



Язык C++

Указатели

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 презентации

1. Определение и объявление указателей
2. Основные операции с указателями
3. Арифметика указателей
4. Константы и указатели
5. Массивы и указатели
6. Динамическое выделение памяти
7. Умные указатели (C++11)
8. Указатели на функции
9. Указатели на члены класса
10. void указатели
11. Best practices и common errors

Что такое указатели?

Указатель - переменная, которая хранит адрес памяти другой переменной

Основные концепции:

- Адрес памяти - уникальный идентификатор ячейки памяти
- Разыменование - доступ к значению по адресу
- Размер указателя зависит от архитектуры (обычно 4 или 8 байт)

Зачем нужны указатели:

- Динамическое выделение памяти
- Работа с массивами
- Передача параметров по ссылке
- Создание сложных структур данных
- Полиморфизм и ООП

Аналогия:

Указатель как номер дома → Дом как переменная → Жители как значение

Объявление указателей

// Базовое объявление

```
int x = 10;
```

```
int* ptr = &x;    // ptr указывает на x
```

// Разные стили объявления

```
int* p1;          // стиль 1: * рядом с типом
```

```
int *p2;          // стиль 2: * рядом с именем
```

```
int * p3;         // стиль 3: * посередине
```

// Указатели разных типов

```
double* dptr;
```

```
char* cptr;
```

```
std::string* sptr;
```

// Множественное объявление

```
int *a, *b, *c;    // все три - указатели
```

```
int* d, e;         // d - указатель, e - int!
```

// Инициализация

```
int* null_ptr = nullptr; // современный C++ (рекомендуется)
```

```
int* old_null = NULL;    // устаревший стиль
```

```
int* zero_ptr = 0;       // не рекомендуется
```

Основные операции

```
int x = 10, y = 20;  
int* ptr = &x;
```

// Разыменование

```
cout << *ptr;      // 10  
*ptr = 15;         // x теперь = 15
```

// Присваивание

```
ptr = &y;           // теперь указывает на y  
cout << *ptr;      // 20
```

// Сравнение указателей

```
int* ptr1 = &x;  
int* ptr2 = &y;  
bool same = (ptr1 == ptr2); // false  
bool null_check = (ptr1 == nullptr); // проверка на null
```

// Указатель на указатель

```
int** pptr = &ptr; // pptr → ptr → y  
cout << **pptr;    // 20  
**pptr = 25;       // y теперь = 25
```

Арифметика указателей

Арифметические операции с указателями:

- Инкремент ($++ptr$) - переход к следующему элементу
- Декремент ($--ptr$) - переход к предыдущему элементу
- Сложение ($ptr + n$) - переход на n элементов вперед
- Вычитание ($ptr - n$) - переход на n элементов назад
- Разность ($ptr1 - ptr2$) - количество элементов между указателями

Важные особенности:

- Размер шага зависит от типа указателя
- int^* увеличивается на $\text{sizeof}(int)$ байт
- $double^*$ увеличивается на $\text{sizeof}(double)$ байт
- $void^*$ арифметика запрещена
- Указатели можно сравнивать ($<$, $>$, $<=$, $>=$)

Область применения:

- Итерация по массивам
- Работа с буферами данных
- Низкоуровневые операции

Примеры арифметики

```
int arr[5] = {10, 20, 30, 40, 50};  
int* ptr = arr;    // указывает на arr[0]
```

// Базовые операции

```
cout << *ptr;      // 10  
ptr++;             // теперь указывает на arr[1]  
cout << *ptr;      // 20  
ptr += 2;          // теперь указывает на arr[3]  
cout << *ptr;      // 40  
ptr--;             // теперь указывает на arr[2]  
cout << *ptr;      // 30
```

// Разность указателей

```
int* start = arr;  
int* end = arr + 5;  
int distance = end - start; // 5 элементов
```

// Итерация по массиву

```
for (int* p = arr; p < arr + 5; p++) {  
    cout << *p << " ";    // 10 20 30 40 50  
}
```

