类的声明

- 在声明之后，定义之前，只知道Screen是一个类名，但不知道包含哪些成员。**只能以有限方式使用它**，不能定义该类型的对象，只能用于定义指向该类型的指针或引用，声明（不是定义）使用该类型作为形参类型或返回类型的函数。

```
class Screen;
void Test1(Screen &a){};        // OK
void Test1(Screen *a){};        // OK
void Test1(Screen a){};         // ------错误
```

如果一个类只声明，则该类大小不确定，只能以有限的方式来使用该类

1. 创建指针
2. 创建引用
3. 注意：不能创建对象(不知道类有哪些成员)

首先说一下一个C++的空类，编译器会加入哪些默认的成员函数

·默认构造函数

·拷贝构造函数

·析构函数

·赋值函数（赋值运算符）

·取值函数

即使程序没定义任何成员，编译器也会插入以上的函数！

注意：构造函数可以被重载，可以多个，可以带参数；

析构函数只有一个，不能被重载，不带参数。

而默认构造函数没有参数，它什么也不做。

• 隐含的 this 指针

- 成员函数具有一个附加的隐含形参，即this指针，它由编译器隐含地定义。成员函数的函数体（成员变量也可以使用）可以显式使用 this 指针。
- this访问类的非静态成员

```
class Sales_item{
private:
    string booksName;
    double booksPrice;
public:
    double getBooksPrice(){
```

```
        return this->booksPrice;        // this的隐式用法(缺省)
    }
};

int main(){
    Sales_item item;
    item.getBooksPrice();
    return 0;
}
```

何时使用 this 指针

- 当我们需要将一个对象作为整体引用而不是引用对象的一个成员时。最常见的情况是在这样的函数中使用 this：该函数返回对调用该类对象的引用。

```
class Screen {
...
public:
    Screen &set(char);
};

Screen &Screen::set(char c) {
    contents[cursor] = c;
    return *this;
}
```

隐式类类型转换

只含单形参的构造函数能够实现从形参类型到该类类型的一个隐式转换

```
class A{
public:
    A(int x){b = x;}
    bool EqualTo(const A& a1){return b == a1.b;}
private:
    int b;
};

A a(1);
bool bEq = a.EqualTo(1);    //参数为1，实现从int型到A的隐式转换
```

抑制由构造函数定义的隐式转换

通过将构造函数声明为 explicit，来防止在需要隐式转换的上下文中使用构造函数：

```
class A{
public:
    explicit A(int a)
    {
        ia = a;
    }
    bool EqualTo(const A& a)
    {
        return ia == a.ia;
    }
private:
    int ia;
};
```

- **通常，除非有明显的理由想要定义隐式转换，否则，单形参构造函数应该声明为 explicit。将构造函数设置为explicit 可以避免错误。**

析构函数

```
class A{
public:
    A(){}
    ~A(){}  // 析构函数
};
```

析构函数作用：

用来释放相关资源

析构函数声明时候被调用：

1. 当创建对象以后，对象被释放(占空间被释放)
2. 调用delete关键字的时候会调用析构函数

面试题：

构造函数能不能重载

析构函数能不能重载

构造函数能不能是虚函数

析构函数能不能是虚函数

拷贝构造函数

```
class A{
public:
    A(){}
    ~A(){}

    A(const A&){}       // 默认拷贝构造函数
};
```

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

class Student{
public:
     Student();
     Student(const Student &);       // 拷贝构造函数/深拷贝
     ~Student();

private:
     char *m_pName;
};

Student::Student(){
    m_pName= new char(20);
     cout << "Student constructor" << endl;
}

// 深拷贝
Student::Student(const Student &s){
    m_pName = new char(20);
    memcpy(m_pName, s.m_pName, strlen(s.m_pName));
    cout << "Student copyconstructor" << endl;
}

Student::~Student(){
    cout << "Student destructor" << endl;
    delete m_pName;
    m_pName = nullptr;
}

