COMP140-GAM160: Game Architecture

4: Data Structures, Collections, & Generic Types

Learning outcomes

- **Understand** the various collection classes in C++ & C#
- **Compare** the collection classes
- **Implement** an application which uses some of the collection classes

Common Data Structures

Introduction

- In Programming we have concept of reusable data structures which can be used to build applications
- These can be used in order to build larger systems (e.g. Inventory Systems, AI Navigation etc)
- Most programming languages have these built in
- Before writing any system you should always examine these data structures and pick the appropriate one for your Use Case

Dynamic Array

The Problem

- Arrays in C++ & C# are fixed in size
- During development you need to know exactly how many item are going to be in the array
- If you need to add elements and you don't have enough space, you will need to carry out the following
  - Create a new array of the appropriate size
  - Copy elements from the old array into this new one
  - Destroy the old array
  - Add in the new element
- The above process can be quite costly

The Solution

- Luckily in most programming languages we have a Data Structure which grows in size when we require it
  - In C# we have the **List** class
  - In C++ we have the **vector** class
- These classes have the same properties as an array
  - Items are located contiguously in memory
  - We can randomly access elements using an index
  - We can iterate through each element
- You should consider using a Dynamic Array over a normal array
- One caveat, Dynamic Arrays are more expensive

Use Case

- Manage Enemies as they are spawned into the scene
- Keep track of players as they are added into the game
- Inventory systems

C# List Example

```
List<int> scores=new List<int>();
scores.Add(100);
scores.Add(200);
foreach(int score in scores)
{
    Debug.Log("Score is "+score.ToString() ↩
        );
}
int player1Score=scores[0];
scores.Remove(100);
```

C++ Vector Example

```
vector<int> scores;
scores.push_back(100);
scores.push_back(200);
for (int score : scores)
{
    std::cout<<"Score is "<<score<<std::endl;
}
int player1Score=scores[0];
scores.Remove(100);
```

Additional Notes

- Try to avoid insertion/deleting in the middle of the collection
- Searching the collection is linear elements and will increase as more elements are added (O(n))
- insertion/deleting at the end of the collection is constant in performance (O(1))

Generic Types

Quick Aside - Generic Programming

- Generic Programming is where you write one piece of code which operates on many different types
- This uses a concept called Templates which act in proxy for the type
- The Compiler then generates the code which uses the actual type

Look back at Vector/List

- In the previous section you would have noticed the following
  - vector<int> or List<int>
- These are know as generic parameters and you should insert the data type that the collection will handle (including your own data types aka classes and structs)

Generic Programming

- You can write your own generic classes and functions but this is beyond the scope of this class
- C++ examples - `https://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part`
- C# examples - `http://www.tutorialsteacher.com/csharp/csharp-generics`
- Word of warning, it is often difficult to write generic code
- If you have errors they are often difficult to isolate as the compiler messages are so cryptic

Linked List

The Problem

- You have started using a dynamic array and you have notice performance is poor on adding/removing
- You then realise that you are adding/removing elements from the middle of the collection
- You also realise that you don't require random access to elements in the collection

The Solution

- In this case a Linked List would be a better choice
  - In C# we have the **LinkedList** class
  - In C++ we have the **list** class
- Linked Lists contain elements (called Nodes) which usually have a reference (or pointer) to the previous and next Node in the list
- This means that there is a slight increase in memory needed when working with lists

Use Case

- If you AI character has to visit a series of waypoints, these could be stored in a list
- Your Player has a number of quests they can
- If the AI/Player carries an action and a number of systems need to be notified of the event

C# Linked List Example

```
LinkedList<Transform> waypoints=new LinkedList< ↩
    Transform>();

waypoints.AddLast(GameObject.Find("Waypoint1"). ↩
    Transform);
waypoints.AddLast(GameObject.Find("Waypoint2"). ↩
    Transform);
waypoints.AddLast(GameObject.Find("Waypoint3"). ↩
    Transform);

foreach(Transfrom t in waypoints)
{
    Debug.Log("Waypoint Locations "+t.position. ↩
        ToString());
}
```

C# Linked List Example

```
waypoints.AddFirst(GameObject.Find("Waypoint0"). ↩
    Transform);

LinkedListNode<Transform> waypoint2Node = linked.Find( ↩
    GameObject.Find("Waypoint2"));
waypoints.AddAfter(waypoint2Node,GameObject.Find(" ↩
    SpecialQuest"));
```

