Теория:

Часть 1:

1. **Пространство имён и ссылки**
   Пространство имён: что такое и зачем нужно. `using`-объявление. Анонимное пространство имён. Что такое ссылки. Различие ссылок и указателей. Ссылки на константу. Три типа передачи аргументов в функцию: передача по значению, передача по ссылке и передача по ссылке на константу. Преимущества/недостатки каждого метода. Возвращение ссылки из функции.

2. **Перегрузка функций**
   Сигнатуры функций в языках C и C++. Перегрузка функций. Манглирование имён. Ключевое слово `extern "C"`. Правила разрешения перегрузки функций.

3. **Перегрузка операторов**
   Перегрузка операторов в языке C++. Перегрузка арифметических операторов. Перегрузка унарных операторов. Перегрузка операторов как методов класса. Перегрузка оператора присваивания. Перегрузка оператора присваивания сложения. Реализация оператора сложения с помощью оператора присваивания сложения (`+=`). Перегрузка операторов ввода вывода `<<` и `>>` с `cin` и `cout`. Перегрузка оператора взятия индекса. Перегрузка операторов инкремента и декремента. Перегрузка оператора стрелочка (`->`). Перегрузка операторов `new` и `delete`. Перегрузка оператора вызова функции.

4. **Классы. Инкапсуляция**
   Что такое объектно-ориентированное программирование. Основные принципы ООП: инкапсуляция, композиция, наследование и полиморфизм. Классы. Поля и методы класса. Константные методы класса. Модификаторы доступа `private` и `public`. Указатель `this`. Различие ключевых слов `struct` и `class` в языке C++. Конструкторы и деструкторы. Список инициализации членов класса. Какие поля можно инициализировать с помощью списка инициализации, но нельзя инициализировать обычным образом. Перегрузка конструкторов. Конструктор по умолчанию. Конструктор копирования. Делегирующий конструктор. Ключевое слово `explicit`. Перегрузка оператора присваивания. Конструкторы и перегруженные операторы, создаваемые по умолчанию. Друзья. Ключевое слово `friend`.

5. **Инициализация, ключевое слово auto и другое**
   Инициализация. Default initialization. Value initialization. Direct initialization. Direct list initialization. Copy initialization. Copy list initialization. Ключевое слово `auto`. Range-based циклы. Пользовательские литералы. Structure bindings. Copy elision. Return value optimization.

6. **Динамическое создание объектов в Куче**
   Создание экземпляров класса в стеке и куче в языке C++. Использование операторов `new` и `delete`. Основные отличия `new` и `delete` от `malloc` и `free`. Операторы `new[]` и `delete[]`. Создание массива объектов в Куче. Оператор `placement new`.

7. **Реализация строки. Классы `std::string` и `std::string_view`**
   Реализация своей строки с выделением памяти в Куче. Методы такой строки:

   - Конструктор по умолчанию
   - Конструктор, принимающий строку в стиле C (`const char*`)
   - Конструктор копирования
   - Деструктор
   - Оператор присваивания
   - Оператор сложения
   - Оператор присваивания сложения(`+=`).

   Стандартная строка `std::string`. Преимущества строки `std::string` по сравнению со строкой в стиле `C`. Класс `std::string_view`. Строение объектов этого класса, его размер. Конструкторы этого класса. Методы `remove_prefix` и `remove_suffix`. В чём преимущество передачи `string_view` в функцию. Опасность возврата `string_view` из функции.

8. **Шаблоны.**
   Шаблонные функции. Использование шаблонных функции в языке C++. Шаблоны классов. Инстанцированием шаблона. Вывод шаблонных аргументов функций и классов. Специализация шаблона.

9. **STL. Контейнеры `std::vector` и `std::array`**
   Контейнер `std::vector`. С помощью какой структуры данных реализован. Как устроен вектор, где и как хранятся данные в векторе. Размер и вместимость вектора, методы `resize` и `reserve`. Методы `push_back`, `pop_back`, `insert`, `erase` и их вычислительная сложность. Когда происходит инвалидация итераторов вектора? Контейнер `std::array`. Как устроен, где и как хранятся данные в массиве.

10. **STL. Итераторы**
    Идея итераторов. В чём преимущество итераторов по сравнению с обычным обходом структур данных. Операции, которые можно производить с итератором. Категории итераторов (Random access, Biderectional, Forward, Output, Input). Обход стандартных контейнеров с помощью итераторов. Константные и обратные итераторы. Методы `begin`, `end`, `cbegin`, `cend` и другие. Итератор `std::back_insert_iterator`. Использование функции `std::copy` для вставки элементов в контейнер. Итератор `std::ostream_iterator`. Функции `std::advance`, `std::next` и `std::distance`.

