Полонейчик Ульяна, 9 группа, Вариант 2

Общие вопросы:
1) Что такое ООП? – полное определение

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции объектов. Объекты инкапсулируют данные и поведение, связанное с ними, предоставляя чёткие интерфейсы для взаимодействия. Основные принципы ООП:

1. Инкапсуляция — скрытие деталей реализации объекта и предоставление доступа к данным только через методы.
2. Наследование — создание новых классов на основе уже существующих, что позволяет повторно использовать код.
3. Полиморфизм — способность объекта принимать разные формы, позволяя переопределять методы в производных классах.
4. Абстракция — выделение значимых характеристик объекта, игнорирование несущественных деталей.

Можно взять в пример создание различных машин для таксопарка, то есть мы берём какой-то общий класс Car(), который содержит в себе общие поля такие как цвет, марку, время начала и конца поездки, номер техосмотра. А от него создадим дочерний класс Taxi(), который будет иметь тарифы, возможность провоза животных/детей. Так при необходимости можно будет создать другие дочерние классы от Car(), которые не будут конфликтовать дргу с другом.

2) Магическое число 7 Миллера? – привести не менее 7 примеров из IT

Концепция Джорджа Миллера предполагает, что человек способен одновременно удерживать в кратковременной памяти около 7 (±2) элементов информации. Примеры в IT:

1. Количество параметров функции — стараются не превышать 7.
2. Размер команды разработчиков — идеальная команда 5–9 человек.
3. Меню интерфейсов — не более 7 пунктов для удобства восприятия.
4. Длина строки кода — стараются не превышать 80–100 символов, чтобы за раз охватить взглядом.
5. Количество классов в пакете/модуле — оптимально до 7.
6. Ограничение вкладок в браузере при UX-дизайне.
7. Число веток принятия решения в условных конструкциях.

3) Энтропия ПО?

Энтропия ПО — это степень беспорядка и сложности в программном обеспечении, которая растёт по мере его разработки, если не предпринимать меры для контроля. Примеры негативных мер:

1. Отсутствие модульных тестов.

2. Использование глобальных переменных.
3. Непоследовательные соглашения об именах.
4. Дублирование кода.
5. Сложные зависимости между модулями.

4) 5 признаков сложной системы по Гради Бучу

1. Иерархическая структура:

В таксопарке — иерархия классов Vehicle → Car → Taxi.

В проекте с процессами — структура Main → Creator → Reporter.

2. Эмерджентные свойства:

Система бронирования такси: взаимодействие клиентов и машин создаёт динамическое расписание.

Проект с процессами: последовательность создания и отчётности даёт конечный результат — управление процессами.

3. Повторное использование шаблонов:

Использование базового класса Vehicle для создания разных видов транспорта.

В лабораторной работе с процессами — повторяющиеся шаблоны для создания дочерних процессов.

4. Постоянные изменения:

В проекте таксопарка — добавление новых функций, например, расчёт стоимости поездки.

В проекте с процессами — адаптация логики взаимодействия процессов в зависимости от требований.

5. Многоуровневая абстракция:

Интерфейс Vehicle для описания базовых методов всех транспортных средств.

Создание класса Process для управления потоками информации между программами.

5) Закон иерархических компенсаций Седова

Закон Седова - сложные системы развиваются иерархически, причём каждая новая иерархия компенсирует недостатки предыдущей. Примеры в IT:

1. Развитие языков программирования: от ассемблера к C, затем к C++ с ООП.
2. Создание операционных систем: от однозадачности к многозадачности.
3. Сетевые протоколы: от прямых соединений к многоуровневой модели OSI.
4. Архитектура программ: от монолитных приложений к микросервисам.
5. Развитие версионного контроля: от локальных копий к распределённым системам Git.

Нулевые вопросы:

1) Приведите Win API, необходимое для решения Лабораторной работы номер 2.

