**写在前面：**

      关于C++的赋值运算符重载函数(operator=)，网络以及各种教材上都有很多介绍，但可惜的是，内容大多雷同且不全面。面对这一局面，在下在整合各种资源及融入个人理解的基础上，整理出一篇较为全面/详尽的文章，以飨读者。

**正文：**

**Ⅰ.举例**

例1

[复制代码](javascript:void(0);)

#include<iostream>

#include<string>

using namespace std;

class MyStr

{

private:

char \*name;

int id;

public:

MyStr():id(0),name(NULL) {}

MyStr(int \_id, char \*\_name) //constructor

{

cout << "constructor" << endl;

id = \_id;

name = new char[strlen(\_name) + 1];

strcpy\_s(name, strlen(\_name) + 1, \_name);

}

MyStr(const MyStr& str)

{

cout << "copy constructor" << endl;

id = str.id;

name = new char[strlen(str.name) + 1];

strcpy\_s(name, strlen(str.name) + 1, str.name);

}

MyStr& operator =(const MyStr& str)//赋值运算符

{

cout << "operator =" << endl;

if (this != &str)

{

if (name != NULL)

delete[] name;

this->id = str.id;

int len = strlen(str.name);

name = new char[len + 1];

strcpy\_s(name, strlen(str.name) + 1, str.name);

}

return \*this;

}

~MyStr()

{

delete[] name;

}

};

int main()

{

MyStr str1(1, "hhxx");

cout << "====================" << endl;

MyStr str2;

str2 = str1;

cout << "====================" << endl;

MyStr str3 = str2;

return 0;

}

[复制代码](javascript:void(0);)

**结果：**

**Ⅱ.参数**

一般地，赋值运算符重载函数的参数是函数所在类的const类型的引用（如上面例1），加const是因为：

①我们不希望在这个函数中对用来进行赋值的“原版”做任何修改。

②加上const，对于const的和非const的实参，函数就能接受；如果不加，就只能接受非const的实参。

用引用是因为：

这样可以避免在函数调用时对实参的一次拷贝，提高了效率。

**注意**：

上面的规定都不是强制的，可以不加const，也可以没有引用，甚至参数可以不是函数所在的对象，正如后面例2中的那样。

**Ⅲ.返回值**

一般地，返回值是被赋值者的引用，即\*this（如上面例1），原因是

①这样在函数返回时避免一次拷贝，提高了效率。

②更重要的，这样可以实现连续赋值，即类似a=b=c这样。如果不是返回引用而是返回值类型，那么，执行a=b时，调用赋值运算符重载函数，在函数返回时，由于返回的是值类型，所以要对return后边的“东西”进行一次拷贝，得到一个未命名的副本（有些资料上称之为“匿名对象”），然后将这个副本返回，而这个副本是右值，所以，执行a=b后，得到的是一个右值，再执行=c就会出错。

**注意**：

这也不是强制的，我们可以将函数返回值声明为void，然后什么也不返回，只不过这样就不能够连续赋值了。

**Ⅳ.调用时机**

      当为一个类对象赋值（注意：可以用本类对象为其赋值（如上面例1），也可以用其它类型（如内置类型）的值为其赋值，关于这一点，见后面的例2）时，会由该对象调用该类的赋值运算符重载函数。

如上边代码中

str2 = str1;

一句，用str1为str2赋值，会由str2调用MyStr类的赋值运算符重载函数。

需要注意的是，

MyStr str2;

str2 = str1;

和

MyStr str3 = str2;

在调用函数上是有区别的。正如我们在上面结果中看到的那样。

      前者MyStr str2;一句是str2的声明加定义，调用无参构造函数，所以str2 = str1;一句是在str2已经存在的情况下，用str1来为str2赋值，调用的是拷贝赋值运算符重载函数；而后者，是用str2来初始化str3，调用的是拷贝构造函数。

**Ⅴ.提供默认赋值运算符重载函数的时机**

      当程序没有显式地提供一个以本类或本类的引用为参数的赋值运算符重载函数时，编译器会自动生成这样一个赋值运算符重载函数。注意我们的限定条件，不是说只要程序中有了显式的赋值运算符重载函数，编译器就一定不再提供默认的版本，而是说只有程序显式提供了以本类或本类的引用为参数的赋值运算符重载函数时，编译器才不会提供默认的版本。可见，所谓默认，就是“以本类或本类的引用为参数”的意思。

见下面的例2

[复制代码](javascript:void(0);)

#include<iostream>

#include<string>

using namespace std;

class Data

{

private:

int data;

public:

Data() {};

Data(int \_data)

:data(\_data)

{

cout << "constructor" << endl;

}

Data& operator=(const int \_data)

{

cout << "operator=(int \_data)" << endl;

data = \_data;

return \*this;

}

};

int main()

{

Data data1(1);

Data data2,data3;

cout << "=====================" << endl;

data2 = 1;

cout << "=====================" << endl;

data3 = data2;

return 0;

}

[复制代码](javascript:void(0);)

