拷贝构造函数和赋值运算符

C语言面变量的初始化和赋值操作都是在基本数据类型间进行的，所以对这块没有什么太值得说的。但是C++中因为引入了类，我们可以把每个类理解为用户自己定义的新数据类型，类里面可以定义各种数据类型，这就导致类在进行初始化和赋值时需要考虑得更多。

本文将针对拷贝构造函数和赋值运算符进行讨论，也就是在讨论用同一个类的对象初始化另一个对象。接下来先了解下什么是拷贝构造函数以及区分深拷贝和浅拷贝的问题。

通常在创建一个类的对象时，经常用一些常数对类中的变量进行赋值，这就需要调用类中定义的构造函数，当然，即使不写构造函数，系统也会生成相应的默认构造函数。但如果我们在定义类的同时是通过另一个同类的对象去初始化对象的话，在写构造函数时就有些区别了，完成这一工作的函数我们称为拷贝构造函数。

拷贝构造函数有两种，一种是默认拷贝构造函数，一种是程序员自己定义的拷贝构造函数，简单来看就是一个是系统自带的一个是进行个性化定制的。之所以系统会自带默认拷贝构造函数是为了应对最常规的用对象初始化另一个对象的情况，对应的是之前提到的浅拷贝，为了能够写出适应性强的代码，还需要我们自己重写。

我们可以通过下面一个简单例子来比较浅拷贝和深拷贝。

|  |
| --- |
| #include <iostream>  using namespace std;  class A  {  public:  A(int i) { num = i; }  A(const A &temp) { num = temp.num; }  ~A(){}  int get() { return num; }  private:  int num;  };  int main()  {  A a(10);  A b(a);  cout << "a.num = " << a.get() << endl;  cour << "b.num = " << b.get() << endl;  return 0;  } |

这里A类对象a先通过带参数的构造函数进行初始化，之后我们用已经生成的对象a来初始化同样是A类的对象b，最终运行结果是对象b中的num变量被正确赋值为10。上面这个例子只是最简单的通过对象初始化对象的场景，即使我们把自己写的那部分拷贝构造函数删除掉依旧能够准确运行，这是因为系统在我们没有自己定义拷贝构造函数时会自动为我们加上默认拷贝构造函数。

下面我们考虑一种比较困难的情况：类中含有指针变量。在下一个例子中我会把浅拷贝（默认拷贝构造函数，不过我会自己显式实现）和深拷贝两种方法都写出来方便比较。

|  |
| --- |
| #include <iostream>  #include <cstring>  using namespace std;  class A  {  public:  // 带参构造函数  A(const char \*s):str(s){}  // 浅拷贝  A(const A &temp):str(temp.str){}  // 深拷贝  A(const A &temp)  {  int len = strlen(temp.str);  str = new char[len + 1];  if(str != NULL)  strcpy(str, temp.str);  }  ~A(){}  char\* get() { return str; }  private:  char \*str;  };  int main()  {  A a("Hello World");  A b(a);  A c(a);  cout << "a.str = " << a.get() << endl;  cout << "b.str = " << b.get() << endl;  cout << "c.str = " << c.get() << endl;  return 0;  } |

在定义类A的对象b和c时分别使用了深拷贝和浅拷贝，很明显对象b在进行深拷贝时先分配了能够存储字符的空间，之后通过strcpy函数将对象a中字符串逐字符地拷贝过来。但是对象c进行浅拷贝时，仅仅将字符指针设置成a中字符指针的字面值，这样就相当于在内存中存储了一个字符串但是有两个字符指针指向它，这样当其中一个对象生命期结束调用析构函数时，就会把指向的字符串释放掉，此时另一个对象中指向此字符串的指针相当于失效了。

所以在这个问题上我们得到的教训是，为了保证类的安全性，当类的变量中含有指针变量时一定需要自己写拷贝构造函数，且是通过深拷贝的方式。简单的默认拷贝构造函数只能复制其他变量的字面值，如果含有指针变量，会造成严重的内存问题。

在这个问题上，可能会有疑惑，哪些场景会出现通过对象初始化对象呢？除了上面那种情况，其实最常见的是在函数调用时通过值传递一个对象或者返回值是对象时，在调用函数中都会先生成一个临时的对象并调用拷贝构造函数，在函数调用结束后自动销毁调用析构函数。试想如果调用函数或者返回时用某个对象初始化了临时对象，之后临时对象生存期到了之后调用析构函数，若临时对象中有指针变量，则会销毁这个指针指向的数据，当回到主函数后再使用这块数据时就会发生错误。

接下来说一下赋值运算和拷贝构造函数的区别。其实简而言之就是拷贝构造函数调用时先产生同类型的新的对象，再对这个对象初始化，在初始化对向前不会检查源对象和新建对象是否相同。但是赋值运算是在已经存在的对象上使用赋值操作，要先检查源对象和赋值对象是否是同一个，并且如果之前已经有内存分配，要先将那块内存释放掉。同样的，如果类中有指针变量时，一定要重写拷贝构造函数和赋值运算符，而不是使用默认的。下面给个小例子来对比深拷贝和赋值运算的区别。

|  |
| --- |
| #include <iostream>  #include <cstring>  using namespace std;  class A  {  public:  // 带参构造函数  A(const char \*s):str(s){}  // 深拷贝  A(const A &temp)  {  int len = strlen(temp.str);  str = new char[len + 1];  if(str != NULL)  strcpy(str, temp.str);  }  // 赋值运算符  A & operator = (const A &temp)  {  if(this == &temp)  return \*this;  delete []str;  int len = strlen(temp.str);  str = new char[len + 1];  if(str != NULL)  strcpy(str, temp.str);  return \*this;  }  ~A(){}  char\* get() { return str; }  private:  char \*str;  };  int main()  {  A a("Hello World");  // A b(a);等价于A b = a;  A b(a);  // 赋值运算,务必把定义和赋值分开写  A c;  c = a;  cout << "a.str = " << a.get() << endl;  cout << "b.str = " << b.get() << endl;  cout << "c.str = " << c.get() << endl;  return 0;  } |

在上例中，拷贝构造函数的内容之前已经说过了，赋值运算确实相比简单的拷贝要多出1.判断源对象和目的对象是否是同一个对象，2.释放原有分配内存。另外需要说的是，赋值运算是在对象已经生成之后再去赋值，所以必须把定义和赋值两个操作分开，如果写在一行就是创建新对象和初始化再同一步，这就不再是赋值运算而是直接调用拷贝构造函数了。