1. Указатели на функции

Определение:
Указатель на функцию — это переменная, которая хранит адрес функции в памяти. Это позволяет вызывать функцию косвенно, работать с функциями как с данными и передавать их в качестве параметров.

cpp
```cpp
#include <iostream>
#include <cmath>
using namespace std;

// Современное объявление типа указателя на функцию
using MathOperation = int(int, int);

// Различные функции с одинаковой сигнатурой
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; }

// Функция, принимающая указатель на функцию как параметр
void calculate(int x, int y, MathOperation* operation) {
    cout << "Result: " << operation(x, y) << endl;
}

// Массив указателей на функции
void demonstrateFunctionArray() {
    MathOperation* operations[] = {add, subtract, multiply};
    const char* names[] = {"Add", "Subtract", "Multiply"};

    for (int i = 0; i < 3; i++) {
        cout << names[i] << ": ";
        calculate(10, 5, operations[i]);
    }
}

// Callback функция для обработки данных
void processNumbers(int arr[], int size, int (*processor)(int)) {
    for (int i = 0; i < size; i++) {
        arr[i] = processor(arr[i]);
    }
}
```

```cpp
int square(int x) { return x * x; }
int cube(int x) { return x * x * x; }

int main() {
    // Использование указателей на функции
    MathOperation* op = add;
    cout << "10 + 5 = " << op(10, 5) << endl;

    op = multiply;
    cout << "10 * 5 = " << op(10, 5) << endl;

    // Передача функции как callback
    int numbers[] = {1, 2, 3, 4, 5};
    processNumbers(numbers, 5, square);

    cout << "Squared: ";
    for (int i = 0; i < 5; i++) {
        cout << numbers[i] << " ";
    }
    cout << endl;

    demonstrateFunctionArray();

    return 0;
}
```

2. Перегрузка функций в языке C++

Определение:
Перегрузка функций — это возможность создавать несколько функций с одинаковым именем, но разными параметрами (по количеству, типам или порядку). Компилятор выбирает appropriate функцию на основе переданных аргументов.

cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Перегрузка по типам параметров
void display(int value) {
    cout << "Integer: " << value << endl;
}
```

```cpp
void display(double value) {
    cout << "Double: " << value << endl;
}

void display(const string& value) {
    cout << "String: " << value << endl;
}

void display(const char* value) {
    cout << "C-string: " << value << endl;
}

// Перегрузка по количеству параметров
int calculateSum(int a, int b) {
    return a + b;
}

int calculateSum(int a, int b, int c) {
    return a + b + c;
}

int calculateSum(const vector<int>& numbers) {
    int total = 0;
    for (int num : numbers) {
        total += num;
    }
    return total;
}

// Перегрузка с параметрами по умолчанию
void showMessage(string message, int repeat = 1, char separator = ' ') {
    for (int i = 0; i < repeat; i++) {
        cout << message;
        if (i < repeat - 1) cout << separator;
    }
    cout << endl;
}

// Перегрузка для разных категорий значений
void processValue(int& value) { // Для изменяемых значений
    value *= 2;
    cout << "Modified value: " << value << endl;
}
```

```cpp
void processValue(const int& value) { // Для неизменяемых значений
    cout << "Read-only value: " << value << endl;
}

int main() {
    // Демонстрация перегрузки по типам
    display(42);
    display(3.14159);
    display("Hello");
    display(string("World"));

    // Демонстрация перегрузки по количеству
    cout << "Sum of 2: " << calculateSum(10, 20) << endl;
    cout << "Sum of 3: " << calculateSum(10, 20, 30) << endl;

    vector<int> nums = {1, 2, 3, 4, 5};
    cout << "Sum of vector: " << calculateSum(nums) << endl;

    // Параметры по умолчанию
    showMessage("Hello");
    showMessage("Hello", 3);
    showMessage("Hello", 3, '-');