**结果：**

     上面的例子中，我们提供了一个带int型参数的赋值运算符重载函数，data2 = 1;一句调用了该函数，如果编译器不再提供默认的赋值运算符重载函数，那么，data3 = data2;一句将不会编译通过，但我们看到事实并非如此。所以，这个例子有力地证明了我们的结论。

**Ⅵ.构造函数还是赋值运算符重载函数**

     如果我们将上面例子中的赋值运算符重载函数注释掉，main函数中的代码依然可以编译通过。只不过结论变成了

可见，当用一个非类A的值（如上面的int型值）为类A的对象赋值时

①如果匹配的构造函数和赋值运算符重载函数同时存在（如例2），会调用赋值运算符重载函数。

②如果只有匹配的构造函数存在，就会调用这个构造函数。

**Ⅶ.显式提供赋值运算符重载函数的时机**

①用非类A类型的值为类A的对象赋值时（当然，从Ⅵ中可以看出，这种情况下我们可以不提供相应的赋值运算符重载函数而只提供相应的构造函数来完成任务）。

②当用类A类型的值为类A的对象赋值且类A的成员变量中含有指针时，为避免浅拷贝（关于浅拷贝和深拷贝，下面会讲到），必须显式提供赋值运算符重载函数（如例1）。

**Ⅷ.浅拷贝和深拷贝**

      拷贝构造函数和赋值运算符重载函数都会涉及到这个问题。

      所谓浅拷贝，就是说编译器提供的默认的拷贝构造函数和赋值运算符重载函数，仅仅是将对象a中各个数据成员的值拷贝给对象b中对应的数据成员（这里假设a、b为同一个类的两个对象，且用a拷贝出b或用a来给b赋值），而不做其它任何事。

      假设我们将例1中显式提供的拷贝构造函数注释掉，然后同样执行MyStr str3 = str2;语句，此时调用默认的拷贝构造函数，它只是将str2的id值和nane值拷贝到str3，这样，str2和str3中的name值是相同的，即它们指向内存中的同一区域（在例1中，是字符串”hhxx”）。如下图

     这样，会有两个致命的错误

①当我们通过str2修改它的name时，str3的name也会被修改！

②当执行str2和str3的析构函数时，会导致同一内存区域释放两次，程序崩溃！

      这是万万不可行的，所以我们必须通过显式提供拷贝构造函数以避免这样的问题。就像我们在例1中做的那样，先判断被拷贝者的name是否为空，若否，delete[] name（后面会解释为什么要这么做），然后，为name重新申请空间，再将拷贝者name中的数据拷贝到被拷贝者的name中。执行后，如图

      这样，str2.name和str3.name各自独立，避免了上面两个致命错误。

      我们是以拷贝构造函数为例说明的，赋值运算符重载函数也是同样的道理。

**Ⅸ.赋值运算符重载函数只能是类的非静态的成员函数**

       C++规定，赋值运算符重载函数只能是类的非静态的成员函数，不能是静态成员函数，也不能是友元函数。关于原因，有人说，赋值运算符重载函数往往要返回\*this，而无论是静态成员函数还是友元函数都没有this指针。这乍看起来很有道理，但仔细一想，我们完全可以写出这样的代码

static friend MyStr& operator=(const MyStr str1,const MyStr str2)

{

……

return str1;

}

      可见，这种说法并不能揭露C++这么规定的原因。

      其实，之所以不是静态成员函数，是因为静态成员函数只能操作类的静态成员，不能操作非静态成员。如果我们将赋值运算符重载函数定义为静态成员函数，那么，该函数将无法操作类的非静态成员，这显然是不可行的。

      在前面的讲述中我们说过，当程序没有显式地提供一个以本类或本类的引用为参数的赋值运算符重载函数时，编译器会自动提供一个。现在，假设C++允许将赋值运算符重载函数定义为友元函数并且我们也确实这么做了，而且以类的引用为参数。与此同时，我们在类内却没有显式提供一个以本类或本类的引用为参数的赋值运算符重载函数。由于友元函数并不属于这个类，所以，此时编译器一看，类内并没有一个以本类或本类的引用为参数的赋值运算符重载函数，所以会自动提供一个。此时，我们再执行类似于str2=str1这样的代码，那么，编译器是该执行它提供的默认版本呢，还是执行我们定义的友元函数版本呢？

       为了避免这样的二义性，C++强制规定，赋值运算符重载函数只能定义为类的成员函数，这样，编译器就能够判定是否要提供默认版本了，也不会再出现二义性。

**Ⅹ. 赋值运算符重载函数不能被继承**

见下面的例3

[复制代码](javascript:void(0);)

#include<iostream>

#include<string>

using namespace std;

class A

{

public:

int X;

A() {}

A& operator =(const int x)

{

X = x;

return \*this;

}

};

class B :public A

{

public:

B(void) :A() {}

};

int main()

{

A a;

B b;

a = 45;

//b = 67;

(A)b = 67;

return 0;

}

[复制代码](javascript:void(0);)