C++ List Example

```
list<vec2> waypoints;

waypoints.push_back(vec2(10,10));
waypoints.push_back(vec2(15,15));
waypoints.push_back(vec2(20,20));

for(vec2 position:waypoints)
{
    std::cout<<"Waypoint Locations "<<position.x<< ↩
        " "<<position.y<<std::endl;
}
```

C++ List Example

```
waypoints.push_front(vec2(0,0));

auto iter=std::find(waypoints.begin(),waypoints.
    end(),vec2(15,15));
waypoints.insert(iter, vec3(25,25));
```

Additional Notes

- Linked Lists usually support constant time insertions and deletions in the collection (O(1))
- Also perform better than dynamic arrays for moving elements around the collection
- This feature means that Linked Lists are a good data structure if you need to sort your data
- Main drawback of Linked Lists is that you can't have direct access to elements in the list, it takes linear time (O(n)) to access

Queue

The Problem

- If you need to visit items in a certain (e.g front to back)
- Examples of this could be waypoints or commands to an AI character

The Solution

- In this case a Queue would be a good choice
  - In C# we have the **Queue** class
  - in C++ we have the **queue** class
- This is **F**irst-**I**n-**L**ast-**O**ut data structure
- You add elements to the end of the queue and you remove elements from the start

Use Case

- An RTS game where you can add orders to a unit, these are then carried out sequence
- An RTS where you have a base which produces units
- A spawning system, where you have to defeat enemies in a specific order

C# Queue Example

```
Queue<GameObject> unitsToBuild=new Queue<GameObject>() ↩
    ;

unitsToBuild.Enqeue(soliderPrefab);
unitsToBuild.Enqeue(builderPrefab);
unitsToBuild.Enqeue(tankPrefab);

foreach(GameObject go in unitsToBuild)
{
    Debug.Log("Units to build "+go.name);
}
```

C# Queue Example

```
GameObject nextUnitToBuild=unitsToBuild.Peek();

unitsToBuild.Dequeue();
```

C++ Queue Example

```
queue<Command> aiCommands;

aiCommands.push(Command("Attack"));
aiCommands.push(Command("Recharge"));
aiCommands.push(Command("Run"));
```

C++ Queue Example

```
Command nextCommand=aiCommands.front();

aiCommands.pop();
```

Stack

The Problem

- If you need to manage the state of an AI character
- If you need to implement a Undo system

The Solution

- A Stack would be a good choice
  - In C# we have the **Stack** class
  - in C++ we have the **stack** class
- This is **L**ast-**I**n-**F**irst-**O**ut data structure
- You add elements to the top of the stack and you remove elements from the top

C# Stack Example

```
Stack<Command> issuedCommands=new Stack<Command>();

issuedCommands.Push(new Command("Edit"));
issuedCommands.Push(new Command("Create"));
issuedCommands.Push(new Command("Updat"));
```

C# Stack Example

```
Command lastCommandIssued=issuedCommands.Peek();

Command lastCommandIssued=issuedCommands.Pop();
```

C++ Stack Example

```
stack<AIState> aiStates;

aiStates.push(Command("Attack"));
aiStates.push(Command("Idle"));
aiStates.push(Command("Run"));
```

C++ Stack Example

```
Command lastState=aiStates.top();

aiStates.pop()
```

Associative Array: Map & Dictionary

The Problem

- If you need to store one unique copy of an element
- You want to access an element via a key
- You are doing lots of searches for an element

The Solution

- You should use an Associative array
  - In C# we have the **Dictionary** class
  - in C++ we have the **map** or **unordered_map** class
- These data structures are structured as key-value pair
- It allows you to retrieve the items via the key
- This makes it a good choice for looking up large data sets

Use Case

- If you are creating a resource management system for handling textures, models or other assets
- Localisation system, each language is stored in an Associative Array
- Unit Manager, a class to manage units created in the game
- Save Game System

C# Dictionary Example

```
Dictonary<string,int> highScoreTable=new Dictonary<
    string,int>();

highScores.Add("Brian",200);
highScores.Add("Sarah",2000);
highScores[Julia]=4000;

foreach(KeyValuePair<string,int> pair in
    highScoreTable)
{
    Debug.Log("High Score "+pair.Key+" "+pair.Value);
}
```

