Задания к работе №1 по языкам и методам программирования.

Все задания реализуются на языке программирования C++ (стандарт C++14 и выше). Реализованные в заданиях приложения не должны завершаться аварийно; все возникающие исключительные ситуации должны быть перехвачены и обработаны.

Во всех заданиях запрещено пользоваться функциями, позволяющими завершить выполнение приложения из произвольной точки выполнения.

Во всех заданиях при реализации необходимо разделять контексты работы с данными (поиск, сортировка, добавление/удаление, модификация и т. п.) и отправка данных в поток вывода / выгрузка данных из потока ввода.

Во всех заданиях все вводимые (с консоли, файла, командной строки) пользователем данные должны (если не сказано обратное) быть подвергнуты валидации в соответствии с типом валидируемых данных.

Во всех заданиях необходимо контролировать ситуации с невозможностью [пере]выделения памяти; во всех заданиях необходимо корректно освобождать всю выделенную динамическую память.

Все ошибки, связанные с операциями открытия системных ресурсов уровня ОС (файлы, средства синхронизации, etc.), должны быть обработаны; все открытые системные ресурсы должны быть возвращены ОС.

Реализованные компоненты должны зависеть от абстракций, а не от конкретных реализаций абстракций. Для реализованных компонентов должны быть переопределены (либо перекрыты - при обосновании) следующие механизмы классов C++: конструктор копирования, деструктор, оператор присваивания, конструктор перемещения, присваивание перемещением.

Для задач, каталоги которых в репозитории содержат папку *tests*, требуется демонстрация прохождения всех описанных тестов для реализованных компонентов. Модификация кода тестов запрещена.

1. Реализуйте логгер (repo path: */logger/client_logger*) на основе контракта *logger* (repo path: */logger/logger*). Ваша реализация логгера должна конфигурироваться двумя способами:
   - на основе данных из конфигурационного файла (структуру файла продумайте самостоятельно);
   - на основе порождающего паттерна проектирования *builder*.

Конфигурирование объекта логгера позволяет задавать потоки вывода (файловые и консольный), структуру лога в виде форматной строки в стиле C (в форматной строке допустимо использование следующих флагов: %d - текущая дата (григорианский календарь, GMT+0); %t - текущее время (GMT+0); %s - уровень жёсткости логгирования в виде строки; %m - логгируемое сообщение), а также для каждого потока вывода множество *severity*, для которых должен производиться вывод сообщения в данный поток вывода (пример: если для потока вывода задан набор *severity*: *trace*, *critical*, то через данный объект логгера в данный поток вывода будут выводиться логи только с этими *severity*). При повторной настройке уже открытого для объекта логгера потока необходимо обновить уже настроенное множество *severity*.

Учтите, что один и тот же поток вывода может использоваться единовременно разными объектами логгеров (и набор *severity* этого потока вывода для разных логгеров также может различаться). При разрушении последнего объекта логгера, связанного с заданным файловым потоком вывода, этот файловый поток вывода должен быть закрыт.

Реализуйте и продемонстрируйте работу приложения, в рамках которого

- различными способами конфигурируются несколько объектов логгера (при этом хотя бы два логгера связаны с одним файловым потоком вывода);
- для записи логов в файл/на консоль используются сконфигурированные объекты логгеров;
- при разрушении объектов логгеров закрываются связанные с ними потоки вывода.

