Язык C++. Функции

Александра Волосова,
к.т.н., доцент кафедры
ИУ5

Презентация разработана в рамках гранта «Грант на обучение студентов по образовательным программам высшего образования для топ-специалистов в сфере информационных технологий. Договор № 70-2025-000850 с АНО «Аналитический центр при Правительстве Российской Федерации»

План лекции

1. Зачем нужны функции? Принцип DRY
2. Объявление, определение, вызов
3. Сигнатура функции. Прототипы
4. Передача параметров: по значению
5. Передача по ссылке
6. Передача по указателю
7. Возвращаемые значения
8. Перегрузка функций
9. Параметры по умолчанию
10. Указатели на функции
11. Рекурсия
12. Области видимости
13. Время жизни переменных
14. Inline функции
15. constexpr функции
16. Шаблонные функции
17. Lambda выражения
18. noexcept функции
19. Атрибуты функций
20. Best Practices
21. Оптимизация функций
22. Отладка функций
23. Тестирование функций
24. Современный C++20/23
25. Заключение

1. Зачем нужны функции? Принцип DRY

Принцип DRY (Don't Repeat Yourself):
• Исключение дублирования кода
• Единая точка изменения

Структурирование программы:
• Разбиение сложной задачи на подзадачи
• Иерархическая организация кода

Абстракция:
• Сокрытие сложной реализации
• Простой интерфейс использования

Упрощение тестирования:
• Изолированное тестирование компонентов
• Unit testing*

Повторное использование:
• Библиотеки функций
• Кросспроектное использование

Unit testing
(модульное тестирование, юнит-тестирование)

- — это процесс в разработке программного обеспечения, при котором отдельные небольшие части приложения (называемые **«юнитами»** или **«модулями»**) проверяются на корректность работы
- **Unit testing** — это практика написания автоматических тестов для небольших, изолированных частей кода, которая позволяет создавать более надежное, предсказуемое и легко сопровождаемое программное обеспечение. Это неотъемлемая часть профессиональной разработки и таких подходов, как **TDD (Test-Driven Development — Разработка через тестирование)**
- **Это не интеграционные тесты.** Интеграционные тесты проверяют взаимодействие нескольких модулей между собой (например, работает ли связь вашего сервиса с реальной базой данных).
- **Это не end-to-end (E2E) тесты.** E2E-тесты проверяют работу всего приложения целиком от начала до конца с точки зрения пользователя (например, «нажать кнопку → увидеть всплывающее окно»)

Пример

Допустим, у нас есть функция, которую мы хотим протестировать:

 # calculator.py (наш модуль)
def add(a, b):
return a + b

Unit-тест для этой функции (используя популярный фреймворк pytest) будет выглядеть так:

```
# test_calculator.py (наш тест)
import pytest
from calculator import add  # импортируем функцию, которую тестируем

def test_add_positive_numbers():
    # Arrange (Подготовка): задаем входные данные и ожидаемый результат
    a = 5
    b = 3
    expected_result = 8

    # Act (Действие): вызываем тестируемый код
    actual_result = add(a, b)

    # Assert (Проверка): проверяем, что результат соответствует ожиданию
    assert actual_result == expected_result

def test_add_negative_numbers():
    assert add(-1, -2) == -3

def test_add_zero():
    assert add(5, 0) == 5
```

Ключевые характеристики Unit-тестов:

