# PART 1 MEMORY MANAGER

Data Structures and Algorithms

David Gray s2947315
david.gray2@griffithuni.edu.au

# Table of Contents

## Problem Statement

The goal of Part 1 of this assignment was to write a dynamic memory manager class that is used to manage memory from an array. The memory manager class had to consist of multiple functions that consist of: creating a memory array of any size; allowing the allocation and reallocation of memory within this array; to free memory stored within the array; to return the amount of free memory within the array; to compact the memory within the array; print out the array contents as hexadecimal values and string values; to copy the memory manager object into another of the same type; and a function to delete the object.

## User Requirements

There are no user requirements for this program apart from running the executable.

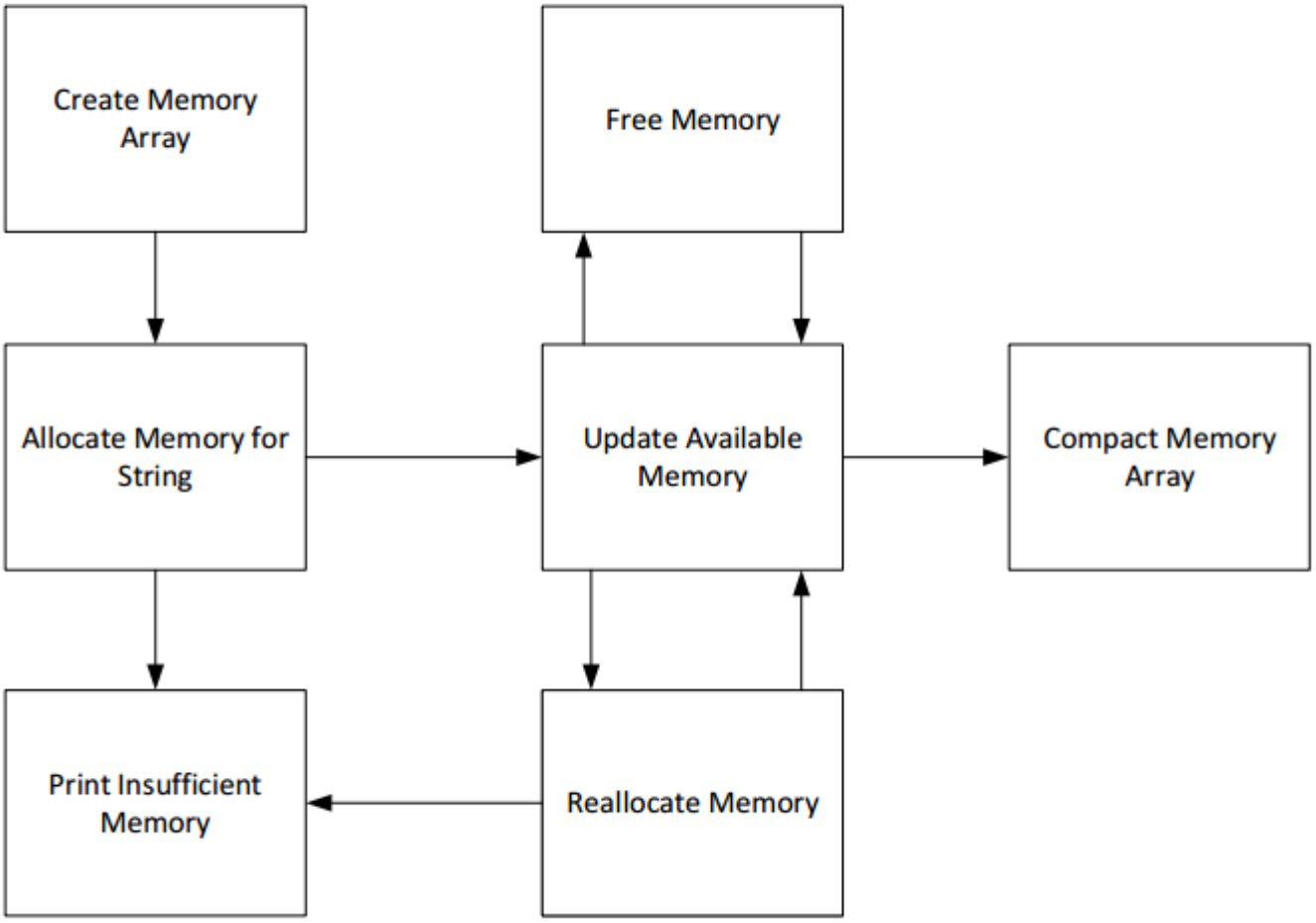
## Software Requirements

The following outlines the software requirements for the program:

