implemented by derived
components
    virtual ~Component() = default;  // Virtual destructor for proper cleanup
};
// Health Component
class HealthComponent : public Component {
private:
    int health;
public:
    HealthComponent(int h) : health(h) {}
    void Update() override {
        if (health <= 0)
            std::cout << "Entity is dead!" << std::endl;
        else
            std::cout << "Health: " << health << std::endl;
    }
};
// Movement Component
class MovementComponent : public Component {
private:
    float speed;
public:
    MovementComponent(float s) : speed(s) {}
    void Update() override {
        std::cout << "Moving with speed " << speed << std::endl;
    }
};
// Render Component
class RenderComponent : public Component {
public:
    void Update() override {
        std::cout << "Rendering entity..." << std::endl;
    }
};
```

```cpp
// GameObject class that holds multiple components
class GameObject {
private:
    std::unordered_map<std::string, std::shared_ptr<Component>> components;  // Store components
by type name
public:
    // Add a component to the GameObject
    template <typename T>
    void AddComponent(std::shared_ptr<T> component) {
        components[typeid(T).name()] = component;
    }
    // Get a component from the GameObject
    template <typename T>
    std::shared_ptr<T> GetComponent() {
        return std::static_pointer_cast<T>(components[typeid(T).name()]);
    }
    // Update all components of the GameObject
    void Update() {
        for (auto& pair : components) {
            pair.second->Update();
        }
    }
};

int main() {
    // Create a GameObject (Entity)
    GameObject player;
    // Add different components to the GameObject
    player.AddComponent(std::make_shared<HealthComponent>(100));
    player.AddComponent(std::make_shared<MovementComponent>(5.0f));
    player.AddComponent(std::make_shared<RenderComponent>());
    // Simulate game loop, where each entity updates its components
    player.Update();
    // Access individual component to modify or inspect
    auto health = player.GetComponent<HealthComponent>();
    std::cout << "Player health: " << health->Update() << std::endl;
    return 0;
}
```

State Pattern

- Problem: Managing complex game objects (like characters or AI) that have multiple states can become unwieldy and hard to maintain with traditional conditional logic (e.g., if-else or switch statements). As states multiply (e.g., idle, walking, jumping, attacking), these conditionals grow into a mess.

- Solution: The State Pattern allows game objects to change their behavior when their internal state changes by encapsulating state-specific behavior into separate state classes. This makes the code more modular and easier to extend or modify.

- State encapsulation: represent each state as a separate class, encapsulating specific behaviour

- State Transitions: objects maintain reference to current state and changes behaviour dynamically during run time

- Extensibility: new states can be added easily by creating new classes

```cpp
// Forward declaration of Player class
class Player;
// Base class for all states
class PlayerState {
public:
   virtual ~PlayerState() = default;
   virtual void handleInput(Player& player, const std::string& input) = 0;
   virtual void update(Player& player) = 0;
};
// Context class: Player holds the current state
class Player {
private:
   std::shared_ptr<PlayerState> currentState;
public:
   void setState(std::shared_ptr<PlayerState> state) {
      currentState = state;
   }
   void handleInput(const std::string& input) {
      currentState->handleInput(*this, input);
   }
   void update() {
      currentState->update(*this);
   }
};
// Concrete State: Idle
class IdleState : public PlayerState {
public:
   void handleInput(Player& player, const std::string& input) override {
      if (input == "walk") {
         std::cout << "Transitioning from Idle to Walking" << std::endl;
         player.setState(std::make_shared<class WalkingState>());
      } else if (input == "jump") {
         std::cout << "Transitioning from Idle to Jumping" << std::endl;
         player.setState(std::make_shared<class JumpingState>());
      }
   }
   void update(Player& player) override {
      std::cout << "Player is standing idle." << std::endl;
   }
};
```

```cpp
// Concrete State: Walking
class WalkingState : public PlayerState {
public:
   void handleInput(Player& player, const std::string& input) override {
      if (input == "stop") {
         std::cout << "Transitioning from Walking to Idle" << std::endl;
         player.setState(std::make_shared<IdleState>());
      } else if (input == "jump") {
         std::cout << "Transitioning from Walking to Jumping" <<
std::endl;
         player.setState(std::make_shared<JumpingState>());
      }
   }
   void update(Player& player) override {
      std::cout << "Player is walking." << std::endl;
   }
};
// Concrete State: Jumping
class JumpingState : public PlayerState {
public:
   void handleInput(Player& player, const std::string& input) override {
      if (input == "land") {
         std::cout << "Transitioning from Jumping to Idle" << std::endl;
         player.setState(std::make_shared<IdleState>());
      }
   }
   void update(Player& player) override {
      std::cout << "Player is jumping." << std::endl;
   }
};
```

