Sabanci University

Faculty of engineering and Natural Sciences

CS 300 Data Structures

Assigned: **Nov 6, 2020** Due: **Nov 16, 2020 @ 11:55pm**

Please note:

SOLUTIONS HAVE TO BE YOUR OWN. NO COLLABORATION OR COOPERATION AMONG STUDENTS IS PERMITTED.

10% PENALTY WILL BE INCURRED FOR EACH DAY OF OVERTIME. SUBMISSIONS THAT ARE LATE MORE THAN 3 DAYS WILL NOT GET ANY CREDITS.

SUBMISSIONS WILL BE MADE TO THE SUCOURSE SERVER. NO OTHER METHOD OF SUBMISSION WILL BE ACCEPTED.

Introduction

Memory allocated in a C++ is separated into two regions of memory, the stack and the heap. The heap is the part of memory that contains any variable you create using the "`new`" keyword. In this homework, you are going to simulate your own heap memory region. You will allocate memory (new), and deallocate memory (delete) from this region. You will also explore how different algorithms handle re-allocation, or the process of reusing memory that has been freed.

Heaps can be implemented in many different ways. The most common method is using **a linked list representation** (this is the method used in the `g++` compiler in linux, in fact). Of course, different implementations on different compilers and OSes will have differences in the representations and algorithms, etc. but the idea is similar; use a linked list to control the memory allocations.

Heaps give the programmer the ability to allocate blocks of memory dynamically during the runtime of the program. However, as you will see soon, they suffer from the problem of **fragmentation;** after a while, the heap memory ends up becoming split into a lot of small non-contiguous regions. This could lead to degrading the speed of memory allocation and might even lead to running out of memory for smaller devices (arduinos, for example.)

Implementation

You are going to simulate a heap implementation that uses a **doubly linked list** data structure to manage its allocations. When implementing your heap, you will assume that you have an imaginary region of memory called `Img_heap` that starts at the address `0x0` and that has a capacity of 512 bytes. All the allocations and deallocations will be made from this space. Your `My_heap` linked list will keep track of **which areas of `Img_heap` are full and which are empty.** Whenever a new allocation happens, some part of `Img_heap` will be reserved, and when a deallocation happens, a part of it will be freed. All of these operations must be reflected on our `My_heap` linked list.

It should be emphasized that `Img_heap` does not actually exist; you will simulate it with your `My_heap` linked list. Your linked list must always reflect the status of `Img_heap` through the addresses and sizes its nodes contain.

The nodes in the `My_heap` linked list are `memory_block` structs. Each one of these structs is responsible for an area of the memory region `Img_heap`**.** The struct is show below:

```
struct memory_block{
      memory_block* right; // node to the right
      memory_block* left;  // node to the left
      bool used; // if this memory block is reserved or not
      int size;  // the number of bytes reserved in Img_heap
      int starting_address; // the starting address in Img_heap
                            // of the memory reserved by this block
};
```

The area that a memory block `B` is responsible for, is the area within the bytes `[B->starting_address, B->starting_address+B->size)`. If the area that a block `B` is responsible for, is allocated, the `used` boolean must be set to `true`. Otherwise, it is `false`. The right and left pointers point to the next and previous nodes in `My_heap`.

The following is the data structure of `My_heap`. You will need to implement all the functions in this class. The explanations of these functions and data members are shown in the following subsections.

```
class My_heap{
private:
      memory_block* heap_begin;
      memory_block* blk;
      int used_bytes;
      const int MAX_CAPACITY = 512; // this must remain constant;
                                    // you don't need to change it
public:
```

```
        My_heap();
        ~My_heap();
        memory_block* bump_allocate(int num_bytes);
        memory_block* first_fit_allocate(int num_bytes);
        memory_block* best_fit_allocate(int num_bytes);
        memory_block* first_fit_split_allocate(int num_bytes);
        void deallocate(memory_block* block_address);
        void print_heap();
        float get_fragmantation();
};
```

Data members

`heap_begin`: a pointer at the **first** `memory_block` in `My_heap`.
`blk`: a pointer at the **last** `memory_block` in `My_heap`.
`used_bytes`: the total number of allocated bytes in `Img_heap`**.**
`MAX_CAPACITY`: the total number of bytes available in the memory region `Img_heap`. This is a constant value that will not change throughout the execution of the program.