11. **STL. Контейнеры `std::list` и `std::deque`**
    С помощью какой структуры данных реализован. Как устроен список, где и как хранятся данные в списке. Методы списка: `insert`, `erase`, `push_back`, `push_front`, `pop_back`, `pop_front`. Вычислительная сложность этих операций. Когда происходит инвалидация итераторов списка? Как удаляются элементы списка во время прохода по нему. Контейнер `std::deque`, как реализован, операций, которые можно с ним провести и их вычислительная сложность. Когда происходит инвалидация итераторов deque? Контейнеры адаптеры `std::stack`, `std::queue` и `std::priority_queue`.

12. **STL. 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. Можно ли 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. Когда происходит инвалидация итераторов множества?

13. **STL. Контейнеры-словари**
    Контейнер `std::map` – словарь. Его основные свойства. Методы `insert`, `operator[]`, `erase`, `find`, `count`, `lower_bound`, `upper_bound` и их вычислительная сложность. Контейнер `std::unordered_map`. С помощью какой структуры данных этот контейнер реализован. Его основные свойства и методы и их вычислительная сложность. Как изменить ключ элемента словаря? Когда происходит инвалидация итераторов словаря? Пользовательский компаратор для упорядоченных ассоциативных контейнеров. Контейнеры `multumap` и `unordered_multumap`. Как удалить из `multimap` все элементы с данным ключом. Как удалить из `multimap` только один элемент с данным ключом? Пользовательский компаратор и пользовательская хеш-функция для неупорядоченнных ассоциативных контейнеров.

14. **Move-семантика.**
    Глубокое копирование и поверхностное копирование. Копирование объекта. Перемещение объекта. Стандартная функция `std::move`. В чём преимущества перемещения над копированием? Перемещение объекта в функцию, если функция принимает объект по значению. Перемещение объекта при возврате из функции. Что такое выражение? lvalue-выражения и rvalue-выражения. Приведите примеры lvalue и rvalue выражений.

15. **Умные указатели.**
    Недостатки обычных указателей. Умный указатель `std::unique_ptr`. Шаблонная функция `std::make_unique`. Перемещение объектов типа `unique_ptr`. Умный указатель `std::shared_ptr`. Работа с таким указателем. Шаблонная функция `std::make_shared`. Базовая реализация `std::shared_ptr`. Умный указатель `std::weak_ptr`.

16. **rvalue-ссылки и универсальные ссылки**
    Что такое lvalue ссылки, а что такое rvalue ссылки, в чём разница? Зачем нужно разделение выражений на lvalue и rvalue. rvalue-ссылки. Что на самом деле делает функция `std::move`? Конструктор перемещения и оператор присваивания перемещения. Создание класса, с пользовательским конструктором перемещения и пользовательским оператором перемещения. Правило пяти. Правила свёртки ссылок. Универсальные ссылки, чем они отличаются от lvalue и rvalue ссылок? Реализация функции `std::move`. Идеальная передача. Функция `std::forward`.

Часть 2:

17. **Раздельная компиляция**
Что такое файл исходного кода и исполняемый файл. Этап сборки программы: препроцессинг, ассемблирование, компиляция и линковка. Директивы препроцессора `#include` и `#define`. Компиляция программы с помощью `g++`. Header-файлы. Раздельная компиляция. Преимущества раздельной компиляции. Статические библиотеки и их подключение с помощью компилятора `gcc`. Динамические библиотеки и их подключение.

18. **Событийно-ориентированное программирование и библиотека SFML**
Библиотека SFML. Класс `sf::RenderWindow`. Системы координат SFML (координаты пикселей, глобальная система координат, локальные системы координат). Методы `mapPixelToCoords` и `mapCoordsToPixel`. Основной цикл программы. Двойная буферизация. Понятие событий. Событийно-ориентированное программирование. События SFML: `Closed`, `Resized`, `KeyPressed`, `KeyReleased`, `MouseButtonPressed`, `MouseButtonReleased`, `MouseMoved`. Очередь событий. Цикл обработки событий.

19. **Наследование.**
Наследование в языке `C++`. Добавление новых полей и методов в наследуемый класс. Вызов конструкторов наследуемого класса. Модификатор доступа `protected`. Переопределение методов. Чем отличается переопределение от перегрузки? Ключевые слова `override` и `final`. Чистые виртуальные функции. Абстрактные классы и интерфейсы. Срезка объектов. Множественное наследование. Виртуальное множественное наследование.

20. **Полиморфизм.**
Полиморфизм в `C++`. Указатели на базовый класс, хранящие адрес объекта наследуемого класса. Виртуальные функции. Реализация механизма виртуальных функций. Таблица виртуальных функций. Виртуальный деструктор. Статический и динамический типы. Хранение объектов разных динамических типов в векторе. Оператор `dynamic_cast`. В каких случая он используется? Что происходит если `dynamic_cast` не может привести тип? Рассмотрите случай приведения указателей и случай приведения ссылок. Использование `static_cast` для приведения типов и указателей на типы в иерархии наследования.

