Assignment: Custom Memory Allocation

> ***If your submission generates the correct output without implementing the assignment exactly as described below (redefining global `operator new` and `operator delete` functions; defining `operator new` and `operator delete` member functions; defining a custom allocator class), then don't submit to the auto-grader!!! What happens if you decide to ignore this warning? A human grader will regrade your submission to zero points. Second, subverting assessments by passing something off as something else is a violation of the academic integrity policy and appropriate remedies as cited in that policy will be enforced.***

No standard library

Learning Outcomes

This assignment will provide you with the knowledge and practice required to develop and implement software involving:

1. Practice implementation of own `operator new` and `operator delete`.
2. Practice implementation of a custom memory allocator class.
3. Get familiar with use of placement `new` expressions and explicit destructor calls.
4. Practice bitwise operations.
5. Get exposed to C++11 concepts including function declarations using trailing return type and `decltype` type specifier.

Task

Consider a hypothetical 3D graphics engine that defines a structure called `vector` and a union called `vertex`. You'll augment file `allocator.hpp` with an implementation of the following:

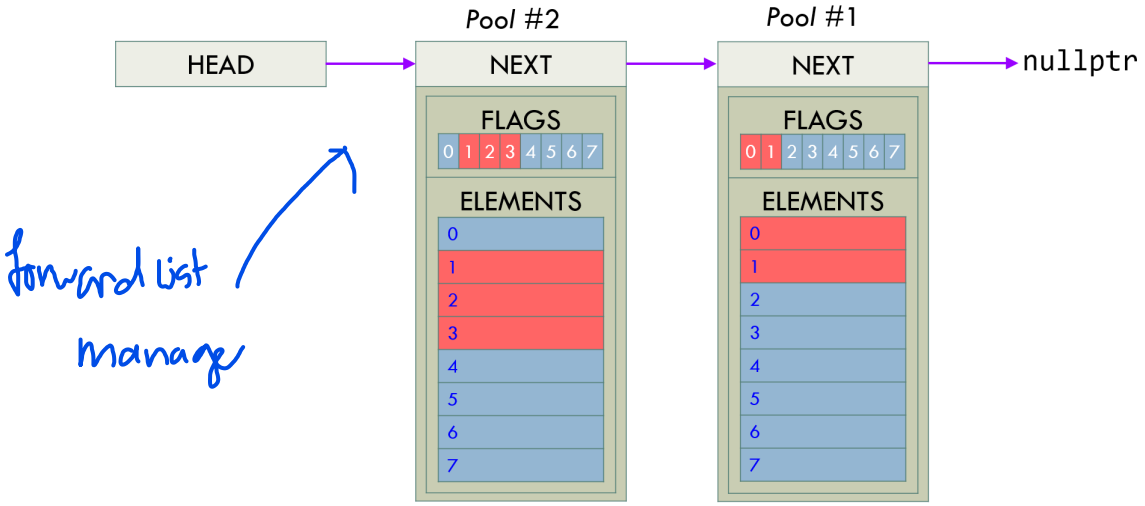
1. Redefine global `operator new` and `operator delete` functions that generate console output as per the sample (driver source file and corresponding output text file) and use standard library functions `malloc` and `free` to manage the memory.
2. Define in-class local `operator new` and `operator delete` functions that generate console output as per the sample (driver source file and corresponding output text file) and that use the redefined global `operator new` and `operator delete` functions to manage the memory.
3. A custom allocator class template for performing memory management inside `std::vector` instances. The allocator's behavior is described below.

member functions

On the outside, the `allocator<TDataType, TFlags>` class template must behave in a way compatible with `std::allocator<T>` (where `T` corresponds to our `TDataType` template parameter). Internally, the memory allocation process must reflect the following policy:

1. The allocator stores the allocated memory in a singly-linked list `std::forward_list` of memory pools. Each pool contains a block of memory for storing up to `N` elements of `TDataType`, and a bit mask of type `TFlags`, where `N` is exactly the number of bits in a bit mask. For example, on a platform where `unsigned short int` occupies 16 bits, `allocator<MyClass, unsigned short int>` can store up to 16 objects of a type `MyClass` in each pool.

2. An i^{th} bit in the bit mask is set if and only if the element at index `i` in the pool is deemed allocated, and the bit is not set if the element is deemed unused and available for future allocation. **Take note** that you are not allowed to use `std::bitset<N>` to store bit mask in a pool; a pool should store bit mask in an object of type `TFlags` directly.
3. The singly-linked list starts empty. When the program requests the allocator to acquire `count` elements (1 or more) of type `TDataType`, the allocator must find the first pool with `count` continuous bits that have not been set, mark them as set, and return the address of the element corresponding to the index of the first bit found. If there are no pools with count continuous bits, a new pool can be added at the beginning of the singly-linked list, and the allocation can happen in this block.
4. If the program requests the allocator to release `count` elements at a certain address, the allocator must find the pool with an element at that address, get its index, and mark bits of `count` consecutive elements from that index as unused. Lastly, if there are no elements used in a memory pool after deallocation, the pool should be removed. Research the interface for `std::forward_list<T>` to determine the appropriate function to remove the pool.
5. The following diagram illustrates a sample scenario, where a pool contains a block with 8 elements, which is possible if the `TFlags` bit mask type uses a type that consists of 8 bits. This allocation layout could be the result of the following sequence of allocations and deallocations:
    1. The list starts empty.
    2. Allocate 2 elements – the allocator creates the first pool (Pool $\#1$).
    3. Allocate 6 elements – the allocator fills out the rest of the pool (Pool $\#1$).
    4. Allocate 1 element – as the right pool is full, the allocator inserts a new pool (Pool $\#2$).
    5. Allocate 3 elements – the allocator fills out subsequent 3 elements (Pool $\#2$).
    6. Deallocate elements allocated in step 3 causing 6 elements from the pool on the right to be marked as unused (Pool $\#1$).
    7. Deallocate elements allocated in step 4 causing 1 element from the pool on the left to be marked as unused (Pool $\#2$).



6. If the program requests the allocator to acquire `count` elements of type `TDataType`, where `count` is greater than the number of elements in a single pool, `std::bad_alloc` should be thrown.
7. If the program requests the allocator to deallocate memory at the address that does not belong to any of the pools, `std::bad_alloc` should be thrown.
8. **Note** that the allocator is not responsible for calling the placement `new` expression and the explicit destructor.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions. **Valgrind** and documentation using **Doxygen** is required.

Submission files

An incomplete `allocator.hpp` is provided as part of the interface. You will augment this file with your solution to the above task and upload it as your submission.

Compiling, linking, and testing

A driver file `allocator-test.cpp` is provided to test your implementation. `allocator-output.txt` contains the correct output generated by the driver. Practice using makefiles by refactoring the `makefile` from previous assignments to compile, link, run the executable, and test the output.

Automatic evaluation

1. In the course web page, click on the appropriate submission page to submit `allocator.hpp`.
2. Please read the following rubrics to maximize your grade:
    - Your submission will receive an F grade if your submission doesn't compile with the full suite of `g++` options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if output of function matches correct output of auto grader.
    - A maximum of D grade if Valgrind detects even a single memory leak or error. A teaching assistance will check you submission for such errors.
    - A deduction of one letter grade for missing file-level documentation in `allocator.hpp`. A deduction of one letter grade for each missing function definition documentation block in `allocator.hpp`. Your submission must have **one** file-level documentation block and function-level documentation blocks for ***every function you're defining***. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to E.