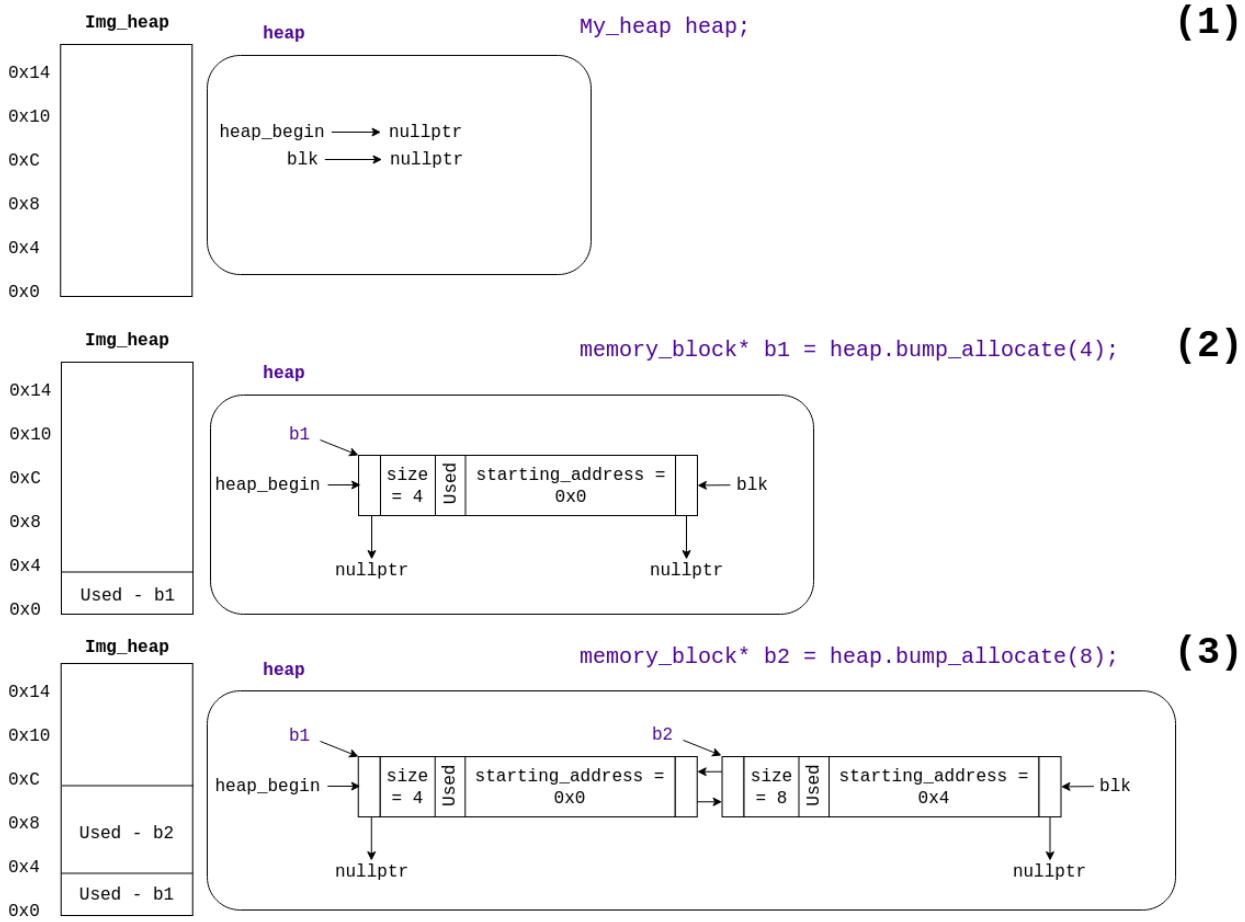
Member functions

`My_heap()`

Creates an empty `My_heap`.

`memory_block * bump_allocate(int num_bytes)`

This function will simply allocate a new region in `Img_heap` regardless of the state of the other blocks. In other words, every time this function is called, a new `memory_block` is added at the end of `My_heap`, and that block will be responsible for a new region at the end of `Img_heap`. If there is not enough memory at the end of the heap to allocate `num_bytes`, the function should return `nullptr` since no new allocations can be made in Img_heap. Below is a demonstration of this function. The figures show what the execution of the purple lines of code above them will do.



(1) When an instance of `My_heap` is created, it starts out as an empty linked list. This represents an empty `Img_heap`.