      注释掉的一句无法编译通过。报错提示：没有与这些操作数匹配的”=”运算符。对于b = 67;一句，首先，没有可供调用的构造函数（前面说过，在没有匹配的赋值运算符重载函数时，类似于该句的代码可以调用匹配的构造函数），此时，代码不能编译通过，说明父类的operator =函数并没有被子类继承。

     为什么赋值运算符重载函数不能被继承呢？

     因为相较于基类，派生类往往要添加一些自己的数据成员和成员函数，如果允许派生类继承基类的赋值运算符重载函数，那么，在派生类不提供自己的赋值运算符重载函数时，就只能调用基类的，但基类版本只能处理基类的数据成员，在这种情况下，派生类自己的数据成员怎么办？

     所以，C++规定，赋值运算符重载函数不能被继承。

    上面代码中， (A)b = 67; 一句可以编译通过，原因是我们将B类对象b强制转换成了A类对象。

**Ⅺ.赋值运算符重载函数要避免自赋值**

      对于赋值运算符重载函数，我们要避免自赋值情况（即自己给自己赋值）的发生，一般地，我们通过比较赋值者与被赋值者的地址是否相同来判断两者是否是同一对象（正如例1中的if (this != &str)一句）。

     为什么要避免自赋值呢？

 ①为了效率。显然，自己给自己赋值完全是毫无意义的无用功，特别地，对于基类数据成员间的赋值，还会调用基类的赋值运算符重载函数，开销是很大的。如果我们一旦判定是自赋值，就立即return \*this，会避免对其它函数的调用。

②如果类的数据成员中含有指针，自赋值有时会导致灾难性的后果。对于指针间的赋值（注意这里指的是指针所指内容间的赋值，这里假设用\_p给p赋值），先要将p所指向的空间delete掉（为什么要这么做呢？因为指针p所指的空间通常是new来的，如果在为p重新分配空间前没有将p原来的空间delete掉，会造成内存泄露），然后再为p重新分配空间，将\_p所指的内容拷贝到p所指的空间。如果是自赋值，那么p和\_p是同一指针，在赋值操作前对p的delete操作，将导致p所指的数据同时被销毁。那么重新赋值时，拿什么来赋？

      所以，对于赋值运算符重载函数，一定要先检查是否是自赋值，如果是，直接return \*this。

**结束语：**

      至此，本文的所有内容都介绍完了。由于在下才疏学浅，错误纰漏之处在所难免，如果您在阅读的过程中发现了在下的错误和不足，请您务必指出。您的批评指正就是在下前进的不竭动力！

标签: [C++](https://www.cnblogs.com/zpcdbky/tag/C%2B%2B/), [赋值运算符重载函数](https://www.cnblogs.com/zpcdbky/tag/%E8%B5%8B%E5%80%BC%E8%BF%90%E7%AE%97%E7%AC%A6%E9%87%8D%E8%BD%BD%E5%87%BD%E6%95%B0/), [友元函数](https://www.cnblogs.com/zpcdbky/tag/%E5%8F%8B%E5%85%83%E5%87%BD%E6%95%B0/), [浅拷贝](https://www.cnblogs.com/zpcdbky/tag/%E6%B5%85%E6%8B%B7%E8%B4%9D/), [深拷贝](https://www.cnblogs.com/zpcdbky/tag/%E6%B7%B1%E6%8B%B7%E8%B4%9D/), [继承](https://www.cnblogs.com/zpcdbky/tag/%E7%BB%A7%E6%89%BF/), [自赋值](https://www.cnblogs.com/zpcdbky/tag/%E8%87%AA%E8%B5%8B%E5%80%BC/)

[**好文要顶**](javascript:void(0);) [**关注我**](javascript:void(0);) [**收藏该文**](javascript:void(0);) **[https://common.cnblogs.com/images/icon_weibo_24.png](javascript:void(0);)** **[https://common.cnblogs.com/images/wechat.png](javascript:void(0);)**

[https://pic.cnblogs.com/face/642223/20141117233155.png](https://home.cnblogs.com/u/zpcdbky/)

[同勉共进](https://home.cnblogs.com/u/zpcdbky/)  
[关注 - 0](https://home.cnblogs.com/u/zpcdbky/followees/)  
[粉丝 - 87](https://home.cnblogs.com/u/zpcdbky/followers/)

[+加关注](javascript:void(0);)

24

0

[«](https://www.cnblogs.com/zpcdbky/p/4902688.html)上一篇： [理清C++常量指针和指针常量这团乱麻](https://www.cnblogs.com/zpcdbky/p/4902688.html)   
[»](https://www.cnblogs.com/zpcdbky/p/5275959.html)下一篇： [话说C++中的左值、纯右值、将亡值](https://www.cnblogs.com/zpcdbky/p/5275959.html)

posted @ 2015-12-07 21:23  [同勉共进](https://www.cnblogs.com/zpcdbky/)  阅读(106945)  评论(22)  [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=5027481)  [收藏](javascript:void(0))