- **Изоляция (Isolation):** Это самое важное. Юнит-тест проверяет *только один конкретный модуль* в полной изоляции от всех остальных частей программы.
  - **Как достигается?** Внешние зависимости (базы данных, файловая система, сетевые запросы, другие классы) не используются. Вместо них подставляются **«заглушки» (mocks, stubs)**. Это специальные объекты, которые *имитируют* поведение реальных зависимостей.
  - **Цель:** Если тест упал, вы *точно знаете*, что проблема в коде именно этого модуля, а не в базе данных, которая временно не работает.
- **Маленький и быстрый:** Юнит-тесты должны выполняться миллисекунды. Поскольку их сотни или тысячи, вы можете запускать их постоянно и получать мгновенную обратную связь.
- **Детерминированность:** Результат unit-теста всегда должен быть одинаковым при одинаковых входных данных. Он не должен зависеть от внешнего состояния (например, от времени суток или данных в базе).
- **Что такое «Юнит» (Модуль)?**
- Чаще всего «юнитом» является:
- **Одна функция** (в функциональном программировании)
- **Один метод** класса (в объектно-ориентированном программировании)
- **Один класс**
- **Зачем это нужно? (Цели и преимущества)**
- **Раннее обнаружение ошибок:** Ошибки находятся на самом раннем этапе, сразу после написания кода. Исправлять их в этот момент в десятки раз дешевле и быстрее, чем на этапе тестирования всего приложения.
- **Облегчение рефакторинга:** Рефакторинг (изменение структуры кода без изменения его поведения) всегда рискован. Unit-тесты служат «страховочной сеткой». Если после изменений все тесты проходят, вы с высокой долей уверенности можете сказать, что не сломали существующую функциональность.
- **Живая документация:** Хорошо написанные тесты наглядно показывают, *как именно* должен использоваться тот или иной модуль и что от него ожидается. Это гораздо более точная и актуальная документация, чем та, что лежит в Word-файле и быстро устаревает.
- **Улучшение дизайна кода:** Чтобы код было легко тестировать, его приходится писать хорошо структурированным, с четкими границами ответственности и минимальными зависимостями. Таким образом, сама необходимость писать unit-тесты заставляет разработчика писать более качественный код.

2. Объявление, определение, вызов

```
// ОБЪЯВЛЕНИЕ (прототип) в .h файле
int calculate_sum(int a, int b);
double compute_average(const double* data, size_t size);

// ОПРЕДЕЛЕНИЕ в .cpp файле
int calculate_sum(int a, int b) {
    return a + b;
}

double compute_average(const double* data, size_t size) {
    double sum = 0.0;
    for (size_t i = 0; i < size; ++i) {
        sum += data[i];
    }
    return sum / size;
}

// ВЫЗОВ в программе
int result = calculate_sum(5, 3);
double avg = compute_average(values, count);
```

3. Сигнатура функции. Прототипы

СИГНАТУРА включает:
• Имя функции
• Количество параметров
• Типы параметров (порядок важен!)
• Квалификаторы (const, volatile)

НЕ входит в сигнатуру:
• Возвращаемый тип
• Имена параметров

ПРОТОТИПЫ необходимы для:
• Раздельной компиляции
• Проверки типов на этапе компиляции
• Создания библиотек

Пример разных сигнатур:

```
void process(int value);
void process(double value);
void process(int value, int precision);
void process(const std::string& text);
void process(int value) const; // Для методов
```

4. Передача параметров: по значению

```
// Создается копия параметра
void modify_value(int x) {
    x = 100; // Изменяется копия
    std::cout << "Inside: " << x << std::endl;
}

// Использование
int number = 5;
modify_value(number);
std::cout << "Outside: " << number << std::endl;
// Output: Inside: 100, Outside: 5
```

Плюсы:

- Простота использования
- Безопасность - оригинал защищен

Минусы:

- Накладные расходы на копирование
- Не подходит для больших объектов

Рекомендации:

- Используйте для простых типов (int, double, char)
- Избегайте для больших структур и классов

5. Передача по ссылке

```
// Работа с оригинальным объектом
void modify_reference(int &x) {
    x = 100; // Изменяется оригинал
    std::cout << "Inside: " << x << std::endl;
}

// Использование
int number = 5;
modify_reference(number);
std::cout << "Outside: " << number << std::endl;
// Output: Inside: 100, Outside: 100
```

Плюсы:
- Эффективность - нет копирования
- Возможность модификации

Минусы:
- Риск непреднамеренных изменений
- Может быть менее безопасно

Константные ссылки:

```
void read_only(const std::string& str) {
    // str нельзя изменить
    std::cout << str.length();
}
```

6. Передача по указателю

```
// Работа через указатель
void modify_pointer(int *x) {
    if (x != nullptr) { // Всегда проверяйте!
        *x = 100; // Изменяем значение по указателю
    }
}
// Использование
int number = 5;
modify_pointer(&number);
std::cout << "Result: " << number << std::endl; // 100
```

Плюсы:
- Явное указание на возможность изменения
- Возможность передачи nullptr
- Поддержка legacy C кода

Минусы:
- Синтаксически сложнее
- Риск nullptr dereference
- Менее безопасно чем ссылки