```cpp
int main() {
   Player player;
   // Start with the Idle state
   player.setState(std::make_shared<IdleState>());
   // Simulate input and updates
   player.update();
   player.handleInput("walk");
   player.update();
   player.handleInput("jump");
   player.update();
   player.handleInput("land");
   player.update();
   return 0;
}
```

Object Pool Pattern

- Problem: Games often need to create and destroy objects frequently (e.g., bullets, enemies, particles). Constantly allocating and deallocating memory for these objects can lead to performance issues like slowdowns and excessive garbage collection, especially in resource-constrained environments.
- Solution: The Object Pool Pattern pre-allocates a fixed set of objects and reuses them, avoiding costly memory allocations and deallocations during gameplay. When an object is no longer in use (e.g., a bullet hits a target), it is "returned" to the pool and reused later, rather than being destroyed and recreated.

- Pre-allocation: create a pool of objects upfront, not dynamically during runtime
- Reusability: objects can be returned to the pool and reused, reducing overhead
- Memory management: prevents frequent allocation/deallocation

```cpp
#include <iostream>
#include <vector>
#include <memory>

// Bullet class representing an object that will be pooled
class Bullet {
private:
    bool active;  // Determines if the bullet is in use or not

public:
    Bullet() : active(false) {}

    void fire() {
        active = true;
        std::cout << "Bullet fired!" << std::endl;
    }

    void reset() {
        active = false;
        std::cout << "Bullet returned to pool." << std::endl;
    }

    bool isActive() const {
        return active;
    }
};

// Object pool class for managing bullets
class BulletPool {
private:
    std::vector<std::shared_ptr<Bullet>> bullets;

public:
    BulletPool(int poolSize) {
        for (int i = 0; i < poolSize; ++i) {
            bullets.push_back(std::make_shared<Bullet>());
        }
    }

    // Get an inactive bullet from the pool
    std::shared_ptr<Bullet> getBullet() {
        for (auto& bullet : bullets) {
            if (!bullet->isActive()) {
                bullet->fire();
                return bullet;
            }
        }
        // If all bullets are active, return a nullptr (or could expand pool)
        std::cout << "No available bullets in the pool!" << std::endl;
        return nullptr;
    }

    // Reset a bullet (return it to the pool)
    void returnBullet(std::shared_ptr<Bullet> bullet) {
        bullet->reset();
    }
};

// Example usage
int main() {
    // Create a bullet pool with a size of 3
    BulletPool pool(3);

    // Fire some bullets
    auto bullet1 = pool.getBullet();
    auto bullet2 = pool.getBullet();
    auto bullet3 = pool.getBullet();

    // Try to fire a fourth bullet (none available)
    auto bullet4 = pool.getBullet();

    // Return a bullet to the pool and reuse it
    pool.returnBullet(bullet1);
    auto bullet5 = pool.getBullet();  // Reuses the first bullet

    return 0;
}
```

Event Queue Pattern

- Problem: In complex games, many objects need to communicate with one another. For example, when a player shoots an enemy, multiple systems may need to react: the enemy takes damage, sound effects play, particles appear, and scores update. Direct communication between objects can lead to tight coupling, making the game hard to maintain or extend.
- Solution: The Event Queue Pattern allows game objects to communicate indirectly by sending and receiving events. Objects post events to a centralized queue, and other objects that are interested in those events respond to them. This decouples the sender from the receiver, promoting modularity and scalability.

- Decoupling: objects do not need direct references, only post or listen for events
- Centralised communication: allows many objects or systems to react to the same event without inter-object communication
- Efficiency: the queue can batch and distribute event in a timely manner, ensuring smooth game performance

```cpp
#include <iostream>
#include <functional>
#include <unordered_map>
#include <vector>
#include <memory>

// Base Event class
class Event {
public:
   virtual ~Event() = default;
};
// Example event: EnemyHitEvent
class EnemyHitEvent : public Event {
public:
   int enemyID;
   EnemyHitEvent(int id) : enemyID(id) {}
};
// Event Queue class
class EventQueue {
private:
   // Mapping of event type to listeners
   std::unordered_map<std::string,
std::vector<std::function<void(std::shared_ptr<Event>)>>> listeners;

public:
   // Register listener for a specific event type
   void addListener(const std::string& eventType,
std::function<void(std::shared_ptr<Event>)> listener) {
      listeners[eventType].push_back(listener);
   }

   // Post an event to the queue and notify all listeners
   void postEvent(const std::string& eventType, std::shared_ptr<Event> event) {
      if (listeners.find(eventType) != listeners.end()) {
         for (auto& listener : listeners[eventType]) {
            listener(event);  // Notify each listener
         }
      }
   }
};