    // Обработка разных категорий значений
    int val = 5;
    processValue(val);       // Изменяемое значение
    processValue(10);        // Временное значение

    return 0;
}
```

3. Шаблоны функций

Определение:
Шаблоны функций — это механизм, позволяющий создавать обобщенные функции, которые могут работать с различными типами данных. Компилятор автоматически генерирует конкретные реализации для каждого используемого типа.

cpp

```cpp
#include <iostream>
#include <vector>
#include <string>
```

```cpp
#include <algorithm>
using namespace std;

// Базовый шаблон функции для нахождения максимума
template<typename T>
T findMaximum(T a, T b) {
    return (a > b) ? a : b;
}

// Шаблон с автоматическим выводом возвращаемого типа
template<typename T1, typename T2>
auto combineValues(T1 a, T2 b) {
    return a + b;
}

// Шаблон функции для работы с массивами
template<typename T, size_t Size>
void displayArray(const T (&array)[Size]) {
    cout << "Array[" << Size << "]: ";
    for (size_t i = 0; i < Size; i++) {
        cout << array[i] << " ";
    }
    cout << endl;
}

// Шаблон с non-type параметром
template<typename T, int Factor>
T scaleValue(T value) {
    return value * Factor;
}

// Явная специализация шаблона для C-строк
template<>
const char* findMaximum<const char*>(const char* a, const char* b) {
    return (strcmp(a, b) > 0) ? a : b;
}

// Шаблонная функция сортировки
template<typename T>
void sortArray(T array[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (array[j] > array[j + 1]) {
                swap(array[j], array[j + 1]);
```

```
            }
        }
    }
}

// Шаблон для вывода элементов контейнера
template<typename Container>
void printContainer(const Container& container) {
    for (const auto& element : container) {
        cout << element << " ";
    }
    cout << endl;
}

// Variadic template (шаблон с переменным числом параметров)
template<typename... Args>
void printAll(Args... args) {
    (cout << ... << args) << endl; // Fold expression (C++17)
}

int main() {
    // Использование шаблонных функций
    cout << "Max int: " << findMaximum(10, 20) << endl;
    cout << "Max double: " << findMaximum(3.14, 2.71) << endl;

    // Автоматический вывод типов
    cout << "Combined: " << combineValues(10, 3.14) << endl;
    cout << "Combined: " << combineValues(string("Age: "), 25) << endl;

    // Работа с массивами
    int integers[] = {5, 2, 8, 1, 9};
    double doubles[] = {3.14, 2.71, 1.618};

    displayArray(integers);
    displayArray(doubles);

    // Сортировка массивов
    sortArray(integers, 5);
    cout << "Sorted integers: ";
    displayArray(integers);

    // Non-type параметры
    cout << "Scaled x2: " << scaleValue<int, 2>(5) << endl;
    cout << "Scaled x10: " << scaleValue<double, 10>(2.5) << endl;
```

```
    // Работа с контейнерами STL
    vector<string> words = {"template", "function", "cpp"};
    printContainer(words);

    // Variadic templates
    printAll(1, " + ", 2, " = ", 3);
    printAll("Hello", " ", "World", "!");

    return 0;
}
```

Дополнительная информация

Объявление указателей на функции

Указатели на функции объявляются с указанием типа возвращаемого значения и типов параметров. Синтаксис требует круглых скобок вокруг имени указателя.

cpp

```
#include <iostream>
using namespace std;

// Пример функции
double process(int x) {
    return x * 1.5;
}

// Правильное объявление указателя на функцию
double (*func_ptr)(int);  // Указатель на функцию, принимающую int и возвращающую dou
ble

int main() {
    func_ptr = process;    // Присваивание адреса функции
    cout << func_ptr(10) << endl;  // Вызов через указатель
    return 0;
}
```

Инициализация указателей на функции

Указатели на функции можно инициализировать различными способами, включая прямое присваивание и использование auto для автоматического вывода типа.