Современный подход:
- Предпочитайте ссылки над указателями
- Используйте умные указатели для владения

7. Возвращаемые значения

```
// Возврат по значению (рекомендуется)
std::string create_greeting(const std::string& name) {
    return "Hello, " + name + "!";
}

// Возврат по ссылке (осторожно!)
const std::string& get_default_name() {
    static const std::string name = "Guest";
    return name; // Безопасно - static живет всегда
}
// Возврат по указателю (еще осторожнее!)
int* create_array(size_t size) {
    return new int[size]; // Вызывающий должен удалить!
}
// Современный C++: умные указатели
std::unique_ptr<int[]> create_safe_array(size_t size) {
    return std::make_unique<int[]>(size); // Автоматическое управление
}
// Multiple return values (C++17)
std::tuple<int, double, std::string> get_data() {
    return {42, 3.14, "result"};
}
```

8. Перегрузка функций

```
// Разные типы параметров
int add(int a, int b) { return a + b; }
double add(double a, double b) { return a + b; }
std::string add(const std::string& a, const std::string& b) {
    return a + b;
}

// Разное количество параметров
int sum(int a, int b) { return a + b; }
int sum(int a, int b, int c) { return a + b + c; }
int sum(const std::vector<int>& numbers) {
    return std::accumulate(numbers.begin(), numbers.end(), 0);
}

// Разный порядок параметров
void print(int count, const std::string& message) { /* ... */ }
void print(const std::string& message, int count) { /* ... */ }

// Ограничения:
• Возвращаемый тип НЕ учитывается!
• Только const/volatile квалификаторы не достаточно
• Разные пространства имен - разные функции
```

9. Параметры по умолчанию

```
// Объявление в заголовочном файле
void draw_rectangle(int width,
                    int height = 10,
                    const std::string& color = "black",
                    bool filled = true);

// Определение в .cpp файле (без значений по умолчанию!)
void draw_rectangle(int width, int height,
                    const std::string& color, bool filled) {
    // Реализация...
}

// Разные способы вызова:
draw_rectangle(100, 50); // height=50, color="black", filled=true
draw_rectangle(100);     // height=10, color="black", filled=true
draw_rectangle(100, 20, "red"); // color="red", filled=true
draw_rectangle(100, 30, "blue", false); // все параметры
```

Правила:

- Параметры по умолчанию должны быть в конце
- Значения по умолчанию указываются только в объявлении
- Можно иметь несколько объявлений с разными значениями по умолчанию

10. Указатели на функции

```
// Традиционный синтаксис
typedef int (*MathFunction)(int, int);

// Современный синтаксис (C++11)
using MathFunction = int (*)(int, int);

// Функции для примера
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }

// Использование
MathFunction operation = add;
int result = operation(5, 3); // 8

// Массив функций
MathFunction operations[] = {add, subtract, multiply};
for (auto func : operations) {
    std::cout << func(10, 2) << std::endl; // 12, 8, 20
}
```

11. Рекурсия

```
// Факториал - классический пример
int factorial(int n) {
    if (n <= 1) return 1;           // Базовый случай
    return n * factorial(n - 1);    // Рекурсивный вызов
}

// Числа Фибоначчи (неэффективная версия)
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// Хвостовая рекурсия (может быть оптимизирована)
int factorial_tail(int n, int accumulator = 1) {
    if (n <= 1) return accumulator;
    return factorial_tail(n - 1, n * accumulator);
}

// Рекурсия с мемоизацией
std::unordered_map<int, int> cache;
int fibonacci_memo(int n) {
    if (n <= 1) return n;
    if (cache.find(n) != cache.end()) return cache[n];
    cache[n] = fibonacci_memo(n-1) + fibonacci_memo(n-2);
    return cache[n];
}
```

```
// Когда использовать рекурсию:
• Деревья и графы
• Разделяй и властвуй алгоритмы
• Backtracking задачи
```

