本章所含内容

概述

对于一个类，我们显式或隐式地指定该类型对象的拷贝，赋值，移动和销毁时该做什么

拷贝构造函数`copy constructor`和移动构造函数`move constructor`定义了用另一个同类型对象**初始化**本对象时该做的事

拷贝赋值运算符`copy assignment operator`和移动赋值运算符`move assignment operator`定义了用另一个同类型对象**赋值**本对象时该做的事

析构函数`destructor`定义了销毁本对象时该做的事

问题

- 为什么拷贝构造函数的参数是引用类型?

当我们调用拷贝函数时，如果参数是`pass by value`，那么当我们需要用参数来初始化本对象时，我们需要先对参数调用一次拷贝构造函数

为了对参数调用拷贝构造函数，此时又会回到刚才的过程，也就是拷贝构造函数的调用永远不会成功

也就是:

为了调用拷贝构造函数，我们需要先拷贝实参，而为了拷贝实参，我们需要调用拷贝构造函数

拷贝构造函数

拷贝构造函数的第一个参数必须是**引用**，并且由于拷贝构造函数通常会被隐式调用，因此**不应当被声明为**`explicit`

拷贝构造函数在一下情况下会调用:

- 用 等号`=` **初始化**对象

```
string s = "NishikiChisato
```

- 函数形参为`pass by value`

```
void fun(string rhs)
```

- 从一个返回类型为**非引用**类型的函数返回一个对象

```
string fun();
string res = fun();
```

- 用**花括号列表**初始化一个**数组**或者一个**聚合类**的成员

```
string arr[2] = { "Nishiki", "Chisato" };
```

注：当我们对标准库容器调用push或insert时，会调用其拷贝构造函数，如果调用emplace时，则会直接初始化（后者开销更小）

拷贝赋值运算符

区分两种情况分别调用什么函数：

```
//情况一
string s1 = "NishikiCHisato";

//情况二
string s2 = "NishikiCHisato";
string s3 = s2;
```

第一种情况，会**隐式**调用**拷贝构造函数**

第二种情况，才会调用**拷贝赋值运算符**

拷贝赋值运算符写法

```
class Foo
{
public:
    Foo& operator=(const Foo& rhs);
}
```

拷贝赋值运算符的返回值通常为一个**指向左侧对象的引用**

如果用户没有定义copy-assignment operator，那么编译器为自动为该类生成一个copy-assignment operator，称之为**合成拷贝赋值运算符**synthesized copy-assignment operator

对于合成拷贝赋值运算符，会将**右侧**运算对象的所有**非static**对象赋值给左侧运算对象的对应成员，并返回一个指向左侧对象的引用

析构函数

构造函数会初始化**非static**成员的值，析构函数会销毁**非static**成员的值

析构函数不接受参数，因此它**不能被重载**，因此对于一个给定类，只会有**一个**析构函数

在构造函数中，成员的初始化是在**构造函数的函数体执行之前完成的**，并且按照它们在**类中出现的顺序初始化**

在析构函数中，**首先执行函数体**，随后**再销毁成员**，销毁顺序会按照成员出现的顺序销毁

调用析构函数的时机：

- 当对象离开其作用域时，会调用该对象的析构函数

- 对一个对象被调用析构函数时，其成员也会被调用析构函数
- 容器（标准库或数组）被销毁时，其元素同样会被销毁
- 对指针采取`delete`时，会销毁该指针所指向的对象
- 对于临时对象，在本条语句执行完之后调用其析构函数将其销毁

如果类未定义析构函数，那么编译器会自动生成一个析构函数，称为**合成析构函数**`synthesized destructor`，该函数内部为**空实现**

需要注意的是，成员的销毁是在**析构函数体执行完之后再执行的**，即析构函数体是作为成员销毁的另一部分而进行的