```cpp
#include <iostream>
using namespace std;

double calculate(int x) {
    return x * 2.5;
}

int main() {
    // Способ 1: Явное указание типа
    double (*ptr1)(int) = calculate;

    // Способ 2: Использование auto (C++11)
    auto ptr2 = calculate;

    // Способ 3: Присваивание после объявления
    double (*ptr3)(int);
    ptr3 = calculate;

    cout << ptr1(5) << " " << ptr2(5) << " " << ptr3(5) << endl;
    return 0;
}
```

Вызов через указатель на функцию

Функцию можно вызвать через указатель двумя способами: напрямую используя имя указателя или с оператором разыменования.

```cpp
#include <iostream>
using namespace std;

double transform(int value) {
    return value + 0.5;
}

int main() {
    double (*func)(int) = transform;
```

```
    // Способ 1: Прямой вызов
    double result1 = func(10);

    // Способ 2: С оператором разыменования
    double result2 = (*func)(10);

    cout << result1 << " " << result2 << endl;  // Оба способа работают
    return 0;
}
```

Массивы указателей на функции

Массивы указателей на функции позволяют создавать таблицы функций для вызова по индексу.

cpp

```
#include <iostream>
using namespace std;

void func1(int x) { cout << "Func1: " << x << endl; }
void func2(int x) { cout << "Func2: " << x * 2 << endl; }
void func3(int x) { cout << "Func3: " << x * 3 << endl; }

int main() {
    // Массив указателей на функции
    void (*func_array[3])(int) = {func1, func2, func3};

    for (int i = 0; i < 3; i++) {
        func_array[i](i + 1);  // Вызов по индексу
    }
    return 0;
}
```

Условия перегрузки функций

Перегрузка функций требует различий в сигнатурах - количестве или типах параметров.

cpp

```
#include <iostream>
using namespace std;
```

```cpp
// Разное количество параметров
int sum(int a, int b) {
    return a + b;
}

int sum(int a, int b, int c) {
    return a + b + c;
}

// Разные типы параметров
double sum(double a, double b) {
    return a + b;
}

int main() {
    cout << sum(1, 2) << endl;        // Вызов первой версии
    cout << sum(1, 2, 3) << endl;     // Вызов второй версии
    cout << sum(1.5, 2.5) << endl;    // Вызов третьей версии
    return 0;
}
```

Разрешение перегрузки

Компилятор выбирает наиболее подходящую перегруженную функцию на основе точного совпадения типов или доступных преобразований.

cpp

```cpp
#include <iostream>
using namespace std;

void process(int x) {
    cout << "Integer: " << x << endl;
}

void process(double x) {
    cout << "Double: " << x << endl;
}

void process(const char* x) {
    cout << "String: " << x << endl;
}

int main() {
```

```
    process(10);        // Точное совпадение - int
    process(3.14);      // Точное совпадение - double
    process("hello");   // Точное совпадение - const char*
    process(10.5f);     // Стандартное преобразование float -> double
    return 0;
}
```

Параметры по умолчанию и перегрузка

Параметры по умолчанию могут создавать неоднозначности при вызове перегруженных функций.

cpp

```
#include <iostream>
using namespace std;

void display(int x) {
    cout << "One param: " << x << endl;
}

void display(int x, int y = 0) {
    cout << "Two params: " << x << ", " << y << endl;
}

int main() {
    display(5);  // Неоднозначность - какая функция вызваться?
    return 0;
}
```

Константные параметры и перегрузка

Const влияет на перегрузку для ссылочных параметров и указателей, но не для параметров по значению.

cpp

```
#include <iostream>
using namespace std;

void process(int& x) {
    cout << "Non-const reference" << endl;
    x = 10;
}
```