12. Области видимости

```
// Глобальная область видимости
int global_counter = 0;

void example_function() {
    // Локальная область видимости функции
    int local_var = 10;

    { // Блоковая область видимости
        int block_var = 5;
        std::cout << local_var + block_var << std::endl; // OK: 15
    }
    // std::cout << block_var << std::endl; // Ошибка: block_var не виден
}

// Пространства имен
namespace math {
    const double PI = 3.14159;

    namespace geometry {
        double circle_area(double radius) {
            return PI * radius * radius;
        }
    }
}

// Анонимное пространство имен (только в этом файле)
namespace {
    int file_local_variable = 42;
}
```

```
// Правила видимости:
• Внутренние области видят внешние
• Внешние области не видят внутренние
• Имена могут быть скрыты (shadowing)
```

13. Время жизни переменных

```
// Автоматическое (стековое) время жизни
void automatic_example() {
    int x = 10; // Создается при входе в функцию
    // Уничтожается при выходе из функции
}

// Статическое время жизни
void static_example() {
    static int counter = 0; // Инициализируется один раз
    counter++; // Сохраняется между вызовами
}

// Динамическое время жизни (куча)
void dynamic_example() {
    int* ptr = new int(42); // Создается в куче
    // Живет до explicit delete
    delete ptr; // Явное освобождение
}

// Thread-local время жизни (C++11)
thread_local int thread_specific = 0;

// Время жизни объектов:
• Automatic: до конца блока
• Static: вся программа
• Dynamic: до delete
• Thread: до конца потока

// Умные указатели для автоматического управления:
std::unique_ptr<int> smart_ptr = std::make_unique<int>(42);
```

14. Inline функции

```
// inline предложение компилятору
inline int square(int x) {
    return x * x;
}

// Современный подход: компилятор сам решает
int cube(int x) {
    return x * x * x;
} // Может быть inlined автоматически

// Inline в заголовочных файлах
// math.h:
inline int max(int a, int b) {
    return (a > b) ? a : b;
}
```

```
// Плюсы inline функций:
• Устранение накладных
расходов вызова
• Возможность оптимизации
• Подходят для маленьких
функций

// Минусы:
• Увеличение размера кода
• Может замедлить выполнение
(cache misses)
• Перекомпиляция при
изменении

// Когда использовать
inline:
• Маленькие функции (1-3
строки)
• Часто вызываемые функции
• Функции в заголовочных
файлах
```

15. constexpr функции

```
// constexpr - вычисление на этапе компиляции
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}

// Использование во время компиляции
constexpr int fact_5 = factorial(5); // 120
int array[factorial(3)]; // Массив размером 6

// constexpr с C++14 допускает больше:
constexpr int sum_squares(int n) {
    int result = 0;
    for (int i = 1; i <= n; ++i) {
        result += i * i;
    }
    return result;
}

// constexpr if (C++17)
template<typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>) {
        return *t;
    } else {
        return t;
    }
}
```

```
// Преимущества:
• Вычисления во время компиляции
• Нет runtime накладных расходов
• Безопасность типов
• Возможность использования в константных выражениях
```

16. Шаблонные функции

```
// Базовый шаблон
template<typename T>
T max(T a, T b) {
    return (a > b) ? a : b;}

// Несколько параметров
template<typename T, typename U>
auto mixed_max(T a, U b) {
    return (a > b) ? a : b;}

// Шаблонные параметры не базового типа
template<typename T, int Size>
class FixedArray {
    T data[Size];
public:
    T& operator[](int index) { return data[index]; }};

// Явное инстанцирование.
template int max<int>(int, int);
template class FixedArray<double, 10>;

// SFINAE (Substitution Failure Is Not An Error)
template<typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
process_integral(T value) {
    return value * 2;}

// Concepts (C++20) - улучшение шаблонов
template<std::integral T>
T safe_divide(T a, T b) {
    if (b == 0) throw std::invalid_argument("Division by zero");
    return a / b;}
```

1. Явное инстанцирование (Explicit Instantiation)

- **Способ** явно **попросить компилятор сгенерировать код** для конкретной версии шаблонной функции или класса в данной единице трансляции (.cpp файле).
- **Зачем это нужно?**
- **Контроль над компиляцией:** Обычно код для шаблонов генерируется в каждом .cpp файле, где они используются (неявное инстанцирование). Это может увеличивать время компиляции.
- **Сокращение времени сборки:** Вы можете "принудительно" создать все нужные версии в одном .cpp файле (например, templates.cpp), а в других файлах просто объявить их как extern. Это предотвращает многократную компиляцию одного и того же шаблонного кода.
- **Предотвращение ошибок линковки:** Гарантирует, что нужная версия шаблона будет доступна линковщику.