(2) when we call the function `bump_allocate(4)`, we are going to allocate an area of 4 bytes in `Img_heap`. This is represented by creating a new `memory_block` in the linked list which contains the starting address of the created memory region in `Img_heap (0x0)`, the size of the allocated memory region (4 bytes), a boolean indicating that the block is not free (`used = true`), and pointers to the next and previous nodes in `My_heap`. This function will return a pointer to the created block in `My_heap` so that our users can access the memory region.

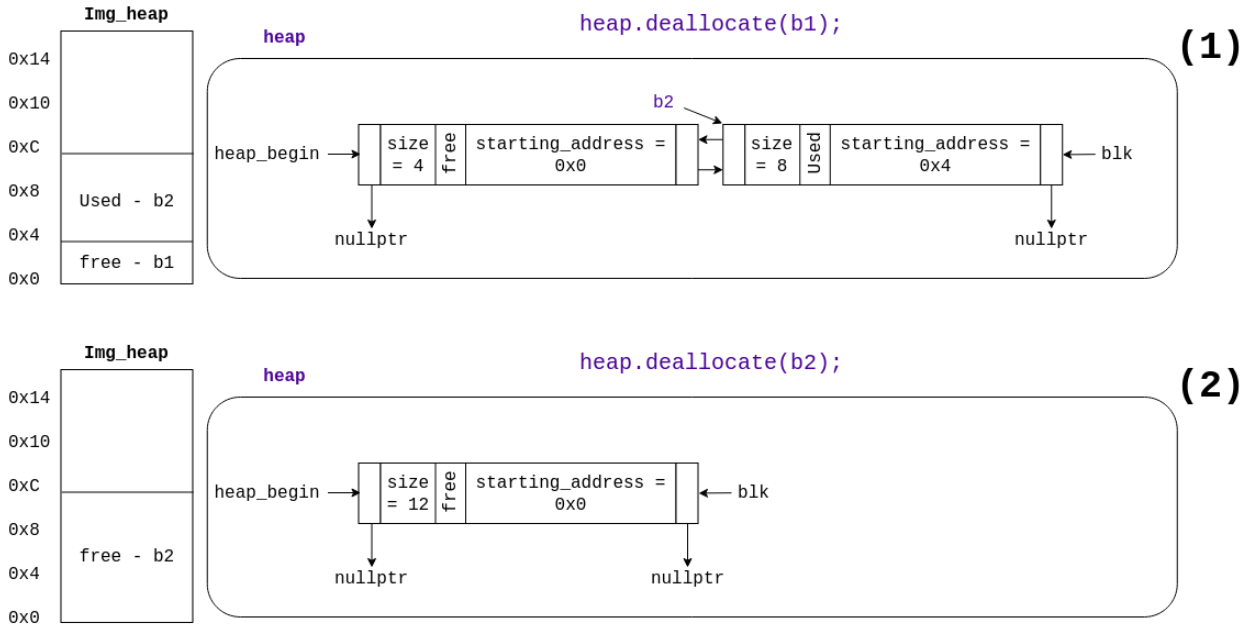
(3) Calling `bump_allocate(8)` reserves 8 more bytes of `Img_heap`. This in turn creates a second `memory_block` that's responsible for a new region of the heap. The block encapsulates the starting address and size of the new memory region, and that this memory block is being used (not free). Notice that the new node was created **to the right** of the first one, and that it is pointed at by the pointer **`blk`,** while the first node is pointed at by **`heap_begin`.**

`void deallocate(memory_block* to_delete)`

This function will deallocate the memory that block `to_delete` is responsible for on `Img_heap`. Also, a call to this function can merge free blocks. Deallocation from `my_heap` will do two things

a. It will set the status of the block pointed at by `to_delete` to be free, i.e., `to_delete->used = false;`
b. If the block is surrounded by other free blocks, these blocks will merge into a single block with the combined sizes of all the blocks and status `used = false`. Please note that this will result in **deleting nodes from `My_heap`.**

The following figure demonstrates this process.



(1) When we deallocate block `b1`, we are going to **set the status of the block pointed at by `b1` to be free.** This signals that this memory region is now free to be used by other allocations. Also, we will check to see if any of the neighboring nodes of `b1` are also free, if they are, **we will merge these blocks to create a single bigger free block.** In this case, there were no free blocks around `b1` so we don't do any merging.