```cpp
void process(const int& x) {
    cout << "Const reference: " << x << endl;
}

void process(const int* ptr) {
    cout << "Pointer to const: " << *ptr << endl;
}

void process(int* ptr) {
    cout << "Pointer to non-const" << endl;
    *ptr = 20;
}

int main() {
    int a = 5;
    const int b = 10;

    process(a);    // Non-const reference
    process(b);    // Const reference
    process(&a);   // Pointer to non-const
    process(&b);   // Pointer to const

    return 0;
}
```

Шаблоны функций

Шаблоны позволяют создавать обобщенные функции, которые инстанцируются компилятором для конкретных типов.

cpp

```cpp
#include <iostream>
using namespace std;

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

template<typename T, typename U>
auto multiply(T a, U b) -> decltype(a * b) {
    return a * b;
```

```
}

int main() {
    cout << max(10, 20) << endl;           // int version
    cout << max(3.14, 2.71) << endl;       // double version
    cout << multiply(2, 3.5) << endl;      // mixed types
    return 0;
}
```

Синтаксис шаблонов

Шаблоны объявляются с ключевым словом template, за которым следуют параметры в угловых скобках.

cpp

```
#include <iostream>
using namespace std;

// Правильный синтаксис объявления шаблона
template<typename T>
T square(T x) {
    return x * x;
}

template<class U>  // 'class' эквивалентно 'typename'
U cube(U x) {
    return x * x * x;
}

int main() {
    cout << square(5) << endl;    // 25
    cout << cube(3.0) << endl;    // 27.0
    return 0;
}
```

Вывод типов в шаблонах

Компилятор автоматически выводит типы шаблонных параметров из аргументов функции.

cpp

```
#include <iostream>
```

```cpp
using namespace std;

template<typename T>
void printType(T value) {
    cout << "Type: " << typeid(T).name() << ", Value: " << value << endl;
}

template<typename T, typename U>
auto combine(T a, U b) {
    return a + b;
}

int main() {
    printType(42);           // T выводится как int
    printType(3.14);         // T выводится как double
    printType("hello");      // T выводится как const char*

    auto result = combine(10, 2.5);  // auto выводит тип double
    cout << result << endl;
    return 0;
}
```

Явное указание типов

Явное указание типов необходимо когда вывод невозможен или нужен конкретный тип.

cpp

```cpp
#include <iostream>
using namespace std;

template<typename T>
T create() {
    return T();
}

template<typename T>
T convert(double value) {
    return static_cast<T>(value);
}

int main() {
    // Явное указание типа когда вывод невозможен
```

```cpp
    int x = create<int>();
    double y = create<double>();

    // Явное указание когда нужен конкретный тип
    int intVal = convert<int>(3.14);
    float floatVal = convert<float>(2.718);

    cout << x << " " << y << " " << intVal << " " << floatVal << endl;
    return 0;
}
```

Специализация шаблонов

Специализация позволяет предоставить особую реализацию шаблона для конкретного типа.

cpp
```cpp
#include <iostream>
#include <cstring>
using namespace std;

template<typename T>
bool compare(T a, T b) {
    return a == b;
}

// Специализация для C-строк
template<>
bool compare<const char*>(const char* a, const char* b) {
    return strcmp(a, b) == 0;
}

int main() {
    cout << compare(10, 10) << endl;              // true
    cout << compare("hello", "world") << endl;    // false
    cout << compare("test", "test") << endl;      // true (специализация)
    return 0;
}
```

Перегрузка и шаблоны

Обычные функции и шаблоны могут сосуществовать, при этом обычные функции имеют приоритет при разрешении перегрузки.

cpp
```cpp
#include <iostream>
using namespace std;

// Обычная функция
void process(int x) {
    cout << "Ordinary function: " << x << endl;
}

// Шаблонная функция
template<typename T>
void process(T x) {
    cout << "Template function: " << x << endl;
}

int main() {
    process(10);      // Вызов обычной функции (приоритет)
    process(3.14);    // Вызов шаблонной функции
    process("hello"); // Вызов шаблонной функции
    return 0;
}
```

Указатели на шаблонные функции

Указатели на шаблонные функции объявляются с указанием шаблонных параметров.

cpp
```cpp
#include <iostream>
using namespace std;

template<typename T>
T transform(T x) {
    return x * 2;
}

int main() {
    // Указатель на конкретную инстанцированную функцию
    int (*int_ptr)(int) = transform<int>;
```

