类和对象的特性

面向对象程序设计方法概论

什么是面向对象程序设计

面向对象的五个特点

> 1. 对象
> 2. 封装和信息隐蔽
> 3. 抽象
> 4. 继承和重用
> 5. 多态
>
> 数据封装解决的问题是:防止不同模块间数据的非法访问

类的说明和对象的定义

类和对象的关系

类是对象的抽象，而对象是类的具体实例

类是抽象的,不占用内存,对象是具体的,占用内存

类定义

以下为类的定义的例子:

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

class Dog {
private:
    std::string name; // 狗的名字
    int age;          // 狗的年龄

public:
    // 构造函数
    Dog(std::string n, int a) : name(n), age(a) {}

    // 狗的叫声方法
    void bark() const {
        std::cout << "Woof!" << std::endl;
    }

    // 获取狗的信息方法
    std::string get_info() const {
        return name + " is " + std::to_string(age) + " years old.";
    }
};

int main() {
    Dog myDog("Buddy", 3);
```

```
    myDog.bark();
    std::cout << myDog.get_info() << std::endl;
    return 0;
}
```

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

class Car {
private:
    std::string make;  // 制造商
    std::string model; // 型号
    int year;          // 生产年份
    int odometer_reading; // 里程表读数

public:
    // 构造函数
    Car(std::string m, std::string mo, int y) : make(m), model(mo), year(y),
odometer_reading(0) {}

    // 描述汽车的方法
    std::string describe_car() const {
        return std::to_string(year) + " " + make + " " + model;
    }

    // 读取里程表的方法
    int read_odometer() const {
        return odometer_reading;
    }

    // 更新里程表的方法
    void update_odometer(int mileage) {
        if (mileage >= odometer_reading) {
            odometer_reading = mileage;
        } else {
            std::cout << "You can't roll back an odometer!" << std::endl;
        }
    }
};

int main() {
    Car myCar("Toyota", "Corolla", 2020);
    std::cout << myCar.describe_car() << std::endl;
    myCar.update_odometer(50);
    std::cout << "Odometer reading: " << myCar.read_odometer() << std::endl;
    return 0;
}
```

定义对象的方法

先定义类型,再定义变量

如

```
Car myCar("Toyota", "Corolla", 2020);
```

类和结构体的异同

struct虽然也能定义类,但二者有差别:

用struct声明的类，如果对其成员不作private或public的声明，$\blue{默认为piblic。}$

而用class定义的类，如果不作private或public声明$\blue{默认为private。}$

如果希望成员是公用的，用struct比较方便，如果希望部分成员是私有的，宜用class。

类的成员函数

成员函数的性质

私有成员函数只能被本类中的其他成员函数调用，不能被类外调用。成员函数可以访问本类中的任何成员，可以引用在本作用域中有效的数据。

说明:一般的做法是将需要被外界调用的声明为public，它们是类的对外接口。只在本类中使用的成员函数，定义为private.

类外定义成员函数

类外定义成员函数遵循下列模板

```
returnType ClassName::memberFunction(parameters) {
    // 函数体
}
```

以下为一个例子

```
#include <iostream>

// 声明类
class Rectangle {
private:
    double width;
    double height;

public:
    Rectangle(double w, double h); // 构造函数声明
    double getArea() const;       // 成员函数声明
```

```
    void setDimensions(double w, double h); // 设置尺寸的成员函数声明
};

// 在类外定义构造函数
Rectangle::Rectangle(double w, double h) : width(w), height(h) {}

// 在类外定义getArea成员函数
double Rectangle::getArea() const {
    return width * height;
}

// 在类外定义setDimensions成员函数
void Rectangle::setDimensions(double w, double h) {
    width = w;
    height = h;
}
```

内置成员函数

在C++中，inline 关键字用于建议编译器在编译时将成员函数的定义直接插入到每个函数调用的地方，以减少函数调用的开销。inline 成员函数通常用于小型、频繁调用的函数，以减少函数调用的额外开销，如参数传递和栈帧的创建。

以下为一个例子

```
#include <iostream>

class SimpleClass {
public:
    // inline 用于建议编译器在每个调用点展开这个函数
    inline void display() const {
        std::cout << "Display function called." << std::endl;
    }
};

int main() {
    SimpleClass obj;
    obj.display(); // 调用成员函数
    return 0;
}
```

注意:如果在类体外定义inline函数，则必须将类的声明和成员函数的定义都放在同一个头文件中(或者写在同一个源文件中)，否则编译时无法进行置换。只有在类外定义的成员函数规模很小而调用频率较高时，才指定为内置函数

