

COMP140-GAM160: Further Programming

3: 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

Classes Review

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++

```
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

- ▶ A **Struct** is pretty much the same as a **Class**
- ▶ The only difference in functionality, 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
player1.TakeDamage(20);

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

//Call take damage function, note we use -> to access functions
player2->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 & Destructor

- ▶ **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
- ▶ Destructors 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
- ▶ Destructors have the same name as the class but prefixed with ~ (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 parameters constructor  
Player player3=new Player(120,50);
```

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;
    }
}
```


Inheritance

Introduce 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]
    protected 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;
        DamageMultiplier=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;
        };

        ~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;
}
}
```

Polymorphism

Collections & Polymorphism

Coffee Break

Static Keyword & Singletons

Exercise

References