C++. Модуль 1. Вопросы.

1. **Пространства имён, ссылки, перегрузка функций, строка и вектор**

   a. **Пространства имён**
   Зачем нужны пространства имён? Оператор разрешения области видимости. Директива `using namespace`. `using`-объявление. Пространства имён и заголовочные файлы. Псевдонимы пространств имён. Пространство имён `std`.

   b. **Ссылки**
   Отличие ссылок от указателей. Инициализация ссылок. Константные ссылки. Продление времени жизни временных объектов. Три типа передачи аргументов в функцию: передача по значению, передача по ссылке и передача по константной ссылке. Преимущества/недостатки каждого метода. Возврат ссылки из функции. Висячие ссылки.

   c. **Перегрузка функций**
   Сигнатура функции. По каким составляющим можно перегружать функцию, а по каким нельзя? Перегрузка при неявных преобразованиях. Выбор функции при перегрузке.

   d. **Манглирование имён**
   Исполняемые и объектные файлы. Символы. Утилиты `nm`, `strings` и `c++filt`. `extern "C"`.

   e. **Класс `std::string`**
   Стандартная строка `std::string`. Преимущества строки `std::string` по сравнению с C-строкой. Конструкторы класса `std::string`. Работа с отдельными символами в строке. Библиотека `cctype`. Методы класса строки:

   - `begin`, `end`, `rbegin`, `rend` и т. д.
   - `operator[]`, `at`, `front` и `back`
   - `clear` и `empty`
   - `push_back` и `pop_back`
   - `insert` и `erase`
   - `substr`
   - `find`
   - `starts_with` и `ends_with`
   - `swap`
   - `c_str`, `size` и `capacity`
   - `reserve` и `resize`
   - `shrink_to_fit`

   Статическая константа `std::string::npos`. Перегруженные операторы `std::string`. Вычислительная сложность всех методов. Передача строк в функции. Конвертация чисел в строки и наоборот. Чтение в строку. Функция `std::getline`. Литералы типа `std::string` из пространства имён `std::string_literals`. Строение объектов класса `std::string` и их размер. Оптимизация малой строки (SSO).

   f. **Класс `std::vector`**
   Контейнер `std::vector`. Конструкторы вектора. Доступ к отдельным элементам в векторе. Присваивание векторов. Методы вектора:

   - `begin`, `end`, `rbegin`, `rend` и т. д.
   - `front` и `back`
   - `operator[]` и `at`
   - `clear` и `empty`
   - `push_back` и `pop_back`
   - `insert` и `erase`
   - `swap`
   - `data`, `size` и `capacity`
   - `reserve` и `resize`
   - `shrink_to_fit`

   Вычислительная сложность всех методов. Передача вектора в функции. Внутренее устройство вектора. Стратегия роста.

   g. **Приведение типов в C++**
   Опасность приведения в стиле C. Оператор приведения `static_cast`. Отличие приведения типов с помощью оператора `static_cast` от приведения типов в стиле C. То есть чем

   ```
   static_cast<type>(a)
   ```

   отличается от:

   ```
   (type)(a)
   ```

   Оператор приведения `reinterpret_cast`. Оператор приведения `const_cast`.

2. **Классы**

   a. **Классы**
   Объектно-ориентированное программирование. Класс. Члены класса. Поля и методы класса. Инкапсуляция. Константные методы класса. Указатель `this`. Сокрытие данных. Модификаторы доступа `private` и `public`. Различие ключевых слов `struct` и `class` при объявлении классов. Геттеры и сеттеры. Друзья классов. Вложенные классы. Псевдонимы типа внутри класса. Использование `typedef` и `using` для создания псевдонимов типов внутри класса.

   b. **Конструкторы и деструкторы**
   Когда вызываются конструкторы, а когда деструкторы? Можно ли перегружать конструкторы и деструкторы? Различные синтаксисы вызова конструктора: с использованием знака `=`, с использованием круглых скобок и с использованием фигурных скобок. Конструкторы и передача в функции/возврат из функций. Использование конструктора для явного и неявного приведения типов. Делегирующий конструктор. Идиома RAII.

   c. **Перегрузка операторов**
   Перегрузка операторов как свободных функций и как методов класса. Какие операторы можно перегружать, а какие нельзя? Перегрузка оператора присваивания. Перегрузка операторов инкремента и декремента. Перегрузка оператора стрелочка (`->`). Перегрузка операторов приведения и ключевое слово `explicit`. Перегрузка операторов `new` и `delete`. Перегрузка операторов `<<` и `>>` с объектами типа `std::ostream` и `std::istream`.

