4- Практическое задание – Конструкторы и Деструкторы и перегрузка методов

Table of Contents

Цель

Ознакомить студентов с ключевыми аспектами использования конструкторов, деструкторов и перегрузки методов в C++. Через серию задач студенты научатся реализовывать и применять различные типы конструкторов, включая конструкторы по умолчанию и параметризованные конструкторы, а также изучат роль деструкторов в управлении ресурсами. С помощью практических заданий они исследуют концепции копирования объектов с использованием конструкторов копирования, управления динамическим поведением объектов через конструкторы и деструкторы, и навыки перегрузки методов для создания гибких и мощных программных решений.

1. Конструктор по умолчанию и параметризованный конструктор

Для выполнения задачи 1 мы последовательно создадим класс Person с базовым и параметризованным конструкторами. Эта задача нацелена на обучение использованию конструкторов для инициализации объектов с заданными или значениями по умолчанию, когда данные не предоставлены. Давайте разберемся с каждым шагом.

Шаг 1: Определение класса Person

Во-первых, мы определяем класс Person с двумя членами-переменными: name и age. В классе также будут включены два конструктора: конструктор по умолчанию и параметризованный конструктор.

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

class Person {
public:
    std::string name;
    int age;

    // Конструктор по умолчанию
    Person() : name("Unknown"), age(0) {}

    // Параметризованный конструктор
    Person(std::string n, int a) : name(n), age(a) {}
};
```

Пояснение к шагу 1:

- Определение класса: Класс Person определен с публичными членами-переменными name и age для простоты.
- Конструктор по умолчанию: Person() инициализирует name значением "Unknown" и age значением 0. Этот конструктор использует список инициализации (: name("Unknown"), age(0)) для инициализации, что является рекомендуемой практикой для эффективности и ясности.
- Параметризованный конструктор: Person(std::string n, int a) принимает два параметра и использует их для инициализации name и age, соответственно. Это позволяет создавать объекты Person с пользовательскими значениями.

Шаг 2: Основная функция для демонстрации использования конструкторов

Далее, мы напишем функцию main, чтобы создать объекты Person с использованием обоих конструкторов и отобразить инициализированные значения.

```
int main() {
    // Использование конструктора по умолчанию
    Person person1;
    std::cout << "Person 1: " << person1.name << ", " << person1.age << std::endl;

    // Использование параметризованного конструктора
    Person person2("John Doe", 30);
    std::cout << "Person 2: " << person2.name << ", " << person2.age << std::endl;

    return 0;
}
```

Пояснение к шагу 2:

- Создание объектов: Два объекта Person созданы: person1 с использованием конструктора по умолчанию и person2 с использованием параметризованного конструктора с пользовательскими значениями ("John Doe", 30).
- Отображение значений: Программа использует std::cout для вывода имени и возраста обоих объектов Person на консоль. Это демонстрирует, как конструкторы по-разному инициализируют объекты.

2. Реализация Деструкторов в Классе LogFile

Для выполнения задачи 2, которая включает реализацию деструкторов в классе LogFile, мы последовательно разработаем класс, сосредоточив внимание на симуляции сообщений операций с файлами с помощью конструкторов и деструкторов. Эта задача направлена на обучение использованию деструкторов для выполнения задач очистки, таких как освобождение ресурсов или отображение сообщений перед уничтожением объекта. Давайте пройдемся по шагам:

Шаг 1: Определение класса LogFile

Во-первых, мы определим класс LogFile с членом-переменной logMessage для симуляции содержимого файла. Класс будет включать конструктор для симуляции открытия файла и инициализации logMessage, а также деструктор для симуляции закрытия файла с отображением сообщения "Файл закрыт".

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

class LogFile {
public:
    std::string logMessage;

    // Конструктор
    LogFile(std::string message) : logMessage(message) {
        std::cout << "Файл открыт: " << logMessage << std::endl;
    }

    // Деструктор
    ~LogFile() {
        std::cout << "Файл закрыт" << std::endl;
    }
};
```

Пояснение к шагу 1:

- Определение класса: Класс LogFile определен с публичной член-переменной logMessage.
- Конструктор: Конструктор принимает параметр std::string для лог-сообщения и выводит сообщение "Файл открыт" вместе с лог-сообщением, симулируя открытие файла. Используется список инициализации для установки logMessage.
- Деструктор: Деструктор выводит сообщение "Файл закрыт" для симуляции закрытия файла. Он автоматически вызывается, когда объект LogFile выходит из области видимости или удаляется, обеспечивая должное очищение ресурсов.