```
    double (*double_ptr)(double) = transform<double>;

    cout << int_ptr(5) << endl;       // 10
    cout << double_ptr(2.5) << endl;  // 5.0

    return 0;
}
```

Параметры шаблонов

Шаблоны могут принимать различные типы параметров: типы, значения и другие шаблоны.

cpp

```
#include <iostream>
using namespace std;

// Параметр-тип
template<typename T>
T getDefault() {
    return T();
}

// Параметр-значение
template<int Size>
class Buffer {
    int data[Size];
public:
    int getSize() { return Size; }
};

// Шаблонный параметр
template<template<typename> class Container, typename T>
void processContainer(Container<T>& cont) {
    cout << "Processing container" << endl;
}

int main() {
    cout << getDefault<int>() << endl;

    Buffer<10> buffer;
    cout << buffer.getSize() << endl;
```

```
    return 0;
}
```

Шаблоны с несколькими параметрами

Шаблоны могут иметь несколько параметров разных категорий.

cpp
```cpp
#include <iostream>
using namespace std;

template<typename T, typename U>
auto calculate(T a, U b) -> decltype(a + b) {
    return a + b;
}

template<typename T, int N>
class Array {
    T elements[N];
public:
    T& operator[](int index) { return elements[index]; }
    int size() { return N; }
};

int main() {
    cout << calculate(10, 3.14) << endl;  // 13.14

    Array<int, 5> arr;
    for (int i = 0; i < arr.size(); i++) {
        arr[i] = i * 2;
    }

    return 0;
}
```

Автоматический вывод типов

Ключевое слово auto позволяет автоматически выводить типы в шаблонах.

cpp
```cpp
#include <iostream>
using namespace std;
```

```cpp
template<typename T, typename U>
auto multiply(T a, U b) {
    return a * b;
}

template<typename T>
auto getSize(const T& container) -> decltype(container.size()) {
    return container.size();
}

int main() {
    auto result = multiply(2, 3.5);  // auto выводит double
    cout << result << endl;           // 7.0

    return 0;
}
```

SFINAE и шаблоны

SFINAE (Substitution Failure Is Not An Error) - принцип, при котором неудачная подстановка шаблонных параметров не приводит к ошибке, а просто исключает шаблон из consideration.

cpp

```cpp
#include <iostream>
#include <type_traits>
using namespace std;

template<typename T>
typename enable_if<is_integral<T>::value, T>::type
process(T x) {
    return x * 2;
}

template<typename T>
typename enable_if<is_floating_point<T>::value, T>::type
process(T x) {
    return x * 3;
}

int main() {
    cout << process(5) << endl;     // 10 (целочисленная версия)
```

```
    cout << process(2.5) << endl;   // 7.5 (версия для чисел с плавающей точкой)
    // process("hello");            // Ошибка - нет подходящей функции
    return 0;
}
```

Практическое применение указателей на функции

Указатели на функции широко используются для реализации callback-ов, плагинов и динамического поведения.

cpp

```
#include <iostream>
#include <vector>
using namespace std;

// Callback система
using Callback = void(int);

void callback1(int value) {
    cout << "Callback 1: " << value << endl;
}

void callback2(int value) {
    cout << "Callback 2: " << value * 2 << endl;
}

class EventSystem {
    vector<Callback*> callbacks;
public:
    void registerCallback(Callback* cb) {
        callbacks.push_back(cb);
    }

    void triggerEvent(int value) {
        for (auto cb : callbacks) {
            cb(value);
        }
    }
};

int main() {
    EventSystem system;
    system.registerCallback(callback1);
```

```
    system.registerCallback(callback2);

    system.triggerEvent(42);
    return 0;
}
```