5- Практическое задание – Наследование и Демонстрации использования защищенного спецификатора доступа

Table of Contents

Цель

Изучение наследования и защищенных членов в объектно-ориентированном программировании на C++ имеет цель обеспечить глубокое понимание механизмов построения гибких и масштабируемых программных систем. Наследование позволяет классам наследовать или расширять функциональность других классов, способствуя повторному использованию кода и уменьшению его избыточности. Защищенные (protected) члены класса поддерживают принцип инкапсуляции, ограничивая доступ к данным класса, но делая их доступными для классов-наследников. Это обеспечивает безопасность данных и контролируемое их наследование и модификацию, что является ключевым для создания эффективных и надежных объектно-ориентированных приложений.

1. Наследование

В рамках данной задачи мы затрагиваем один из основополагающих принципов объектно-ориентированного программирования (ООП) — наследование. Наследование позволяет создавать новые классы на основе существующих, расширяя их функциональность и переиспользуя код, что способствует уменьшению дублирования кода и упрощению его поддержки.

Пошаговое объяснение кода

Класс Vehicle

Конструктор: Принимает два параметра (string m, int y) и инициализирует приватные атрибуты make и year. Эти параметры представляют марку и год выпуска транспортного средства соответственно.

Методы setMake и getMake: Предоставляют доступ к приватному атрибуту make. Метод setMake позволяет изменить марку транспортного средства, а getMake возвращает текущую марку.

Методы setYear и getYear: Аналогичны методам для работы с маркой, но предназначены для работы с годом выпуска транспортного средства.

Метод displayInfo: Виртуальный метод, который выводит информацию о марке и годе выпуска. Использование ключевого слова virtual позволяет производным классам переопределять этот метод.

Класс Car

Конструктор: Вызывает конструктор базового класса Vehicle для инициализации общих атрибутов, а также инициализирует атрибут isSedan, указывающий, является ли автомобиль седаном.

Методы setIsSedan и getIsSedan: Предоставляют доступ к атрибуту isSedan.

Метод displayInfo: Переопределяет метод базового класса для включения информации о том, является ли автомобиль седаном.

Класс Motorcycle

Конструктор, методы доступа и displayInfo: Работают аналогично классу Car, но для специфичного атрибута hasSidecar, который указывает, имеет ли мотоцикл коляску.

Функция main

Создает объекты классов Car и Motorcycle, инициализируя их с помощью конструкторов.

Вызывает метод displayInfo для каждого объекта, демонстрируя полиморфное поведение: хотя метод вызывается через указатель на базовый класс, выполняется версия метода, соответствующая реальному типу объекта.

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

class Vehicle {
private:
    string make;
    int year;

public:
    Vehicle(string m, int y) : make(m), year(y) {}

    void setMake(string m) {
        make = m;
    }

    string getMake() const {
        return make;
    }

    void setYear(int y) {
        year = y;
    }

    int getYear() const {
        return year;
    }

    virtual void displayInfo() {
        cout << "Make: " << make << ", Year: " << year << endl;
    }
};

class Car : public Vehicle {
private:
    bool isSedan;

public:
    Car(string m, int y, bool isS) : Vehicle(m, y), isSedan(isS) {}

    void setIsSedan(bool isS) {
        isSedan = isS;
    }

    bool getIsSedan() const {
        return isSedan;
    }

    void displayInfo() override {
        Vehicle::displayInfo();
        cout << "Is Sedan: " << (isSedan ? "Yes" : "No") << endl;
    }
};

class Motorcycle : public Vehicle {
private:
    bool hasSidecar;

public:
    Motorcycle(string m, int y, bool hasS) : Vehicle(m, y), hasSidecar(hasS) {}

    void setHasSidecar(bool hasS) {
        hasSidecar = hasS;
    }

    bool getHasSidecar() const {
```

```
        return hasSidecar;
    }

    void displayInfo() override {
        Vehicle::displayInfo();
        cout << "Has Sidecar: " << (hasSidecar ? "Yes" : "No") << endl;
    }
};

int main() {
    Car car1("Toyota", 2020, true);
    Motorcycle motorcycle1("Harley-Davidson", 2019, false);

    car1.displayInfo(); // Демонстрация информации о машине
    motorcycle1.displayInfo(); // Демонстрация информации о мотоцикле

    return 0;
}
```

Заметка:

A. Ключевое слово virtual в C++ используется для объявления функции как виртуальной. Виртуальные функции позволяют реализовать полиморфизм времени выполнения, один из основных принципов объектно-ориентированного программирования. Рассмотрим приведенный синтаксис подробнее:

Синтаксис:

