Session 02

Generic Types & Collections

Session Overview

- Generic Types
- Namespaces
- Dictionaries
- Lists
- Sorting
- Statics (& Singletons)
- Control Scripts

Generic Types

- Some classes and methods require a type to be specified within angle brackets. For example: `GetComponent<XYZ>()`
- This allows the same method to work with different types. For example, we might use GetComponent to get a reference first to a Rigidbody component and then later to an AudioSource

```
rb = GetComponent<RigidBody>();
as = GetComponent<AudioSource>();
```

- All collections we will use make use of these Generic Parameters
  - C# - `List<int> numbers = new List<int>();`
  - C++ - `list<int> numbers;`

Namespaces

- Namespaces are a means of organising classes into groups.
- Namespaces allow us to avoid naming collisions and is good practice for API writers to wrap their code in a namespace.
- In order to use collections and some other types we have to add the follow at the top of our code file
  - C# - `using System.Collections.Generic;`
  - C++ - `using namespace std;`

Research Task

- BA - Research the following C# collections:
  - List
  - Dictionary
- BSc - Research the following C++ collections
  - vector or TArray
  - map or TMap
- How are these different to arrays?

Lists (vector or TArray)

- Lists are dynamically-sized collections.
- This means that without re-declaring the collection we can add more data to the end, or we can remove an element from the middle and all subsequent elements automatically move one index position down.
- Lists are especially useful when keeping track of a frequently changing set of items, for example: enemies which are currently active in the scene, or players who have joined your multiplayer game.

List - C#

```
// Create new List to hold scores as ints
List<int> scores = new List<int>();
// Add 100 to scores
scores.Add(100);
// Add 2000 to scores
scores.Add(2000);
// retrieve the 2nd scores from the list (starts at 0)
int player2Score=scores[1];
// Remove 100 from the list
scores.remove(100);
```

List (aka vector) - C++

```
// Create new List to hold scores as ints
vector<int> scores;
// Add 100 to scores
scores.push_back(100);
// Add 2000 to scores
scores.push_back(2000);
// retrieve the 2nd scores from the list (starts at 0)
int player2Score=scores[1];
// Remove 100 from the list
scores.erase(0);
```

List (TArray) – C++ & Unreal

```
// Create new List to hold scores as ints
TArray<int32> scores;
// Add 100 to scores
scores.Add(100);
// Add 2000 to scores
scores.Add(2000);
// retrieve the 2nd scores from the list (starts at 0)
int player2Score=scores[1];
// Remove 100 from the list
scores.Remove(100);
```

Sorting – C#

- If you want a custom class to be sortable then you must implement the IComparable<T> interface, where T is the type of your class.

  `public class Loot : IComparable<Loot> { …`

- This interface ensures that your class contains the method:

  `public int CompareTo(T other)`

- This method takes the same type as the class implementing the method, in our case `CompareTo(Loot other)`
  - It must return a positive integer if it is of higher rank or value than other.
  - It must return a negative integer if it is of lower rank or value than other.
  - It must return zero if the two instances are equal in rank or value.

Sorting - C#

- For example:

```
using System;  // required for IComparable interface

public class Loot : IComparable<Loot> {
        public int value;

        public int CompareTo(Loot other) {
                if (other == null)
                {
                        return 1;
                }
                return this.value - other.value;
        }
}
```

Sorting – C++

- There is a function in the Standard Template Library which sorts elements in a collection.
- **sort** function receives the following as parameters
  - **Iterator** to first element in the collection to sort
  - **Iterator** to last element in the collection to sort
  - **Compare** function, this is a function which receives two parameters of the type in the Container (you can omit this and sort will use < operator to sort)
- See for more details - http://www.cplusplus.com/reference/algorithm/sort/

Sorting – C++ & Unreal

- **TArray** has several sort functions which operator very similar to C++ sort e.g.
  - **Sort()** – assume that operator < is defined for the type in the container
- There are also versions of this function which take in a predicate class to carry out a custom function
- More Details:
  - https://docs.unrealengine.com/latest/INT/API/Runtime/Core/Containers/TArray/index.html
  - https://answers.unrealengine.com/questions/4234/question-how-to-sort-array.html

