C++. Модуль 2. Вопросы.

1. **Наследование**

   a. **Основы наследования**
   Наследование в языке `C++`. Добавление новых полей и методов в наследуемый класс. Модификатор доступа `protected`. Публичное и приватное наследование. Имеют ли друзья базового класса доступ к приватным полям класса-наследника? Порядок вызовов конструкторов при создании экземпляра класса-наследника. Как сделать так, чтобы вызывалась необходимая перегрузка конструктора базового класса при создании экземпляра класса-наследника?

   b. **Перегрузка и переопределение методов в классе наследнике**
   Перегрузка методов в базовом и наследуемом классе. Как проходит отбор перегрузки? Переопределение методов в классе-наследнике. Вызов методов базового класса из класса наследника.

   c. **Приведение типов**
   Присваивание объекта класса наследника объекту базового класса (`base = derived`). Срезка. Строение объекта класса-наследника. Размер объекта класса-наследника. Empty base optimisation. Присваивание указателя на объект класса наследника указателю базового класса (`pbase = pderived`). Иерархия наследования. Использование `static_cast` для перемещения по иерархии наследования. В каких случаях это может привести к неопределённому поведению?

   d. **Множественно наследование**
   Строение объекта класса наследника при обычном (не виртуальном) множественном наследовании. Сдвиг указателей при присваивании в случае множественного наследования. Ромбовидное наследование. Как в языке C++ решается проблема ромбовидного наследования?

2. **Полиморфизм**

   a. **Основы полиморфизма**
   Статический полиморфизм в языке C++ и других языках. Динамический полиморфизм и его примеры в других языках(например, в языке Python). Для чего нужен полиморфизм?

   b. **Основы динамического полиморфизма в языке C++**
   Указатели на базовый класс, хранящие адрес объекта наследуемого класса (`Base* pbase = &derived`). Методы какого класса будут вызываться, если мы будем вызывать их через такой указатель? Виртуальные функции. Виртуальный деструктор. Ключевые слова `override` и `final`. Уметь написать пример использования полиморфизма (например, вектор указателей типа `Base*`). Приватность и виртуальные функции.

   c. **Абстрактные классы**
   Чистая виртуальная функция. Абстрактный класс. Интерфейс. Наследование от интерфейса. Ошибка pure virtual call.

   d. `dynamic_cast`
   Полиморфные типы. Использование `static_cast` для приведения типов и указателей на типы в иерархии наследования. Когда использование `static_cast` может привести к неопределённому поведению? Оператор `dynamic_cast`. Чем он отличается от `static_cast` и в каких случая он используется? Что происходит если `dynamic_cast` не может привести тип (рассмотрите случай приведения указателей и случай приведения ссылок)?.

   e. **Реализация механизма виртуальных функций**
   Скрытое поле - указатель на таблицу виртуальных функций. Сколько таблиц виртуальных функций хранится в памяти при работе программы? Как устроены таблицы виртуальных функций?

3. **Вывод типов и идеальная передача**

   a. **Вывод типов в шаблонах и при использовании** `auto`
   Вывод типов в шаблонах при передаче по значению. Вывод типов в шаблонах при передаче по ссылке/константной ссылке. Вывод типов при использовании `auto`. `std::initializer_list`.

   b. **Вывод типов при использовании** `decltype`
   Правила `decltype`. Вывод возвращаемого значения функции. `decltype(auto)`.

   c. **Вывод аргументов шаблонного класса (CTAD)**
   Руководства вывода (deduction guides).

d. **Универсальные ссылки**
Правила свёртки ссылок. Что такое универсальные ссылки? Как написать функцию, которая принимает по унивесальной ссылке? Какой тип выводится при передаче в такую функцию lvalue-выражения и какой тип выводится при передаче в неё rvalue-выражения?

e. **Типы передачи объекта в функцию**
- Передача по значению копированием
- Передача по значению перемещением
- Передача по lvalue-ссылке
- Передача по rlvalue-ссылке
- Передача по константной lvalue-ссылке
- Передача по универсальной ссылке

Преемущества и недостатки каждого из видов передачи в функцию.

f. **Идеальная передача**
Функция `std::forward`, что делает и зачем она нужна? Чем функция `std::forward` отличается от `std::move`. Как реализована функция `std::forward`? Примеры использования идеальной передачи: `emplace_back` и `make_unique`.

g. **Вариативные шаблоны**
Функция, которая принимает переменное количество аргументов произвольных типов. Шаблонные классы с произвольным количеством шаблонных параметров. Пакет параметров шаблона. Раскрытие пакета. Где можно раскрывать пакет параметров шаблона? Выражения всёртки (fold expressions). Оператор `sizeof...`. Применение вариативных шаблонов совместно с идеальной передачей.

