OOP 之类和对象

--面向对象的第一个特征是：数据抽象

目录

带有函数的结构体：

前面我们写过这样一个结构体：

```
struct Person{
    int _id;
    string _name;
    int _age;
};
```

在 C 语言中，结构体就是这样定义，只可以含有成员变量，不能含有函数，但在 C++中的结构体可以加上函数。

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

struct Person {
```

```
    int _id;
    string _name;
    int _age;

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

int main() {

    Person p1;
    p1._age = 99;
    p1._id = 00123;
    p1._name = "Jack";

    p1.print(cout);
}
```

从上面的代码可以看出： 结构体内部的确可以带有函数，而且这些函数可以直接访问里面的数据。

两种方式访问对象成员

对于结构体内的成员，包括变量和函数，有两种访问方式：

如果使用的是结构体类型的变量，则采用.的形式。

如果使用的是指针操纵结构体，采用->的形式。例如：

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

struct Person {
    int _id;
    string _name;
    int _age;

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};
```

```
int main() {

    Person p1;
    p1._age = 99;
    p1._id = 00123;
    p1._name = "Jack";
    p1.print(cout);

    Person *p2 = new Person;
    p2->_age = 34;
    p2->_id = 12345;
    p2->_name = "hello";
    p2->print(cout);


}
```

这里提前说明一点：结构体内部的函数包含了一个隐藏的参数，所以这使得这个参数可以直接调用他所在结构体的成员变量。

此时我们的结构体已经具有了成员变量和成员函数，这就是类。

类

每个人都有自己的个人信息，其中的有些向周围的人公开，如姓名。这样的信息被成为公有的。然后，有些个人信息可能不想让别人知道，例如收入。这种信息是私有的，通常保密。

C++允许程序员将类的属性和方法设为公有的，这意味着有了对象之后就可以获取他们；也可以将其声明为私有的，这意味着只能在类的内部去访问。C++提供了两个关键字，private 和 public 分别可以把元素设为私有的和公开的。

public：元素是公开的，任何位置都可以访问。

private：成员是私有的，只能在类的内部访问。

我们一般采取的方案是：把类的成员变量设为 private，把类的成员函数设为 public。

上面的结构体中，我们把三个属性设为 private，但是这样一来，三个变量无法访问，于是我们为每个变量提供 get 和 set 方法。

代码如下：

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

class Person {
private:
    int _id;
    string _name;
    int _age;

public:
    int get_id() const {
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
```

```
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

int main() {

    //编译错误
    Person p1;
    p1._age = 99;
    p1._id = 00123;
    p1._name = "Jack";
    p1.print(cout);

    Person *p2 = new Person;
    p2->_age = 34;
    p2->_id = 12345;
    p2->_name = "hello";
    p2->print(cout);

}
```

很显然代码编译有问题，因为三个属性已经设为 private，不可以直接访问它们。这时只能通过 get 和 set 来访问某个变量。

将 main 里面的内容改为：

```
Person p1;
p1.set_age(23);
p1.set_id(1234);
p1.set_name("hello");
p1.print(cout);
```

此时就没有问题了。

成员函数

类的成员函数与其他的函数类似，和任何函数一样，成员函数也包含下面四个部分：

1.函数返回类型

2.函数名

3.形参表

4.函数体

在前面的代码中，我们已经很清楚的看到成员函数如何编写，下面是一些注意点：

1.成员函数含有额外的、隐含的形参

调用成员函数时，实际上是使用对象来调用的。例如上面的 p1.print(cout);，在这个调用中，传递了对象 cout，用 cout 初始化形参 os，但是 print 如何知道打印哪个对象的属性呢？这里实际上把 p1 也作为一个参数传递给了 print。这个隐含的参数将该成员函数和调用该函数的对象捆绑在一起。所以这个函数调用中，print 隐式调用了 p1 这个对象的成员。

2.this 指针的引入

每个成员函数都有一个额外的隐含的形参，这个参数就是 this 指针，它指向调用对象的地址。

在 p1.print(cout)这个代码中，print 中的 this 指针就是 p1 这个对象的地址。

this 指针一般用于解决重名问题和返回自身的值或者引用。例如：

