COMP140-GAM160: Further Programming

4: Inheritance and Polymorphism

Learning outcomes

- **Understand** Inheritance in Object Orientated Programming
- **Understand** Polymorphism role in creating Games
- **Apply** your knowledge of Inheritance and Polymorphism to programming problems

FALMOUTH UNIVERSITY

Classes Review

Classes

Classes

- Let us look at Classes again

Classes

- Let us look at Classes again
- Classes allow us to create our own data types

Classes

- Let us look at Classes again
- Classes allow us to create our own data types
- They consist of a series of data(variables) and functions that operate on the data

Classes

- Let us look at Classes again
- Classes allow us to create our own data types
- They consist of a series of data(variables) and functions that operate on the data
- Functions and variables inside the class can be marked with the following **access specifiers**

Classes

- Let us look at Classes again
- Classes allow us to create our own data types
- They consist of a series of data(variables) and functions that operate on the data
- Functions and variables inside the class can be marked with the following **access specifiers**
  - **Public**: Can be accessed directly

Classes

- Let us look at Classes again
- Classes allow us to create our own data types
- They consist of a series of data(variables) and functions that operate on the data
- Functions and variables inside the class can be marked with the following **access specifiers**
  - **Public**: Can be accessed directly
  - **Private**: Can only be accessed inside the class

Classes

- Let us look at Classes again
- Classes allow us to create our own data types
- They consist of a series of data(variables) and functions that operate on the data
- Functions and variables inside the class can be marked with the following **access specifiers**
  - **Public**: Can be accessed directly
  - **Private**: Can only be accessed inside the class
  - **Protected**: Acts like private, but child classes can access

Class Examples - C++

```cpp
class Player
{
public:
    Player()
    {
        Health=100;
    };

    void TakeDamage(int health)
    {
        Health-=health;
    };

    void HealDamage(int health)
    {
        Health+=health;
    };

    ~Player(){};
private:
    int Health;
};
```

Class Examples - C# Unity

```
public class Player
{
    private int Health;

    public Player()
    {
        Health=100;
    }

    public void TakeDamage(int health)
    {
        Health-=health;
    }

    public void HealDamage(int health)
    {
        Health+=health;
    }
}
```

Classes vs Structs

Classes vs Structs

- A **Struct** is pretty much the same as a **Class**

Classes vs Structs

- A **Struct** is pretty much the same as a **Class**
- The only difference in functionally, by default:

Classes vs Structs

- A **Struct** is pretty much the same as a **Class**
- The only difference in functionally, by default:
  - Everything in a **Class** is **private**

Classes vs Structs

- A **Struct** is pretty much the same as a **Class**
- The only difference in functionally, by default:
  - Everything in a **Class** is **private**
  - Everything in a **Struct** is **public**

Classes vs Structs

- A **Struct** is pretty much the same as a **Class**
- The only difference in functionally, by default:
  - Everything in a **Class** is **private**
  - Everything in a **Struct** is **public**
- Difference by convention:

Classes vs Structs

- A **Struct** is pretty much the same as a **Class**
- The only difference in functionally, by default:
  - Everything in a **Class** is **private**
  - Everything in a **Struct** is **public**
- Difference by convention:
  - Structs are used for holding related data and tend not to have functions

Classes vs Structs

- A **Struct** is pretty much the same as a **Class**
- The only difference in functionally, by default:
  - Everything in a **Class** is **private**
  - Everything in a **Struct** is **public**
- Difference by convention:
  - Structs are used for holding related data and tend not to have functions
  - Classes hold data and functions

Creating an Instance - C++

```
//Creating on the stack, this will be deleted when it drops out of scope
Player player1=Player();

//Call take damage function, notice we use . to access functions
player.TakeDamage(20);

//Creating on the Heap, please delete!!
Player * player2=new Player();

//Call take damage function, note we use -> to access functions
player->TakeDamage(20);

//Deleting player2 on the heap
if (player2)
{
    delete player2;
    player2=nullptr;
}
```

