**Коллоквиум**

**Операционный Системы**

**Лисовский Алесь 8 группа**

**Вариант 6**

**0.1: Приведите Win API, необходимое для решения Лабораторной работы номер 2**

То, что я использовал

DWORD WINAPI average(LPVOID) {

for (size\_t i = 0; i < array\_size; i++)

{

sum += arr[i];

Sleep(12);

}

std::cout << "Average:\t" << sum / array\_size << "\n";

return 0;

}

DWORD WINAPI min\_max(LPVOID) {

min\_element = arr[0];

max\_element = arr[0];

for (size\_t i = 1; i < array\_size; i++)

{

if (arr[i] > max\_element) {

max\_element = arr[i];

max\_index = i;

}

Sleep(7);

if (arr[i] < min\_element) {

min\_element = arr[i];

min\_index = i;

}

Sleep(7);

}

std::cout << "Minimal element:\t" << min\_element << "\n";

std::cout << "Maximal element:\t" << max\_element << "\n";

return 0;

}

**0.2: Что такое процесс в ОС Windows**

Процесс в операционной системе Windows - это запущенная программа или задача, которая выполняется в оперативной памяти компьютера. Каждый процесс имеет свою собственную область памяти, где хранятся данные и переменные, необходимые для его работы. Процессы могут взаимодействовать друг с другом, передавая данные и используя общие ресурсы. Windows управляет процессами, позволяя им выполняться параллельно и обеспечивая безопасность и стабильность работы системы. Некоторые процессы являются системными и отвечают за основные функции операционной системы, а другие - пользовательскими, которые запускаются приложениями и программами, которые мы используем на компьютере.

**0.3: Что такое Критическая секция**

Критическая секция в WinAPI (Windows API) - это участок кода, который должен быть выполнен только одним потоком одновременно. Она используется для обеспечения синхронизации и защиты общих данных от одновременного доступа нескольких потоков, что может привести к непредсказуемым результатам или ошибкам.

В контексте многопоточности, критическая секция позволяет потокам входить в нее и выполнять код, который является критическим, только по очереди, один за другим. Когда один поток входит в критическую секцию, он блокирует доступ к ней для других потоков, пока сам не завершит работу и не освободит критическую секцию.

Для работы с критическими секциями в WinAPI используются функции, такие как InitializeCriticalSection, EnterCriticalSection, LeaveCriticalSection, и DeleteCriticalSection. Они позволяют создавать, входить в критическую секцию, покидать ее и удалять ее после использования соответственно.

Использование критических секций помогает гарантировать корректное выполнение кода в многопоточной среде и избежать состояний гонки.

**0.4: Что такое Семафор**

Семафор в WinAPI (Windows API) - это объект синхронизации, который используется для контроля доступа к ресурсам или выполнения операций в многопоточной среде. Он позволяет ограничивать количество потоков одновременно выполняющихся операций и синхронизировать их работу.

Семафор имеет счётчик, который инициализируется определенным значением и может быть изменен потоками при каждой операции "ожидание" и "освобождение". Если счетчик больше нуля, поток освобождает семафор и продолжает свою работу. Если счетчик равен нулю, поток ожидает своей очереди на доступ к ресурсу.

Один из наиболее распространенных примеров применения семафора - это управление доступом к ограниченному количеству ресурсов, таких как потоки, память, файлы и т.д. Семафоры также могут использоваться для синхронизации потоков, задержки исполнения до определенного события или для упорядочивания операций в многопоточной среде.

В WinAPI для работы с семафорами используются функции, такие как CreateSemaphore, WaitForSingleObject, ReleaseSemaphore и другие. Они позволяют создавать, ожидать и освобождать семафоры, а также управлять их свойствами.

Использование семафоров в многопоточных приложениях помогает эффективно контролировать доступ к ресурсам и синхронизировать операции между потоками.

**0.5: Сравнительный анализ стандарта C++ 98 с и без применения библиотеки boost (в контексте лабораторных)**

Стандарт C++ 11 включает несколько изменений в самом языке(например, приведён в порядок механизм компиляции шаблонов, что позволяет использовать «внешние шаблоны» («extern template»), а как же проще инициализировать объекты-контейнеры; и даже мехнизм определения типов(type inference mechanism)). Но так же стандарт содержит множество улучшений в Standard Library.