```
struct A{
    int a;

    void test(int a){
        this->a = a;
    }
```

};

test 函数的形参 a 和类成员 a 成名，根据就近原则，直接使用 a，调用的是形参 a，那么如何使用被屏蔽的成员 a 呢，这里就是采用 this 指针。

this 指针的其他用途在本章后面可以看到。

3.const 成员函数

上面我们给每个成员加入了 get 和 set 方法，大家可能已经注意到我们给 get 方法加上了一个 const。这里加 const 的含义是，这个函数不能修改本对象，其实就是函数体内不得对类的成员进行修改。const 主要起到保护的作用。

这里有一点：普通对象可以调用 const 函数，也可以调用非 const 函数，但是 const 对象只能调用 const 函数。

为什么是这样？ 我们不去试图从复杂的语法角度解释，而是联想程序的语义。const 关键字的含义就是**不希望**我们去更改，而不加 const 就是**期望（不一定真的修改，但是程序希望这么做）**我们去修改对象。这样我们在 const 对象里面调用非 const 函数，**我们的意图就显得有些矛盾！**

例如：

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

class Person {
private:
    int _id;
    string _name;
    int _age;
```

```
public:
    int get_id() const {
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};



int main() {

    const Person p;
    p.get_age(); //OK
    p.set_age(12); //编译错误!!!!

}
```

请自行查看编译错误的信息。

4. const 成员函数和普通函数可以构成重载

例如：

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

class Person {
private:
    int _id;
    string _name;
    int _age;

public:
    int get_id() const {
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
```

```
};

int main() {

    const Person p1;
    Person p2;
    p2.set_age(12);

    p1.print(cout);
    p2.print(cout);

}
```

这里我们提供了一个 print 的 const 版本，编译通过，说明没有发生函数的重新定义。所以这里的两个 print 构成了重载的关系。

到此为止，构成函数重载的要素有：

1.函数名

2.函数形参表

3.类的名称

4.**成员函数的 const 属性**

事实上，如果你听说过函数签名的概念，那么函数的签名就是由这几个部分构成。

在这里我们解释一个问题： 为什么 C 语言里面没有函数重载？ 在编译器编译程序的时候会维护一张**符号表**，C 语言在记载函数的时候就是简单的记录函数的**名字**，所以名字就是 C 函数的唯一标识。当我们试图定义两个名字相同的函数时，就发生了重定义。

C++是怎么做的呢？ 很显然，对于普通函数，它的符号（**唯一标识**）是根据名字和参数表生成的，对于类的成员函数，还要加上类名和

const 属性，所以我们进行函数重载的时候，这些函数在符号表中的标识是不相同的。 **C++正是通过这种机制实现了函数的重载**。

注意：C++编译器生成函数符号的时候没有考虑返回值，这也是函数重载和返回值无关的原因。

构造函数

构造函数是特殊的成员函数，与其他成员函数不同，构造函数与类同名，而且没有返回类型。而与其他成员函数相同的是，构造函数也有形参表（可能为空）和函数体。一个类可以有多个构造函数，每个构造函数必须有与其他构造函数不同数目或类型的形参。

对于 Person 类，可以这样编写构造函数：

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

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person(int id, const string &name, int age){
        _id = id;
        _name = name;
        _age = age;
    }


    int get_id() const {
```

```
        return _id;
    }

    void set_id(int id) {
        _id = id;
    }

    string get_name() const {
        return _name;
    }

    void set_name(const string &name) {
        _name = name;
    }

    int get_age() const {
        return _age;
    }

    void set_age(int age) {
        _age = age;
    }

    void print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

int main() {

    Person p(12, "zhangsan", 12345); //调用我们自己编写的构造函数
    p.print(cout);

}
```