(2) In the second call, when we deallocate `b2`, we find that **the node to the left of `b2` is free, so we merge the two blocks into a single free block.**

A note about dangling pointers: notice that in both these deallocation calls, we did not set the values of `b1` or `b2` to nullptr. This means that these pointers are now dangling pointers! **Both of these pointers are pointing at memory blocks in the heap that have been deallocated.**

`memory_block * first_fit_allocate(int num_bytes)`

This allocation algorithm does not always create a new memory block for every allocation. Instead, this algorithm will search, starting from the beginning of `My_heap` (starting at `heap_begin`), for the first `memory_block` that is not allocated into (`used == false`) and that can fit at least `num_bytes` bytes. If it finds such a block, it will set its status to `used = true` and will return a pointer to it, i.e it will allocate that block of memory. If such a block is not found, the algorithm will allocate a new region at the end of `Img_heap` by creating a new block in `My_heap` and will return a pointer to this new block. If there is not enough memory left in `Img_heap`, the function should return `nullptr`.

`memory_block * best_fit_allocate(int num_bytes)`

Similar to `first_fit_allocate`, this algorithm will search `My_heap` (starting from `heap_begin`) for a free block that can fit the required bytes. However, this algorithm does not use the *first* block it finds. Instead, it will **use the block that will lead to the smallest amount of wasted memory.** In other words, if `num_bytes == 10` and we had two free blocks in `My_heap`, one with `size = 15` and one with `size = 20`, we will choose the block with `size = 15` since it will result in wasting 5 bytes only instead of 10 bytes. If this algorithm does not find any free blocks that can fit the data, it follows the `bump_allocate` routine of adding to the end of `Img_heap`. In the case of the heap not having enough free space, do not make any allocations and return `nullptr`.

`memory_block * first_fit_split_allocate(int num_bytes)`

This algorithm is very similar to `first_fit_allocate`, except that, instead of wasting memory, it will split free blocks into smaller blocks and allocate the exact amount of memory it requires. More precisely, this algorithm begins a search from `heap_begin` and searches for a free block with `size >= num_bytes`. Once it finds it, if the block's `size == num_bytes` it will simply set its status `used = true` and return its address. However, if `size > num_bytes`, it will split the block into two blocks, a used block with `size = num_bytes`, and a free block with `size = size-num_bytes`.

For example, if `num_bytes = 10` and during your search you find a block with `size = 15`, you must split this block into two blocks, the first must have `size = 10` and `used = true`, and the second will have `size = 5` and be `used = false`. You will return the **first block**.

Similarly to the other allocation strategies, if there is not enough space in the heap to contain `num_bytes`, no allocations should be made and `nullptr` is returned.

`float get_fragmantation()`

This function will calculate the fragmentation percentage of your heap and will return that value as a float. We can calculate the fragmentation of a heap using this equation:

`fragmentation% = (free_memory - biggest_free_block)/free_memory * 100%`

where,
`free_memory` = the total memory that isn't allocated in your heap
`biggest_free_block` = the biggest contiguous block in your memory that is free.
For example, looking at the second figure, in the upper image, we can see that:
`free_memory = 512-8 = 504`
`biggest_free_block = 500 (which is the size of the area of memory *after* the second block)`
`Fragmentation = 0.79%`

This value is excellent since it means there aren't too many small free memory blocks scattered around the heap.

`void print_heap()`

Prints statistics about the heap, as well as information about **all** the blocks in the heap. Any call to `print_heap` will result in the following lines, where the red strings represent variables that will change depending on the heap status:

```
Maximum capacity of heap: 512B
Currently used memory (B): O1
Total memory blocks: O2
Total used memory blocks: O3
Total free memory blocks: O4
Fragmentation: O5%
------------------------------
```

Where:
`O1`: The total allocated memory in the heap
`O2`: The number of memory_blocks in My_heap
`O3`: The number of memory_blocks which are used
`O4`: The number of memory_blocks which are free
`O5`: Fragmentation of the heap, calculated using the function `get_fragmentation()` defined above.

