



Указатели на функции в C++

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»

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

План лекции

1. Концепция указателей на функции
2. Синтаксис объявления и инициализации
3. Вызов функций через указатели
4. Практические примеры использования
5. Указатели на функции как параметры
6. Возврат указателей на функции
7. Массивы указателей на функции
8. typedef и using для упрощения
9. Указатели на функции-члены класса
10. Сравнение с std::function
11. Лямбда-выражения
12. Обратные вызовы (Callbacks)
13. Pattern: Strategy
14. Best practices
15. Типичные ошибки

1. Концепция указателей на функции

- Указатель на функцию хранит адрес функции в памяти
- Аналогичен обычным указателям, но для исполняемого кода
- Позволяет вызывать функции динамически во время выполнения
- Основывается на принципе: "код тоже данные"
- Мощный инструмент для создания гибких архитектур

Пример использования:

- Механизмы обратных вызовов (callbacks)
- Системы плагинов
- Реализация паттернов проектирования

2. Синтаксис объявления

// Общий синтаксис:

```
return_type (*pointer_name) (parameter_types);
```

// Конкретные примеры:

```
int (*math_op) (int, int);
```

```
// Для int func(int, int)
```

```
double (*transform) (double);
```

```
// Для double func(double)
```

```
void (*callback) (const char*);
```

```
// Для void func(const char*)
```

// typedef для упрощения:

```
typedef int (*MathFunction) (int, int);
```

```
MathFunction add_ptr = add;
```

3. Инициализация указателей

// Примеры функций:

```
int add(int a, int b) { return a + b; }
```

```
int multiply(int a, int b) { return a * b; }
```

// Способы инициализации:

```
int (*op1)(int, int) = add;
```

// Прямое присваивание

```
int (*op2)(int, int) = &multiply;
```

// Явное взятие адреса

// Массив указателей:

```
int (*operations[2])(int, int) = {add, multiply};
```

// Автоматическое определение (C++11):

```
auto auto_ptr = add;
```

// Тип выводится автоматически

4. Вызов через указатели

```
int result;
int (*operation)(int, int) = add;

// Эквивалентные способы вызова:
result = operation(5, 3);           // Неявное разыменование
result = (*operation)(5, 3);        // Явное разыменование

// Избыточное разыменование (работает!):
result = (*****operation)(5, 3);

// Пример с массивом:
int (*ops[2])(int, int) = {add, multiply};
result = ops[0](4, 5);  // 4 + 5 = 9
result = ops[1](4, 5);  // 4 * 5 = 20
```

5. Практический пример: Калькулятор

```
#include <iostream>
#include <map>
#include <string>

// Базовые операции:
double add(double a, double b) { return a + b; }
double sub(double a, double b) { return a - b; }
double mul(double a, double b) { return a * b; }
double div(double a, double b) { return b != 0 ? a / b : 0; }

// Тип для указателей:
typedef double (*CalculatorOp)(double, double);

// Карта операций:
std::map<std::string, CalculatorOp> operations = {
    {"+", add}, {"-", sub}, {"*", mul}, {"/", div}
};
```

Пример калькулятора (реализация)

```
int main() {
    double a, b;
    std::string op;

    std::cout << "Введите выражение (a op b): ";
    std::cin >> a >> op >> b;

    // Динамический выбор операции:
    auto it = operations.find(op);
    if (it != operations.end()) {
        CalculatorOp operation = it->second;
        double result = operation(a, b);
        std::cout << "Результат: " << result << std::endl;
    } else {
        std::cout << "Неизвестная операция!" << std::endl;
    }

    return 0;
}
```

6. Указатели как параметры функций

```
// Функция принимает указатель на функцию как параметр
void process_array(int* arr, size_t size, int (*processor)(int)) {
    for (size_t i = 0; i < size; ++i) {
        arr[i] = processor(arr[i]);
    }
}
```

```
// Примеры функций-обработчиков:
int square(int x) { return x * x; }
int increment(int x) { return x + 1; }
int negate(int x) { return -x; }
```

```
// Использование:
int data[] = {1, 2, 3, 4, 5};
process_array(data, 5, square);    // Возводит в квадрат
process_array(data, 5, increment); // Увеличивает на 1
```

7. Возврат указателей из функций

```
// Функция возвращает указатель на функцию
int (*get_operation(const std::string& op))(int, int) {
    if (op == "add") return add;
    if (op == "multiply") return multiply;
    return nullptr;
}
```

```
// С использованием typedef:
typedef int (*MathOp)(int, int);
```

```
MathOp get_operation_simple(const std::string& op) {
    if (op == "add") return add;
    if (op == "multiply") return multiply;
    return nullptr;
}
```

```
// Использование:
auto op = get_operation("add");
if (op) int result = op(5, 3); // 8
```