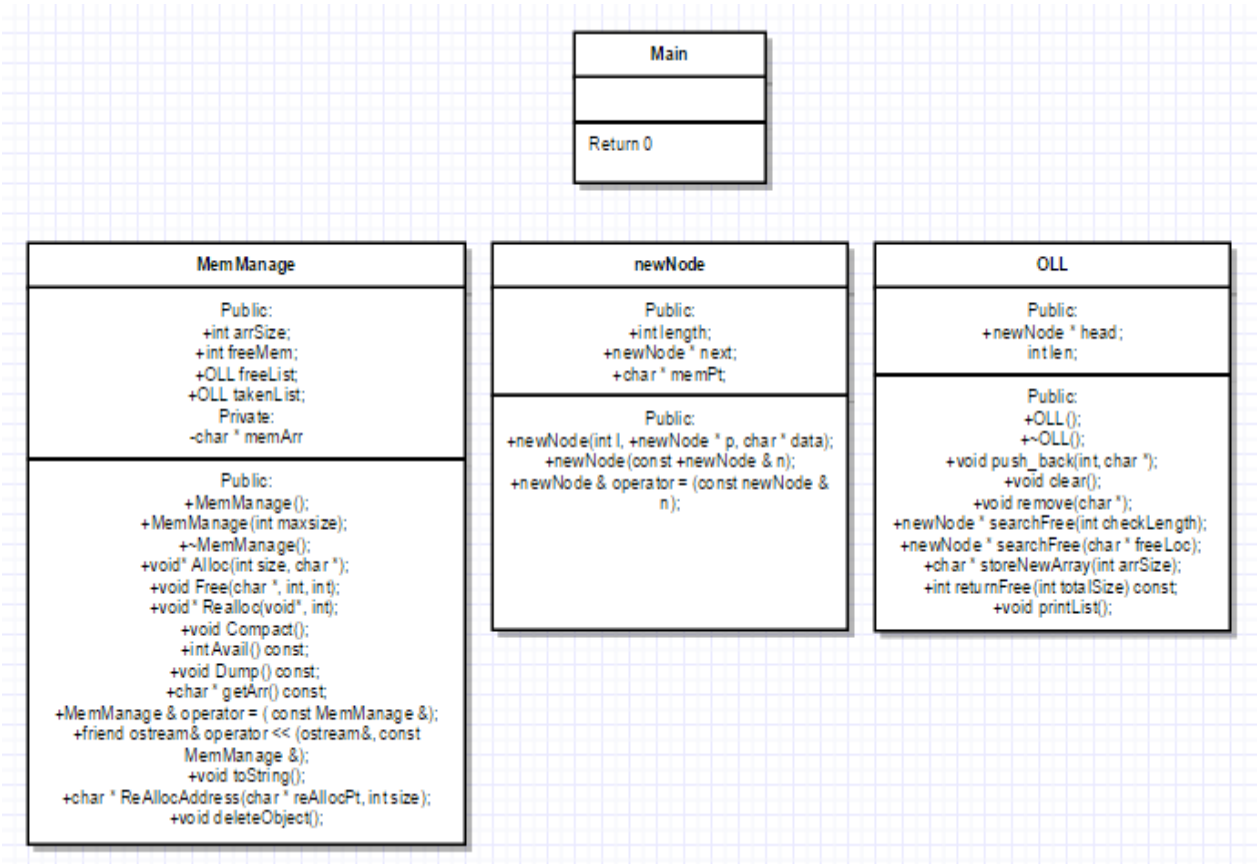
1. The program allows a memory array of any size to be set.
2. The program shall display the contents stored within the memory array created in both hexadecimal values and string values.
3. The program shall display an error message if memory cannot be allocated, reallocated or if there is insufficient memory available.
4. The program shall display the amount of free memory available.
5. The memory array that is allocated is set to private so the user cannot directly access it.
6. The program shall use linked lists to keep track of the location and size of free and taken memory.
7. The program's memory allocation function will try to reuse the smallest memory fragments it can before using memory from larger blocks.
8. The program's memory reallocation function will first try to enlarge the size of the current allocated memory if there is enough free space and if there is not enough free space it will free its current memory and reallocate it somewhere else within the memory array.
9. The program shall compact the memory array when asked.
10. The program shall be able to make a deep copy of the Memory Manage class by overloading the = operator.

## Software Design

**High Level Design – Logical Block Diagram**



## Structure Chart



## All main functions in the software.

**MemManage Constructor:** takes an integer as a parameter to which the maximum size of the memory array is stored as. A free memory linked list and a taken memory linked list is initialized and all of the memory values within the memory array are set to 0.

**MemManage Destructor:** clears both of the linked lists and deletes the memory array.

**MemManage Alloc:** takes in an integer and a string as parameters and tries to find a location to store this data within the memory array. The function first tries to look to see if there is enough free memory. If there is not the function returns null. If there is, it first loops through the free linked list to see if there is a smaller chunk of memory that can fit there data and stores it there. If there is not, then the data is appended to the end of the array where there is free space.

**MemManage Free:** takes in a memory pointer, an integer to set the new values, and an integer length as parameters. The function finds the piece of memory within the memory array to where the pointer points to and then loops through the length size and replaces the data stored in those memory locations to the integer value that was passed through.