   d. **Реализация своего класса строки**
   Уметь писать свой простейший класс строки. Методы такой строки:

   - Конструктор по умолчанию
   - Конструктор, принимающий строку в стиле C (`const char*`)
   - Конструктор копирования
   - Деструктор
   - Оператор присваивания
   - Оператор присваивания сложения(`+=`).
   - Оператор сложения. Его реализация с помощью операторов `=` и `+=`.
   - Операторы сравнения.
   - Оператор индексирования.

   e. **Раздельная компиляция**
   Вынос определений функций из класса. Forward declaration. Вынос определений функций в другие `.cpp` файлы.

3. **Инициализация**

   a. **Виды инициализации**
   Что такое инициализация? Классификация типов на скалярные типы, агрегатные типы и нормальные классы. Виды инициализации: *default initialization*, *value initialization*, *direct initialization*, *copy initialization*. `explicit`-конструкторы.

   b. **Инициализация полей класса**
   Когда инициализируются поля класса? Инициализация полей класса по умолчанию. Список инициализации полей класса. Порядок инициализации полей класса. Инициализация константных полей и полей-ссылок.

   c. **Особые методы класса**

   - Конструктор по умолчанию
   - Деструктор
   - Конструктор копирования
   - Оператор присваивания копирования
   - Конструктор перемещения
   - Оператор присваивания перемещения

   При каких условиях компилятор автоматически создаёт эти методы? Что делают особые методы, автоматически созданные компилятором? Правило пяти. Удалённые функции и методы, ключевое слово `delete`. Ключевое слово `default` для особых методов класса.

d. **Избегание копирования**
Оптимизация Copy Elision и когда она происходит?

e. **Динамическое создание объектов в куче**
Создание/удаление объектов в куче с помощью операторов `new` и `delete`. Создание/удаление массива объектов в куче с помощью операторов `new[]` и `delete[]`. Основные отличия `new` и `delete` от `malloc` и `free`. Оператор `placement new`. Как оператор `new` возвращает ошибку при нехватки памяти?

f. **Статические поля и методы**
Статическое поле. Инициализация статического поля. Инициализация `const static` полей. Статические методы.

g. **Обработка ошибок**
Классификация ошибок. Ошибки времени компиляции, ошибки линковки, ошибки времени выполнения, логические ошибки. Внутренние и внешние ошибки. Обработка ошибок с помощью макроса `assert`. Обработка ошибок с использованием глобальной переменной `errno`. Обработка ошибок с помощью кодов возврата. В чём недостатки кодов возврата? Использование класса `std::optional` для кодов возврата. Обработка ошибок с использованием исключений.

h. **Основы работы с исключениями**
Преимущества и недостатки исключений перед другими методами обработки ошибок. Оператор `throw`. Какие типы объектов можно "бросать"? Что происходит после достижения программой оператора `throw`? Раскручивание стека. Блок `try-catch`. Перехват по типу. Что произойдёт, если выброшенное исключение не будет поймано? `catch (...)`. Повторное выбрасывание исключения. Исключения в деструкторах. Стандартные классы исключений: `std::exception`, `std::runtime_error`, `std::logic_error`. Метод `what`. Конструкции языка, которые бросают исключения. `std::bad_alloc`. Функции стандартной библиотеки, которые бросают исключения.

4. **Шаблоны**

a. **Шаблоны функций**
Шаблоны функций. Инстанцирование шаблона. Автоматический вывод типа для шаблона функции. Ограничения, накладываемые на тип шаблона функции. Шаблонные аргументы по умолчанию. Шаблоны функций и перегрузка. Шаблоны с нетиповыми параметрами. Два этапа компиляции шаблона. Правила вывода типа шаблонной функции.

b. **`auto` и вывод типа**
Ключевое слово `auto`. Вывод типа при использовании `auto`. Различие этого вывода от вывода типа шаблонной функции (`std::initializer_list`). `auto` и возвращаемые значения. Trailing return type. Использование `auto` для параметров функций. Range-based цикл `for`. Структурное связывание. `decltype`.

c. **Шаблоны классов**
Шаблоны классов. Вывод шаблонных аргументов классов (CTAD). Зависимые имена в шаблонах. Использование ключевых слов `typename` и `template` для правильной интерпретации зависимых имён. Специализация шаблонов. Полная специализация. Частичная специализация. `std::vector<bool>`.