Creating an Instance - C#

```
//Create a player
Player player1=new Player();

//Call take Damage
player1.TakeDamage(50);
```

Constructor & Deconstructor

Constructor & Deconstructor

- **Constructors** are called when you create an instance

Constructor & Deconstructor

- **Constructors** are called when you create an instance
- Constructors can take in zero or many parameters

Constructor & Deconstructor

- **Constructors** are called when you create an instance
- Constructors can take in zero or many parameters
- You need to declare different version of the constructor

Constructor & Deconstructor

- **Constructors** are called when you create an instance
- Constructors can take in zero or many parameters
- You need to declare different version of the constructor
- Deconstructors are called when the instance has been deleted (by the dropping out of scope, or explicitly deleted in C++)

Constructor & Deconstructor

- **Constructors** are called when you create an instance
- Constructors can take in zero or many parameters
- You need to declare different version of the constructor
- Deconstructors are called when the instance has been deleted (by the dropping out of scope, or explicitly deleted in C++)
- Constructors have to be names the same as the class

Constructor & Deconstructor

- **Constructors** are called when you create an instance
- Constructors can take in zero or many parameters
- You need to declare different version of the constructor
- Deconstructors are called when the instance has been deleted (by the dropping out of scope, or explicitly deleted in C++)
- Constructors have to be names the same as the class
- Deconstructors have the same name as the class but prefixed with $\sim$ (tilde symbol)

Constructors C++

```
public class Player
{
    public:
        Player()
        {
            Health=100;
            Strength=10;
        };

        Player(int health)
        {
            Health=health;
            Strength=10;
        };

        Player(int health,int strength)
        {
            Health=health;
            Strength=strength;
        };
        ~Player(){};
private:
    int Health;
    int Strength;
};
```

Constructors C++

```
//Create a player
Player * player1=new Player();

//Create another player with the one parameter constructor
Player player2=Player(10);

//Create another player with the two parameter constructor
Player * player3=new Player(100,20);

delete player1;
delete player2;
```

Constructors C#

```
class Player
{
    private int Health;
    private int Strength;

    public Player()
    {
        Health=100;
        Strength=10;
    }

    public Player(int health)
    {
        Health=health;
        Strength=10;
    }

    public Player(int health,int strength)
    {
        Health=health;
        Strength=strength;
    }

}
```

Using Constructors C#

```
//Create a player with the default no parameter constructor
Player player1=new Player();

//Create a player with one parameter constructor
Player player2=new Player(50);

//Create a player with two parametes constructor
Player player3=new Player(120,50);
```

Encapsulation

Encapsulation

- In OOP, Encapsulation is a key principle

Encapsulation

- In OOP, Encapsulation is a key principle
- This refers to the idea that all data in a class should be hidden by the caller

Encapsulation

- In OOP, Encapsulation is a key principle
- This refers to the idea that all data in a class should be hidden by the caller
- This means that **all** variables should be marked **private** or **protected**

Encapsulation

- In OOP, Encapsulation is a key principle
- This refers to the idea that all data in a class should be hidden by the caller
- This means that **all** variables should be marked **private** or **protected**
- And only functions inside the class can operate on the data

Encapsulation

- In OOP, Encapsulation is a key principle
- This refers to the idea that all data in a class should be hidden by the caller
- This means that **all** variables should be marked **private** or **protected**
- And only functions inside the class can operate on the data
- **Unity - but what about exposing variables to the editor?**

Encapsulation

- In OOP, Encapsulation is a key principle
- This refers to the idea that all data in a class should be hidden by the caller
- This means that **all** variables should be marked **private** or **protected**
- And only functions inside the class can operate on the data
- **Unity - but what about exposing variables to the editor?**
  - You should still make everything private