**MemManage Realloc:** takes in a memory pointer and an integer that represents length as parameters. The function calls the linked list reallocation address function within the MemManage class. The reallocation address function first finds the node within the taken list that matches the memory pointer that is passed through. From that it checks to see if there is enough space after the memory location to increase its current length to the new integer length that is passed through. If there is, the free memory list and taken memory list are updated. If there is not enough space to increase its size at the current location, the memory is freed and then a new place within the memory array is found to store the data after calling the allocation function on the data.

**MemManage Dump and << operator:** prints out the contents of the memory array byte by byte as consecutive rows of 16 hexademical values with a single space between each value.

**MemManage toString:** prints out the contents of the memory array similar to Dump and << but it returns the memroy content as a string.

**MemManage Compact:** defragments the memory array within the MemManage class by creating a second array and looping through the taken memory list and taking the contents of each node in the list and stores that within the new array, removing all spaces and packing the data. The old array is then deleted and the new array created becomes the memory array.

**MemManage = operator:** passes through a MemManage object as its parameter. Takes the MemManage object that is passed through and copies its data into the MemManage object that wants to equal it. To make a deep copy of the memory arrays from the new object and the passed through object, a second memory array is created to copy the data values and pointers stored in the passed through objects array so that the new object that is going to store the passed through objects details does not point to the same memory locations for their memory arrays. Once this is complete the new objects original memory array is deleted and replaced byt the second memory array that was just created and the free and taken lists are updated accordingly.

## All data structures in the software.

A linked list is the main data structure that is used within this program.

It is used to keep track and store the data locations of free and taken memory within the memory array in the memory manager class. The linked lists are traversed in order compare values passed through to a function to the values within the linked lists so that the functions are able to allocate memory using the free memory list and deallocate/free memory using the taken memory list.

The linked list utilises a node struct to store memory pointers and lengths of locations within the memory array.

All of the functions within the Memory Manager class makes use of the linked lists in order to complete the allocation/reallocation of memory, the printing of memory and the freeing of memory.

The secondary data structure used within this program is an array that is used to point to, store and locate the data that is being managed from the main MemManage class.

## Requirement Acceptance Tests

| Software Requirement No | Test | Implemented (Full /Partial/ None) | Test Results (Pass/ Fail) | Comments (for partial implementation or failed test results) |
|---|---|---|---|---|
| 1 | The memory array allocated by the constructor must be private | Full | Pass | |
| 2 | Use a linked list to keep track of the location and size of free memory fragments. You must write your own, you can not use std::list | Full | Pass | |
| 3 | Avail() returns the total amount of free memory in all free fragments | Full | Pass | |
| 4 | Realloc() first attempts to enlarge the size allocated at the location given by the pointer and if that is not possible makes a new allocation and frees the existing. It returns a pointer to the allocation or NULL if reallocation was not possible | Partial | Fail | Does full functionality of reallocating memory however it does not update the free memory sizes properly for the Avail function. |
| 5 | Dump() and operator << prints the contents of the memory out byte by byte as consecutive rows of 16 hexadecimal values with a single space between each value. Note that that hex values are always printed out using 2 characters such that '3' => '03' to do this you will need to set up your printing function with the correct flags (or manipulators for cout) | Full | Pass | |
| 6 | char* toString() – similar to dump but returns memory content as a string. | Full | Pass | |
| 7 | Compact() essentially defragments your memory resulting in having a single free fragment. | Full | Pass | |
| 8 | Operator= makes a deep copy of a MemManage object | Partial | Pass | Main functionaly of = works however if you try to use Avail() afterwards it will not return the correct free memory size as the taken list is not updated correctly. |

## Detailed Software Testing

| Test | Expected Results | Actual Results |
|---|---|---|
| **Memory Allocation** | | |
| Tested memory allocation of “zero” | Tested allocating memory of size 5 with string“zero”.<br>Expected result z e r o for toString and 7a 65 72 6f | As expected |
| Tested Reallocation for “zero” increasing size from 5 to 7 whilst there is another element directly after “zero” | Program should detect that is not enough space to reallocate at current position and should instead add it to the end of the memory array. | As expected |
| Tested Compact by printing out the dump before and after the compact function is called and all the elements should be the same however with no spaces. | The Dump should be the same before and after the compact however due to how I programmed my linked lists the memory will be put back in in size order. | As expected |
| Tested = operator by making a second copy of MemManage object and making it equal a previously created object. | Dump should return same values for both objects however the first memory pointer should not be the same. | As expected |
| Tested Alloc to see if it returns insufficient space if the data being allocated is greater than the data available. | Should show “Insufficient Space” when trying to allocate a size of 101 when memory array is of size 100 | As expected |
| Tested Alloc to see if it allocated “one” where “two” was previously freed and they both have the same size. | “one” should be where “two” originally was. | As expected |

## User Instructions

If running the program using the Visual Studio 2010, open the project and then run the program in debug mode.

If using the executable, double click on the file and let it run.