类的拷贝控制

作者：heyang5188
原文链接：https://ld246.com/article/1504172291734
来源网站：链滴

类的拷贝控制和拷贝控制函数的说明

当你程序需要拷贝的时候，就会调用类里面的拷贝构造函数

比如以下场合：

vector<int> v2;

vector<int> v3 = v2;

这样是拷贝初始化。

当使用直接初始化的时候，编译器使用普通的参数匹配。当我们使用的是拷贝的初始化，

我们要求编译器将右边的对象拷贝到左边正在创建的对象中去。

拷贝函数何时发生?

```
* asd
class HasPtr {
public:
    HasPtr(const std::string &s = std::string())
    :ps(new std::string(s)),i(0){    }
    HasPtr(const HasPtr &hp)
        :ps(new string(*hp.ps)), i(hp.i) {}
    HasPtr& operator=(const HasPtr &hp)
     {
        auto newps = new std::string(*hp.ps);
        delete(ps);
        ps = newps;
        i = hp.i;
        return *this;
    }
    string& show() { return *ps; }
private:
    std::string *ps;
    int i;
};
int  main()
{
    HasPtr hp("heyang15928");
    HasPtr mp;
    mp = hp;
    cout << mp.show() << endl;
    system("pause");
    return 0;
}
```

对象计数程序告诉我们什么时候对象拷贝

<code>当我们需要自己设置一个析构函数的时候，我们也需要一个拷贝构造函数和一个自定义拷贝算符</code>

因为需要自定义一个析构函数的场合一般是有动态内存的，合成的默认拷贝构造函数不会新创建一块

的动态内存，只会拷贝指针，这样，两个不同的类尽然有相同的内存地址，显然这就是错的 例如上面例子，拷贝构造函数 new了一块新的动态内存，这样两个对象的地址才是分开的。

另外，我们需要知道的是第二个原则

<code>如果类需要一个拷贝狗仔函数，几乎可以肯定，它也需要一个拷贝赋值运算符。</code>

下面的这个程序，就是一个对象的计数程序，当一个对象拷贝，他的识数字增加1；

```
class numbered {
private:
    static int sql;
public:
    numbered() { mysn=sql++; }
    numbered(const numbered &)
    {    mysn = sql++;    }
    int mysn;
};
int numbered::sql = 0;
void f(numbered &num) { cout << num.mysn << endl; }
int  main()
{
    numbered a, b = a, c = b;
    f(a); f(b); f(c);
    system("pause");
    return 0;
}
```

阻止拷贝

```
Nocopy(const Nocopy&)=delete; //阻止拷贝
Nocopy& operator=(const Nocopy&)=delete; //阻止赋值
```

如果一个类有数据成员不能默认构造，拷贝，复制或者销毁，则对应成员函数将被定义为删除的。

行为像值的类

一个行为像值的类，对于类的资源管理就有一定的要求，每个对象都应该有自己的一份拷贝。

为了实现这样的行为，Hasptr就需要以下的三点

- 一个拷贝构造函数，完成string的拷贝，而不是拷贝指针
- 定义一个析构函数来释放string
- 定义一个拷贝赋值运算符来释放当前对象的string，并从右侧运算对象拷贝string

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string())
    :ps(new std::string(s)),i(0){    }
    HasPtr(const HasPtr &hp)
```

```
    :ps(new string(*hp.ps)), i(hp.i) {}  //拷贝构造函数，拷贝的是值而不是指针！！！
  HasPtr& operator=(const HasPtr &hp)
   {
      auto newps = new std::string(*hp.ps); //当拷贝异常的时候，如果先释放资源了就会出错
      delete ps;   //先拷贝了再删除，防止拷贝异常，
       ps = newps;
      i = hp.i;
      return *this;
  }
  ~HasPtr() { delete ps; }  //析构函数来释放资源
  string& show() { return *ps; }
private:
  std::string *ps;
  int i;
};
```

行为像指针的类

行为像指针的类就是类似于智能指针shared_ptr的类，他有一个非常重要的特点就是引用技术

- 引用计数在拷贝的时候加一，在调用析构函数的时候减一

难点，怎么让每一个对象使用同一个引用计数呢?

计数器不能作为HasPtr的成员

一个解决方法是，在构造的时候用一个动态内存来保存这个引用计数，把这个引用计数的指针作为类成员，在拷贝的时候拷贝这个指针就可以了，使用这种方法，副本的对象和原对象都会指向相同的计器。

<code>赋值计算应该先处理自赋值，理由在上面有介绍到</ode>

```
class HasPtr {
public:
  HasPtr(const std::string &s = std::string())
  :ps(new std::string(s)),i(0),use(new std::size_t(1)){   }
  HasPtr(const HasPtr &hp)
      :ps(new string(*hp.ps)), i(hp.i), use(hp.use) {
      ++*use;
  }
  HasPtr& operator=(const HasPtr &hp)
   {
      ++*hp.use;
      if (--*use == 0) {
          delete use;
          delete ps;
      }
      ps = hp.ps;
      i = hp.i;
      use = hp.use;
      return *this;
  }
  ~HasPtr() {
```

```
        if (--*use == 0)
        {
            delete ps;
            delete use;
        }
    }
```

类的交换控制--自定义swap函数

自定义交换函数的好处是可以自定义效率比较高的方式来代替标准库的swap函数

```
class HasPtr {
    friend void swap(HasPtr &lhs, HasPtr &rhs);
public:
    HasPtr(const std::string &s = std::string())
    :ps(new std::string(s)),i(0){    }
    HasPtr(const HasPtr &hp)
        :ps(new string(*hp.ps)), i(hp.i){    }
    HasPtr& operator=(const HasPtr &hp)
     {
        auto *newptr = new string(*hp.ps);
        delete ps;
        ps = newptr;
        i = hp.i;
        return *this;
    }
    ~HasPtr() {    delete ps;}
    string& show() {
        return *ps;
    }
    bool operator<(const HasPtr& hp)const {
        return *ps > *hp.ps;
    }
private:
    std::string *ps;
    int i;

};
inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;

    cout << "exchange " << *lhs.ps << "and " << *rhs.ps << endl;
    swap(lhs.ps, rhs.ps);
    swap(lhs.i, rhs.i);
}
std::allocator<std::string> StrVec::alloc;
int  main()
{

    HasPtr h("ahi mom!");
    HasPtr h2(h);
    HasPtr h3 = h;
```

```
    h2 = string("chi dad!");
    h3 = string("bhi son");
    HasPtr h4("nimabi");

    HasPtr h5("fuck you");
    vector<HasPtr> v3ec{string("f232"),string("qq253") ,string("as2d3") ,string("2f3") ,string("g2
") ,string("2a3") ,string("23g") ,string("2s3") ,string("hh23") ,string("ss23") ,string("2ww3") ,strin
("gg2aa3") };
    vector<HasPtr> vec(v3ec.begin(), v3ec.end());
    vec.insert(vec.begin(), v3ec.begin(), v3ec.end());
    vec.insert(vec.begin(), v3ec.begin(), v3ec.end());
    vec.insert(vec.begin(), v3ec.begin(), v3ec.end());
    vec.insert(vec.begin(), v3ec.begin(), v3ec.end());
    vec.insert(vec.begin(), v3ec.begin(), v3ec.end());
    vec.push_back(h);
    vec.push_back(h2);
    vec.push_back(h3);
    vec.push_back(h4);
    vec.push_back(h5);
    sort(vec.begin(), vec.end());
    for (auto &x : vec)
        cout << x.show() << endl;



    system("pause");
    return 0;
}
```