

《面向对象程序设计》朋辈课程 · 第七辑

# 对象生灭进阶

信息学院 赵家宇



PART 01

# 复制构造函数

## 回顾：已经接触过的对象

<code>string s1;</code>	默认构造函数，s1为空串
<code>string s2(s1);</code>	将s2初始化为s1的一个副本(s1可以是字符数组或string)
<code>string s3("Hello");</code>	将s3初始化为一个字符串字面值副本
<code>string s4(n,'c');</code>	将s4初始化为字符'c'的n个副本

- 在创建一个新对象时，用另一个同类的对象对其进行初始化，实现对象的克隆，即用一个已有的对象为依据，快速地复制出多个完全相同的同类对象；把实现这样功能的构造函数称为复制构造函数。

# 复制构造函数

- 复制构造函数也是构造函数，是普通构造函数的重载；其功能为根据实参对象创造新的对象，并将实参对象的各数据成员值一一赋给新的对象中对应的数据成员，使用示例：

`Date d3(d1);`

- 建立对象d3的语句，实参是对象d1；由于在括号内给定的实参是对象，因此将调用复制构造函数，而不会调用其他构造函数；

# 复制构造函数

- 以下三种情况发生时复制构造函数被调用：

①用类的对象去初始化该类的另一个对象时；

```
Cat cat1;
```

```
Cat cat2(cat1); //创建cat2时系统自动调用拷贝构造函数，用cat1初始化cat2
```

```
Cat cat3=cat2; //用类的一个对象去初始化另一个对象时的另外一种形式
```

```
Cat cat4; cat4=cat3; //错误
```

# 复制构造函数

- 以下三种情况发生时复制构造函数被调用：

②对象作为函数的实参传递给函数的形参时，传递过程就是将已有对象拷贝到一个新建的形参对象；

```
feed(Cat a){    }    // 定义feed函数，形参为cat类对象  
Cat b;          // 定义对象b  
feed(b); //进行f函数调用时，系统自动调用拷贝构造函数
```

# 复制构造函数

- 以下三种情况发生时复制构造函数被调用：

③如果函数的返回值是类的对象，函数调用返回时将调用拷贝构造函数。

```
Cat feed(Cat a){  
    return a;  
}    // 定义feed函数，函数的返回值为cat类的对象  
  
Cat b;    // 定义对象b  
  
Cat c = feed(b); //系统自动调用拷贝构造函数，将对象a复制到新创建的临时对象中
```

# 对象的本体和实体

- 声明了类，定义了对象，在内存中就为开辟了一块对象空间。

```
class Date{  
    private:  
        int year,month,day;  
    public:  
        ...  
};
```

year	2024
month	5
day	8

- 如果没有指针关系时，一切成员变量的内存排列在一起，可以连续的进行赋值，我们称之为**对象的本体和实体一致**。



# 对象的本体和实体

- 但当成员中出现了指针变量时，问题变得复杂了一些。

```
class Student {  
    private:  
        string name;  
        int age;  
        bool sex;  
        char* resume;  
        Person* father;  
        Person* mother;  
    public: ...  
};
```

- 此时，如果创建一个对象，则存储的信息不只出现在对象的内存空间中；相应的指针成员不能直接通过简单的赋值来创建，被称为**对象的本体和实体不一致**。
- 此时需要依赖于构造函数，在函数体中对整个对象实体进行充分创建，也即需要分配动态内存空间给对象中的指针成员。

# 对象的本体和实体

- 需要解决对象本体与实体不一致下的函数问题。

```
class Student {  
    private:  
        string name;  
        int age;  
        bool sex;  
        char* resume;  
        Person* father;  
        Person* mother;  
    public: ...  
};
```

- 析构函数：用户自定义析构函数；
- 赋值运算：对于深拷贝，需要对运算符进行重载；
- 对象复制：对于深拷贝，需要自定义拷贝构造函数。

# 深拷贝与浅拷贝

- 在进行数据的复制（拷贝）时，根据拷贝的目标和操作方式的不同，可以分为**深拷贝**和**浅拷贝**两种实现。
- **浅拷贝**是创建新对象时只进行简单的值传递，其成员为原始对象成员的直接复制，在指针出现时会出现重复指向，使得新旧对象无法隔离；
- **深拷贝**是创建新对象时递归复制所有对象的成员，使新对象和原始对象不共享任何内存地址，修改新对象的任何成员都不会影响原始对象。