2. Реализуйте (repo path: */allocator/allocator_global_heap*) аллокатор на основе контракта *allocator* (repo path: */allocator/allocator*). Выделение и освобождение динамической памяти реализуйте посредством глобальных операторов *new* и *delete* соответственно. В типе аллокатора допускается единственное поле типа *logger* * - указатель на объект логгера, используемый объектом аллокатора в процессе работы. При невозможности выделения памяти должна быть сгенерирована исключительная ситуация типа *std::bad_alloc*, а также сгенерированный объект исключения должен быть перехвачен и обработан в вызывающем коде. При освобождении памяти должна осуществляться проверка на принадлежность освобождаемого блока к текущему объекту аллокатора (если это не так, необходимо сгенерировать исключительную ситуацию типа *std::logic_error*, а также перехватить и обработать объект исключения в вызывающем коде). Продемонстрируйте работу объекта аллокатора, разместив в нём объекты различных типов данных. Предусмотрите логгирование (на уровне объекта реализованного аллокатора) следующих данных/ситуаций:
- проброс исключительной ситуации - приоритет *logger::severity::error*;
- переопределение запроса пользователя на выделение памяти - приоритет *logger::severity::warning*;
- состояние блока (без служебных данных) перед освобождением (в виде массива байт) - приоритет *logger::severity::debug*;
- начало/окончание вызова любого метода уровня интерфейса аллокатора - приоритет *logger::severity::debug*;
- начало/окончание вызова любого метода уровня реализованного компонента - приоритет *logger::severity::trace*.

3. Реализуйте (repo path: */allocator/allocator_sorted_list*) аллокатор на основе контракта *allocator_with_fit_mode* (repo path: */allocator/allocator*). Выделение памяти реализуйте при помощи методов (с возможностью конфигурации конкретной реализации через конструктор объекта и через метод *set_fit_mode* контракта *allocator_with_fit_mode*) первого подходящего, лучшего подходящего, худшего подходящего, а освобождение памяти - при помощи метода освобождения в рассортированном списке. В типе аллокатора допускается единственное поле типа *void* * - указатель на доверенную объекту аллокатора область памяти. Служебные данные для работы аллокатора размещайте в доверенной ему области памяти (размер доверенной области памяти задаётся на уровне конструктора объекта аллокатора и доверенная память при этом запрашивается из объекта аллокатора, передаваемого как параметр по умолчанию конструктору (если объект аллокатора отсутствует, память запрашивается из глобальной кучи)). При освобождении памяти в объект аллокатора должна осуществляться проверка на принадлежность освобождаемого блока к текущему объекту аллокатора. Обращения к объекту аллокатора должны быть синхронизированы (должно гарантироваться, что в произвольный момент времени жизни объекта аллокатора выделение/освобождение памяти в нём выполняется максимум в одном потоке исполнения). Продемонстрируйте работу объекта аллокатора, разместив в нём объекты различных типов данных. Предусмотрите логгирование (на уровне объекта реализованного типа аллокатора) следующих данных/ситуаций:
   - проброс исключительной ситуации - приоритет *logger::severity::error*;
   - переопределение запроса пользователя на выделение памяти - приоритет *logger::severity::warning*;
   - после выполнения операции выделения/освобождения памяти: объём доступной для выделения памяти в байтах - приоритет *logger::severity::information*;
   - состояние блока (без служебных данных) перед освобождением (в виде массива байт) - приоритет *logger::severity::debug*;
   - после выполнения операции выделения/освобождения памяти: состояние всей неслужебной памяти, управляемой объектом аллокатора (формат строкового представления блока: “<*block availability*> <*block size*>”, где <*block availability*> - признак свободности/занятости блока (для свободного блока - строка “avail”, для занятого - строка “occup”), <*block size*> - размер текущего блока в байтах (без учёта служебной памяти уровня блока; строковые представления блоков сепарированы символом ‘|’); блоки в строковом представлении отсортированы по возрастанию по ключу адреса байта памяти начала) - приоритет *logger::severity::debug*;
   - начало/окончание вызова любого метода уровня интерфейса аллокатора - приоритет *logger::severity::debug*;
   - начало/окончание вызова любого метода уровня реализованного компонента - приоритет *logger::severity::trace*.