Пример:

// Явно просим компилятор сгенерировать функцию max для типа int template int max<int>(int, int); *// Явно просим компилятор сгенерировать класс FixedArray с параметрами double и 10* template class FixedArray<double, 10>;

2. SFINAE (Substitution Failure Is Not An Error)

- **Ф**ундаментальное правило компиляции шаблонов в C++. Оно гласит: **"Если подстановка параметра шаблона при инстанцировании приводит к некорректному коду, это не ошибка компиляции, а просто исключение этого кандидата из множества перегрузок"**.

Когда компилятор ищет, какую перегруженную функцию или шаблон вызвать, он пробует "подставить" типы аргументов. Если при этой подстановке в каком-то шаблоне получается бессмыслица (например, попытка создать ссылку на void или вызвать несуществующий метод), компилятор не ругается, а просто тихо отбрасывает этот вариант и ищет дальше.

- **Зачем это нужно?**
  Чтобы писать шаблонный код, который по-разному ведет себя для разных типов, и создавать перегрузки, которые работают только с определенными категориями типов.

Пример:

```
#include <type_traits> // Для std::enable_if и std::is_integral // Эта версия функции process_integral будет инстанцирована ТОЛЬКО для целочисленных типов (int, char, long, etc.)
template<typename T> // Возвращаемый тип: enable_if<условие, тип>::type. // Если условие true, то возвращаемый тип - T.
 // Если условие false, то возникает ошибка подстановки (SFINAE), и эта функция исключается из consideration.
typename std::enable_if<std::is_integral<T>::value, T>::type
                    process_integral(T value) {
 return value * 2; }
// Для нецелочисленных типов (double, std::string, etc.) компилятор не найдет подходящую перегрузку process_integral
// и выдаст ошибку, если мы попытаемся её вызвать.
```

- **Как работает std::enable_if?**
  Это простой шаблон, реализованный примерно так:

```
template<bool B, class T = void>
struct enable_if {};
template<class T> // Специализация для true
struct enable_if<true, T> {
 typedef T type; }; //Если B = true, то у enable_if есть член type, равный T. Если B = false, то у enable_if нет члена type. Попытка обратиться к enable_if<false, T>::type вызовет ошибку подстановки, которую SFINAE благополучно "поглотит".
```

- **Недостатки SFINAE:** Синтаксис очень громоздкий и сложный для чтения.

Concepts (C++20) - улучшение шаблонов

- **Н**овая языковая возможность, которая призвана заменить и кардинально улучшить технику SFINAE. Concepts позволяют **явно указывать ограничения (constraints)** на параметры шаблонов.
- **Зачем это нужно?**
  - **Читаемость:** Код становится чище и нагляднее.
  - **Понятные сообщения об ошибках:** Вмезаместо многостраничных ошибок из недр шаблонов вы получаете четкое сообщение: "ошибка: std::string не удовлетворяет ограничению integral".
  - **Простота использования:** Гораздо проще выражать сложные требования к типам.

```
#include <concepts> // Для std::integral
// Эта функция будет инстанцирована ТОЛЬКО для целочисленных типов.
// Синтаксис намного чище и понятнее, чем SFINAE!

template<std::integral T> // Ограничение прямо в объявлении шаблона
T safe_divide(T a, T b) {
    if (b == 0) throw std::invalid_argument("Division by zero");
    return a / b;
}
```

Сравнение SFINAE и Concepts:

Характеристика	SFINAE	Concepts (C++20)
Синтаксис	Громоздкий, typename...::type	Чистый, template<concept_name T>
Читаемость	Низкая, намерения скрыты	Высокая, намерения явны
Ошибки	Длинные и запутанные	Короткие и понятные
Мощность	Мощный, но сложный в использовании	Такой же мощный, но гораздо более простой