这里注意，构造函数是自动执行，不需要我们去显示调用，而且也不允许手工调用。

1.初始化式

构造函数有一个特殊的地方，就是它可以包含一个构造函数初始化列表：

```
Person(int id, const string &name, int age)
        :_id(id), _name(name), _age(age){
}
```

构造函数初始化列表有个地方难以理解，因为我们这样写也完全可以达到目的：

```
Person(int id, const string &name, int age){
        _id = id;
        _name = name;
        _age = age;
}
```

实际上在有些时候我们必须使用初始化列表：

a)没有默认构造函数的类成员

b)const 成员

c)引用类型的成员

d)有的类成员需要显式调用含参数的构造函数

练习：　自行写程序验证以上的几种类型。

这里有一处陷阱：

考虑下面的类：

```
class X {
    int i;
    int j;
public:
    X(int val) :
            j(val), i(j) {
    }
};
```

我们的设想是这样的，用 val 初始化 j，用 j 的值初始化 i，然而这

里初始化的次序是先 i 然后 j。

记住：类成员初始化的顺序是它们在类中声明的顺序，而不是初始化列表中列出的顺序！

2.默认构造函数

前面我们编写了 Person 类，但是现在这样语句会发生错误：

Person p;

本来这样是正确的，为什么我们自定义了构造函数就错误了？原因是因为如果我们什么都不做，那么编译器自动为我们合成一个默认的无参数的构造函数，类似于：

```
Person() :
                _id(), _name(), _age() {

    }
```

里面每个成员都进行默认初始化。

但是，当我们自行编写了构造函数的时候，编译器就不再为我们提供默认无参数的构造函数，所以我们只能自己提供上面的无参数的构造函数。

当使用 Person *p = new Person[10]; 这段代码生成了 10 个 Person 对象，如果 Person 没有无参数的构造函数，那么这段代码无法通过编译。

练习： 先不为 Person 提供无参数的构造函数，看看编译的错误信息是什么。

3.构造函数的重载

上面我们提供了 Person 的两个构造函数，这就构成了构造函数的

重载。

```
Person() :
        _id(), _name(), _age() {

}

Person(int id, const string &name, int age) :
        _id(id), _name(name), _age(age) {
}
```

最后记住：　构造函数不能为 const。

析构函数

与构造函数一样，析构函数也是一种特殊的函数。构造函数在对象被创建时调用，析构函数则是在对象被销毁时被调用。

析构函数看起来有些奇怪，它的名字和类名也相同，只是前面多了一个~，Person 类的析构函数如下：

```
    ~Person(){

        }
```

同样没有返回值，而且析构函数没有任何参数。

再谈 this 指针

this 指针最大的作用是返回自身的引用，刚才我们把 Person 的 set 函数返回值设为 void，现在我们改成这样：

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

class Person {
```

```
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
            _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
            _id(id), _name(name), _age(age) {
    }

    Person &set_id(int id) {
        _id = id;
        return *this;
    }

    Person &set_name(const string &name) {
        _name = name;
        return *this;
    }

    Person &set_age(int age) {
        _age = age;
        return *this;
    }

    Person &print(std::ostream &os) {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
        return *this;
    }

    const Person &print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
        return *this;
    }

};