Шаг 2: Демонстрация использования деструктора

Далее мы напишем функцию вне main для демонстрации автоматического вызова деструктора, когда объект LogFile выходит из области видимости. Кроме того, мы покажем его использование в функции main.

```
void simulateFileOperation() {
    LogFile log("Это лог-сообщение.");
    // Деструктор будет автоматически вызван, когда эта функция закончит выполнение
    // и объект `log` выйдет из области видимости.
}

int main() {
    simulateFileOperation();
    std::cout << "Вернулись в main, после simulateFileOperation" << std::endl;
    return 0;
}
```

Пояснение к шагу 2:

- Функция simulateFileOperation: Эта функция создает объект LogFile с именем log с образцовым лог-сообщением. Цель состоит в том, чтобы симулировать операцию с файлом, где файл "открывается" (создается) и "закрывается" (уничтожается) в пределах области видимости функции. Деструктор класса LogFile вызывается автоматически, как только область видимости функции завершается, демонстрируя, как деструкторы вызываются, когда объекты выходят из области видимости.
- Функция main: Вызывает simulateFileOperation() и затем выводит сообщение, показывающее, что управление вернулось в функцию main после уничтожения объекта LogFile.

3. Конструкторы Копирования в Классе Box

Для выполнения задачи 3, сосредоточенной на реализации копирующих конструкторов в классе Box, мы проведем вас через создание класса с размерами и покажем, как копирующие конструкторы позволяют создавать один объект как копию другого. Эта задача подчеркивает концепцию глубокого копирования, обеспечивая понимание студентами правильной реализации и использования копирующих конструкторов.

Шаг 1: Определение класса Box

Сначала мы определим класс Box с размерами в качестве членов переменных и включим как параметризованный конструктор, так и копирующий конструктор.

```
#include <iostream>

class Box {
public:
    double length;
    double width;
    double height;

    // Параметризованный конструктор
    Box(double l, double w, double h) : length(l), width(w), height(h) {}

    // Копирующий конструктор
    Box(const Box& b) : length(b.length), width(b.width), height(b.height) {
        std::cout << "Вызван копирующий конструктор" << std::endl;
    }

    // Функция для отображения размеров
    void display() const {
        std::cout << "Длина: " << length << ", Ширина: " << width << ", Высота: " << height << std::endl;
    }
};
```

Пояснение к шагу 1:

- Определение класса: Класс Box включает три публичных члена-переменных для представления размеров коробки: length, width и height.
- Параметризованный конструктор: Этот конструктор инициализирует объект Box с указанными размерами. Он полезен для создания нового Box с пользовательскими размерами.
- Копирующий конструктор: Копирующий конструктор принимает ссылку на другой объект Box в качестве своего параметра и использует размеры этого объекта для инициализации нового объекта. Квалификатор const гарантирует, что исходный объект не будет изменен. Обратите внимание на сообщение о том, когда вызывается копирующий конструктор, что помогает понять его использование.
- Функция отображения: Член функция display предоставляется для вывода размеров коробки в стандартный вывод, помогая демонстрировать результаты.

Шаг 2: Демонстрация использования копирующего конструктора

Затем мы используем функцию main, чтобы создать объект Box, а затем создать другой объект Box как копию первого. Этот шаг демонстрирует копирующий конструктор на практике.

```
int main() {
    // Создание объекта Box с использованием параметризованного конструктора
    Box box1(10.5, 8.3, 2.4);
    std::cout << "Размеры Box 1: ";
    box1.display();

    // Создание другого объекта Box как копии первого
    Box box2 = box1; // Вызывает копирующий конструктор
```

```
    std::cout << "Размеры Box 2 (скопированы из Box 1): ";
    box2.display();

    return 0;
}
```

Пояснение к шагу 2:

- Создание первого Box: box1 создается с указанными размерами с использованием параметризованного конструктора.
- Копирование Box: box2 создается как копия box1 с использованием копирующего конструктора. Копирующий конструктор класса Box автоматически вызывается здесь, копируя размеры из box1 в box2.
- Отображение размеров: Размеры как box1, так и box2 отображаются, подтверждая, что box2 действительно инициализирован с теми же размерами, что и box1.