Константы и указатели

Комбинации const и указателей:

1. Указатель на константу:

```
const int* ptr
```

- Менять указатель можно
- Менять значение нельзя

2. Константный указатель:

```
int* const ptr
```

- Менять указатель нельзя
- Менять значение можно

3. Константный указатель на константу:

```
const int* const ptr
```

- Менять указатель нельзя
- Менять значение нельзя

Правила чтения:

- Читать справа налево
- const перед * - константное значение
- const после * - константный указатель

Пример:

```
const int* ptr1;    // pointer to const int
```

```
int* const ptr2;    // const pointer to int
```

```
const int* const ptr3; // const pointer to const int
```

Примеры const указателей

```
int x = 10, y = 20;  
const int z = 30;
```

// 1. Указатель на константу

```
const int* ptr1 = &x; // ОК  
ptr1 = &y;           // ОК - можно изменить указатель  
// *ptr1 = 15;       // ОШИБКА - нельзя изменить значение  
ptr1 = &z;           // ОК - указывает на константу
```

// 2. Константный указатель

```
int* const ptr2 = &x; // ОК  
*ptr2 = 15;          // ОК - можно изменить значение  
// ptr2 = &y;        // ОШИБКА - нельзя изменить указатель
```

// 3. Константный указатель на константу

```
const int* const ptr3 = &z; // ОК  
// ptr3 = &x;         // ОШИБКА  
// *ptr3 = 25;        // ОШИБКА
```

Массивы и указатели

Тесная связь массивов и указателей:

Факты:

- Имя массива - указатель на первый элемент
- `arr[i]` эквивалентно `*(arr + i)`
- `&arr[0]` эквивалентно `arr`
- Массивы передаются в функции как указатели

Различия:

- `sizeof(arr)` возвращает размер всего массива
- `sizeof(ptr)` возвращает размер указателя
- Массив нельзя перенаправить
- Указатель можно перенаправить

Многомерные массивы:

- `int matrix[2][3]` - массив массивов
- `matrix[0]` - указатель на первую строку
- Элементы хранятся в памяти последовательно

Связь массивов и указателей

```
int arr[5] = {10, 20, 30, 40, 50};
```

```
// Эквивалентные формы доступа
```

```
cout << arr[0];      // 10
```

```
cout << *arr;        // 10
```

```
cout << *(arr + 0);   // 10
```

```
cout << arr[2];       // 30
```

```
cout << *(arr + 2);    // 30
```

```
// Разница между массивом и указателем
```

```
int* ptr = arr;
```

```
cout << sizeof(arr);  // 20 (5 * 4 байта)
```

```
cout << sizeof(ptr);  // 4 или 8 (размер указателя)
```

```
// arr = ptr;        // ОШИБКА - массив нельзя перенаправить
```

```
ptr = arr;           // ОК - указатель можно перенаправить
```

Динамическая память

Динамическое выделение памяти:

Отличие от статической памяти:

- Статическая: размер известен на этапе компиляции
- Динамическая: размер определяется во время выполнения

Преимущества:

- Гибкость размера
- Память живет до явного освобождения
- Можно создавать большие структуры

Недостатки:

- Риск утечек памяти
- Риск висячих указателей
- Сложнее управлять

Область применения:

- Структуры данных переменного размера
- Большие объекты
- Полиморфизм
- Ресурсы, время жизни которых нужно контролировать



new и delete

// Выделение памяти для одного объекта

```
int* single = new int;    // выделение
```

```
*single = 42;           // использование
```

```
delete single;          // освобождение
```

```
single = nullptr;       // хорошая практика
```

// Выделение памяти для массива

```
int* array = new int[5]; // массив из 5 int
```

```
for (int i = 0; i < 5; i++) {
```

```
    array[i] = i * 10;
```

```
}
```

```
delete[] array;          // освобождение массива
```

```
array = nullptr;
```

// Инициализация при выделении

```
int* value = new int(100); // инициализация значением
```

```
int* arr_init = new int[5]{1, 2, 3, 4, 5}; // инициализация массива
```