17. Lambda выражения (C++11)

```
// Базовая lambda
auto simple = []() { std::cout << "Hello Lambda!"; };
simple();

// Lambda с параметрами
auto adder = [](int a, int b) { return a + b; };
int sum = adder(5, 3); // 8

// Захват переменных
int x = 10, y = 20;
auto capture_by_value = [x, y]() { return x + y; };
auto capture_by_reference = [&x, &y]() { x++; y++; };
auto capture_all_by_value = [=]() { return x + y; };
auto capture_all_by_reference = [&]() { x++; y++; };

// mutable lambda
auto counter = [count = 0]() mutable { return ++count; };

// Возвращаемый тип
auto precise_division = [](double a, double b) -> double {
    if (b == 0) return 0;
    return a / b;};

// Использование с алгоритмами
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::sort(numbers.begin(), numbers.end(),
          [](int a, int b) { return a > b; });
// Generalized lambda (C++14)
auto generic_add = [](auto a, auto b) { return a + b; };
```

18. noexcept функции

```
// Функция, которая не создает исключения
void safe_function() noexcept {
    // Гарантированно не создает исключений}
// noexcept с условием (C++11)
template<typename T>
void swap(T& a, T& b) noexcept(std::is_nothrow_move_constructible_v<T> &&
                               std::is_nothrow_move_assignable_v<T>) {
T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);}
```

Что значит "Создать исключение"?
Сначала вспомним, что такое исключение. Исключение (exception) — это механизм для обработки ошибок и нештатных ситуаций. Когда функция сталкивается с проблемой, которую не может решить самостоятельно, она может "бросить" исключение с помощью оператора throw, чтобы сигнализировать об этом вызывающему коду.
Пример функции, которая МОЖЕТ создать исключение:
cpp

```
int divide(int a, int b) {
if (b == 0) { throw
std::invalid_argument("Division by
zero!"); // Создаем исключение }
return a / b; }
```

Функция, которая не создает исключения, обещает, что ни при каких обстоятельствах (или при корректных входных данных) оператор throw внутри нее не будет выполнен. Либо же она сама обрабатывает все возможные ошибки внутри себя другими способами.
Пример функции, которая НЕ создает исключения:

```
// Вариант 1: Гарантия с помощью noexcept
int safe_divide(int a, int b) noexcept {
    // Вместо исключения возвращаем
специальное значение или используем
std::optional
    if (b == 0) {
        return 0; // Просто возвращаем 0
вместо броска исключения }
    return a / b;}

// Вариант 2: Функция настолько проста,
что в ней просто нечему ломаться
int increment(int x) noexcept {
    return x + 1; // Никаких ошибок здесь
произойти не может}
```

```
// Преимущества noexcept:
• Возможность оптимизации
компилятором
• Лучшая документация кода
• Использование в constexpr функциях
• Требование для некоторых операций
move

// Проверка noexcept
static_assert(noexcept(safe_function(
)), "Function should be noexcept");

// Если noexcept функция все же
создаст исключение:
• Вызывается std::terminate()
• Программа аварийно завершается

// Рекомендации:
• Помечайте noexcept функции, которые
не создает исключения
• Используйте для move
конструкторов/операторов
• Используйте для деструкторов
```

Ключевое слово noexcept

- В C++ эта гарантия выражается с помощью ключевого слова **noexcept**.
- **Для чего объявлять функцию с noexcept?**
- **Оптимизация компилятора:** Компилятор знает, что ему не нужно генерировать дополнительный код для раскрутки стека (stack unwinding) при исключениях внутри этой функции. Это может сделать код более эффективным.
- **Договор с пользователем:** Это явное обещание другим программистам, что ваша функция безопасна и не потребует блоков try/catch.
- **Требование для move-операций:** Во многих случаях реализации стандартной библиотеки C++ используют более эффективные move-конструкторы и move-операторы присваивания (вместо копирования) только если они объявлены как noexcept.
- **Безопасность и предсказуемость:** Вы знаете, что функция либо выполнится успешно, либо, в худшем случае, завершит программу (если случится непредвиденное), но не создаст исключение, которое вы не ожидаете.