// Systems that listen to events
class AudioManager {
public:
   void onEnemyHit(std::shared_ptr<Event> event) {
      auto hitEvent = std::static_pointer_cast<EnemyHitEvent>(event);
      std::cout << "AudioManager: Playing sound for enemy " << hitEvent->enemyID << std::endl;
   }
};

class ParticleManager {
public:
   void onEnemyHit(std::shared_ptr<Event> event) {
      auto hitEvent = std::static_pointer_cast<EnemyHitEvent>(event);
      std::cout << "ParticleManager: Showing particles for enemy " << hitEvent->enemyID << std::endl;
   }
};

// Example usage
int main() {
   EventQueue eventQueue;

   AudioManager audioManager;
   ParticleManager particleManager;

   // Register listeners for EnemyHitEvent
   eventQueue.addListener("EnemyHit", [&](std::shared_ptr<Event> event) {
audioManager.onEnemyHit(event); });
   eventQueue.addListener("EnemyHit", [&](std::shared_ptr<Event> event) {
particleManager.onEnemyHit(event); });

   // Post an event
   std::shared_ptr<Event> event = std::make_shared<EnemyHitEvent>(42);  // Enemy with ID 42 hit
   eventQueue.postEvent("EnemyHit", event);

   return 0;
}
```

Command Pattern

- Problem: In many games, user inputs (like keyboard or controller actions) must trigger various commands (e.g., move, attack, jump) that can be complex and require multiple steps. Managing these commands directly within input handlers can lead to tangled code, making it hard to manage and extend game functionality.
- Solution: The Command Pattern encapsulates requests as objects, allowing you to parameterize clients with queues, requests, and operations. It separates the responsibility of handling user inputs from the execution of commands, enabling easier management of user actions and adding features like undo/redo, macros, and logging.
- Encapsulation: encapsulate commands within their own objects with all information for actions
- Decoupling: input handling is decoupling from logic that executes commands
- Extensibility: able to add new commands without modifying existing code
- Support for undo/redo: since objects can store state, this allows for easy undo/redo functionality.

```cpp
// Command Interface
class Command {
public:
    virtual ~Command() = default;
    virtual void execute() = 0; // Method to execute the command
};
// Receiver class
class Player {
public:
    void move() {
        std::cout << "Player moves forward!" << std::endl;
    }

    void attack() {
        std::cout << "Player attacks!" << std::endl;
    }
};
// Concrete Command for moving
class MoveCommand : public Command {
private:
    Player& player; // Reference to the receiver

public:
    MoveCommand(Player& p) : player(p) {}
    void execute() override {
        player.move(); // Call the receiver's move method
    }
};
// Concrete Command for attacking
class AttackCommand : public Command {
private:
    Player& player; // Reference to the receiver

public:
    AttackCommand(Player& p) : player(p) {}
    void execute() override {
        player.attack(); // Call the receiver's attack method
    }
};
```

```cpp
// Concrete Command for attacking
class AttackCommand : public Command {
private:
    Player& player; // Reference to the receiver
public:
    AttackCommand(Player& p) : player(p) {}
    void execute() override {
        player.attack(); // Call the receiver's attack method
    }
};
// Invoker class to handle commands
class InputHandler {
private:
    std::vector<std::unique_ptr<Command>> commands; // List of commands

public:
    void bindCommand(std::unique_ptr<Command> command) {
        commands.push_back(std::move(command)); // Store commands
    }

    void invokeCommands() {
        for (auto& command : commands) {
            command->execute(); // Execute all commands in the list
        }
    }
};
// Example usage
int main() {
    Player player; // Create player
    InputHandler inputHandler; // Create input handler

    // Bind commands to the player
    inputHandler.bindCommand(std::make_unique<MoveCommand>(player));
    inputHandler.bindCommand(std::make_unique<AttackCommand>(player));

    // Simulate input handling
    std::cout << "Handling input..." << std::endl;
    inputHandler.invokeCommands(); // Invoke the commands
    return 0;
}
```

Worth taking a look on your own

- Singleton
- Observer
- Command

- Also worth mentioning Game Loop, Update Method and Double Buffer sequencing patterns are pretty foundational and applied to ALL games, hence not elaborated on.

Alternative Research

- https://www.linkedin.com/pulse/essential-design-patterns-game-developers-envision-studio-pvt-ltd-kbirc/
- https://sourcemaking.com/design_patterns
- https://refactoring.guru/design-patterns
- https://www.youtube.com/watch?v=0A0-nkB1fxQ&ab_channel=ThisisGameDev

- Recategorises the patterns into creational, behavioural and structural patterns.
- For the purpose of this course, we'll stick with Robert Nystrom's definitions for the coursework assessment, but you will find some intersect.
- Not crucial to apply, as there is some criticism about the approach
- You might need to justify WHY you applied the pattern and argue if it actually helps your project!

Thanks so far!

Any Questions?