4. Динамическое Поведение с Конструкторами и Деструкторами в Классе Counter

Для решения задачи 4 мы сосредоточимся на реализации класса Counter, чтобы отслеживать динамическое поведение создания и уничтожения объектов с использованием конструкторов и деструкторов. Эта задача демонстрирует использование статических членов переменных и методов для управления и отслеживания состояния через множество экземпляров класса. Давайте пройдемся по шагам:

Шаг 1: Определение класса Counter

Сначала мы определим класс Counter, включая статическую член-переменную для отслеживания количества экземпляров и как конструктор, так и деструктор для модификации этого счетчика. Мы также включим статический метод для отображения текущего счетчика.

```
#include <iostream>

class Counter {
private:
    static int count; // Статическая член-переменная для отслеживания количества экземпляров

public:
    // Конструктор
    Counter() {
        count++; // Увеличение счетчика при создании нового объекта
        std::cout << "Вызван конструктор. Текущий счет: " << count << std::endl;
    }

    // Деструктор
    ~Counter() {
        count--; // Уменьшение счетчика при уничтожении объекта
        std::cout << "Вызван деструктор. Текущий счет: " << count << std::endl;
    }

    // Статический метод для отображения текущего счета
    static void displayCount() {
        std::cout << "Текущий счет: " << count << std::endl;
    }
};

// Инициализация статической член-переменной
int Counter::count = 0;
```

Пояснение к шагу 1:

- Статическая член-переменная: count используется для отслеживания количества экземпляров Counter. Она общая для всех экземпляров класса.
- Конструктор: Увеличивает count каждый раз, когда создается новый объект, демонстрируя динамическое поведение при создании объекта.
- Деструктор: Уменьшает count каждый раз, когда объект уничтожается, показывая динамический аспект уничтожения объекта.
- Статический метод: displayCount используется для отображения текущего значения count. Будучи статическим, он может быть вызван без экземпляра класса.

Шаг 2: Демонстрация поведения класса

Затем мы используем функцию main, чтобы создать и уничтожить объекты Counter в различных областях видимости, наблюдая за автоматическим вызовом конструкторов и деструкторов.

```
int main() {
    Counter::displayCount(); // Отображение начального счета

    Counter a; // Создан первый объект
    Counter::displayCount(); // Отображение счета после создания одного объекта

    {
        Counter b, c; // Созданы еще два объекта
        Counter::displayCount(); // Отображение счета во внутренней области видимости
    } // b и c уничтожены здесь

    Counter::displayCount(); // Отображение счета после уничтожения b и c

    return 0;
}
```

Пояснение к шагу 2:

- Начальное отображение счета: Показывает счет до создания любых объектов.
- Создание первого объекта: Демонстрирует увеличение счетчика конструктором.
- Внутренняя область видимости: Создает два дополнительных объекта b и c, показывая, как счет увеличивается с каждым из них. Область видимости обеспечивает уничтожение b и c, когда она заканчивается, активируя их деструкторы и уменьшая счет.
- Финальное отображение счета: После уничтожения объектов b и c обновленный счет отображается, отражая динамическое поведение конструкторов и деструкторов в управлении жизненным циклом объектов.

5. Перегрузка методов

Цель

Цель данного урока - понять и применить перегрузку методов в C++, расширив функциональность простого класса Calculator. Перегрузка методов позволяет классу иметь несколько методов с одинаковым именем, но разными параметрами. Эта возможность будет продемонстрирована путем введения дополнительного метода calculate для выполнения операций с разным количеством операндов.

Решение и объяснение

Шаг 1: Calculator

Класс Calculator предназначен для выполнения основных арифметических операций: сложения, вычитания, умножения и деления, на основе символа, представляющего операцию. Он включает конструктор для инициализации операции и метод calculate для выполнения операции над двумя операндами.

```
#include <iostream>
#include <cmath> // For additional operations in overloaded method

class Calculator {
private:
    char operation;

public:
    // Constructor
    Calculator(char op) : operation(op) {}

    // Method to perform operation on two doubles
    void calculate(double a, double b) const {
        switch (operation) {
        case '+':
            std::cout << a << " + " << b << " = " << (a + b) << std::endl;
            break;
        case '-':
            std::cout << a << " - " << b << " = " << (a - b) << std::endl;
            break;
        case '*':
            std::cout << a << " * " << b << " = " << (a * b) << std::endl;
            break;
        case '/':
            if (b != 0) std::cout << a << " / " << b << " = " << (a / b) << std::endl;
            else std::cout << "Division by zero!" << std::endl;
            break;
        default:
            std::cout << "Invalid operation" << std::endl;
        }
    }

    // Overloaded method to perform a unary operation (e.g., square root)
};
```