d. **Стандартные шаблонные классы**

i. Класс вектора `std::vector<T>`.

ii. Класс массива `std::array<T, Size>`. Чем `std::array` отличается от `std::vector`? Чем `std::array` отличается от массива языка C?

iii. Класс пары `std::pair`. Поля `first` и `second`. Создание пары. Сравнение пар.

iv. Класс `std::optional<T>`. Конструкторы класса `optional`. Методы класса `std::optional`:

- `operator*`
- `operator bool()`
- `value`
- `reset`
- `has_value`
- `value_or`

Объект `std::nullopt`.

e. **Вариативные шаблоны**
Пак типов. Пак параметров. Распаковка пака типов и пака параметров. Оператор `sizeof...`. Рекурсивная обработка пакета. Fold expressions. Использование оператора запятая в fold expressions.

5. **Контейнеры**

   a. **Итераторы**
   Идея итераторов. В чём преимущество итераторов по сравнению с обычным обходом структур данных? Операции, которые можно производить с итератором вектора. Обход стандартных контейнеров с помощью итераторов. Передача итераторов в функции. Константные и обратные итераторы. Методы `begin`, `end`, `cbegin`, `cend`, `rbegin` и `rend`.

   b. **Класс списка** `std::list`
   С помощью какой структуры данных реализован список `std::list`? Как устроен список, где и как хранятся данные в списке? Методы списка: `insert`, `erase`, `push_back`, `push_front`, `pop_back`, `pop_front`. Вычислительная сложность всех методов. Итератор списка `std::list<T>::iterator`. Операции, которые можно производить с итератором списка. Сортировка списка. Как удалить элементы списка во время прохода по нему? Контейнер `std::forward_list`.

   c. **Класс двухсторонней очереди** `std::deque`
   Как устроена двухсторонняя очередь, где и как хранятся данные в ней? Операций, которые можно с ней провести и их вычислительная сложность.

   d. **Контейнеры-множества**
   Контейнер `std::set` – множество. Его основные свойства. С помощью какой структуры данных он реализован? Методы `insert`, `erase`, `find`, `count`, `lower_bound`, `upper_bound` и их вычислительная сложность. Как изменить элемент множества? Контейнер `std::unordered_set` – неупорядоченное множество. Его основные свойства. С помощью какой структуры данных он реализован? Основные методы этого контейнера и их вычислительная сложность. Контейнеры `multuset` и `unordered_multuset`.

   e. **Контейнеры-словари**
   Контейнер `std::map` – словарь. Его основные свойства. С помощью какой структуры данных он реализован? Методы `insert`, `operator[]`, `erase`, `find`, `count`, `lower_bound`, `upper_bound` и их вычислительная сложность. Контейнер `std::unordered_map`. С помощью какой структуры данных реализован? Его основные свойства и методы и их вычислительная сложность. Как изменить ключ элемента словаря? Контейнеры `multumap` и `unordered_multumap`. Как удалить из `multimap` все элементы с данным ключом? Как удалить из `multimap` только один элемент с данным ключом?

   f. **Инвалидация итераторов**
   Инвалидация итераторов вектора. Инвалидация итераторов списка. Инвалидация итераторов множества и словаря.

   g. **Контейнерные адаптеры**
   Шаблонный параметр шаблона. Контейнерный адаптер `std::stack` и его методы `push`, `pop` и `top`. Почему метод `pop` не возвращает элемент? Контейнерный адаптер `std::queue` и его методы `push`, `pop`, `front` и `back`. Контейнерный адаптер `std::priority_queue` и его методы `push`, `pop` и `top`. Задание базового контейнера для адаптера.

   h. **Настройка множеств и словарей**
   Пользовательский компаратор для упорядоченных ассоциативных контейнеров. Пользовательский компаратор и пользовательская хеш-функция для неупорядоченнных ассоциативных контейнеров.