Encapsulation

- In OOP, Encapsulation is a key principle
- This refers to the idea that all data in a class should be hidden by the caller
- This means that **all** variables should be marked **private** or **protected**
- And only functions inside the class can operate on the data
- **Unity - but what about exposing variables to the editor?**
  - You should still make everything private
  - Then use the **(SerializeField)** attribute to make the variable visible in the inspector

Class Examples - C# Unity

```
using UnityEngine;

public class Player : MonoBehaviour
{
    (SerializeField)
    private int Health;

    public Player()
    {
        Health=100;
    }

    public void TakeDamage(int health)
    {
        Health-=health;
    }

    public void HealDamage(int health)
    {
        Health+=health;
    }
}
```

FALMOUTH UNIVERSITY

Inheritance

Introduction to Inheritance

Introduction to Inheritance

- One of the key features of OOP languages is **Inheritance**

Introduction to Inheritance

- One of the key features of OOP languages is **Inheritance**
- This allows you to **Derive** a new class from an existing one

Introduction to Inheritance

- One of the key features of OOP languages is **Inheritance**
- This allows you to **Derive** a new class from an existing one
- When this is done, the new class automatically inherits the variables and functions of the **parent** class

Introduction to Inheritance

- One of the key features of OOP languages is **Inheritance**
- This allows you to **Derive** a new class from an existing one
- When this is done, the new class automatically inherits the variables and functions of the **parent** class
- Advantages of inheritance includes

Introduction to Inheritance

- One of the key features of OOP languages is **Inheritance**
- This allows you to **Derive** a new class from an existing one
- When this is done, the new class automatically inherits the variables and functions of the **parent** class
- Advantages of inheritance includes
  - **Code reuse:** There is no need to redefine functionality, you can just inherit from a base class

Introduction to Inheritance

- One of the key features of OOP languages is **Inheritance**
- This allows you to **Derive** a new class from an existing one
- When this is done, the new class automatically inherits the variables and functions of the **parent** class
- Advantages of inheritance includes
    - **Code reuse:** There is no need to redefine functionality, you can just inherit from a base class
    - **Fewer errors:** If you build on existing class that is bug free then you are more likely to have less errors

Introduction to Inheritance

- One of the key features of OOP languages is **Inheritance**
- This allows you to **Derive** a new class from an existing one
- When this is done, the new class automatically inherits the variables and functions of the **parent** class
- Advantages of inheritance includes
  - **Code reuse:** There is no need to redefine functionality, you can just inherit from a base class
  - **Fewer errors:** If you build on existing class that is bug free then you are more likely to have less errors
  - **Cleaner code:** because of the increase of code reuse then your code is more modular and reusable.

Inheritance Example - C#

```
public class Enemy : MonoBehaviour
{
    (SerializeField)
    proteced int Damage;

    void Start()
    {
        Damage=1;
    }

    public void Attack()
    {
        Debug.Log("The attack causes "+Damage.ToString()+" damage");
    }
}
```

Inheritance Example - C#

```
public class Boss : Enemy
{
    (SerializeField)
    private int DamageMultiplier;

    void Start()
    {
        Damage=5;
        DamageMultipler=2;
    }

    public void Attack()
    {
        Debug.Log("The attack causes "+Damage.ToString()+" damage");
    }

    public void SpecialAttack()
    {
        int totalDamage=Damage*DamageMultiplier;
        Debug.Log("Special attack causes "+totalDamage.ToString()+" damage");
    }
}
```

Inheritance Example - C++

```
public class Enemy
{
    public:
        Enemy()
        {
            Damage=1;
        };

        virtual ~Enemy()
        {
        }

        void Attack()
        {
            std::cout<<"The attack causes "<<Damage<<" damage"<<std::endl;
        }
    protected:
        int Damage;
}
```

Inheritance Example - C++

```
public class Boss : public Enemy
{
    public:
        Boss()
        {
            Damage=5;
            DamageMultiplier=2;
        };

        ~Boss()
        {
        }

        void SpecialAttack()
        {
            int totalDamage=Damage*DamageMultiplier;
            std::cout<<"Special attack causes "<<totalDamage<<" damage"<<std::
                endl;
        }
    protected:
        int DamageMultiplier;


}
```