Following that, a single line will be printed for each memory_block in My_heap, starting with the node pointed at by heap_begin, in the following format:

```
Block O6\t\tUsed: O7\tSize (B): O8\tStarting Address: 0xO9\n
```

Where:
`O6`: The index of the block (starting from 0), i.e it is the order of this block's occurrence from the beginning of the heap.
`O7`: This is going to be the string "`True`" if the block is occupied, and "`False`" if it's free.
`O8`: The size of the memory region this block is responsible for in `Img_heap` in bytes.
`O9`The starting address of the memory region this block is responsible for written as a **hexadecimal** number.
Please note that `\t` is the tab character.

After the lines for all the nodes have been printed, print out the following two lines:
```
------------------------------
------------------------------
```

An example of what the output should look like is shown in the test run at the end of this document.

`~My_heap()`

The destructor must safely deallocate all the memory of the linked list. But before that, it should **print the memory leakage in the heap.** In other words, it will print the total number of bytes which are allocated at the time of the destruction of `My_heap()`. The constructor must print the following line when it is executed.

```
At destruction, the heap had a memory leak of X bytes.
```

Where:
`X`: The number of leaked bytes as an integer.

Submission

In this homework, you will submit two files, `my_heap.h` and `my_heap.cpp`. `my_heap.h` must contain the definitions of the class `My_heap` and the structure `memory_block`. Please note that you are **not allowed** to make any changes to memory_block's *data members*, however, you can add additional functions to the struct (a constructor, helper functions, etc.). Similarly, for the class `My_heap`, you cannot change the data members of the class, and you must implement **all** the functions in the definition we have given you in this document without any changes to the parameters, return types, or function names . However, you may add *additional* helper functions if you find it necessary.

The file `my_heap.cpp` will contain the implementations of all the functions defined in `my_heap.h`.

DO NOT submit a main program. **YOUR .cpp FILE SHOULD NOT HAVE FUNCTION `main(...)` EITHER**. During the grading process, we will compile your class files and link them with our own (hidden) main programs to evaluate your outputs.

You *should,* however, write your own main program to test your class functionality with it, but you should not submit it.

Your final submission should be a zip file named:

`suname_suid_cs300_hw1.zip`
`(amroa_26001_cs300_hw1.zip)`

It will contain exactly two files:

`my_heap.h`
`my_heap.cpp`

And no additional files/folders.

Sample Run

main.cpp

```
#include "my_heap.h"


int main(){
	My_heap heap;

	heap.print_heap();

	memory_block* block = heap.bump_allocate(8);
	memory_block* block1 = heap.bump_allocate(12);
	memory_block* block2 = heap.bump_allocate(9);
	memory_block* block3 = heap.bump_allocate(23);

	heap.print_heap();

	heap.deallocate(block1);
	heap.print_heap();

	heap.deallocate(block2);
	heap.print_heap();
	return 0;
}
```

Console output:

```
Maximum capacity of heap: 512B
Currently used memory (B): 0
Total memory blocks: 0
Total used memory blocks: 0
Total free memory blocks: 0
Fragmentation: 0%
------------------------------
------------------------------
------------------------------
Maximum capacity of heap: 512B
Currently used memory (B): 52
Total memory blocks: 4
Total used memory blocks: 4
Total free memory blocks: 0
Fragmentation: 0%
------------------------------
Block 0     Used: True  Size (B): 8  Starting address: 0x0
Block 1     Used: True  Size (B): 12 Starting address: 0x8
Block 2     Used: True  Size (B): 9  Starting address: 0x14
Block 3     Used: True  Size (B): 23 Starting address: 0x1d
------------------------------
------------------------------
Maximum capacity of heap: 512B
Currently used memory (B): 40
Total memory blocks: 4
Total used memory blocks: 3
Total free memory blocks: 1
Fragmentation: 2.54237%
------------------------------
Block 0     Used: True  Size (B): 8  Starting address: 0x0
Block 1     Used: False Size (B): 12 Starting address: 0x8
Block 2     Used: True  Size (B): 9  Starting address: 0x14
Block 3     Used: True  Size (B): 23 Starting address: 0x1d
------------------------------
------------------------------
Maximum capacity of heap: 512B
Currently used memory (B): 31
Total memory blocks: 3
Total used memory blocks: 2
Total free memory blocks: 1
Fragmentation: 4.3659%
------------------------------
Block 0     Used: True  Size (B): 8  Starting address: 0x0
Block 1     Used: False Size (B): 21 Starting address: 0x8
Block 2     Used: True  Size (B): 23 Starting address: 0x1d
------------------------------
------------------------------
At destruction, the heap had a memory leak of 31 bytes
```