```
virtual void displayInfo(){}
```

- virtual указывает, что метод может быть переопределен в производном классе. Это ключевая часть реализации полиморфизма: виртуальные методы вызываются в соответствии с типом объекта, на который указывает или которым является указатель или ссылка, а не типом указателя или ссылки. Таким образом, если производный класс предоставляет свою версию (переопределение) виртуальной функции, то именно эта версия будет вызвана, даже если вызов происходит через ссылку или указатель на базовый класс.
- void означает, что функция не возвращает значение.
- displayInfo() является именем функции. Эта функция предназначена для отображения (или "показа") информации, и скобки после имени указывают, что функция не принимает параметров.

Значение и использование:

Использование виртуальной функции displayInfo в базовом классе позволяет производным классам переопределить эту функцию для предоставления собственной реализации отображения информации. Это означает, что поведение функции может отличаться в зависимости от типа объекта, для которого она вызывается, даже если этот вызов производится через указатель или ссылку на базовый класс.

Использование virtual обеспечивает гибкость и расширяемость в объектно-ориентированных системах, позволяя новым классам изменять и расширять поведение базовых классов без изменения существующего кода, который использует эти классы

B. override

```
void displayInfo() override {}
```

- void указывает на тип возвращаемого значения функции. В данном случае void означает, что функция не возвращает значение.

- displayInfo() является именем функции. Это имя выбрано для метода, который, как предполагается, выводит (или "отображает") информацию об объекте. Скобки после имени функции указывают на то, что это функция без параметров.
- override является спецификатором, который используется при переопределении функции. Этот спецификатор говорит компилятору, что данная функция предназначена для переопределения виртуальной функции, объявленной в базовом классе. Использование override необязательно для функционирования полиморфизма, но это хорошая практика, так как оно:
  - ✓ Повышает читаемость кода, ясно указывая на намерение переопределить функцию базового класса.
  - ✓ Помогает избежать ошибок: если функция с override не переопределяет виртуальную функцию базового класса (например, из-за опечатки в имени функции или несоответствия сигнатур), компилятор выдаст ошибку.

Зачем использовать override?

Использование override при переопределении виртуальных функций в производных классах обладает несколькими преимуществами:

- Проверка компилятором: Компилятор проверяет, что переопределяемая функция действительно существует в базовом классе и имеет соответствующую сигнатуру. Это предотвращает ошибки, связанные с неправильным переопределением функций.
- Ясность намерений: Слово override ясно указывает другим разработчикам (или вам в будущем), что данная функция предназначена для переопределения функции базового класса, а не для введения новой функциональности.

Поддержка лучших практик программирования: Использование override способствует написанию более чистого и безопасного кода.

В этом контексте, **Vehicle::displayInfo();** демонстрирует механизм вызова метода базового класса из переопределенного метода в производном классе. Этот прием используется в C++ для реализации расширения или дополнения функциональности метода базового класса, а не его полной замены. Давайте разберем этот пример более подробно:

Контекст:

Предположим, у нас есть базовый класс Vehicle с методом displayInfo(), который выводит общую информацию о транспортном средстве. В производном классе, например, Car, мы хотим добавить вывод дополнительной информации, специфичной для автомобилей, такой как информация о том, является ли автомобиль седаном.

Пример использования

```
void displayInfo() override {
    Vehicle::displayInfo(); // Вызов метода displayInfo базового класса Vehicle
    cout << "Is Sedan: " << (isSedan ? "Yes" : "No") << endl; // Дополнительный вывод, специфичный для класса
Car
}
```

Объяснение

Vehicle::displayInfo();:

Эта строка вызывает реализацию метода displayInfo() из базового класса Vehicle. Синтаксис Vehicle:: указывает компилятору на то, что необходимо использовать версию метода из класса Vehicle, даже если мы находимся в контексте производного класса, который переопределяет этот метод.

Этот прием часто используется, когда переопределенный метод в производном классе хочет расширить функциональность метода базового класса, а не полностью заменить ее. Сначала выполняется "базовая" часть операции, общая для всех транспортных средств, а затем добавляется специфическая логика, уникальная для производного класса.

Дополнительный вывод:

После вызова метода базового класса, в методе displayInfo() производного класса Car выполняется дополнительный вывод, который информирует, является ли автомобиль седаном. Это расширяет базовую функциональность, предоставляя более специфичную информацию, связанную с объектом Car.

Важность

Использование Vehicle::displayInfo(); в производном классе для вызова метода базового класса подчеркивает мощь и гибкость наследования в объектно-ориентированном программировании. Это позволяет разработчикам строить сложные иерархии классов, где производные классы могут не только добавлять новую функциональность, но и строить на уже существующей базе, предоставляемой их базовыми классами, обеспечивая тем самым повторное использование кода и сокращение избыточности.

C. **Конструктор класса Person**

