Project Description

Implement a custom memory manager in C++ that compares the performance/efficiency of three different allocation strategies/algorithms:

1. **Best Fit** - The allocator chooses the smallest area of memory that is available that is large enough for the desired allocation.
2. **First Fit** - The allocator chooses the first area of memory that is of sufficient size for the desired allocation
3. **Custom Designed** - The algorithm chooses a location for the allocation based on a scheme you design yourself.

The custom allocator should overload the `new`, `new[]`, `delete`, `delete[]` operators as well as any other versions of those operators you see fit. They should be overloaded at the global scope.
The allocation and de-allocation operators should make use of a custom Allocator class whose functionality is exposed through an `AllocatorSingleton` object using the Singleton Design Pattern. The implementation of each allocation strategy should be through a subclass of Allocator. Implementing the relationship between the allocator and its subclasses should be done using the Strategy Design Pattern. Note: Allocator and AllocatorSingleton are different classes.

Stress Testing your Implementation

You'll design and implement two versions of stress testing:

1. Artificial Testing - Designing a strategy of generating allocation sets of increasing size and complexity solely for the purpose of testing allocations. Example of one data set: randomly generate allocation sizes between 4 and 1024 bytes. Devise a way to randomly allocate and de-allocate (intermingled) chunks of memory for each of those sizes. Based on your knowledge of machine organization, memory management, etc., you are encouraged to devise at least four different experiments you will carry out.

2. Real World Testing - Test it with increasingly large data sets.

Deliverables

Implementation

- You should submit a complete implementation of your memory manager along with the relevant stress testing code.
- Your code base should adhere to the following standards and guiding principles:
    i. To the extent possible, your code should be self documenting.
    ii. Use code documentation (comments) where needed to contextualize or otherwise explain challenging or dense blocks of code.
    iii. Your code should build without warnings when using the `-Wall` flag with g++.
    iv. A full and complete guide to running your project in the spirit or "Reproducible Research"