int main(){
    Student s1;
    Student s2 = s1;
    return 0;
}
```

执行结果：调用一次构造函数，调用两次析构函数，两个对象的指针成员所指内存相同，这会导致什么问题呢？name指针被分配一次内存，但是程序结束时该内存却被释放了两次，会导致崩溃！这是由于编译系统在我们没有自己定义拷贝构造函数时，会在拷贝对象时调用默认拷贝构造函数，进行的是浅拷贝！即对指针name拷贝后会出现两个指针指向同一个内存空间。

总结1:

在对含有指针成员的对象进行拷贝时，必须要自己定义拷贝构造函数，使拷贝后的对象指针成员有自己的内存空间，即进行深拷贝，这样就避免了内存重复释放发生。

总结2：

> 浅拷贝只是对指针的拷贝，拷贝后两个指针指向同一个内存空间，深拷贝不但对指针进行拷贝，而且对指针指向的内容进行拷贝，经深拷贝后的指针是指向两个不同地址的指针。

再说几句：

当对象中存在指针成员时，除了在复制对象时需要考虑自定义拷贝构造函数，还应该考虑以下两种情形：

1.当函数的参数为对象时，实参传递给形参的实际上是实参的一个拷贝对象，系统自动通过拷贝构造函数实现；

2.当函数的返回值为一个对象时，该对象实际上是函数内对象的一个拷贝，用于返回函数调用处。

3.浅拷贝带来问题的本质在于析构函数释放多次堆内存，使用std::shared_ptr，可以完美解决这个问题。

友元(friend)

- 友元机制允许一个类将对其非公有成员的访问权授予指定的函数或类。
- 友元可以出现在类定义的内部的任何地方。
- 友元不是授予友元关系的那个类的成员，所以它们不受声明出现部分的访问控制影响。
- 友元不能被继承
- 建议：将友元声明成组地放在类定义的开始或结尾。

友元：

1. 友元函数
2. 友元类

友元类

```
class Husband{
public:
    friend class Wife;
private:
    double money;               // 钱是老公私有的，别人不能动，但老婆除外
};

class Wife{
public:
    void Consume(Husband& h){
        h.money -= 10000;       // 老婆可以花老公的钱
    }
};

int main(){
    Husband h;
    Wife w;
    w.Consume(h);
    return 0;
```

```
}
```

友元函数

```
class Husband;          //1.声明Husband

class Wife              //2.定义Wife类
{
public:
    void Consume(Husband& h);
};

class Husband           //3.定义Husband类
{
public:
    friend void Wife::Consume(Husband& h);          // 声明Consume函数。
private:
    double money;                                   // 钱是老公私有的，别人不能动，但老婆除外
};

void Wife::Consume(Husband& h)                      // 4.定义Consume函数。
{
    h.money -= 10000;                               // 老婆可以花老公的钱
}
```

C++的扩展：单例模式

1.1 单例模式分类

单例模式可以分为懒汉式和饿汉式，区别在于创建实例的阶段不同

懒汉式（线程不安全）：程序运行时，当需要使用该实例时，才创建并使用实例。
饿汉式（线程安全）：程序启动时就创建实例并初始化，当需要时直接调用即可。

1.2 单例类特点

构造函数和析构函数为private类型，禁止外部构造和析构
拷贝构造和赋值构造函数为private类型，目的是禁止外部拷贝和赋值，确保实例的唯一性
类中有一个获取实例的静态函数getInstance，可以全局调用

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

class Singleton{
public:
    static Singleton *getInstance(){
        if(m_pInstance == nullptr)
            m_pInstance = new Singleton;
        return m_pInstance;
    }
private:
```

```
    Singleton(){
        cout << "This is Singleton constructor" << endl;
    }
    ~Singleton(){
        cout << "This is Singleton destructor" << endl;
    }

private:
    static Singleton *m_pInstance;
};

// static成员变量需要在类外初始化
Singleton *Singleton::m_pInstance = nullptr;

int main(){
    Singleton *pInstance_1 = Singleton::getInstance();  // 必须得通过这个接口来创建单例
    Singleton *pInstance_2 = Singleton::getInstance();  // 必须得通过这个接口来创建单例
    return 0;
}
```

缺省参数

C++支持函数缺省参数

注意：C语言不支持默认参数

```
// 在函数参数中,如果出现等号"=",则表明参数缺省
// 缺省参数特点: 如果参数不写,则默认使用等号后面的数据,如果添加了参数,则使用添加的参数
void func1(int x = 100){
    cout << "x = " << x << endl;
}

void func2(int x, int y = 300){
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}

void func3(int x = 160, int y = 200){
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}

int main(){
    // func1(200);
    // func2(400, 120);
    func3(99, 88);
    return 0;
}
```

注意1：缺省参数是由顺序的，当参数个数不止一个时，参数缺省应该从右向左缺省

注意2：当缺省参数的函数重载时要注意传参的类型和个数，否则会造成歧义错误

```
void func3(int x = 160, int y = 200){
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
}

void func3(int x = 160){
    cout << "x = " << x << endl;
}

int main(){
    // func1(200);
    // func2(400, 120);
    func3(99);      // 歧义错误：ambiguous
    return 0;
}
```

类的构造函数也支持缺省参数

```
class A{
public:
    A(int x = 100){
        cout << "x = " << x << endl;
        cout << "This is A's constructor" << endl;
    }
};

int main(){
    A a1;       // x=100
    A a2(300);  // x=300
    return 0;
}
```

问题：

C语言支不支持函数重载（C语言不支持重载）

在这里printf函数是不定参数,参数形式：printf(...)

```
#include <stdio.h>

int main(){
    int a = 10;
    printf("a = %d\n", a);
    printf("Hello world\n");
    return 0;
}
```