# Task 09 - Game Data Structures Short Report

---

**Spike:** Task_09
**Title:** Game Data Structures
**Author:** Thomas Horsley, 103071494

**Aim:** To determine which data structure is most effective for use within an inventory system. The report must outline and critique four unique data-types based on ease of implementation, effectiveness and performance.

## Criteria

| | Very effective | Effective | Satisfactory | Non-satisfactory |
|---|---|---|---|---|
| **Speed of Relevant operations*** | Linked List | Dynamic Array, Vector | | Static Array |
| **Ease of Implementation** | | Vector | Dynamic Array, Linked List | Static Array |
| **Scalability** | Linked List** | | Dynamic Array, Vector | Static Array |

***** Relevant operations include traversing, accessing, reading, inserting and deleting elements

****** Custom classes allow custom solutions. Making additional features such as item stacks, quick switching and crafting easier to implement if the game allows.

### Result

It was  determined through research that a Linked List would be the best data structure to construct an inventory management system. This is due to it’s quick insertion and deletion operations and the scalability it provides to the developer.

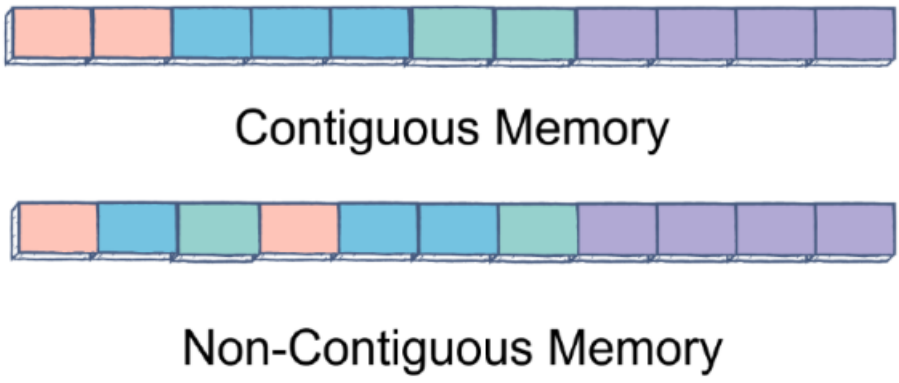
---

# Task 09 - Game Data Structures Short Report

## Data Structures

### Static & Dynamic Arrays

Arrays store elements contiguously in memory (one after the other see diagram), this allows for very fast random-access to elements as their addresses are easy to traverse and known before runtime. A Static Array is initialized on the stack and therefore it's elements are much faster to access than of a Dynamic Array. However, as a consequence of stack allocation, a Static Array must have a known, fixed size and it's elements are immutable. Additionally, the process of inserting, deleting and reshuffling elements within a Static Array is slower than a Dynamic Array as the entire set must be manipulated before being copied into a new array of relevant size (which also must be known at compile time).




On the contrary, Dynamic Arrays are initialized on the heap allowing for faster run-time modification to elements and resizing of the array. Dynamic Arrays are easier to implement, more efficient and hold less memory overhead if many insert / remove and reshuffle operations are required (such is as required for an effective inventory system). As a result, for the purposes of an inventory a Dynamic Array must be used over a Static Array if an array is to be used at all.

### Vectors

Vectors are essentially Dynamic Arrays with a bit of extra flavoring, they employ contiguous memory allocation on the heap similar to Dynamic Arrays. However, Vectors are template classes, this means that memory allocation and deallocation is encapsulated within the object and there's embedded functionality to aid the programmer with manipulating the data set contained within the Vector. Vectors are also easier to work with than Dynamic Arrays as

the size of the Vector doesn't have to be specified as a parameter as (because Vectors support random-access,) it can be determined in **O(1)** time. Additionally, a Vector has the ability to automatically resize itself upon growing greater than it's capacity and can be specified as a return type. Similar to Dynamic Arrays, Vectors are designed for reading and accessing elements of a set but struggle to insert elements anywhere but the tail of the vector. A vector will perform arbitrary insertion / deletion of elements in linear-time (**O(n)**) due to the required shifting of elements to maintain contiguity.

Vectors contain objects called iterators which allow for traversal through a Vector. The objects exist to further abstract the *'bare metal'* implementation of pointer manipulation and acts to increase the robustness of the Template. As a Vector belongs to the STL (Standard Template Library), it's heavily documented and will be compatible with other algorithms, functions and classes contained within the STL.

## Linked Lists

Linked lists are comprised of two class objects a Node and a Node Manager.  The Node holds an element of data and a pointer to the next Node with the final node of the list (the tail) referencing a nullptr. Nodes are stored non-contiguously on the Heap.

The Node Manager holds the functionality required to traverse the list and insert or delete elements. The Linked list can be singular (mono-directional traversal), double-ended (bi-directional traversal) or circular (go anywhere and do anything traversal) depending on implementation. A linked list does not support random-access and therefore must be traversed manually (starting at the head) through the Node pointers to find an arbitrary Node. Subsequently, the accessing and reading of elements is slower than that of a Vector or Array. Even so, a Linked List allows easier random insertion and deletion operations as only the Nodes reference to the next Node must be modified rather than copied and/or shifted.  It's primarily due to the ease of arbitrary insertion / deletion which makes the Linked List the most appropriate data structure for an inventory system.