**Spike:** Task 10

**Title:** Game Data Structures

**Author:** Sam Huffer, 101633177

# Goals / deliverables:

# Technologies, Tools, and Resources used:

* Visual Studio 2017
* Microsoft Word
* Draw.io

# Tasks undertaken

* Considered what features the data structure underlying an inventory would need to have.
* Looked at lecture notes and conducted research on data structures that could be used, noting down their features in relation to the criteria.
* Selected a data structure to form the basis of player inventories and general storage containers.
* I created the Item and Container classes, Item holding strings of information about an item, and Container building on top of vectors to allow players to store items, retrieve them and view them. I also created the ContainerItem class which inherits from both Item and Container to allow for items that are also containers, such as bags. Once the classes were complete, I made Location inherit from Container and added into the test World’s starting location some test items.
* I added the Player class (which inherits from Container to allow for a player inventory), and began on look command functionality so that the player could look at items at their location or in their inventory. I started adding functionality for players to be able to look at items inside container items at their location or in their inventory, but ran into issues accessing the contents of the container item the player was looking into. I noticed that when I was getting that item from the location or player, I was casting it from the type Item to the type Container rather than Container Item. Once I changed that, the game then allowed me to look at the contents of container items.
* I considered how to implement the commands “take from”, “put in” and “drop”; I originally implemented Container.ViewItem() to be able to get an “item in container in container in . . .”, but I figured that would be fiddley to implement with the take command, so I cut back on that in the look command to only be able to access items in a container in the player’s inventory or their current location.
* I implemented the “take from” command, allowing players to take items out of their current location or a container in their inventory or current location, and deposit said items into their inventory. Initially I only let players specify “take \_\_ from \_\_”, but then added “take \_\_ from \_\_ in [inventory or location]”. However, I later figured that was beyond the scope of this task, and removed it to keep it consistent with the look command.
* I added the “drop” command, allowing players to drop items in their inventory into their current location, or drop items in a container item in their inventory into their current location.

# What we found out

* Some access methods (search, random reads) can be costly when applied to linked lists, as the list has to be traversed to get a result.
* The standard library includes singly-linked and doubly-linked lists (forward\_list and list respectively), which I originally assumed programmers would have to create manually if they wanted to use them.
* The standard library includes multimaps: a map variant that allows key-value pairs to have identical keys.
* If class A inherits class B and C and is stored as class B, casting it as class C won’t work; you have to cast it as class A to access class C members.

# Game Data Structures for Zorkish Adventures Inventory

### Inventory Requirements

A player’s inventory (or any container object generally) would have the following requirements:

* Items can be added or removed as necessary.
  + It needs to be resizable, as inventories could contain any number of items (unless the game has an item count; Zorkish Adventures is assumed to not limit the player’s inventory capacity).
  + Items’ removability should be independent of when they were added.
* Multiple identical copies of an object type can be stored if necessary.

### Data Structures

#### Array

Arrays are a basic data structure, providing storage for a specified number of elements of a single data type. Given that they are of a fixed size, they are unsuitable for inventories that aren’t assumed to be of a fixed size; this is true for both basic C++ arrays and the array class/struct.

#### Vectors

Vectors build upon the basic array to provide storage for items of a single type, but can add or remove more items as needed, including functionally identical items, and not being bound to a fixed size. As such, they would be usable for player inventories.

#### Linked List

Linked lists are a list where the “list” class points to an element (of a given type) in the list, which has a pointer to the next items in the list, and so on. (The “doubly linked” variation features pointers to both the next and the previous elements in the list.) Items can be added and removed as required, and functionally identical items can be included. As such, they would also be suitable for player inventories. However, the programmer would need to be aware of the costliness of some access methods when used on linked lists.

#### Queue

A queue is a list variant where all inserted items go to the back of the queue, and only the first item in the queue is removed. Inventories need every item to be retrievable at any time, making queues (including priority, circular and double-ended queues (“deques”)) ill-suited for serving as an inventory.

#### Stack

A stack is a list variant similar to a queue in that items get added and removed at specific points in the list. For a stack, they’re added and removed from the same end, meaning the most recently added is the first to be removed. Again, since not just any item can be removed at any time, stacks aren’t suited for being inventories either.

#### Map

A map stores elements as key-value pairs of specified data types, each unique key element allowing access to a specific value element. Size is not fixed, and key-value pairs can be added and removed as is necessary. Items stored in players’ inventories would need to be searchable by a name or ID field, making maps appropriate in that regard. However, functionally identical but distinct items with identical names or IDs (e.g. multiple “apple”s) would need to be storable in the same inventory. Map keys need to be unique, rendering maps unsuitable for player inventories unless every item in the game has a unique identifier (which is not the assumption for Zorkish Adventures).

#### Multimap

C++ offers multimaps, maps where keys of key-value pairs do not need to be unique. Pairs are ordered by keys; pairs with identical keys are ordered by their order of insertion. As multimaps allow multiple functionally identical items where maps do not, and as maps would otherwise be an appropriate data structure for a player’s inventory, multimaps would therefore be usable for such a purpose.

### Chosen Data Structure

In the above discussion of data structures, I listed vectors, linked lists and multimaps as usable data structures for player inventories. With linked lists, I noted some potential issues that vectors and multimaps do not face. As such, I will not be using linked lists.

Between vectors and multimaps, vectors would be the easier to implement, as I have already worked with those before. Given that multimaps store not just a value, but a class combining the key and the value, they would require more memory to store items (which would be minor for the scale of game being produced here). If I were to use vectors, to search for items in the inventory, I would need to include a method that loops through elements checking if their name or ID was that being searched for. However, such a method would be very straightforward and nothing that I haven’t done before. As such, I feel vectors would be the more suitable for me to use for this task.

### Bibliography

Clinton Woodward, *COS30031 Games Programming Lecture 4: Data Structures*, on Canvas.

<https://en.cppreference.com/w/cpp/container>

<https://en.cppreference.com/w/cpp/container/multimap>