19. Атрибуты функций (C++11/14/17)

```
// [[noreturn]] - функция не возвращает управление
[[noreturn]] void fatal_error(const std::string& message) {
    std::cerr << message << std::endl;
    std::exit(1);
}

// [[deprecated]] - устаревшая функция
[[deprecated("Use new_function() instead")]]
void old_function() { /* ... */ }

// [[nodiscard]] - результат нельзя игнорировать
[[nodiscard]] int create_resource() {
    return 42; // Ресурс который нужно освободить
}
// create_resource(); // Предупреждение: результат игнорируется

// [[maybe_unused]] - подавление предупреждений
void unused_function([[maybe_unused]] int parameter) {
    // parameter может не использоваться
}

// Атрибуты для оптимизации
__attribute__((always_inline)) void force_inline() { /* ... */ }
__attribute__((hot)) void frequently_called() { /* ... */ }
__attribute__((cold)) void rarely_called() { /* ... */ }

// Стандартные атрибуты (переносимы между компиляторами)
```

20. Лучшие практики

Именование функций:
• Глаголы или глагольные фразы: calculate_total(), process_data()
• Ясные и описательные имена
• Единый стиль именования (camelCase или snake_case)

Размер функций:
• 10-20 строк идеально
• Максимум 50-100 строк
• Одна ответственность на функцию

Параметры:
• Максимум 3-4 параметра
• Используйте структуры для групп параметров
• const ссылки для больших объектов
• Значения для простых типов

Документация:
• Doxygen комментарии
• Описание пред- и постусловий
• Описание исключений
• Примеры использования

Безопасность:
• Проверка входных параметров
• Обработка ошибок
• Гарантии исключений (noexcept)
• const корректность

21. Оптимизация функций

Inline оптимизация:
• Маленькие функции (1-3 строки)
• Часто вызываемые функции
• Используйте __force_inline если необходимо

Оптимизация передачи параметров:
• const& для больших объектов
• Значения для маленьких объектов (до 2-3 машинных слов)
• Rvalue ссылки для move семантики

Оптимизация возвращаемых значений:
• Return Value Optimization (RVO)
• Named Return Value Optimization (NRVO)
• Используйте move семантику для больших объектов

Кэш-дружественный код:
• Локализация данных
• Предсказуемый доступ к памяти
• Избегайте прыжков по памяти

Профилирование:
• Используйте perf, vtune, valgrind
• Оптимизируйте только hot paths
• Измеряйте до и после оптимизаций

Компиляторные оптимизации:
• -O2, -O3 для production
• LTO (Link Time Optimization)
• PGO (Profile Guided Optimization)

24. Современный C++20/23

```
Concepts (C++20):
template<std::integral T>
T square(T x) { return x * x; }

Ranges (C++20):
auto result = numbers | std::views::filter([](int x) { return x % 2 == 0; })
                      | std::views::transform([](int x) { return x * x; });

Format (C++20):
std::string message = std::format("Hello {}, your score is {}", name, score);

Modules (C++20):
import math; // Вместо #include <cmath>

Coroutines (C++20):
generator<int> range(int start, int end) {
    for (int i = start; i < end; ++i) {
        co_yield i;}}

constexpr улучшения (C++20/23):
constexpr std::vector<int> create_data() { /* ... */ }
constexpr virtual функции

Pattern matching (C++23):
auto result = inspect(value) {
    <int> i => std::format("int: {}", i),
    <std::string> s => std::format("string: {}", s),
    _ => "unknown type"
};
```

25. Заключение

Ключевые принципы работы с функциями в C++:

• Модульность: Разбивайте программу на небольшие, focused функции
• Переиспользование: Следуйте принципу DRY (Don't Repeat Yourself)
• Безопасность: Используйте const корректность и проверку параметров
• Производительность: Правильно выбирайте способы передачи параметров
• Читаемость: Давайте функциям понятные имена, документируйте их

Современные возможности C++:
• Lambda выражения для анонимных функций
• constexpr для вычислений во время компиляции
• Шаблоны и Concepts для обобщенного программирования
• Модули для лучшей инкапсуляции
• Ranges для функционального стиля

Лучшие практики:
• Одна функция - одна ответственность
• Максимум 3-4 параметра на функцию
• Ясные имена и хорошая документация
• Полное тестовое покрытие
• Постоянный рефакторинг и улучшение

Ресурсы для изучения

Книги:

- "Язык программирования C++" - Бьярн Страуструп
- "Эффективный современный C++" - Скотт Мейерс
- "Учебник по C++" - Стэнли Липпман

Онлайн ресурсы:

- cppreference.com - полная информация
- learncpp.com - учебник для начинающихх
- isocpp.org - официальный сайт C++

Статьи и руководства:

- **C++ Core Guidelines**
- Документация Microsoft C++