Тут есть важная оговорка, упомянутая ранее: несмотря на то, что стандарт включает описание Standard Library, он не включает имплементацию ни компилятора, ни языка. Всё это остаётся на совести третьих лиц, таких как поставщики компиляторов. Не существует эталонной реализации. Стандарт — это не код, это *описание*.

Так же, одна вещь, которую нужно знать о Boost — это что она на самом деле состоит из нескольких библиотек, и её дизайнеры намеренно раздели их. Это значит, что вы можете использовать только те библиотеки из Boost, которые вам нужны. Там есть несколько взаимозависимостей, но они хорошо документированы. Таким образом, вы можете комбинировать Boost со Standard Library, если необходимо. Например, старые добрые iostream являются частью Standard Library со времён первого стандарта 1998 года, и Boost легко работает с ними(но так же включает свои собственные iostream, если вам так удобнее).

**2.1: Что такое Процедурная декомпозиция**

Процедурная декомпозиция - это метод структурирования программного кода путем разбиения его на более мелкие процедуры или функции. Она представляет собой процесс разделения сложных задач на более простые и понятные шаги, которые можно реализовывать отдельно.

Процедурная декомпозиция позволяет облегчить понимание и поддержку кода, а также упростить его разработку и тестирование. Код разбивается на небольшие самодостаточные модули, каждый из которых выполняет определенную задачу. Это позволяет легко читать, понимать и отлаживать код, а также повторно использовать модули в других частях программы.

Процедурная декомпозиция способствует улучшению читаемости кода, снижению сложности программы, упрощению рефакторинга и обеспечивает модульность программы. Когда код разбит на более мелкие процедуры, каждая процедура выполняет только одну задачу и может быть переиспользована в различных контекстах. Это способствует повышению гибкости и переносимости программы.

Кроме того, процедурная декомпозиция помогает распределить задачи программы между разработчиками, упростить планирование и управление разработкой, а также повысить возможности параллельной разработки.

Процедурная декомпозиция является одним из основных принципов структурированного программирования и широко применяется в различных программных языках и методологиях разработки.

**2.2: Что такое динамический полиморфизм**

Динамический полиморфизм - это свойство объектно-ориентированного программирования, которое позволяет вызывать методы одного и того же имени у разных объектов и выполнять разные действия в зависимости от типа объекта, к которому они относятся.

В основе динамического полиморфизма лежит идея наследования и виртуальных функций. Когда класс наследуется от другого класса, он может переопределить виртуальную функцию этого класса, предоставляя свою собственную реализацию. В результате, при вызове виртуальной функции через указатель или ссылку на базовый класс, будет выполнена соответствующая функция из наследника.

Динамический полиморфизм позволяет обрабатывать различные объекты одним и тем же кодом, так как объекты могут быть одного базового типа и содержать разные реализации виртуальных функций. Это позволяет сократить дублирование кода и упростить его поддержку.

Примером динамического полиморфизма является использование наследования и виртуальных функций в языке программирования C++. Когда метод вызывается через указатель или ссылку на базовый класс, компилятор определяет, какая реализация метода должна быть вызвана в зависимости от типа объекта, на который указывает указатель или ссылка.

Динамический полиморфизм применяется для создания гибких и расширяемых систем, где различные объекты могут работать с одним и тем же интерфейсом, но с различным поведением. Это позволяет программистам писать более обобщенный и гибкий код, способный обрабатывать широкий спектр объектов и их различные типы.

**2.3: Что такое инкапсуляция**

Инкапсуляция - это принцип объектно-ориентированного программирования, который объединяет данные (переменные) и методы, работающие с этими данными, в единый объект, скрывая внутреннюю реализацию и предоставляя интерфейс для взаимодействия с внешним миром.

Основная идея инкапсуляции заключается в том, что данные и методы, оперирующие с этими данными, объединены внутри объекта, и доступ к ним осуществляется через определенный интерфейс. Это позволяет скрыть детали реализации и изменять внутреннюю структуру объекта без влияния на код, использующий объект.

Инкапсуляция обеспечивает важный аспект объектно-ориентированного программирования - сокрытие информации. При инкапсуляции данные объекта могут быть доступны только через определенные методы, которые контролируют и обеспечивают безопасное использование данных. Это позволяет предотвратить неконтролируемые изменения данных и обеспечивает более гибкую и надежную систему.

