OOP 之 复制控制

目录：

对象复制的时机：

根据一个类去显式或者隐式初始化一个对象
复制一个对象，将它作为实参传给一个函数
从函数返回时复制一个对象

那么如何完成对象复制的工作？这里需要的就是复制构造函数

复制构造函数

只有单个形参，而且该形参是本类类型对象的引用（常用 const 修饰），这样的构造函数成为复制控制函数

复制构造函数调用的时机就是在对象复制的时候。

如果什么也不做，编译器会自动帮我们合成一个默认的复制

构造函数，例如：

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

using namespace std;

class Student {
public:
	Student() :
		_id(0), _name("none"), _score(0) {
	}

	void debug() {
		cout << _id << " " << _name << " " << _score << endl;
	}


private:
	int _id;
	string _name;
	int _score;

};

int main(int argc, char **argv) {
	Student s1(23, "test", 77);
	Student s2;
	s1.debug();
	s2.debug();

}
```

观察程序运行的结果可以看到编译器提供的复制构造函数工作正常。

那么如果我们自己来定义复制构造函数，应该怎么写？

仍然是刚才的代码，我们加上：

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

using namespace std;

class Student {
public:
	Student() :
		_id(0), _name("none"), _score(0) {
	}
	Student(int id, const string &name, int score) :
		_id(id), _name(name), _score(score) {
	}


	void debug() {
		cout << _id << " " << _name << " " << _score << endl;
	}

	Student &operator=(const Student &rhs) {
		_id = rhs._id;
		_name = rhs._name;
		_score = rhs._score;
		return *this;
	}

private:
	int _id;
	string _name;
	int _score;

};

int main(int argc, char **argv) {
	Student s1(23, "test", 77);
	Student s2;
	s1.debug();
	s2.debug();
}
```

结果仍然是正确的。

现在来思考一个问题，既然编译器生成的复制构造函数工作

正常，那么什么时候需要我们自己来编写复制构造函数呢？

这就是下面的深拷贝和浅拷贝的问题。

深拷贝和浅拷贝

下面我们来实现一个简易的 String 类

头文件如下：

```
#ifndef STRING_H_
#define STRING_H_

#include <iostream>
#include <string.h>

namespace __str {

class string {
public:
	string();
	string(const char *);
	void debug();
	std::size_t size() const;
	~string();
private:
	char *_str;

};

} /* namespace __str */

#endif /* STRING_H_ */
```

Cpp 文件如下：

```
#include "_string.h"

namespace __str {

string::string() {
```

```
        _str = new char;
        _str[0] = 0;
}
string::string(const char *s) {
        _str = new char[strlen(s) + 1];
        strcpy(_str, s);
}

std::size_t string::size() const {
        return strlen(_str);
}

void string::debug() {
        std::cout << _str << std::endl;
}

string::~string() {
        delete[] _str;
}

} /* namespace __str */
```

看起来似乎正常，但是在 main 中运行程序就会崩溃，原因在哪里？

因为系统合成的复制构造函数，在复制 String 对象时，只是简单的复制其中的 str 的值，这样复制完毕后，就有两个 String 指向同一个内存区域，当对象析构时，发生两次 delete，导致程序错误

如何解决？

方案很简单，就是我们在复制 String 时，不去复制 str 的值，而是复制其指向的内存区域。

我们自定义复制构造函数如下：

```
string::string(const string &s) {
    _str = new char[s.size() + 1];
    strcpy(_str, s._str);
}
```

这两种复制对象的区别就是深拷贝和浅拷贝，定义如下：

深拷贝：

浅拷贝：

赋值运算符：

前面的复制构造函数说的是对象的复制，对象的赋值调用的则是对象的赋值运算符

这是我们首次接触到运算符重载，运算符重载是什么？

对于我们自定义的类（例如 Student），我们是无法进行比较操作的，因为我们自定义的类没有内置比较运算符（<= < > >= == !=），此时我们就可以通过运算符重载的规则给这些类加上运算符

这里我们需要重载的就是赋值运算符

例如 Student，如下：

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

using namespace std;
```

```
class Student {
public:
    Student() :
            _id(0), _name("none"), _score(0) {
    }
    Student(int id, const string &name, int score) :
            _id(id), _name(name), _score(score) {
    }


    void debug() {
        cout << _id << " " << _name << " " << _score << endl;
    }

    Student &operator=(const Student &rhs) {
        _id = rhs._id;
        _name = rhs._name;
        _score = rhs._score;
        return *this;
    }

private:
    int _id;
    string _name;
    int _score;

};

int main(int argc, char **argv) {
    Student s1(23, "test", 77);
    Student s2;
    s1.debug();
    s2.debug();

    s2 = s1;     // =

    s2.debug();

}
```

当然，如果我们什么也不做，系统也会自动合成一个赋值

运算符，但是什么时候需要我们自己来重载赋值运算符呢，仍然是考虑深拷贝和浅拷贝的问题

String 的赋值运算符如下：

```
string &string::operator =(const string &s) {
    if (this != &s) {
        delete[] _str;
        _str = new char[s.size() + 1];
        strcpy(_str, s._str);
    }
    return *this;
}
```

这里有两处值得注意的地方，一是赋值运算符应该检查自身赋值的情况（想想为什么），另一处是要返回自身的引用。

析构函数

对象销毁时调用的函数。一般用于释放动态分配的资源。

参考这个实例：

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

class Test {
public:
    Test() {
        cout << "construct" << endl;
    }
    ~Test() {
        cout << "destroy" << endl;
    }
};

int main(int argc, char **argv) {
```

```
    //  Test t;
        {
    //      Test t;
            Test *p = new Test[10];
    //      Test &t2 = t;
            delete[] p;
        }

        cout << "test" << endl;
    }
```

对于 String 类则是这样编写的：

```
string::~string() {
    delete[] _str;
}
```

此时，String 类完整的源码如下：

```
#ifndef STRING_H_
#define STRING_H_

#include <iostream>
#include <string.h>

namespace __str {

class string {
public:
    string();
    string(const char *);
    string(const string &);
    void debug();
    std::size_t size() const;
    ~string();
    string &operator=(const string&);

private:
    char *_str;

};
```

```
} /* namespace __str */

#endif /* STRING_H_ */
```

CPP 文件如下：

```
#include "_string.h"

namespace __str {

string::string() {
	_str = new char;
	_str[0] = 0;
}
string::string(const char *s) {
	_str = new char[strlen(s) + 1];
	strcpy(_str, s);
}

string::string(const string &s) {
	_str = new char[s.size() + 1];
	strcpy(_str, s._str);
}

std::size_t string::size() const {
	return strlen(_str);
}

void string::debug() {
	std::cout << _str << std::endl;
}

string::~string() {
	delete[] _str;
}

string &string::operator =(const string &s) {
	if (this != &s) {
		delete[] _str;
		_str = new char[s.size() + 1];
		strcpy(_str, s._str);
	}
	return *this;
}
```

```
} /* namespace __str */
```

禁止复制：

如果想禁止复制一个类，应该怎么办？

显然需要把类的复制构造函数设为 private，但是这样以来类的 friend 仍然可以复制该类，于是我们只声明这个函数，而不去实现。

另外，如果你不需要复制该类的对象，最好把赋值运算也一并禁用掉。

所以这里的做法是：把复制构造函数和赋值运算符的声明设为 private 而不去实现。

后面更通用的做法是写一个类 noncopyable，凡是继承该类的任何类都无法复制和赋值

三法则：

如果类需要析构函数，那么它也需要复制构造函数和赋值运算符。