Overriding

Overriding

- You can override functions in the base class by providing a new version of the function

Overriding

- You can override functions in the base class by providing a new version of the function
- You should mark any function that you are going to override with the **virtual** keyword

Overriding

- You can override functions in the base class by providing a new version of the function
- You should mark any function that you are going to override with the **virtual** keyword
- Then in the child class, you have a function with the same signature which is marked with the **override** keyword

Overriding Example - C#

```
public class Enemy : MonoBehaviour
{
    (SerializeField)
    proteced int Damage;

    void Start()
    {
        Damage=1;
    }

    public virtual void Attack()
    {
        Debug.Log("The attack causes "+Damage.ToString()+" damage");
    }
}
```

Overriding Example - C#

```
public class Boss : Enemy
{

void Start()
{
    Damage=5;
}

public override void Attack()
{
    base.Attack();
    Damage+=1;
    Debug.Log("This is the boss attacking");
}
```

Overriding Example - C++

```
public class Enemy
{
public:
    Enemy()
    {
        Damage=1;
    };

    //Make sure you mark any base class deconstuctor as virtual!
    virtual ~Enemy()
    {
    }

    virtual void Attack()
    {
        std::cout<<"The attack causes "<<Damage<<" damage"<<std::endl;
    }
protected:
    int Damage;
}
```

Overriding Example - C++

```
public class Boss : public Enemy
{
public:
    Boss()
    {
        Damage=5;
    };

    ~Boss()
    {
    }

    void Attack() override
    {
        Enemy::Attack();
        Damage+=1;
        std::cout<<"This is the boss attacking"<<std::endl;
    }
protected:
    int DamageMultiplier;
}
```

Polymorphism

Introduction to Polymorphism

Introduction to Polymorphism

- Polymorphism is another key feature of OOP languages

Introduction to Polymorphism

- Polymorphism is another key feature of OOP languages
- The basic idea is that instances of a derived class can be treated as objects of the basic class

Introduction to Polymorphism

- Polymorphism is another key feature of OOP languages
- The basic idea is that instances of a derived class can be treated as objects of the basic class
- They can be used as parameters for functions and in collections

Introduction to Polymorphism

- Polymorphism is another key feature of OOP languages
- The basic idea is that instances of a derived class can be treated as objects of the basic class
- They can be used as parameters for functions and in collections
- We then call the functions on these objects and our code will called the 'correct' version of the function

Introduction to Polymorphism

- Polymorphism is another key feature of OOP languages
- The basic idea is that instances of a derived class can be treated as objects of the basic class
- They can be used as parameters for functions and in collections
- We then call the functions on these objects and our code will called the ‘correct’ version of the function
- This is best illustrated by an example

Polymorphism example C#

```
class Enemy{/*This has been defined in previous slides*/}
class Boss : Enemy{/*Again see previou slides*/}

//This function will be in monobehavior
void DoAttacks(Enemy enemy)
{
     enemy.Attack();
}

//We probably have grabbed these from other game objects
Enemy goblin=new Enemy();
Eneny orc=new Enemy();
Boss ogre=new Boss();

//Call DoAttack on each one of these
DoAttack(goblin);
DoAttack(orc);
DoAttack(ogre);

//This even works if each instance is in a list
List<Enemy> enemies=new List<Enemy>();
enemies.Add(goblin);
enemies.Add(orc);
enemies.Add(ogre);

foreach(Enemy e in enemies)
{
     DoAttack(e);
}
```