int main(int argc, char **argv) {
    Person p;
```

```
        p.print(cout);
        p.set_id(12).print(cout).set_age(22).print(cout).set_name("hello").print(
                cout);
}
```

我们做了这么几处改动：将函数的返回值改为返回自身引用，同时为 print 提供了一个重载版本。

函数返回对象自身引用的目的是为了写出:

p.set_id(12).print(cout).set_age(22).print(cout).set_name("hello").print(cout);这样的连续调用式。

如果返回值不加引用，就达不到这样的效果，请自行尝试。

还有一处注意点，const 函数在返回对象成员引用（指针）或者对象自身的引用（指针）时，**必须将返回值设为 const**，否则**造成语意上的矛盾**。

这里很好解释：**const 代表我们不希望进行修改，但是返回一个非 const 引用（它可以作为左值）就为外界修改对象提供了途径，这与 const 的语义是矛盾的。**

这里调用完 print 仍需要进行 set，所以我们提供 print 的两个版本。

static 静态成员

前面我们提到了成员变量，那么类的每个对象中均存在一个相应的该变量，假设某个类 Point 中含有 x 变量，Point 生成了 100 个对象，那么这 100 个对象均含有一份 x。

如果某个变量属性在某一个类的所有对象中的值均相同，那么把它声明为普通的成员变量，就会造成内存中大量的重复变量。解决方

案是把该变量声明为 static。这样，这个变量不再单独属于某一个对象，而是属于整个类。

```
class   Account{
public:

    Account(){
        ++sum_;
    }
    ~Account(){
        --sum_;
    }

    int show_num();

private:
    static int sum_;   //统计生成了多少对象
};

int Account::sum_ = 0;
int Account::show_num(){
    return sum_;
}
```

上面例子中，sum 是统计 Account 一共生成了多少对象，所以这个数值应该是所有对象集体共享的。

static 变量不与 this 指针绑定

static 变量既然属于整个类，不属于特定的某一个对象，那么它肯定也无法与某一个对象的 this 指针相互关联。这是它与普通变量的重要区别。

成员变量可以声明为 static，函数当然也可以。

上述例子中的 show_num 函数，目的仅仅是打印 sum 这个 static 变量的值，不与任何集体对象发生作用，所以这个函数也可以声明为 static。

static int show_num();

这里注意：**普通成员函数可以访问类的普通成员和 static 成员，但是 static 函数只能访问 static 数据成员**。

头文件与前向声明

class Acount; 这个叫做类的前向声明。

如果我们在某个类 B 中使用到了 A 的指针或者引用，那么其实不需要包含完整的头文件（因为这里只需获取指针或者引用的大小），只需要在 B 前面前向声明 A 即可。

如果 B 类包含了 A 对象，或者使用 A 指针（引用）调用了 A 的成员函数，那么就需要 include 头文件。

static 的应用实例：Linux Thread 的封装

其他： Linux MutexLock 和 Condition 的封装

friend 友元

前面我们把类的数据成员用 private 修饰，这导致我们无法在类的外部直接访问类的数据成员，这显然是一种安全的做法。但很多时候，

类似于现实中的朋友关系，我们往往需要给某些类或者函数提供一些便利，以便于他们可以方便的访问类的成员。

友元类

友元函数

```
class X {
    friend class Y;
    friend void print(const X &x);
private:
    int x_;
    int y_;
};

class Y {
public:
    void print(const X &x) {
        cout << x.x_ << endl;
        cout << x.y_ << endl;
    }
};

void print(const X &x) {
    cout << x.x_ << endl;
    cout << x.y_ << endl;
}
```

在这个例子中，Y 和函数 print 就是类 X 的友元。他们可以直接访问类 X 的成员。

在实际中也有这样一些场景：我们把某个类的所有成员设为 private，只允许它的 friend 成员访问。

单例模式

有时候我们需要一个类只能生成唯一的一个对象，这就需要用到一种

设计模式-单例模式。

维基百科上对单例模式的解释如下：

“单例模式，也叫单子模式，是一种常用的软件设计模式。在应用这个模式时，单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为。比如在某个服务器程序中，该服务器的配置信息存放在一个文件中，这些配置数据由一个单例对象统一读取，然后服务进程中的其他对象再通过这个单例对象获取这些配置信息。这种方式简化了在复杂环境下的配置管理。”

实现单例模式的步骤如下：

1.把类的构造函数设为私有，这是为了防止在类的外面随意生成对象。（同时我们最好把类的复制和赋值功能禁用掉），代码如下：