成员函数的储存方式

这边比较重要的是this指针,下面是其使用方式

```
#include <iostream>

class Counter {
private:
```

```
    int count;

public:
    Counter() : count(0) {}

    // 使用this指针来区分成员变量和参数
    void increment(int count) {
        this->count += count; // 使用this指针明确指出count是成员变量
        std::cout << "Count after increment: " << this->count << std::endl;
    }

    // 返回对象本身，允许链式调用
    Counter& operator++() {
        ++this->count; // 使用this指针来增加成员变量count
        return *this;   // 返回对象的引用
    }

    // 显示计数器的值
    void displayCount() const {
        std::cout << "Current count: " << this->count << std::endl;
    }
};

int main() {
    Counter c;
    c.increment(5); // 增加计数器的值
    ++c;            // 递增计数器
    c.displayCount(); // 显示当前计数器的值
    return 0;
}
```

成员对象以及引用

引用类中的对象通常有三种方法

1. 通过对象名和成员运算符访问对象中的成员
2. 通过指向对象的指针访问对象中的成员
3. 通过对象的引用访问对象中的成员

通过对象名和成员运算符访问对象中的成员

我们通常通过 . 运算符来访问

以下为一个例子

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

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

public:
    Person(const std::string& n, int a) : name(n), age(a) {}
```

```
    // 获取名字
    std::string getName() const {
        return name;
    }

    // 获取年龄
    int getAge() const {
        return age;
    }

    // 设置名字
    void setName(const std::string& n) {
        name = n;
    }

    // 设置年龄
    void setAge(int a) {
        age = a;
    }
};

int main() {
    // 创建Person类的对象
    Person person("John Doe", 30);

    // 通过对象名和成员访问运算符访问成员函数
    std::cout << "Name: " << person.getName() << std::endl; // 访问getName成员函数
    std::cout << "Age: " << person.getAge() << std::endl;   // 访问getAge成员函数

    // 直接访问公共成员（虽然这不是一个好的实践，因为违反了封装原则）
    std::cout << "Direct Access - Name: " << person.name << std::endl; // 错误，
name是私有的
    std::cout << "Direct Access - Age: " << person.age << std::endl;   // 错误，
age是私有的

    // 正确的方式是通过公共接口访问
    person.setName("Jane Doe"); // 访问setName成员函数
    person.setAge(35);          // 访问setAge成员函数

    std::cout << "Updated Name: " << person.getName() << std::endl; // 使用更新后的
名字
    std::cout << "Updated Age: " << person.getAge() << std::endl;   // 使用更新后的
年龄

    return 0;
}
```

通过指向对象的指针访问对象中的成员

以下为一个例子:

```
#include <iostream>

class Box {
```

```
private:
    double length; // 长度
    double width;  // 宽度
    double height; // 高度

public:
    Box(double l, double w, double h) : length(l), width(w), height(h) {}

    // 计算体积
    double getVolume() const {
        return length * width * height;
    }

    // 显示盒子的尺寸
    void displayDimensions() const {
        std::cout << "Length: " << length << ", Width: " << width << ", Height: "
<< height << std::endl;
    }
};

int main() {
    // 创建Box类的对象
    Box myBox(5.0, 10.0, 15.0);

    // 创建指向Box对象的指针
    Box* ptr = &myBox;

    // 通过指针访问对象的成员函数
    std::cout << "Volume of the box: " << ptr->getVolume() << std::endl; // 使用指
针访问getVolume成员函数

    // 通过指针访问对象的成员变量（不推荐，违反了封装原则）
    // std::cout << "Length: " << ptr->length << std::endl; // 错误，length是私有的

    // 正确的方式是通过公共接口访问
    ptr->displayDimensions(); // 使用指针访问displayDimensions成员函数

    return 0;
}
```

通过对象的引用访问对象中的成员

```
#include <iostream>

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

    // 构造函数
    Person(const std::string& n, int a) : name(n), age(a) {}

    // 显示个人信息
    void displayInfo() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
```

```
    }
};

// 函数，接受Person对象的引用作为参数
void printPersonInfo(const Person& person) {
    std::cout << "Printing info from function: " << std::endl;
    person.displayInfo(); // 通过引用调用成员函数
    std::cout << "Name via reference: " << person.name << std::endl; // 通过引用访问成员变量
}

int main() {
    Person person("Alice", 30); // 创建Person对象

    person.displayInfo(); // 直接在对象上调用成员函数

    printPersonInfo(person); // 将对象传递给函数，并在函数中通过引用访问成员

    return 0;
}
```