Dictionaries (map or TMap)

- A dictionary is a collection of key-value pairs.
- The data types of the key and the pair are declared when the dictionary is created.
- These are useful if you want to quickly access a variable

Dictionary - C#

```
// Create new Dictionary to represent a scoreboard
Dictionary<string,int> scoreboard = new
                                    Dictionary<string,int>();
// Add Brian’s score of 100 to scoreboard
scoreboard.Add(“Brian”,100);
// Add Kate’s score of 2000 to scoreboard
scoreboard.Add(“Kate”,2000);
// retrieve Brian’s score
int brianScore=scoreboard[“Brian”];
// Remove Kate’s score from scoreboard
scoreboard.remove(“Kate”);
```

Map - C++

```
// Create new Dictionary to represent a scoreboard
map<string,int> scoreboard;
// Add Brian’s score of 100 to scoreboard
scoreboard[“Brian”]=100;
// Add Kate’s score of 2000 to scoreboard
scoreboard[“Kate”]=2000;
// retrieve Brian’s score
int brianScore=scoreboard[“Brian”];
// Remove Kate’s score from scoreboard
scoreboard.erase(“Kate”);
```

TMap – C++ & Unreal

```
// Create new Dictionary to represent a scoreboard
TMap<FString,int32> scoreboard;
// Add Brian’s score of 100 to scoreboard
Scoreboard.Add(TEXT(“Brian”), 100);
// Add Kate’s score of 2000 to scoreboard
Scoreboard.Add(TEXT(“Kate”), 2000);
// retrieve Brian’s score
int32 brianScore=scoreboard[“Brian”];
// Remove Kate’s score from scoreboard
scoreboard.Remove(“Kate”);
```

Workshop

- Create a scene which contains a variety of ‘pickup’ items
- On a left-click, the clicked item should be added to an inventory list and then deactivated within the scene.
- Ensure the list is public so it can be seen in the.
- On a right-click, remove the first item from the list and re-activate it.
- Stretch goal: have the pickups sortable by value when [space] is hit.
- Stretch goal: create a UI which allows the player to select the returned item instead of using the first item.

Statics

- Variables, methods and classes can all be declared as static.
- Static elements are not instantiated, instead a single copy of that data is shared by the entire program.
- A class may only be static if all of its variables and methods are also static. For example: Unity’s Time class.
- It is possible to instantiate a class which has some instance variables and also some static variables.
- We can create, for example, an Enemy class which contains a static list of type Enemy, and on Start() each instance adds itself to the list.

Statics

```
static List<Enemy> enemies = new List<Enemy>();

void Start() {
        enemies.Add(this);
}

static void ShrinkAllEnemies() {
        foreach (Enemy e in enemies) {
                e.transform.localScale *= 0.5f;
        }
}
```

Singletons

- Singletons are classes which can only be instantiated once.
- They share several similarities with static classes, however singletons offer additional functionality such as implementing interfaces.
- Singletons should declare a static variable named instance for which the data type is its own class, then a static GetInstance() method returns the one instance (instantiating it if required).
- For more info see: MSDN – Implementing Singleton in C#
- https://sourcemaking.com/design_patterns/singleton/cpp/1

Manager Class

Statics and Singletons are often used to Manage Game Logic.

```
public static class GameManager : MonoBehaviour {
	public static int score;
	public static int lives;

	public static void SetPause(bool paused) {
		// snip …
	}
}
```

Within some other class:

```
	if (GameManager.score > 10000) GameManager.lives++;
```

Discussion

- Suggest an area of your group project where you could (or already do) use a Manager.
  - What are the benefits?
  - What are the shortfalls?
  - How else could the same effect be achieved?

Next Session

- Practice using Lists & Dictionaries
- Bring an example of your work to discuss at the start of next week’s session.