类和对象(下篇)

【本节目标】

- 1. 再谈构造函数
- 2. Static成员
- 3. 友元
- 4. 内部类
- 5. 再次理解封装

1. 再谈构造函数

1.1 构造函数体赋值

> 在创建对象时，编译器通过调用构造函数，给对象中各个成员变量一个合适的初始值。

```
class Date
{
public:
    Date(int year, int month, int day)
    {
        _year = year;
        _month = month;
        _day = day;
    }

private:
    int _year;
    int _month;
    int _day;
};
```

> 虽然上述构造函数调用之后，对象中已经有了一个初始值，但是不能将其称为对对象中成员变量的初始化，**构造函数体中的语句只能将其称为赋初值**，而不能称作初始化。因为**初始化只能初始化一次，而构造函数体内可以多次赋值。**

1.2 初始化列表

> 初始化列表：以一个**冒号开始**，接着是一个以**逗号分隔的数据成员列表**，每个**"成员变量"**后面跟一个**放在括号中的初始值或表达式。**

```
class Date
{
public:
```

```
    Date(int year, int month, int day)
        : _year(year)
        , _month(month)
        , _day(day)
    {}

private:
    int _year;
    int _month;
    int _day;
};
```

【注意】

1. 每个成员变量在初始化列表中**只能出现一次**(初始化只能初始化一次)
2. 类中包含以下成员，必须放在初始化列表位置进行初始化：
   - ☑ 引用成员变量
   - ☑ const成员变量
   - ☑ 自定义类型成员(且该类没有默认构造函数时)

```
class A
{
public:
    A(int a)
        :_a(a)
    {}
private:
    int _a;
};

class B
{
public:
    B(int a, int ref)
        :_aobj(a)
        ,_ref(ref)
        ,_n(10)
    {}
private:
    A _aobj;      // 没有默认构造函数
    int& _ref;    // 引用
    const int _n; // const
};
```

3. 尽量使用初始化列表初始化，因为不管你是否使用初始化列表，对于自定义类型成员变量，一定会先使用初始化列表初始化。

```
class Time
{
public:
```

```
    Time(int hour = 0)
        :_hour(hour)
    {
        cout << "Time()" << endl;
    }
private:
    int _hour;
};

class Date
{
public:
    Date(int day)
    {}

private:
    int _day;
    Time _t;
};

int main()
{
    Date d(1);
}
```

4. **成员变量**在类中**声明次序**就是其在初始化列表中的**初始化顺序**，与其在初始化列表中的先后次序无关

```
class A
{
public:
    A(int a)
        :_a1(a)
        ,_a2(_a1)
    {}

    void Print() {
        cout<<_a1<<" "<<_a2<<endl;
    }
private:
    int _a2;
    int _a1;
};

int main() {
    A aa(1);
    aa.Print();
}

A. 输出1  1
B.程序崩溃
C.编译不通过

D.输出1  随机值
```

1.3 explicit关键字

构造函数不仅可以构造与初始化对象，**对于接收单个参数的构造函数，还具有类型转换的作用。**接收单个参数的构造函数具体表现：

1. 构造函数只有一个参数
2. 构造函数有多个参数，除第一个参数没有默认值外，其余参数都有默认值
3. 全缺省构造函数

```
class Date
{
public:
    // 1. 单参构造函数，没有使用explicit修饰，具有类型转换作用
    // explicit修饰构造函数，禁止类型转换---explicit去掉之后，代码可以通过编译
    explicit Date(int year)
        :_year(year)
    {}

    /*
    // 2. 虽然有多个参数，但是创建对象时后两个参数可以不传递，没有使用explicit修饰，具有类型转换作用
    // explicit修饰构造函数，禁止类型转换
    explicit Date(int year, int month = 1, int day = 1)
        : _year(year)
        , _month(month)
        , _day(day)
    {}
    */

    Date& operator=(const Date& d)
    {
        if (this != &d)
        {
            _year = d._year;
            _month = d._month;
            _day = d._day;
        }

        return *this;
    }
private:
    int _year;
    int _month;
    int _day;
};

void Test()
{
    Date d1(2022);

    // 用一个整形变量给日期类型对象赋值
    // 实际编译器背后会用2023构造一个无名对象，最后用无名对象给d1对象进行赋值
```

```
    d1 = 2023;

    // 将1屏蔽掉，2放开时则编译失败，因为explicit修饰构造函数，禁止了单参构造函数类型转换的作用
}
```

上述代码可读性不是很好，**用explicit修饰构造函数，将会禁止构造函数的隐式转换。**

2. static成员

2.1 概念

声明为**static的类成员**称为**类的静态成员**，用**static**修饰的**成员变量**，称之为**静态成员变量**；用**static修饰**的**成员函数**，称之为**静态成员函数。静态成员变量一定要在类外进行初始化**