4. **Обработка ошибок. Исключения.**

a. **Методы обработки ошибок.**
Классификация ошибок. Ошибки времени компиляции, ошибки линковки, ошибки времени выполнения, логические ошибки. Виды ошибок времени выполнения: внутренние и внешние ошибки. Методы борьбы с ошибками: макрос `assert`, использование глобальной переменной(`errno`), коды возврата и исключения. Преемущества и недостатки каждого из этих методов. Какие из этих методов желательно использовать для внутренних ошибок, а какие для внешних?

b. `assert`
Макрос `assert` и его применения для обнаружения ошибок.

c. **Коды возврата и класс `std::optional`**
Обработка ошибок с помощью кодов возврата. Примеры стандартных фуцнкий, использующих коды возврата. Класс `optional` из стандартной библиотеки. Методы класса `optional`:
- Конструкторы
- Методы `value`, `has_value`, `value_or`.
- Унарные операторы `*` и `->`
- Оператор преобразования к значению типа `bool`.

Для чего можно применять `std::optional`? Использование класса `optional` для обработки ошибок с помощью кодов возврата.

d. **Исключения.**
Зачем нужны исключения, в чём их преимущество перед другими методами обработки ошибок? Оператор `throw`, аргументы каких типов может принимать данный оператор. Что происходит после достижения программы оператора `throw`. Раскручивание стека. Блок `try-catch`. Что произойдёт, если выброшенное исключение не будет поймано? Стандартные классы исключений: `std::exception`, `std::runtime_error`, `std::bad_alloc`, `std::bad_cast`, `std::logic_error`. Почему желательно ловить стандартные исключение по ссылке на базовый класс `std::exception`? Использование `catch` для ловли всех типов исключений. Использование исключений в кострукторах, деструкторах, перегруженных операторах. Спецификатор `noexcept`. Оператор `noexcept`. Гарантии безопасности исключений. Исключения при перемещении объектов. `move_if_noexcept`. Идиома `copy and swap`.

5. **Вычисления на этапе компиляции**

a. **Вычисление на этапе компиляции с использованием `constexpr`**
`constexpr` переменные. `constexpr` функции. Ограничения в функциях, вычисляемых на этапе компиляции. `consteval` и `constinit`.

b. **Специализация шаблонов**
Полная специализация. Частичная специализация.

c. **Метафункции**
Что такое метафункция? Что делают следующие метафункции и как реализовать подобные метафункции самостоятельно: `std::is_integral`, `std::is_pointer`, `std::is_same`, `std::is_lvalue_reference`, `std::true_type`, `std::false_type`, `std::is_function`, `std::remove_pointer`, `std::remove_reference`, `std::remove_cv`, `std::is_copy_constructible`, `std::is_nothrow_move_constructible`, `std::is_standard_layout`, `std::is_trivially_copyable`.
Суффиксы `_v` и `_t` у метафункций. Как реализовать функции `std::move` и `std::move_if_noexcept`?

d. **SFINAE**
Что такое принцип SFINAE и на чём он основан? Метафункция `std::enable_if`. Как использовать метафункцию `std::enable_if`? Проверка на существование определённого метода у класса.

6. **Концепты**

a. **Ограничение шаблонных параметров**
Наложение ограничений на шаблонные параметры с использованием ключевого слова `requires`. Использование метафункций для ограничения шаблонных параметров.

b. **`requires`-выражения**
`requires`-выражения. Типы требований внутри `requires`-выражения:

- Простое требование (Simple requirement)
- Вложенное требование (Nested requirements)
- Типовое требование (Type requirements)
- Состовное требование (Compound requirements)

c. **Концепты**
Ключевое слово `concept`. Создание своих концептов. Использование концептов для ограничения шаблонных параметров. Стандартные концепты: `std::integral`, `std::floating_point`, `std::same_as`, `std::convertible_to`, `std::default_initializable`, `std::copy_constructable`, `std::move_constructable`, `std::movable`, `std::copyable`, `std::semiregular`, `std::regular`. Концепты итераторов. Перегрузка шаблонных функций по концепту.