```
Person(string n, int a) : name(n), age(a) {}
```

- Person(string n, int a) определяет конструктор для класса Person, который принимает два параметра: n (имя, строка) и a (возраст, целое число).
- : name(n), age(a) является списком инициализации членов. Этот синтаксис говорит компилятору инициализировать члены данных name и age значениями n и a соответственно, переданными в конструктор.
- Фигурные скобки {} после списка инициализации членов обозначают тело конструктора. В данном случае тело конструктора пусто, поскольку все необходимые действия по инициализации членов класса выполняются в списке инициализации.

D. **Конструктор класса Student**

```
Student(string n, int a, string id) : Person(n, a), studentID(id) {}
```

- Student(string n, int a, string id) определяет конструктор для класса Student, который принимает три параметра: n (имя, строка), a (возраст, целое число) и id (идентификатор студента, строка).
- : Person(n, a), studentID(id) также является списком инициализации членов. Здесь происходит два действия:
- Person(n, a) вызывает конструктор базового класса Person, передавая ему параметры n и a. Это инициализирует члены данных name и age базового класса значениями, переданными в конструктор Student.
- studentID(id) инициализирует член данных studentID класса Student значением параметра id.
- Пустые фигурные скобки {} указывают на то, что в теле конструктора Student не выполняется никаких дополнительных действий.

2. Демонстрации использования защищенного (protected) спецификатора доступа

Для демонстрации использования защищенного (protected) спецификатора доступа в наследовании, создадим простую иерархию классов, представляющих собой части системы управления университетом. В этой системе будут классы Person, Student, и Teacher. Защищенные члены класса Person будут доступны в классах-наследниках Student и Teacher, но не будут доступны за пределами этих классов, что иллюстрирует особенности работы спецификатора protected.

Пример кода

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

class Person {
protected: // Защищенные члены класса
    string name;
    int age;

public:
    Person(string n, int a) : name(n), age(a) {}

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

class Student : public Person {
protected: // Дополнительные защищенные члены специфичные для студента
    string studentID;

public:
    Student(string n, int a, string id) : Person(n, a), studentID(id) {}

    void display() {
        Person::display(); // Вызов метода display базового класса
        cout << "Student ID: " << studentID << endl;
    }
};

class Teacher : public Person {
protected: // Дополнительные защищенные члены специфичные для учителя
    string subject;

public:
    Teacher(string n, int a, string sub) : Person(n, a), subject(sub) {}

    void display() {
        Person::display(); // Вызов метода display базового класса
        cout << "Teaches: " << subject << endl;
    }
};

int main() {
    Student student1("Alice", 20, "S12345");
    Teacher teacher1("Bob", 45, "Mathematics");

    student1.display();
    cout << "----------------" << endl;
    teacher1.display();

    return 0;
}
```

Пошаговое объяснение

Класс Person

Спецификатор доступа protected: Члены класса name и age объявлены как protected, что делает их доступными в самом классе Person, а также во всех классах, наследующих Person, но недоступными для других частей программы.

Конструктор: Инициализирует name и age при создании объекта класса.

Метод display: Выводит информацию о человеке. Этот метод будет доступен для вызова из объектов класса Person и его подклассов.

Класс Student

Наследование: Класс Student наследуется от класса Person, что позволяет ему наследовать защищенные члены name и age, а также публичные методы.

Конструктор: Вызывает конструктор базового класса Person для инициализации name и age, а также инициализирует специфичный для студента член studentID.

Метод display: Дополняет метод display базового класса, добавляя вывод информации о studentID. Вызов Person::display() обеспечивает вывод общих данных о человеке.

Класс Teacher

Аналогично классу Student, наследует от Person и расширяет его функциональность, добавляя специфичный для учителя член subject.

Конструктор и метод display работают по тому же принципу, что и в классе Student.

Функция main

Создает объекты Student и Teacher, инициализируя их с помощью их конструкторов.

Вызывает метод display для каждого объекта, демонстрируя их специфические и общие данные.

Самостоятельная задача: Иерархия Классов Животных

Цель этого задания - познакомить студентов с базовыми принципами наследования и использованием защищенного (protected) спецификатора доступа в C++, на примере создания простой иерархии классов животных.

Описание задания

Базовый класс Animal:

- Создайте класс Animal, который будет базовым классом для всех животных.
- Защищенные члены данных класса должны включать name (имя, строка) и age (возраст, целое число).
- Публичный конструктор должен инициализировать эти данные.
- Включите публичный метод makeSound() который будет выводить на экран сообщение "Animal makes a sound".

Производный класс Dog:

- На основе Animal создайте производный класс Dog, представляющий собой собаку.
- Переопределите метод makeSound() для отображения сообщения "Dog barks".

Производный класс Cat:

- На основе Animal создайте производный класс Cat, представляющий собой кошку.
- Переопределите метод makeSound() для отображения сообщения "Cat meows".

Задачи для реализации

1. Реализуйте классы, следуя описанию выше.
2. Создайте объекты Dog и Cat в функции main и продемонстрируйте их функциональность, вызвав метод makeSound() для каждого объекта.

.