Polymorphism example C++

```
class Enemy{/*This has been defined in previous slides*/}
class Boss : Enemy{/*Again see previou slides*/}

//This function will be in monobehavior
void DoAttacks(Enemy *enemy)
{
    enemy->Attack();
}

//We probably have grabbed these from other game objects
Enemy goblin=new Enemy();
Eneny orc=new Enemy();
Boss ogre=new Boss();

//Call DoAttack on each one of these
DoAttack(goblin);
DoAttack(orc);
DoAttack(ogre);

//This even works if each instance is in a list
std::vector<Enemy*> enemies;
enemies.push_back(goblin);
enemies.push_back(orc);
enemies.push_back(ogre);

for(Enemy * e : enemies)
{
    DoAttack(e);
}
```

Polymorphism: Some details

Polymorphism: Some details

- This is know as runtime Polymorphism and it works by making use of a construct called a virtual function table (a.k.a vtable)

Polymorphism: Some details

- This is know as runtime Polymorphism and it works by making use of a construct called a virtual function table (a.k.a vtable)
- A compiler builds up a vtable during compilation

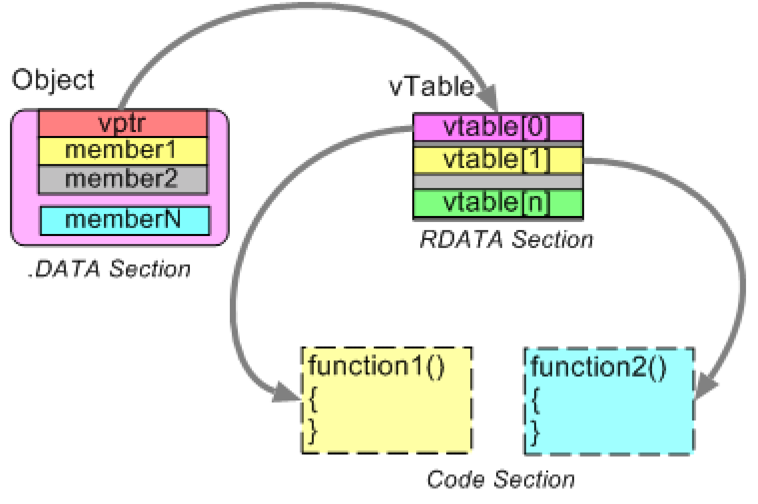
Polymorphism: Some details

- This is know as runtime Polymorphism and it works by making use of a construct called a virtual function table (a.k.a vtable)
- A compiler builds up a vtable during compilation
- Basically a hidden pointer to the vtable is added to the object and is used to call the correct version of the function

Polymorphism: Some details

- This is know as runtime Polymorphism and it works by making use of a construct called a virtual function table (a.k.a vtable)
- A compiler builds up a vtable during compilation
- Basically a hidden pointer to the vtable is added to the object and is used to call the correct version of the function
- Another thing to note, this has a cost so please don't overuse Polymorphism!

Vtable


Object
vptr
member1
member2
memberN
.DATA Section
vTable
vtable[0]
vtable[1]
vtable[n]
RDATA Section
function1()
{
}
function2()
{
}
Code Section

Abstract Classes & Interfaces

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual
- If you then inherit from an interface, you have to provide an implementation of all pure virtual functions

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual
- If you then inherit from an interface, you have to provide an implementation of all pure virtual functions
- The difference between Abstract classes and Interfaces, is that in C# you can only inherit from one class (include abstracts)

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual
- If you then inherit from an interface, you have to provide an implementation of all pure virtual functions
- The difference between Abstract classes and Interfaces, is that in C# you can only inherit from one class (include abstracts)
- But you can inherit from multiple Interfaces

Abstract Classes & Interfaces

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual
- If you then inherit from an interface, you have to provide an implementation of all pure virtual functions

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual
- If you then inherit from an interface, you have to provide an implementation of all pure virtual functions
- The difference between Abstract classes and Interfaces, is that in C# you can only inherit from one class (include abstracts)

Abstract Classes & Interfaces

- An **Interface** is very similar to an abstract class, the only difference is that every function in an Interface is marked as pure virtual
- If you then inherit from an interface, you have to provide an implementation of all pure virtual functions
- The difference between Abstract classes and Interfaces, is that in C# you can only inherit from one class (include abstracts)
- But you can inherit from multiple Interfaces

