Memory-Manager in C++

Memory Manager Justification

- Two types of memory - stack and heap
- The stack is managed by the compiler
- We'll handle the Heap
- why?
    - OS is responsible for the Heap memory management
    - program -> kernel switch -> OS(kernel mode) -> allocates the requested memory -> kernel switch -> program
    - Expensive!
    - No functionality to handle memory fragmentation

Why can't we use the in-built functions in C like malloc and free or new and delete in C++?

- Functions such as malloc and new are general-purpose memory allocators. Your code may be single-threaded, but the malloc function it is linked to can handle multithreaded paradigms just as well. It is this extra functionality that degrades the performance of these routines.
- In their turn, malloc and new make calls to the operating system kernel requesting memory, while free and delete make requests to release memory. This means that the operating system has to switch between the user-space code and kernel code every time a request for memory is made. Programs making repeated calls to malloc or new eventually run slowly because of the repeated context switching.
- The memory that is allocated in a program and subsequently not needed is often unintentionally left undeleted, and C/C++ doesn't provide for automatic garbage collection. This causes the memory footprint of the program to increase. In the case of really big programs, the performance takes a severe hit because available memory becomes increasingly scarce and hard-disk accesses are time intensive.

What are memory leaks?

Memory leaks occur when a new memory is allocated dynamically and never deallocated. In C programs, new memory is allocated by the malloc or calloc functions, and deallocated by the free function. In C++, new memory is usually allocated by the new operator and deallocated by the delete or the delete [] operator. The problem with memory leaks is that they accumulate over time and, if left unchecked, may cripple or even crash a program.

There are many tools available which can detect memory leaks like Valgrind, gdb. Firefox and Mysql have used valgrind in their application.

What is memory fragmentation?

Memory fragmentation is when most of your memory is allocated in a large number of non-contiguous blocks, or chunks - leaving a good percentage of your total memory unallocated, but unusable for most typical scenarios. This results in out of memory exceptions, or allocation errors (i.e. malloc returns null).

How can it be a problem?

When memory is heavily fragmented, memory allocations will likely take longer because the memory allocator has to do more work to find a suitable space for the new object. If in turn, you have many memory allocations (which you probably do since you ended up with memory fragmentation) the allocation time may even cause noticeable delays.

How should we deal with it?

Use a good algorithm for allocating memory. Instead of allocating memory for a lot of small objects, pre-allocate memory for a contiguous array of those smaller objects. Sometimes being a little wasteful when allocating memory can go along way for performance and may save you the trouble of having to deal with memory fragmentation.

Are there any problems using pointers not smart pointers in C++?

A pointer is a variable which stores the address of some variable. Yes, there are many problems that we might encounter while using raw pointers. For example, the scope of raw pointers is local to the function which means it will be destroyed after the function returns that eventually results in memory leaks.
Dangling Pointers: A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

To tackle these problems, we can use smart pointers. Link

Design Goals

- **Speed**
  The memory manager must be faster than the compiler-provided allocators. Repeated allocations and deallocations should not slow down the code. If possible, the memory manager should be optimized for handling certain allocation patterns that occur frequently in the code.

- **Robustness**
  The memory manager must return all the memory it requested to the system before the program terminates. That is, there should be no memory leaks. It should also be able to

handle erroneous cases (for example, requesting too large a size) and bail out gracefully.

Useful strategies for creating a memory manager

- **Request large memory chunks**.
  One of the most popular memory management strategies is to request for large memory chunks during program startup and then intermittently during code execution. Memory allocation requests for individual data structures are carved out from these chunks. This results in far fewer system calls and boosts the performance time.
- **Optimize for common request sizes.**
  In any program, certain specific request sizes are more common than others. Your memory manager will do well if it's optimized to handle these requests better.
- **Pool deleted memory in containers.**
  Deleted memory during program execution should be pooled in containers. Further requests from memory should then be served from these containers. If a call fails, memory access should be delegated to anyone of the large chunks allocated during program start. While memory management is primarily meant to speed up program execution and prevent memory leaks, this technique can potentially result in a lower memory footprint of the program because deleted memory is being reused. Yet another reason to write your own memory allocator!

Memory Manager Features

- Minimal use of kernel swapping
- If it is required then it will be strictly controlled
- Keep the use of standard function calls (new, new[ ], delete, delete[ ])
- Use smart pointers if necessary
- Keep headers small (as small as possible)
- Keep scanning to minimum
- Auto-defrag the memory blocks

Memory Manager Features

- Memory Pools - allocated by the OS at the controlled points
    - Having memory pools will allow us to allocate a big chunk of memory and thereafter dynamically allocate memory to the blocks as per request.
    - There are many advantages of using pools like Memory pools allow memory allocation with constant execution time. The memory release for thousands of objects in a pool is just one operation, not one by one if malloc is used to allocate memory for each object.
    - Important links: Wiki Extra_Info
- Memory Blocks -
    - Name/ID
    - Length of the block
    - Reference counter
    - Flag - this block is defragged (i.e. in the most optimal position in the block)
- Auto-defragmentation functions:
    - future scope

Classes

- **Memory Manager Class**
    - Handles overall memory management
    - List of pools
    - allocate newer pools
    - can query about a specific pool
- **Memory Pool Class**
    - Handles the individual memory pool
    - Header
    - List of Blocks
    - Queries(size, the status of fragmentation, memory left)
    - add blocks
    - delete blocks
- **Memory Block Class**
    - Handles individual memory block
    - Header
    - Data

Resources:
https://csrl.cse.unt.edu/kavi/Research/southeastcon.pdf
https://www.cs.tufts.edu/~nr/cs257/archive/paul-wilson/fragmentation.pdf