6. **Алгоритмы**

   a. **Основные алгоритмы**
   Библиотека `algorithm`. Стандартные шаблонные функции из этой библиотеки: `max_element`, `sort`, `reverse`, `copy`, `count`, `find`, `all_of`, `any_of`, `none_of`, `fill`, `unique`, `remove`. Библиотека `numeric`. Стандартные функции из этой библиотеки: `iota` и `accumulate`. Как написать подобные алгоритмы самостоятельно?

   b. **Output и Input итераторы**
   Output итераторы. Итератор `std::back_insert_iterator`. Как перегружены операторы для этого итератора? Использование функции `std::copy` и этого итератора для вставки элементов в контейнер. Итератор `std::ostream_iterator`, как перегружены операторы для этого итератора? Input итераторы. Итератор `std::istream_iterator`, как перегружены операторы для этого итератора?

   c. **Категории итераторов**
   Различие между итератором вектора и итератором списка. Какие операции можно применять к итератору вектора, но нельзя применять к итератору списка? Категории итераторов: Input, Output, Forward, Bidirectional, Random access. Допустимые операции для каждой категории итераторов. Привести пример итератора из каждой категории. Почему нельзя сортировать контейнер типа `std::list` с помощью стандартной функции `std::sort`? Функции `std::advance`, `std::next` и `std::distance`.

d. **Функциональные объекты**
Тип функция. Тип указатель на функцию. Функтор. Различие между функцией и функтором. Стандартные функторы: `std::less`, `std::greater`, `std::equal_to`, `std::plus` и другие. Лямбда-функции. Передача функциональных объектов в функции.

e. **Указатели на поля и методы**
Тип указателя на поле. Тип указателя на метод. Операторы `.*` и `->*`. Конвертация указателя на метод к функциональному объекту. Функция `std::mem_fn`.

f. **Алгоритмы, принимающие функциональные объекты**
Стандартные функции, принимающие функциональные объекты: `for_each`, `sort`, `stable_sort`, `find_if`, `count_if`, `all_of`, `generate`, `copy_if`, `transform`, `partition`, `stable_partition`. Как написать подобные алгоритмы самостоятельно?

g. **Лямбда-функции**
Простые лямбда-функции без захвата. Лямбда функции как функторы. Создание переменной лямбда-функции. Захват локальных переменных. Захват по ссылке и по значению. Опасность захвата по ссылке. Возвращаемый тип лямбда-функции. Захват `this` и `*this`. Создание и инициализация полей лямбда функции. Захват переменных с перемещением. Модификатор `mutable`.

h. **Класс `std::function`**
Хранение функциональных объектов в объекте класса `std::function`. Современный аналог – класс `std::copyable_function`.

7. **Move семантика**

a. **Перемещение**
Операция копирования. Что происходит при копировании (разберите случаи копирования скаляра, агрегата и нормального класса)? Операция перемешения. Операция перемещающего присваивания. Что происходит при перемещении (разбирите случаи перемещения скаляра, агрегата и нормального класса)? Стандартная функция `std::move`. Что происходит при перемещении объектов типа `int`, `std::string`, `std::vector` и `std::array`? В чём преимущества перемещения над копированием?

b. **lvalue-выражения и rvalue-выражения**
Что такое выражение? Тип выражения и категория выражения. Что такое lvalue-выражение? Что такое rvalue-выражение? Приведите примеры lvalue и rvalue выражений. Зачем нужно разделение выражений на lvalue и rvalue? Когда происходит перемещение? Передача lvalue и rvalue выражений в функции, принимающие по значению. Использование `std::move` при возврате из функции.

c. **lvalue-ссылки и rvalue-ссылки**
Разница между lvalue-ссылками и rvalue-ссылками. Инициализация ссылок. Возврат ссылок из функций. Приведение объектов к ссылкам. Перегрузка по категории выражения. Уметь написать функцию, которая печатает категорию переданого ей выражения. Какую категорию имеет выражение, состоящее только из одного идентификатора – rvalue-ссылки? Что на самом деле делает функция `std::move`? Перегрузка по квалификаторам ссылок.

d. **Особые методы, связанные с перемещением**
Конструктор перемещения и оператор присваивания перемещения. Создание класса, с пользовательским конструктором перемещения и пользовательским оператором перемещения.

8. **Умные указатели**

a. **Ошибки при работе с динамической памятью**
Утечки памяти. Утечки памяти при бросании исключений. Двойное удаление.

b. **Умный указатель `std::unique_ptr`**
Применение класса `std::unique_ptr`. Основные свойства `std::unique_ptr`. Методы `operator*`, `operator->`, `operator bool()`, `get`, `release`, `swap`.Ошибочное использование `std::unique_ptr` при его инициализации с помощью обычного указателя. Шаблонная функция `std::make_unique`. Перемещение объектов типа `unique_ptr`. Передача таких указателей в функции. Циклические ссылки.

c. **Умный указатель `std::shared_ptr`**
Класс `std::shared_ptr` и его отличие от `std::unique_ptr`. Шаблонная функция `std::make_shared`. Как схематически устроен указатель типа `std::shared_ptr`? Циклические ссылки.