Примером инкапсуляции может служить класс в языке программирования, где переменные-члены класса являются закрытыми (private) и доступ к ним осуществляется через открытые (public) методы класса. Такой подход позволяет контролировать доступ к данным и предоставляет возможность изменять внутреннюю реализацию класса, не затрагивая код, который использует класс.

Инкапсуляция вносит важный вклад в создание модульных, гибких и безопасных систем, позволяющих управлять доступом к данным и скрывать детали реализации, что способствует повышению эффективности, надежности и безопасности программного кода.

**3: Сформулировать максимально детальное описание Design Pattern-а через призму инкапсуляции. Привести примеры использования на практике (или теоретические примеры использования)**

**3.1: Абстрактная фабрика**

Абстрактная фабрика - это паттерн проектирования, который позволяет создавать семейства объектов определенного типа, без указания конкретных классов этих объектов. Он основан на принципе инкапсуляции, который позволяет скрыть детали создания объектов и предоставляет унифицированный интерфейс для их создания.

Основная идея абстрактной фабрики заключается в том, что она предоставляет интерфейс для создания объектов, но не указывает конкретные классы этих объектов. Вместо этого она определяет семейство связанных объектов, где каждый класс является вариацией своей абстрактной формы. Таким образом, создание конкретных объектов происходит внутри абстрактной фабрики, и клиентский код взаимодействует с ней только через общий интерфейс.

Примером использования абстрактной фабрики может служить создание пользовательского интерфейса. Предположим, у нас есть несколько тем оформления интерфейса, и каждая тема имеет свои связанные объекты, такие как кнопки, поля ввода и т. д. Мы можем создать абстрактную фабрику, которая определяет методы для создания этих объектов. Затем мы можем создать конкретную фабрику для каждой темы оформления, где реализуем эти методы для создания объектов соответствующей темы оформления. Таким образом, клиентский код будет взаимодействовать с абстрактной фабрикой, вызывая методы для создания объектов интерфейса, и всю логику создания объектов будет скрыта внутри конкретных фабрик.

Преимущества использования абстрактной фабрики включают возможность создания семейств связанных объектов согласованно и легко взаимозаменяемых. Она также обеспечивает гибкость и расширяемость системы, позволяя добавлять новые семейства объектов без влияния на клиентский код. Кроме того, абстрактная фабрика способствует сокрытию деталей создания объектов и обеспечивает лучшую читабельность и поддерживаемость кода.

**3.2: Façade**

Дизайн-паттерн Facade представляет собой структурный паттерн, который объединяет группу связанных интерфейсов в один унифицированный интерфейс, с целью упрощения взаимодействия клиентского кода с комплексной системой или подсистемой. Он инкапсулирует сложную логику и внутреннюю структуру, предоставляя клиенту удобный и простой в использовании интерфейс для взаимодействия с системой.

Основная идея Facade заключается в том, чтобы создать "фасад" – класс, который представляет собой высокоуровневый интерфейс к сложной системе или подсистеме. Этот класс скрывает сложность и детали внутренней реализации, предоставляя клиенту только необходимые методы и функции. Таким образом, клиентский код может взаимодействовать с системой, используя только этот унифицированный интерфейс, не заботясь о сложной логике и внутренней структуре системы.

При использовании Facade происходит следующие шаги:

1. Создание класса Facade, который инкапсулирует сложную систему или подсистему.

2. Внутри класса Facade определяются методы и функции, которые предоставляют унифицированный интерфейс для взаимодействия с системой.

3. Клиентский код взаимодействует с системой, вызывая методы и функции класса Facade.

Пример использования Facade можно найти в разработке программного приложения, где есть сложная система модулей или подсистем, и требуется простой интерфейс для взаимодействия с этой системой. Например, веб-фреймворк может использовать Facade для предоставления упрощенного интерфейса для работы с базой данных. Вместо того, чтобы напрямую работать с компонентами базы данных, клиентский код может использовать методы и функции класса Facade, чтобы выполнять операции чтения, записи или удаления данных. Фасад скрывает все сложности, связанные с управлением подключениями, выполнением запросов и т. д., доставляя простой и понятный интерфейс для работы с базой данных.