Для второй лабораторной работы использовалось несколько функций Win API, которые позволили выполнить задание: создать потоки, синхронизировать их выполнение и освободить ресурсы после окончания работы. Сами функции:

1. CreateThread — создаёт новый поток выполнения. Пример из кода второй лабораторной:
   HANDLE hMinMax = CreateThread(NULL, 0, findMinMax, &data, 0, NULL);
2. WaitForSingleObject — приостанавливает выполнение текущего потока до завершения указанного потока. Пример из лабораторной:
   WaitForSingleObject(hMinMax, INFINITE);
3. CloseHandle — закрывает дескриптор потока, освобождая системные ресурсы. Пример из лабораторной:
   CloseHandle(hMinMax);
4. Sleep — приостанавливает выполнение текущего потока на заданное количество миллисекунд, имитируя задержку для наглядности параллельной работы. Пример из лабораторной:
   Sleep(7);

2) Что такое процесс в ОС Windows?

Процесс в Windows — это экземпляр выполняемой программы, который включает:

1. Адресное пространство — область памяти, выделенная процессу.
2. Контекст выполнения — информация о состоянии процессора, стека, регистров и т. д.
3. Дескрипторы ресурсов — открытые файлы, потоки, семафоры и прочие системные объекты.
4. Потоки выполнения — минимальные единицы выполнения кода внутри процесса.

Пример из лабораторной: моя основная программа — это один процесс, а потоки findMinMax и calculateAverage — параллельные ветки выполнения внутри этого процесса.

3) Что такое критическая секция?

Критическая секция — это участок кода, который может выполняться только одним потоком одновременно для предотвращения состояния гонки при доступе к общим данным.

В WinAPI для работы с критическими секциями используются:

1. InitializeCriticalSection — инициализация секции.
2. EnterCriticalSection — вход в критическую секцию.
3. LeaveCriticalSection — выход из секции.
4. DeleteCriticalSection — освобождение ресурсов.

Пример: в лабораторной, при одновременной записи в min, max и average разными потоками может возникнуть состояние гонки. Добавление критической секции защитит доступ к этим переменным.

4) Что такое семафор?

Семафор — это синхронизационный примитив, ограничивающий количество потоков, которые могут одновременно выполнить определённый участок кода.

В Windows используются:

1. CreateSemaphore — создание семафора.
2. WaitForSingleObject — ожидание, пока семафор не будет освобождён.
3. ReleaseSemaphore — освобождение семафора, увеличивая счётчик.

Применение: если нужно ограничить доступ к ресурсу не более чем для N потоков одновременно — семафоры идеальны. В моей лабораторной, при необходимости, можно было бы применить семафор для ограничения количества потоков, которые одновременно работают с массивом.

5) Сравнительный анализ C++98 с и без Boost / свежего стандарта с или без Qt

C++98 без Boost:

1. Нет потоков — придётся использовать WinAPI (что и использовалось в лабораторной работе).
2. Нет умных указателей — управление памятью вручную.
3. Нет std::thread и std::mutex — вся синхронизация через WinAPI, из-за чего структура кода становится немного сложнее.

C++98 с Boost:

1. boost::thread облегчает создание потоков.
2. boost::mutex заменяет критические секции WinAPI, устраняет состояние “гонки” при запуске нескольких процессов, например, процессы min, max, average в лабораторной работе.
3. boost::bind и boost::function делают код чище при передаче функций в потоки.

Современный C++ (11/14/17/20):

1. Потоки с std::thread проще и безопаснее.
2. std::mutex и std::lock_guard заменяют критические секции, синхронизация потоков встроена в стандарт.
3. std::async и std::future - асинхронное выполнение без явной работы с потоками.

Современный C++ с Qt:

1. QThread позволяет создавать потоки в стиле Qt с возможностью управления и наследования.

2. QMutex и QSemaphore дают высокоуровневые инструменты синхронизации, защищают критические секции.
3. Signal-Slot упрощает коммуникацию между потоками.

Вывод:

1. В C++98 без Boost работа с потоками сложнее, так как нужно использовать WinAPI напрямую.
2. Boost облегчает работу с многопоточностью, но требует дополнительной библиотеки.
3. Современный C++ предлагает встроенные средства работы с потоками, делая код проще и безопаснее.
4. Qt добавляет мощный каркас для построения многопоточных приложений с удобными механизмами синхронизации.