Abstract Class Example C#

```
public abstract class BaseEnemy
{
    public abstract public void Attack();

    public void Jump()
    {
        //Do jump code
    }
}

public class Orc : BaseEnemy
{
    //we have to implement attack but no need to implement Jump
    public void Attack()
    {
        //do attack
    }
}
```

Interface Example C#

```
interface Jump
{
    void DoJump();
}

interace Attack
{
    void DoAttack();
}

public class Orc : Jump, Attack
{
    //we have to implement Attack and Jump Interface
    public void DoAttack()
    {
        //do attack
    }

    public void DoJump()
    {
        //do Jump
    }
}
```

Abstract Class Example C++

```
class BaseEnemy
{
public:
    //Make sure we have a deconstructor defined!
    virtual ~BaseEnemy(){};
    virtual void Attack()=0;
    void Jump()
    {
        //Do jump code
    };
}

class Orc : public BaseEnemy
{
    public:
        //we have to implement attack but no need to implement Jump
        void Attack()
        {
            //do attack
        }
}
```

Interface Example C#

```
class IJump
{
public:
    //We must provide a virtual deconstructor
    virtual ~Jump(){};
    void DoJump()=0;
}

class IAttack
{
public:
    virtual ~Attack(){};
    void DoAttack()=0;
}

class Orc : public IJump, public IAttack
{
//we have to implement Attack and Jump Interface
public:
    void DoAttack()
    {
        //do attack
    }

    void DoJump()
    {
        //do Jump
    }
}
```

Interface Discussion

Interface Discussion

- You can think of an Interface as a contract

Interface Discussion

- You can think of an Interface as a contract
- The derived class must implement the Interface's function

Interface Discussion

- You can think of an Interface as a contract
- The derived class must implement the Interface's function
- I can leverage Polymorphism to work with interfaces

Interface Discussion

- You can think of an Interface as a contract
- The derived class must implement the Interface's function
- I can leverage Polymorphism to work with interfaces
- This means that I can consume derived classes in a function that takes in pointers (in C++) or references (in C#)

Interface Discussion

- You can think of an Interface as a contract
- The derived class must implement the Interface's function
- I can leverage Polymorphism to work with interfaces
- This means that I can consume derived classes in a function that takes in pointers (in C++) or references (in C#)
- Or I can process a collection of instances that implement the Interface

Interface Discussion

Interface Discussion

- Lastly, Interfaces a great tool for working with others. We as a group could create the interface together

Interface Discussion

- Lastly, Interfaces a great tool for working with others. We as a group could create the interface together
- Then another programmer can write Classes which implement the Interface

Interface Discussion

- Lastly, Interfaces a great tool for working with others. We as a group could create the interface together
- Then another programmer can write Classes which implement the Interface
- While another writes code which consumes instances of the Interface

Interface Discussion

- Lastly, Interfaces a great tool for working with others. We as a group could create the interface together
- Then another programmer can write Classes which implement the Interface
- While another writes code which consumes instances of the Interface
- `https://stackoverflow.com/questions/4456424/what-do-programmers-mean-when-they-say-code-aga`

Coffee Break

Exercise

Exercise 1 - Inheritance

- Please use one of the following projects as a starting point
  - C# Unity - `https://github.com/Falmouth-Games-Academy/GAM160-Exercises`
  - C++ - `https://github.com/Falmouth-Games-Academy/COMP140-Exercises`
- You are creating an Fantasy RPG create a class hierarchy which represented the following Ranged Enemies, Melee Enemies, Healer Enemies
- Implement some functions for these classes
- Have you consider having a common base class?

Exercise 2 - Polymorphism

- Now add a pure virtual attack function to the base class
- Change how attack is implemented in each derived class
- Add a few instances of each class to a collection
- Iterate through the collection and call the Attack function on each instance. This should be triggered by a key press

References