Шаг 2: Введение перегрузки методов

Для демонстрации перегрузки методов мы вводим новую версию метода calculate, который работает с одним операндом. Этот метод может быть использован для унарных операций, таких как вычисление квадратного корня.

```
void calculate(double a) const {
    // Реализация для унарных операций
}
```

Шаг 3: Реализация перегруженного метода

Перегруженный метод calculate проверяет символ операции и выполняет соответствующую унарную операцию. Для демонстрации мы используем операцию извлечения квадратного корня, представленную символом 's'.

```
void calculate(double a) const {
    switch (operation) {
    case 's': // Квадратный корень
        std::cout << "sqrt(" << a << ") = " << sqrt(a) << std::endl;
        break;
    default:
        std::cout << "Недопустимая операция для одного операнда" << std::endl;
    }
}
```

Шаг 4: main

Чтобы использовать перегруженные методы, создайте экземпляры класса Calculator для каждой операции, включая унарную операцию (квадратный корень), и вызовите метод calculate с соответствующим количеством аргументов.

```
int main() {
    Calculator add('+');
    Calculator subtract('-');
    Calculator multiply('*');
    Calculator divide('/');
    Calculator sqrt('s'); // Калькулятор для унарной операции

    double a = 10.5, b = 2.5;

    add.calculate(a, b);
    subtract.calculate(a, b);
    multiply.calculate(a, b);
    divide.calculate(a, b);
    sqrt.calculate(a); // Использование перегруженного метода для унарной операции

    return 0;
}
```

Полный код:

```
#include <iostream>
#include <cmath> // For additional operations in overloaded method

class Calculator {
private:
    char operation;

public:
    // Constructor
    Calculator(char op) : operation(op) {}

    // Method to perform operation on two doubles
    void calculate(double a, double b) const {
        switch (operation) {
        case '+':
            std::cout << a << " + " << b << " = " << (a + b) << std::endl;
            break;
        case '-':
            std::cout << a << " - " << b << " = " << (a - b) << std::endl;
            break;
        case '*':
            std::cout << a << " * " << b << " = " << (a * b) << std::endl;
            break;
        case '/':
            if (b != 0) std::cout << a << " / " << b << " = " << (a / b) << std::endl;
```

```
            else std::cout << "Division by zero!" << std::endl;
            break;
        default:
            std::cout << "Invalid operation" << std::endl;
        }
    }

    // Overloaded method to perform a unary operation (e.g., square root)
    void calculate(double a) const {
        switch (operation) {
        case 's': // Assuming 's' for square root
            std::cout << "sqrt(" << a << ") = " << sqrt(a) << std::endl;
            break;
        default:
            std::cout << "Invalid operation for a single operand" << std::endl;
        }
    }
};

int main() {
    Calculator add('+');
    Calculator subtract('-');
    Calculator multiply('*');
    Calculator divide('/');
    Calculator sqrt('s'); // Unary operation calculator

    double a = 10.5, b = 2.5;

    add.calculate(a, b);
    subtract.calculate(a, b);
    multiply.calculate(a, b);
    divide.calculate(a, b);
    sqrt.calculate(a); // Demonstrating overloaded method usage

    return 0;
}
```

Этот продемонстрировал, как использовать перегрузку методов для добавления гибкости и функциональности к классу в C++. Имея несколько методов calculate в классе Calculator, мы можем обрабатывать различные типы вычислений, что демонстрирует мощь перегрузки методов в объектно-ориентированном программировании. Эта возможность позволяет методам быть использованными для разных целей, делая код более модульным и легче расширяемым.

Самостоятельная задача: Реализуйте простую систему управления библиотекой на C++

Реализуйте простую систему управления библиотекой на C++. Ваша основная задача - спроектировать класс Book, который инкапсулирует свойства и поведение книги, включая название, автора, год публикации и статус доступности. Вы изучите основы объектно-ориентированного программирования, используя конструкторы по умолчанию и параметризованные конструкторы для инициализации объектов книг, реализуя деструктор для понимания управления жизненным циклом объекта, и применяя перегрузку методов для обновления информации о книге различными способами.