Задания к работе №2 по Системному Программированию.

Все задания реализуются на языке программирования C++ (стандарт C++20 и выше).

Реализованные в заданиях приложения не должны завершаться аварийно; все возникающие исключительные ситуации должны быть перехвачены и обработаны.

Во всех заданиях запрещено использование: глобальных переменных (включая errno), оператора безусловного перехода (goto).

Во всех заданиях запрещено пользоваться функциями, позволяющими завершить выполнение приложения из произвольной точки выполнения, вне контекста исполнения функции main.

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 параметры функций и вводимые (с консоли, файла, командной строки) пользователем данные должны подвергаться валидации в соответствии с типом валидируемых данных, если не сказано обратное; валидация должна зависеть от типа данных и логики применения этих данных для выполнения целевой подзадачи. При передаче аргументов приложению в командную строку, их количество также должно валидироваться.

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.

Во всех заданиях запрещено использование глобальных переменных. Во всех заданиях при реализации функций необходимо обеспечить возможность обработки ошибок различных типов на уровне вызывающего кода.

Во всех заданиях сравнение (на предмет эквивалентности или отношения порядка) вещественных чисел на уровне функции должно использовать значение эпсилон, которое является параметром этой функции.

Во всех заданиях при реализации функций необходимо максимально ограничивать возможность модификации (если она не подразумевается) передаваемых в функцию параметров (используйте ключевое слово const), а также объекта, в случае метода.

Для реализованных компонентов должны быть переопределены (либо перекрыты - при обосновании) следующие механизмы классов C++: конструктор копирования, деструктор, оператор присваивания, конструктор перемещения, присваивание перемещением.

Во всех заданиях необходимо уменьшать количество копирований нетривиально копируемых объектов.

Во всех заданиях необходимо проектировать компоненты с учетом SOLID принципов. Компонент не должен управлять ресурсом, если это не является его единственной задачей.

Запрещается пользоваться элементами стандартной библиотеки Си, если существует их аналог в стандартной библиотеке языка C++.

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/server_logger*) на основе базового класса *logger* (repo path: */logger/logger*). Ваша реализация логгера должна позволить инициировать отправку запроса на логгирование с заданным *severity* сообщения в серверный процесс уровня того же экземпляра ОС. Запросы формируются на основе логгируемого сообщения и уровня жёсткости логгирования *severity*, а также идентификатора процесса, инициирующего запрос. Используемые средства межпроцессного взаимодействия (IPC) и протокол взаимодействия компонента с серверным процессом определите самостоятельно. Предусмотрите функционал настройки параметров для средств межпроцессного взаимодействия. Обеспечьте конфигурирование исходного кода реализованного компонента под ОС семейств Windows и UNIX. Ваша реализация логгера должна конфигурироваться на основе реализации порождающего паттерна проектирования “строитель”. Поэтапное построение объекта логгера предполагает следующие возможности:
    - настройка потоков вывода (файловые и консольный) с заданием для каждого потока вывода множества severity; созданный реализацией строителя объект логгера должен выводить сообщения с заданным severity только в те настроенные потоки вывода, в множестве severity которых присутствует переданное методу log значение severity;
    - настройка структуры лога, печатаемого логгером в потоки вывода, 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 - строковое представление уровня жёсткости логгирования; %m - логгируемое сообщение;
    - настройка информации о потоках вывода, их severity и структуры лога на основе содержимого конфигурационного файла (в качестве параметров подаются путь к конфигурационному файлу и путь поиска (path) уровня конфигурационного файла, где находится информация о наполнении логгера, т.к. один конфигурационный файл может содержать много конфигураций). Структуру конфигурационного файла определите самостоятельно; предполагается, что найденное содержимое конфигурационного файла с вышеописанной информацией валидно, однако не гарантируется, что это содержимое будет найдено в файле, а также не не гарантируется существование самого файла;
    - удаление всех настроенных параметров с возможностью дальнейшей работы со строителем.
    - настройка пути до процесса сервера

2. Реализуйте (repo path: */allocator/allocator_global_heap*) аллокатор на основе базового класса *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*. 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*;
   - 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 базового класса *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*;
- 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 базового класса *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*;
- 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 базового класса *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*;
- 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 базового класса *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*;
   - 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*.