```
class Singleton{
public:

private:
    Singleton(){};

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);
};
```

2.外面无法生成对象，于是我们尝试在类的内部生成，这里我们添加一个成员函数叫做 getInstance。

```
class Singleton{
public:
    Singleton *getInstance(){
        Singleton *pInstance = new Singleton;
        return pInstance;
    }
private:
    Singleton(){};

    //forbid copy and assign
```

```
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);
};
```

但是此时，我们遇到了这样的情况：getInstance 这个函数是用来生成对象的，但是 getInstance 本身又是一个成员函数，需要对象来调用，这造成了相互矛盾。

3.解决上面问题的方法就是把该函数设为 static：

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

class Singleton{
public:
    static Singleton *getInstance(){
        Singleton *pInstance = new Singleton;
        return pInstance;
    }
private:
    Singleton(){};

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);
};

int main(int argc, char **argv) {
    Singleton *p = Singleton::getInstance();
    cout << p << endl;
}
```

这样我们成功通过这个 static 方法获取了 Singleton 的对象。

4.如何保证对象的唯一性？ 我们添加一个 static 成员变量，每次

生成对象前去检查它是否为 NULL。

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


class Singleton{
public:
    static Singleton *getInstance(){
        //Singleton *pInstance = new Singleton;
        if(pInstance_ == NULL){
            pInstance_ = new Singleton;
        }
        return pInstance_;
    }

private:
    Singleton(){};

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);


    static Singleton *pInstance_;
};

Singleton *Singleton::pInstance_ = NULL;

int main(int argc, char **argv) {
    Singleton *p1 = Singleton::getInstance();
    Singleton *p2 = Singleton::getInstance();
    cout << p1 << " " << p2 << endl;
}
```

到此为止，我们似乎完成了单例模式的编写，但是上面的程序是否真的可靠？我们编程验证：

测试程序如下：

```
#include <iostream>
#include <vector>
#include <pthread.h>
#include <unistd.h>
using namespace std;

class Singleton {
public:
    static Singleton *getInstance() {
        //Singleton *pInstance = new Singleton;
        if (pInstance_ == NULL) {
            sleep(1);
            pInstance_ = new Singleton;
        }
        return pInstance_;
    }

private:
    Singleton() {
    }
    ;

    //forbid copy and assign
    Singleton(const Singleton &);
    Singleton &operator=(const Singleton &);

    static Singleton *pInstance_;
};

Singleton *Singleton::pInstance_ = NULL;

void *threadFunc(void *arg) {
    Singleton *p = Singleton::getInstance();
    cout << p << endl;
    return NULL;
}

int main(int argc, char **argv) {
    vector<pthread_t> vec(50);
    for (auto &it : vec) {
        pthread_create(&it, NULL, threadFunc, NULL);
    }
    for (auto &it : vec) {
        pthread_join(it, NULL);
```

```
        }
    }
```

我们发现，在多线程的环境中，单例模式编写是失败的，如何解决？ 利用互斥锁 MutexLock。

```
static Singleton *getInstance() {
        //Singleton *pInstance = new Singleton;
        lock_.lock();
        if (pInstance_ == NULL) {
            sleep(1);
            pInstance_ = new Singleton;
        }
        lock_.unlock();
        return pInstance_;
    }
```

上面的程序还有一些不完善的地方，例如每次调用 getInstance 都要上锁，开销太大。我们做一些轻微的改动如下：

```
static Singleton *getInstance() {
    //Singleton *pInstance = new Singleton;
    if (pInstance_ == NULL) {
        lock_.lock();
        if (pInstance_ == NULL) {
            pInstance_ = new Singleton;
        }
        lock_.unlock();
    }
    return pInstance_;
}
```

上面的程序就是经典的“**双重锁**”模式。这种模式在某些情况下也有缺陷，但这超出了我们课程的范畴，在此不做讨论。