C# Dictionary Example

```
if (highScores.ContainsKey("Brian"))
{
    int score=highScores["Brian"];
}

highScores.Remove("Sarah");
```

C++ Map Example

```
Map<string,int> highScores;

highScores["Brian"]=200;
highScores["Sarah"]=2000;
highScores["Julia]=4000;

for(auto iter : highScores)
{
    std::cout<<"High Score "+iter.first<<" "<<iter. ↩
        second<<std::endl;
}
```

C++ Map Example

```
auto iter=highScores.find("Brian");
if (iter!=highScores.end())
{
    int score=highScores["Brian];
}

highScores.earse("Sarah");
```

Additional Notes

- Iterating over a map has a slightly annoying syntax
- Associative Arrays tend to have good performance for retrieval (O (log n))
- If you add an item and its key already exists it may overwrite the value

Coffee Break

Operations on collections

Sorting

- Sorting is where we order the items in a collection in a specific order
- There are a whole bunch of sorting algorithms including; Insertion sort, Heap sort, Quick sort (please read about these!)
- In C#, the best sorting algorithm will be picked depending on the size of the collection
- In C++, this depends on the compiler implementation
- Most of the common data types don't need additional work
- For custom classes, we have to write our own sorting algorithm

Sorting C#

- There are few ways to sort a collection
  1. Provide a custom delegate function for the sort
  2. Provide a custom class which inherits from **IComparer**
  3. Your own class has to inherit from **IComparable**
- Often you will use option 3 as the default sort
- Which then be override by option 1

C# Example - Sorting with Delegate

```
struct Character
{
    string name;
    int health;
    int strength;
}

//Adding omitted!
List<Character> characters=new List<Character>();

//Sort by health
characters.Sort(delegate (Character c1, Character c2)
{
    return (c1.health.CompareTo(c2.health));
});
```

C# Example - Sorting with ICompareable

```
struct Character:IComparable<Character>
{
string name;
int health;
int strength;

// sort by name
 public int CompareTo(Character compareCharacter)
 {
    return name.CompareTo(comareCharacter.name);
 }
}

//Adding omitted!
List<Character> characters=new List<Character>();

//Sort will use the CompareTo in the struct or class
characters.Sort()
```

C# - Points to note

- The **CompareTo** function returns an int which can be the following
  - Less than zero: The instance precedes the one passed in
  - Zero: The objects are in the same order
  - Greater than zero: The instance follows the one passed in

Sorting C++

- There are few ways to sort a collection
  1. Provide a custom function for the sort
  2. Provide a lambda expression for the sort
  3. Your own class has to override the < operator
- Often you will use option 3 as the default sort
- Which then be override by option 1
- 2 is probably the more modern way of doing it, but syntax can be confusing

C++ Example - Sorting with Function

```
struct Character
{
std::string name;
int health;
int strength;
}

bool sortByHealth(Character a, Character b){return a. ↩
    health<b.health;}

//Adding omitted!
vector<Character> characters;

//Sort by health
characters.sort(characters.begin(), characters.end(),  ↩
    sortByHealth);
```

C++ Example - Sorting < operator

```
struct Character
{
std::string name;
int health;
int strength;

bool operator <(const Character& other) const {return ↩
    name<other.name;}
}

//Adding omitted!
vector<Character> characters;

//Sort by health
characters.sort(characters.begin(), characters.end());
```

Exercise

Exercise 1 - Collections

1. Download one of the following projects as a zip file
    - BA Students -
    - BSc Students -
2. Add additional items to the collection
3. Display these to the screen

Exercise 2 - Sorting

1. Write a default sort, so that the items are sorted by name
2. Sort the collection when the **s** key is pressed
3. Write another sort, to sort by score, trigger this off by a key press
4. Write another sort, to sort by age, trigger this off by a key press

Exercise 3 - Searching

1. Investigate how to search for collections
2. Add code to search for specific items in the collections
3. Add visual representation to show that the search has completed, this could be a colour change or just displaying the found item elsewhere on the screen

References

https://docs.unrealengine.com/latest/INT/Programming/Development/CodingStandard/