# 复制构造函数

- 在复制构造函数中，往往要完成以下具体操作：
  - ①创建空间，并且数据成员指向该空间；
  - ②赋值操作（地址复制或值复制）；
- 若数据成员中有指针，需要自行设计复制构造函数与析构函数。

# 复制构造函数的分类

- 复制构造函数分为默认复制构造函数和自定义复制构造函数两类。
- 如果用户自己未定义复制构造函数，则编译系统会自动提供一个默认复制构造函数，其作用只是简单地复制类中每个数据成员；
- 当对象本体与对象实体不一致时（例如成员中包含指针关系），如果需要实现深拷贝，则需要自定义复制构造函数；

# 自定义复制构造函数

- 一旦自定义了复制构造函数，默认的复制构造函数也就不再起作用；
- 复制构造函数的参数是本类的对象，而且采用对象的常量引用形式。

```
<类名> (const <类名>&);
```

- 使用const（可以省略），是为了防止在函数中修改实参对象；
- 使用引用&是防止在参数传递中反复调用复制构造函数，从而造成编译器报错。

# 初始化与赋值

- 初始化与赋值有本质区别，赋值出现在两个对象都已经存在，而初始化出现在创建对象时。

```
class Date {  
    private:  
        int year;  
        int month;  
        int day;  
    public:  
        ...  
}
```

//初始化发生在对象创建时

```
Date d1(2016,03,21);
```

```
Date d2(d1);
```

```
Date d3 = d1;
```

//赋值时，两个对象都已经存在

```
Date d1(2016,03,21);
```

```
d2=d1
```



PART 02

# 用于类型转换的 构造函数



# 类型转换构造函数

- 一个运算中可能涉及多种数据类型，则在进行运算之前，经常需要将一种类型转换成另外一种类型：
- 内置标准数据类型间转换：
  - 隐式转换：如 $5/8$ 和 $5.0/8$
  - 显式转换

# 类型转换构造函数

```
class Cube{
    private:
        double side;
    public:
        Cube(double side); //构造函数
        double volume();    //求体积
        bool compareVolume(Cube aCube);
};
double Cube::volume(){
    return side*side*side;
}
bool Cube::compareVolume(Cube aCube){
    return volume()>aCube.volume();
}
```

```
Cube box1(5.0);
Cube box2(3.0);
box1.compareVolume(box2);
box1.compareVolume(50.0));
```

- 编译器意识到compareVolume()函数的参数应该是一个Cube对象，因此会自动调用转换构造函数，把以50.0作为参数来创建类Cube的一个临时对象，转为  
`box1.compareVolume(Cube(50.0))`
- 即函数不是把box1对象的体积与50.0比较，而是与Cube(50.0)即125000.0比较。

# 类型转换构造函数

- 为了实现其他类型到用户自定义类型之间的转换，必须通过编程来实现类型的转换，即定义含一个参数的构造函数，使得编译器可以使用这种构造函数把参数的类型隐式转换为类类型；
- 类型转换构造函数就是将一个其他类型的数据转换成一个指定的对象，其只有一个形参，用户根据需要在函数体中指导编译器如何进行转换。

# 类型转换构造函数

- 抑制由构造函数定义的隐式转化，可以通过将构造函数声明为explicit。explicit关键字只能用于类内部的构造函数声明上，在类的定义体外部所做的定义上不再重复它；
- 为转换而显式地使用构造函数，任何构造函数都可以用来显式地创建临时对象。

```
class Sales_item {  
    public:  
        explicit Sales_item(const string &book=" ");  
}; //抑制由构造函数定义的隐式转化  
  
item.same_isbn(Sales_item(null_book)); //为转换而显式地使用构造函数
```

The background features a vertical gradient from dark blue on the left to light green on the right. Overlaid on this are three large circles: a solid dark green circle on the left, a large semi-transparent light green circle in the center, and a solid purple circle in the bottom-left corner.