Утечки памяти

Распространенные проблемы с памятью:

1. Утечка памяти (Memory Leak):

- Память выделена, но не освобождена
- Программа потребляет все больше памяти

2. Висячий указатель (Dangling Pointer):

- Указатель на уже освобожденную память
- Использование приводит к undefined behavior

3. Двойное освобождение (Double Free):

- Попытка освободить память дважды
- Приводит к crash или corruption

4. Выход за границы (Buffer Overflow):

- Чтение/запись за пределами выделенной памяти

Профилактика:

- Всегда освобождайте память
- Присваивайте nullptr после delete
- Используйте умные указатели
- Проверяйте указатели перед использованием

Умные указатели (C++11)

Умные указатели автоматически управляют памятью:

Типы умных указателей:

1. `std::unique_ptr`

- Единоличное владение
- Нельзя копировать
- Можно перемещать

2. `std::shared_ptr`

- Разделяемое владение
- Счетчик ссылок
- Автоматическое освобождение

3. `std::weak_ptr`

- Слабая ссылка без владения
- Для breaking circular references

Преимущества:

- Автоматическое освобождение памяти
- Исключение утечек памяти
- Безопасность исключений
- Четкая семантика владения

Требуют: `#include <memory>`// это заголовочный файл стандартной библиотеки C++, который содержит умные указатели и средства для управления памятью

std::unique_ptr

```
#include <memory>
```

```
// Создание unique_ptr
```

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(42);
```

```
std::unique_ptr<int[]> arr = std::make_unique<int[]>(5);
```

```
// Использование
```

```
cout << *ptr1;           // 42
```

```
arr[0] = 10;             // доступ к элементам массива
```

```
// Нельзя копировать
```

```
// std::unique_ptr<int> ptr2 = ptr1; // ОШИБКА
```

```
// Можно перемещать
```

```
std::unique_ptr<int> ptr2 = std::move(ptr1); // ptr1 теперь nullptr
```

```
cout << *ptr2;           // 42
```

```
// Освобождение памяти (автоматически при выходе из области видимости)
```

```
void function() {
```

```
    auto local_ptr = std::make_unique<int>(100);
```

```
    // память автоматически освободится
```

```
}
```

std::shared_ptr

```
#include <memory>
```

```
// Создание shared_ptr
```

```
std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
```

```
std::shared_ptr<int[]> arr = std::make_shared<int[]>(5);
```

```
// Копирование разрешено
```

```
std::shared_ptr<int> ptr2 = ptr1; // оба указывают на один объект
```

```
cout << ptr1.use_count(); // 2 - количество владельцев
```

```
// Использование
```

```
cout << *ptr1;      // 42
```

```
cout << *ptr2;      // 42
```

```
// Изменение влияет на все копии
```

```
*ptr1 = 100;
```

```
cout << *ptr2;      // 100
```

```
// Память освободится когда use_count == 0
```

```
ptr1.reset();      // use_count = 1
```

```
cout << ptr2.use_count(); // 1
```

```
ptr2.reset();      // use_count = 0 - память освобождена
```

std::weak_ptr

```
#include <memory>
```

```
// weak_ptr не увеличивает use_count  
std::shared_ptr<int> shared = std::make_shared<int>(42);  
std::weak_ptr<int> weak = shared;  
cout << shared.use_count(); // 1
```

```
// Для доступа нужно создать shared_ptr  
if (auto temp = weak.lock()) { // создает shared_ptr  
    cout << *temp;    // 42  
    cout << temp.use_count(); // 2  
} // temp уничтожается, use_count = 1
```

```
// Проверка валидности  
if (!weak.expired()) {  
    // объект еще существует  
}
```