21. **Функциональные объекты**
Указатели на функции в алгоритмах STL. Функторы. Стандартные функторы: `std::less`, `std::greater`, `std::equal_to`, `std::plus`, `std::minus`, `std::multiplies`. Основы лямбда-функций. Стандартные алгоритмы STL, принимающие функциональные объекты. Тип обёртка `std::function`. Шаблонная функция `std::bind`.

22. **Лямбда-функций**
Лямбда-функций. Объявление лямбда-функций. Передача их в другие функции. Преимущества лямбда-функций перед указателями на функции и функторами. Использование лямбда функций со стандартными алгоритмами `std::sort`, `std::transform`, `str::copy_if`. Лямбда-захват. Захват по значению и по ссылке. Захват всех переменных области видимости по значению и по ссылке. Объявление новых переменных внутри захвата.

23. **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: `assert`, использование глобальной переменной, коды возврата и исключения. Преемущества и недостатки каждого из этих методов. Какие из этих методов желательно использовать для внутренних ошибок, а какие для внешних?

24. **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`. Гарантии безопасности исключений. Исключения при перемещении объектов. `move_if_noexcept`. Идиома `copy and swap`.

25. **Реализация вектора.**
Реализация своего вектора `mipt::Vector<T>` (аналога `std::vector<T>`). Нужно также предусмотреть итераторы этого вектора: `mipt::Vector<T>::iterator`, а также константные и обратные итераторы.
Методы такого вектора:

    - Конструктор по умолчанию
    - Конструктор, принимающий количество элементов
    - Конструктор, принимающий количество элементов и значение элемента
    - Конструктор от `std::initializer_list`.
    - Конструктор копирования
    - Конструктор перемещения
    - Деструктор
    - Оператор присваивания копирования
    - Оператор присваивания перемещения
    - Оператор взятия индекса (`operator[]`)
    - Метод `at`, аналог метода `at` класса `std::vector`
    - Методы `size`, `capacity`, `empty`, `reserve`, `resize`, `shrink_to_fit`.
    - Методы `push_back`, `emplace_back`, `pop_back`.
    - Методы для работы с итераторами `begin`, `end`, `rbegin`, `rend`.

    Безопасность относительно исключений у такого вектора.

26. **Система типов языка `C++`.**
Система типов языка `C++`. Встроенные типы, массивы, структуры, объединения, перечисления, классы, указатели, ссылки, функциональные объекты (функции, указатели и ссылки на функции, функторы, лямбда-функции), указатели на члены класса, битовые поля. Вывод типа выражения с помощью `decltype`. Различие вывода с помощью `decltype`, `auto` и вывода шаблонных аргументов. Разложение типов (type decay) и когда он происходит.

27. **Приведение типов**
В чём недостатки приведения в стиле `C`? Оператор `static_cast` и в каких случая он используется. Операторы `reinterpret_cast` и `const_cast` и в каких случая они используется.

28. **Классы `std::any`, `std::optional` и `std::variant`**
Класс `std::any`. Функция `std::any_cast`. Класс `std::optional`. Методы класса `std::optional`:

    - Конструкторы
    - Методы `value`, `has_value`, `value_or`.
    - Унарные операторы `*` и `->`
    - Оператор преобразования к значению типа `bool`.

    Для чего можно применять `std::optional`?

    Класс `std::variant`. Функции для работы с `std::variant`:

    - `std::get`
    - `std::holds_alternative`
    - `std::visit`

    Для чего можно применять `std::variant`?

29. **Вычисления на этапе компиляции.** `constexpr`
Вычисление на этапе компиляции. В чём преимущества вычисления на этапе компиляции по сравнению с вычислением на этапе выполнения. Ключевое слово `constexpr`. Что означает `constexpr` при объявлении переменной? Что означает `constexpr` при определении функции? Разница между `const` и `constexpr`. Ключевые слова `consteval` и `constinit`. `static_assert`.

30. **Вычисления на этапе компиляции. Шаблонное метапрограммирование.**
Полная специализация шаблона. Частичная специализация шаблона. Что такое шаблонные метафункции и зачем они нужны? Использование специализации шаблона для написание следующих метафункций:

- `IsInt` - проверяет, является ли тип `T` типом `int`.
- `IsIntegral` - проверяет, является ли тип `T` целочисленным типом.
- `IsPointer` - проверяет, является ли тип `T` указателем.
- `IsSame` - проверяет, являются ли 2 типа `T1` и `T2` одинаковыми.
- `RemovePointer` - если тип `T` является указателем, то возвращает тип того, на что такой указатель указывает (то есть убирает одну "звёздочку"у типа).
- `IsHasBegin` - проверяет, есть ли у типа `T` метод begin.

Что такое концепты, как их использовать и зачем они нужны?