PART 03

# 常成员

# const修饰符

- const类型变量
- const对象：常对象
- 常引用
- 常成员函数
- 常数据成员

# const修饰符

- const类型变量
- const对象：常对象
- 常引用
- 常成员函数
- 常数据成员

## const类型变量

- 用const定义的常变量，一定要对其初始化。而在说明时进行初始化是对此变量置值的唯一方法。即：程序中无法使用 赋值运算符对这种常变量进行赋值。
- 格式：const 类型 变量标识符=初始值；

```
const double PI=3.14;
```

```
PI=3.1415;    //编译无法通过
```



# const类型引用

- 格式: `const 类型 &引用名=初始值;`
- 定义了常引用, 不能通过该引用修改它所引用的对象, 一般将常引用用作形参来保证实参产生不希望的更改 (只能访问), 例如复制构造函数中的常引用。

```
int a=5;  
  
const int &ref=a;
```

# const修饰符

- const类型变量
- 常引用
- 常对象
- 常成员函数
- 常数据成员

# const类型对象

- const类型对象（常对象）可以看做特殊的常变量，因此程序中也无法使用赋值运算符对这种常对象进行赋值（常对象的成员无法修改）。
- 格式：const <类名> <对象名>(初始化);

# const类型成员函数

```
class Sample
{
    int n;
public:
    Sample(int i){ n=i;};
    void print() const;
};

void main()
{
    const Sample a(10);
    a.print();
}

void Sample::print() const
{
    cout<<"2:n="<<n<<endl;
}
```

- 格式：  
    <返回类型> <成员函数名>(参数列表) const;
- const是函数类型的一个组成部分，因此在函数实现部分中也要带有const关键字；若定义常成员函数时丢失了const关键字，程序会产生错误。

# const类型成员函数

```
class Sample
{
    int n;
public:
    Sample(int i){ n=i;};
    void print() const;
};
```

```
void Sample::print() const
{
    n=200;           //error C2166: l-value specifies const object
    setValue();      //error C2662: 'setValue' : cannot convert 'this' pointer
                    //from 'const class Sample' to 'class Sample &'

    cout<<"2:n="<<n<<endl;
}
```

- 常成员函数不能更新对象的数据成员，只能调用const的常成员函数；
- const关键字可以用于参与对重载函数的区分。原则是：常对象调用常成员函数，一般对象调用一般成员函数。

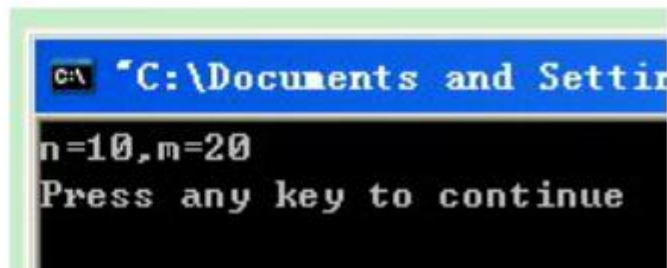
# 常对象与常成员函数

- 常对象不能被更新，通过该常对象只能调用它的常成员函数，而不能调用其他成员函数。

类型	普通数据成员	常数据成员	常对象
普通成员函数	可以访问，也可以修改值	可以访问，但不能修改	不允许访问/修改
常成员函数	可访问，不能修改	可访问，不能修改	可访问，不能修改

# const类型数据成员

```
#include <iostream.h>
class Sample
{
    const int n,m;
public:
    Sample(int i,int j): n(i),m(j){}
    void print()
    {
        cout<<"n="<<n<<","m="<<m<<endl;
    }
};
void main()
{
    Sample a(10,20);
    a.print();
}
```



```
C:\Documents and Settings
n=10,m=20
Press any key to continue
```

- 常数据成员必须进行初始化，且不能被更新；
- 常数据成员的初始化只能通过构造函数的成员初始化列表显式进行。

# 感谢倾听

给个好评！