8. Массивы указателей на функции

```
// Массив математических операций
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int div(int a, int b) { return b != 0 ? a / b : 0; }

// Массив указателей на функции
int (*operations[])(int, int) = {add, sub, mul, div};

// Использование в меню:
void show_menu() {
    std::cout << "1. Сложение\n";
    std::cout << "2. Вычитание\n";
    std::cout << "3. Умножение\n";
    std::cout << "4. Деление\n";
}

int choice;
std::cin >> choice;
if (choice >= 1 && choice <= 4) {
    int result = operations[choice-1](10, 5);
}
```

9. typedef и using для упрощения

```
// Традиционный typedef (C-style):  
typedef int (*TraditionalPtr)(int, int);  
TraditionalPtr ptr1 = add;
```

```
// Современный using (C++11):  
using ModernPtr = int (*)(int, int);  
ModernPtr ptr2 = multiply;
```

```
// Для сложных сигнатур:  
using Callback = void (*)(const std::string&, int);  
using Comparator = bool (*)(const Student&, const Student&);
```

```
// Преимущества using:  
- Более читаемый синтаксис  
- Лучшая поддержка шаблонов  
- Совместимость с современным C++
```

```
// Пример:  
ModernPtr operation = get_operation("add");
```

10. Указатели на функции-члены класса

```
class Calculator {
public:
    int add(int a, int b) { return a + b; }
    int multiply(int a, int b) { return a * b; }
};

// Указатель на функцию-член:
int (Calculator::*member_ptr)(int, int) = &Calculator::add;

// Использование:
Calculator calc;
int result = (calc.*member_ptr)(5, 3); // 8

// Современный подход с std::function:
#include <functional>
std::function<int(Calculator&, int, int)> func_ptr =
&Calculator::add;
result = func_ptr(calc, 5, 3); // 8

// Важно: требуют экземпляр класса для вызова
```

11. Сравнение с std::function

// Традиционные указатели на функции:

```
int (*old_style)(int, int) = add;
```

// Современный std::function:

```
#include <functional>
```

```
std::function<int(int, int)> modern_func = add;
```

// Преимущества std::function:

- Может хранить лямбды, функциональные объекты, методы классов
- Безопаснее и выразительнее
- Поддерживает полиморфизм времени выполнения
- Лучшая интеграция с STL

// Недостатки:

- Небольшие накладные расходы по памяти/производительности
- Требуется включения <functional>

// Рекомендация MIT:

Используйте std::function для нового кода,
но понимайте традиционные указатели для legacy code.

12. Лямбда-выражения (C++11)

// Лямбды как современная альтернатива:

```
auto lambda = [](int a, int b) { return a + b; };
```

// Сохранение в std::function:

```
std::function<int(int, int)> func = [](int a, int b) {  
    return a * b;  
};
```

// Захват переменных из контекста:

```
int multiplier = 5;  
auto capturing_lambda = [multiplier](int x) {  
    return x * multiplier;  
};
```

// Использование с алгоритмами STL:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};  
std::transform(numbers.begin(), numbers.end(), numbers.begin(),  
    [](int x) { return x * x; });
```

// Преимущества:

- Более компактный синтаксис
- Захват контекста
- Лучшая производительность (часто)

13. Обратные вызовы (Callbacks)

```
// Классический пример callback-системы:
typedef void (*EventCallback)(const std::string& event_name);

class EventSystem {
    std::vector<EventCallback> callbacks;
public:
    void register_callback(EventCallback cb) {
        callbacks.push_back(cb);
    }

    void trigger_event(const std::string& event_name) {
        for (auto cb : callbacks) {
            cb(event_name);
        }
    }
};

// Функции-обработчики событий:
void log_event(const std::string& event) {
    std::cout << "Event: " << event << std::endl;
}

void save_event(const std::string& event) {
    // Сохранение в базу данных
}
```

Callback система (использование)

// Продолжение предыдущего примера:

```
int main() {  
    EventSystem events;  
  
    // Регистрация обработчиков:  
    events.register_callback(log_event);  
    events.register_callback(save_event);  
  
    // Генерация событий:  
    events.trigger_event("user_login");  
    events.trigger_event("file_upload");  
    events.trigger_event("system_shutdown");  
    return 0;}
```

```
// Вывод программы:  
// Event: user_login  
// Event: file_upload  
// Event: system_shutdown
```

// Преимущества подхода:

- Гибкая расширяемость
- Разделение ответственности
- Поддержка плагинов

14. Паттерн Strategy

// Реализация паттерна Strategy с указателями на функции:

```
class Sorter {
    using CompareFunction = bool (*)(int, int);
    CompareStrategy compare_strategy;
public:
    void set_strategy(CompareFunction strategy) {
        compare_strategy = strategy;
    }

    void sort(int* array, size_t size) {
        // Использование стратегии для сравнения
        for (size_t i = 0; i < size-1; ++i) {
            for (size_t j = i+1; j < size; ++j) {
                if (compare_strategy(array[i], array[j])) {
                    std::swap(array[i], array[j]);
                }
            }
        }
    };
};

// Стратегии сравнения:
bool ascending(int a, int b) { return a > b; }
bool descending(int a, int b) { return a < b; }
```

Pattern Strategy (использование)