```
// Решение проблемы циклических ссылок  
struct Node {  
    std::shared_ptr<Node> next;  
    std::weak_ptr<Node> prev; // weak_ptr вместо shared_ptr  
};
```

Сравнение умных указателей

Выбор умного указателя:

`std::unique_ptr`:

- Когда у объекта один владелец
- Для исключительного владения
- Более эффективен (нет накладных расходов)
- Нельзя копировать, можно перемещать

`std::shared_ptr`:

- Когда несколько владельцев
- Для разделяемого владения
- Есть накладные расходы (счетчик ссылок)
- Можно копировать

`std::weak_ptr`:

- Для наблюдения без владения
- Для breaking circular references
- Не увеличивает счетчик ссылок

Рекомендации:

- Предпочитайте `unique_ptr` над `shared_ptr`
- Используйте `make_unique` и `make_shared`
- Избегайте циклических ссылок
- Используйте `weak_ptr` для наблюдения

Указатели на функции

// Объявление указателя на функцию

```
int (*func_ptr)(int, int); // указатель на функцию, возвращающую int
```

// Пример функции

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

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

// Присваивание

```
func_ptr = &add; // можно без &
```

```
func_ptr = add; // тоже правильно
```

// Вызов через указатель

```
int result = func_ptr(3, 4); // 7
```

```
result = (*func_ptr)(3, 4); // альтернативный синтаксис
```

// Массив указателей на функции

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

```
result = operations[0](3, 4); // 7
```

```
result = operations[1](3, 4); // 12
```

Указатели на члены класса

```
class MyClass {  
public:  
    int value;  
    void print() { cout << value << endl; }  
    static int static_value;  
};
```

// Указатель на поле класса

```
int MyClass::*ptr_to_member = &MyClass::value;
```

// Указатель на метод класса

```
void (MyClass::*ptr_to_method)() = &MyClass::print;
```

// Использование

```
MyClass obj;
```

```
obj.*ptr_to_member = 42;    // установка значения
```

```
cout << obj.*ptr_to_member; // чтение значения: 42
```

```
(obj.*ptr_to_method)();    // вызов метода: 42
```

// Указатели на статические члены

```
int* ptr_to_static = &MyClass::static_value;
```

// Используются как обычные указатели

void указатели

// void* может указывать на любой тип

```
int x = 10;
```

```
double y = 3.14;
```

```
char z = 'A';
```

```
void* vptr;
```

```
vptr = &x;           // указывает на int
```

```
vptr = &y;           // теперь на double
```

```
vptr = &z;           // теперь на char
```

// Нельзя разыменовывать напрямую

```
// cout << *vptr;    // ОШИБКА
```

// Нужно явное приведение типа

```
cout << *(static_cast<int*>(vptr)); // правильно
```

// Использование в низкоуровневом коде

```
void* memory = malloc(100); // выделение памяти
```

// работа с памятью...

```
free(memory);         // освобождение
```

Лучшие практики

Рекомендации по работе с указателями:

Безопасность:

- Всегда инициализируйте указатели
- Используйте `nullptr` вместо `NULL` или `0`
- Проверяйте указатели перед использованием
- Присваивайте `nullptr` после `delete`

Память:

- Предпочитайте умные указатели над сырыми
- Используйте `make_unique` и `make_shared`
- Освобождайте память в том же контексте, где выделили
- Избегайте циклических ссылок

Кодстайл:

- Используйте `const` с указателями где возможно
- Предпочитайте ссылки над указателями когда возможно
- Избегайте `void*` в высокоуровневом коде
- Документируйте владение памятью

Распространенные ошибки

Типичные ошибки с указателями:

1. Разыменование nullptr:

```
int* ptr = nullptr;  
cout << *ptr; // CRASH!
```

2. Использование после delete:

```
int* ptr = new int(10);  
delete ptr;  
cout << *ptr; // UNDEFINED BEHAVIOR!
```

3. Утечка памяти:

```
void func() {  
    int* ptr = new int[100];  
    // забыли delete[]  
}
```

4. Двойное освобождение:

```
int* ptr = new int(10);  
delete ptr;  
delete ptr; // CRASH!
```

5. Выход за границы:

```
int arr[5];  
int* ptr = arr;  
ptr[10] = 42; // BUFFER OVERFLOW!
```

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++