面试题：实现一个类，计算程序中创建出了多少个类对象。

```
class A
{
public:
    A() { ++_scount; }
    A(const A& t) { ++_scount; }
    ~A() { --_scount; }
    static int GetACount() { return _scount; }
private:
    static int _scount;
};

int A::_scount = 0;

void TestA()
{
    cout << A::GetACount() << endl;
    A a1, a2;
    A a3(a1);
    cout << A::GetACount() << endl;
}
```

2.2 特性

1. **静态成员**为**所有类对象所共享**，不属于某个具体的对象，存放在静态区
2. **静态成员变量**必须在**类外定义**，定义时不添加static关键字，类中只是声明
3. 类静态成员即可用 **类名::静态成员** 或者 **对象.静态成员** 来访问
4. 静态成员函数**没有**隐藏的**this指针**，不能访问任何非静态成员
5. 静态成员也是类的成员，受public、protected、private 访问限定符的限制

【问题】

1. 静态成员函数可以调用非静态成员函数吗?
2. 非静态成员函数可以调用类的静态成员函数吗?

3. 友元

> 友元提供了一种突破封装的方式，有时提供了便利。但是友元会增加耦合度，破坏了封装，所以友元不宜多用。
>
> 友元分为：**友元函数**和**友元类**

3.1 友元函数

> 问题：现在尝试去重载operator<<，然后发现没办法将operator<<重载成成员函数。**因为cout的输出流对象和隐含的this指针在抢占第一个参数的位置。**this指针默认是第一个参数也就是左操作数了。但是实际使用中cout需要是第一个形参对象，才能正常使用。所以要将operator<<重载成全局函数。但又会导致类外没办法访问成员，此时就需要友元来解决。operator>>同理。

```
class Date
{
public:
    Date(int year, int month, int day)
        : _year(year)
        , _month(month)
        , _day(day)
    {}

    // d1 << cout; -> d1.operator<<(&d1, cout);  不符合常规调用
    // 因为成员函数第一个参数一定是隐藏的this，所以d1必须放在<<的左侧
    ostream& operator<<(ostream& _cout)
    {
        _cout << _year << "-" << _month << "-" << _day << endl;
        return _cout;
    }

private:
    int _year;
    int _month;
    int _day;
};
```

> **友元函数**可以**直接访问**类的**私有**成员，它是**定义在类外部**的**普通函数**，不属于任何类，但需要在类的内部声明，声明时需要加**friend**关键字。

```
class Date
{
    friend ostream& operator<<(ostream& _cout, const Date& d);
    friend istream& operator>>(istream& _cin, Date& d);
public:
    Date(int year = 1900, int month = 1, int day = 1)
        : _year(year)
        , _month(month)
        , _day(day)
    {}

private:
```

```
    int _year;
    int _month;
    int _day;
};

ostream& operator<<(ostream& _cout, const Date& d)
{
    _cout << d._year << "-" << d._month << "-" << d._day;
    return _cout;
}

istream& operator>>(istream& _cin, Date& d)
{
    _cin >> d._year;
    _cin >> d._month;
    _cin >> d._day;
    return _cin;
}

int main()
{
    Date d;
    cin >> d;
    cout << d << endl;
    return 0;
}
```

说明:

- **友元函数**可访问类的私有和保护成员，但**不是类的成员函数**
- 友元函数**不能用const修饰**
- **友元函数**可以在类定义的任何地方声明，**不受类访问限定符限制**
- 一个函数可以是多个类的友元函数
- 友元函数的调用与普通函数的调用原理相同

3.2 友元类

友元类的所有成员函数都可以是另一个类的友元函数，都可以访问另一个类中的非公有成员。

- 友元关系是单向的，不具有交换性。

  比如上述Time类和Date类，在Time类中声明Date类为其友元类，那么可以在Date类中直接访问Time类的私有成员变量，但想在Time类中访问Date类中私有的成员变量则不行。

- 友元关系不能传递

  如果B是A的友元，C是B的友元，则不能说明C时A的友元。

- 友元关系不能继承，在继承位置再给大家详细介绍。

```
class Time
{
   friend class Date;   // 声明日期类为时间类的友元类，则在日期类中就直接访问Time类中的私有成员变量
public:
```

```
    Time(int hour = 0, int minute = 0, int second = 0)
        : _hour(hour)
        , _minute(minute)
        , _second(second)
    {}

private:
   int _hour;
   int _minute;
   int _second;
};

class Date
{
public:
   Date(int year = 1900, int month = 1, int day = 1)
        : _year(year)
        , _month(month)
        , _day(day)
   {}

   void SetTimeOfDate(int hour, int minute, int second)
   {
        // 直接访问时间类私有的成员变量
        _t._hour = hour;
        _t._minute = minute;
        _t._second = second;
   }

private:
   int _year;
   int _month;
   int _day;
   Time _t;
};
```