```
// Продолжение примера Strategy:
```

```
int main() {  
    Sorter sorter;  
    int data[] = {5, 2, 8, 1, 9};
```

```
    // Сортировка по возрастанию:  
    sorter.set_strategy(ascending);  
    sorter.sort(data, 5);  
    // data: [1, 2, 5, 8, 9]
```

```
    // Сортировка по убыванию:  
    sorter.set_strategy(descending);  
    sorter.sort(data, 5);  
    // data: [9, 8, 5, 2, 1]  
    return 0;}
```

```
// Преимущества паттерна:
```

- Инкапсуляция алгоритмов
- Возможность смены поведения на лету
- Упрощение тестирования
- Соответствие принципу Open/Closed

15. Best practices MIT

- Используйте typedef/using для сложных сигнатур
- Всегда проверяйте указатели на nullptr перед вызовом
- Предпочитайте std::function для нового кода
- Используйте лямбды для простых операций
- Документируйте ожидаемую сигнатуру функций
- Избегайте избыточного разыменования
- Используйте const где возможно

// Безопасный вызов:

```
if (function_ptr != nullptr) {  
    result = function_ptr(arg1, arg2);  
} else {  
    // Обработка ошибки  
}
```

// Современный подход:

```
std::function<int(int, int)> safe_func = function_ptr;  
if (safe_func) {  
    result = safe_func(arg1, arg2);  
}
```

16. Типичные ошибки

1. Несоответствие сигнатур:

```
int (*ptr)(int) = add; // Ошибка: add ожидает 2 аргумента
```

2. Вызов nullptr:

```
int (*ptr)(int, int) = nullptr;  
int result = ptr(5, 3); // Segmentation fault
```

3. Путаница с синтаксисом:

```
int *ptr(int, int); // Функция, возвращающая int*  
int (*ptr)(int, int); // Указатель на функцию
```

4. Проблемы с областью видимости:

```
void register_callback() {  
    int local_var = 42;  
    // Ошибка: захват локальной переменной  
    callbacks.push_back([&]() { use(local_var); });  
} // local_var уничтожается здесь
```

5. Устаревшие указатели на удаленные функции

17. Отладка и диагностика

// Вывод информации об указателях:

```
#include <cstdio>
```

```
printf("Адрес функции: %p\n", (void*)add);
```

```
printf("Адрес указателя: %p\n", (void*)&function_ptr);
```

// Использование typeid (осторожно!):

```
#include <typeinfo>
```

```
std::cout << "Тип указателя: " << typeid(function_ptr).name() << std::endl;
```

// Современный подход с decltype:

```
using FunctionType = decltype(add);
```

```
FunctionType* ptr = add;
```

// Отладочные макросы:

```
#define CHECK_NULL(ptr) \
```

```
    if ((ptr) == nullptr) { \
```

```
        std::cerr << "Null pointer at " << __LINE__ << std::endl; \
```

```
        return; \
```

```
    }
```

// Инструменты:

- GDB/LLDB: `break *function_address`

- Valgrind для проверки памяти

- Sanitizers (Address, Undefined Behavior)

18. Вопросы производительности

- Прямой вызов функции: самый быстрый
- Вызов через указатель: небольшой overhead (1-2 такта)
- `std::function`: небольшой overhead + возможное выделение памяти
- Влияние на предсказание переходов:
 - Прямые вызовы: легко предсказуемы
 - Косвенные вызовы: сложнее для предсказания

// Бенчмарк пример:

```
void benchmark() {  
    auto start = std::chrono::high_resolution_clock::now();
```

```
    // Прямой вызов (базовый уровень)
```

```
    for (int i = 0; i < 10000000; ++i) {  
        result = direct_call(i, i+1);  
    }
```

```
    // Вызов через указатель
```

```
    for (int i = 0; i < 10000000; ++i) {  
        result = pointer_call(i, i+1);  
    }
```

```
    auto end = std::chrono::high_resolution_clock::now();  
    std::cout << "Time: " << (end - start).count() << " ns\n";
```

```
}
```

19. Вопросы для самопроверки MIT

1. В чем разница между `int (*f)(int)` и `int *f(int)`?
2. Как безопасно вызвать указатель на функцию?
3. Когда использовать `typedef/using` для указателей?
4. Чем `std::function` лучше традиционных указателей?
5. Как реализовать систему обратных вызовов?
6. Какие `overheads` у вызовов через указатели?
7. Как отлаживать проблемы с указателями на функции?
8. Когда предпочесть лямбды указателям?
9. Как реализовать паттерн `Strategy`?
10. Какие типичные ошибки допускают новички?

Практическое задание:

Реализуйте калькулятор с поддержкой пользовательских операций, регистрируемых через указатели на функции.

20. Заключение и ключевые выводы

- Указатели на функции - мощный инструмент для динамического вызова
- Критически важны для обратных вызовов и плагинов
- Современный C++ предлагает `std::function` и лямбды
- Понимание необходимо для работы с legacy code
- Требуют аккуратности и проверок

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 начинающих
- isocpp.org - официальный сайт C++

Статьи и руководства:

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