4. Реализуйте (repo path: */allocator/allocator_boundary_tags*) аллокатор на основе контракта *allocator_with_fit_mode* (repo path: */allocator/allocator*). Выделение памяти реализуйте при помощи методов (с возможностью конфигурации конкретной реализации через конструктор объекта и через метод *set_fit_mode* контракта *allocator_with_fit_mode*) первого подходящего, лучшего подходящего, худшего подходящего, а освобождение памяти - при помощи метода освобождения с дескрипторами границ. В типе аллокатора допускается единственное поле типа *void* * - указатель на доверенную объекту аллокатора область памяти. Служебные данные для работы аллокатора размещайте в доверенной ему области памяти (размер доверенной области памяти задаётся на уровне конструктора объекта аллокатора и доверенная память при этом запрашивается из объекта аллокатора, передаваемого как параметр по умолчанию конструктору (если объект аллокатора отсутствует, память запрашивается из глобальной кучи)). При освобождении памяти в объект аллокатора должна осуществляться проверка на принадлежность освобождаемого блока к текущему объекту аллокатора. Обращения к объекту аллокатора должны быть синхронизированы (должно гарантироваться, что в произвольный момент времени жизни объекта аллокатора выделение/освобождение памяти в нём выполняется максимум в одном потоке исполнения). Продемонстрируйте работу объекта аллокатора, разместив в нём объекты различных типов данных. Предусмотрите логгирование (на уровне объекта реализованного типа аллокатора) следующих данных/ситуаций:
   - проброс исключительной ситуации - приоритет *logger::severity::error*;
   - переопределение запроса пользователя на выделение памяти - приоритет *logger::severity::warning*;
   - после выполнения операции выделения/освобождения памяти: объём доступной для выделения памяти в байтах - приоритет *logger::severity::information*;
   - состояние блока (без служебных данных) перед освобождением (в виде массива байт) - приоритет *logger::severity::debug*;
   - после выполнения операции выделения/освобождения памяти: состояние всей неслужебной памяти, управляемой объектом аллокатора (формат строкового представления блока: “<*block availability*> <*block size*>”, где <*block availability*> - признак свободности/занятости блока (для свободного блока - строка “avail”, для занятого - строка “occup”), <*block size*> - размер текущего блока в байтах (без учёта служебной памяти уровня блока; строковые представления блоков сепарированы символом ‘|’); блоки в строковом представлении отсортированы по возрастанию по ключу адреса байта памяти начала) - приоритет *logger::severity::debug*;
   - начало/окончание вызова любого метода уровня интерфейса аллокатора - приоритет *logger::severity::debug*;
   - начало/окончание вызова любого метода уровня реализованного компонента - приоритет *logger::severity::trace*.

5. Реализуйте (repo path: */allocator/allocator_buddies_system*) аллокатор на основе контракта *allocator_with_fit_mode* (repo path: */allocator/allocator*). Выделение памяти реализуйте при помощи методов (с возможностью конфигурации конкретной реализации через конструктор объекта и через метод *set_fit_mode* контракта *allocator_with_fit_mode*) первого подходящего, лучшего подходящего, худшего подходящего, а освобождение памяти - при помощи метода освобождения в системе двойников. В типе аллокатора допускается единственное поле типа *void ** - указатель на доверенную объекту аллокатора область памяти. Служебные данные для работы аллокатора размещайте в доверенной ему области памяти (размер доверенной области памяти задаётся на уровне конструктора объекта аллокатора и доверенная память при этом запрашивается из объекта аллокатора, передаваемого как параметр по умолчанию конструктору (если объект аллокатора отсутствует, память запрашивается из глобальной кучи)). При освобождении памяти в объект аллокатора должна осуществляться проверка на принадлежность освобождаемого блока к текущему объекту аллокатора. Обращения к объекту аллокатора должны быть синхронизированы (должно гарантироваться, что в произвольный момент времени жизни объекта аллокатора выделение/освобождение памяти в нём выполняется максимум в одном потоке исполнения). Продемонстрируйте работу объекта аллокатора, разместив в нём объекты различных типов данных. Предусмотрите логгирование (на уровне объекта реализованного типа аллокатора) следующих данных/ситуаций:

- проброс исключительной ситуации - приоритет *logger::severity::error*;
- переопределение запроса пользователя на выделение памяти - приоритет *logger::severity::warning*;
- после выполнения операции выделения/освобождения памяти: объём доступной для выделения памяти в байтах - приоритет *logger::severity::information*;
- состояние блока (без служебных данных) перед освобождением (в виде массива байт) - приоритет *logger::severity::debug*;
- после выполнения операции выделения/освобождения памяти: состояние всей неслужебной памяти, управляемой объектом аллокатора (формат строкового представления блока: “<*block availability*> <*block size*>”, где <*block availability*> - признак свободности/занятости блока (для свободного блока - строка “avail”, для занятого - строка “occup”), <*block size*> - размер текущего блока в байтах (без учёта служебной памяти уровня блока; строковые представления блоков сепарированы символом ‘|’); блоки в строковом представлении отсортированы по возрастанию по ключу адреса байта памяти начала) - приоритет *logger::severity::debug*;
- начало/окончание вызова любого метода уровня интерфейса аллокатора - приоритет *logger::severity::debug*;
- начало/окончание вызова любого метода уровня реализованного компонента - приоритет *logger::severity::trace*.

6. Реализуйте (repo path: */allocator/allocator_red_black_tree*) аллокатор на основе контракта *allocator_with_fit_mode* (repo path: */allocator/allocator*). Выделение памяти реализуйте при помощи методов (с возможностью конфигурации конкретной реализации через конструктор объекта и через метод *set_fit_mode* контракта *allocator_with_fit_mode*) первого подходящего, лучшего подходящего, худшего подходящего, а освобождение памяти - при помощи алгоритмов вставки/удаления для красно-чёрного дерева. В типе аллокатора допускается единственное поле типа *void* * - указатель на доверенную объекту аллокатора область памяти. Служебные данные для работы аллокатора размещайте в доверенной ему области памяти (размер доверенной области памяти задаётся на уровне конструктора объекта аллокатора и доверенная память при этом запрашивается из объекта аллокатора, передаваемого как параметр по умолчанию конструктору (если объект аллокатора отсутствует, память запрашивается из глобальной кучи)). При освобождении памяти в объект аллокатора должна осуществляться проверка на принадлежность освобождаемого блока к текущему объекту аллокатора. Обращения к объекту аллокатора должны быть синхронизированы (должно гарантироваться, что в произвольный момент времени жизни объекта аллокатора выделение/освобождение памяти в нём выполняется максимум в одном потоке исполнения). Продемонстрируйте работу объекта аллокатора, разместив в нём объекты различных типов данных. Предусмотрите логгирование (на уровне объекта реализованного типа аллокатора) следующих данных/ситуаций:
   - проброс исключительной ситуации - приоритет *logger::severity::error*;
   - переопределение запроса пользователя на выделение памяти - приоритет *logger::severity::warning*;
   - после выполнения операции выделения/освобождения памяти: объём доступной для выделения памяти в байтах - приоритет *logger::severity::information*;
   - состояние блока (без служебных данных) перед освобождением (в виде массива байт) - приоритет *logger::severity::debug*;
   - после выполнения операции выделения/освобождения памяти: состояние всей неслужебной памяти, управляемой объектом аллокатора (формат строкового представления блока: “<*block availability*> <*block size*>”, где <*block availability*> - признак свободности/занятости блока (для свободного блока - строка “avail”, для занятого - строка “occup”), <*block size*> - размер текущего блока в байтах (без учёта служебной памяти уровня блока; строковые представления блоков сепарированы символом ‘|’); блоки в строковом представлении отсортированы по возрастанию по ключу адреса байта памяти начала) - приоритет *logger::severity::debug*;
   - начало/окончание вызова любого метода уровня интерфейса аллокатора - приоритет *logger::severity::debug*;
   - начало/окончание вызова любого метода уровня реализованного компонента - приоритет *logger::severity::trace*.