4. 内部类

概念：**如果一个类定义在另一个类的内部，这个内部类就叫做内部类。**内部类是一个独立的类，它不属于外部类，更不能通过外部类的对象去访问内部类的成员。外部类对内部类没有任何优越的访问权限。

注意：**内部类就是外部类的友元类**，参见友元类的定义，内部类可以通过外部类的对象参数来访问外部类中的所有成员。但是外部类不是内部类的友元。

特性：

1. 内部类可以定义在外部类的public、protected、private都是可以的。
2. 注意内部类可以直接访问外部类中的static成员，不需要外部类的对象/类名。
3. sizeof(外部类)=外部类，和内部类没有任何关系。

```
class A
```

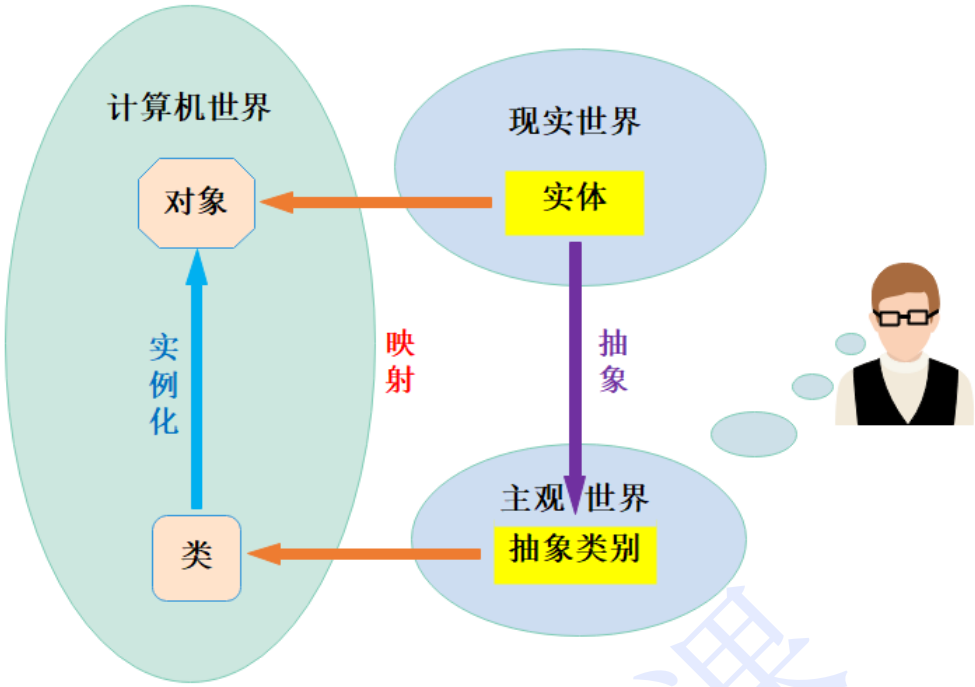
```
{
private:
    static int k;
    int h;
public:
    class B // B天生就是A的友元
    {
    public:
        void foo(const A& a)
        {
            cout << k << endl;//OK
            cout << a.h << endl;//OK
        }
    };
};

int A::k = 1;

int main()
{
    A::B b;
    b.foo(A());

    return 0;
}
```

5. 再次理解类和对象

现实生活中的实体计算机并不认识，计算机只认识二进制格式的数据。如果想要让计算机认识现实生活中的实体，用户必须通过某种面向对象的语言，对实体进行描述，然后通过编写程序，创建对象后计算机才可以认识。比如想要让计算机认识洗衣机，就需要：

1. 用户先要对现实中洗衣机实体进行抽象---即在人为思想层面对洗衣机进行认识，洗衣机有什么属性，有那些功能，即对洗衣机进行抽象认知的一个过程
2. 经过1之后，在人的头脑中已经对洗衣机有了一个清醒的认识，只不过此时计算机还不清楚，想要让计算机识别人想象中的洗衣机，就需要人通过某种面相对象的语言(比如：C++、Java、Python等)将洗衣机用类来进行描述，并输入到计算机中
3. 经过2之后，在计算机中就有了一个洗衣机类，但是洗衣机类只是站在计算机的角度对洗衣机对象进行描述的，通过洗衣机类，可以实例化出一个个具体的洗衣机对象，此时计算机才能洗衣机是什么东西。
4. 用户就可以借助计算机中洗衣机对象，来模拟现实中的洗衣机实体了。

在类和对象阶段，大家一定要体会到，**类是对某一类实体(对象)来进行描述的，描述该对象具有那些属性，那些方法，描述完成后就形成了一种新的自定义类型，才用该自定义类型就可以实例化具体的对象。**



6. 练习题

1.求1+2+3+...+n，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A? B:C[OJ链接](课堂讲解)

2.计算日期到天数的转换 [OJ链接](课堂讲解)

3.日期差值[OJ链接](课后作业)

4.打印日期[OJ链接](课后作业)

5.累加天数[OJ链接](课后作业)