概念题

一. C++ 中为什么需要对操作符进行重载？除了常用的操作符外，还可以对哪些特殊的操作符进行重载？

- **操作符重载原因**：
  - 使得自定义类型的运算也能如同基本类型的运算一样，在**符合人类习惯**的基础上完成相应的运算功能。
  - 操作符重载是C++**多态**的一种体现。
- **特殊操作符**：
  - 赋值操作符"**=**"
  - 访问数组元素操作符"**[]**"
  - 动态对象创建与撤销操作符**new**与**delete**
  - 函数调用操作符"**()**"
  - 类成员访问操作符"**->**"
  - **类型转换操作符**

二. C++中系统提供的隐式赋值操作存在什么问题？如何解决？

- **问题**：
  - 当类的对象会额外申请一些资源时，隐式赋值操作符重载函数不会重新申请这些资源，这会导致新对象与原对象共享资源，在资源使用和释放上会出现问题。若不是有意为之，则需要自定义赋值操作符重载函数。
- **解决方式**：
  - 自定义赋值操作符重载函数，在其中重新申请资源。

编程题

一.

```
//string_operator.h
#include <iostream>
#include <string.h>
using namespace std;

class CustomString{
public:
    //构造函数
    CustomString();
    CustomString(const char* str);
    CustomString(CustomString& c);                                  //拷贝构造
    //析构函数
    ~CustomString();
    //成员函数重载
    char& operator [] (int i);
    CustomString& operator = (const CustomString& c);               //赋值构造
    CustomString& operator += (CustomString& c);
    //友元重载
    friend ostream& operator << (ostream& out, CustomString& c);
    friend istream& operator >> (istream& in, CustomString& c);
    friend CustomString& operator + (CustomString& c1, CustomString& c2);
    friend bool operator == (CustomString& c1, CustomString& c2);
    friend bool operator != (CustomString& c1, CustomString& c2);
private:
    char* p; // 字符串的起始地址
    int len; // 字符串的长度
};
```

```
//string_operator.cpp
#include"string_operator.h"

//析构函数
CustomString::~CustomString(){          //释放空间
    delete[] p;
    p = NULL;
}

CustomString::CustomString(){           //初始为空，长度为1
    p = new char[1];
    len = 1;
    p[0] = '\0';
}

CustomString::CustomString(const char* str){
    len = strlen(str);
    p = new char[len + 1];
    strcpy(p, str);
}

CustomString::CustomString(CustomString& c){
    len = c.len;
    p = new char[len + 1];
    strcpy(p, c.p);
}

//成员函数重载
char& CustomString::operator [] (int i){
    return p[i];
}

CustomString& CustomString::operator = (const CustomString& c){         //const
    delete[] p;
    len = c.len;
    p = new char[len + 1];
    strcpy(p, c.p);
    return *this;
}

CustomString& CustomString::operator += (CustomString& c){          //返回自身，
可迭代执行
    string str1(p);
    string str2(c.p);
    str1 = str1 + str2;
    len = str1.length();
    delete[] p;
    p = new char[len + 1];
    strcpy(p, str1.c_str());
    return *this;
}


//友元重载
ostream& operator << (ostream& out, CustomString& c){
    out << c.p;
    return out;
}

istream& operator >> (istream& in, CustomString& c){
    in >> c.p;
    c.len = strlen(c.p);
    return in;
}

CustomString& operator + (CustomString& c1, CustomString& c2){              //返
回引用！否则a+b临时变量无法被引用
    string str1(c1.p);
    string str2(c2.p);
    string str3 = str1 + str2;
    CustomString* c = new CustomString(str3.c_str());
    return *c;
}

bool operator == (CustomString& c1, CustomString& c2){
    return (c1.len == c2.len && !strcmp(c1.p, c2.p));
}

bool operator != (CustomString& c1, CustomString& c2){
    return (c1.len != c2.len || strcmp(c1.p, c2.p));
}

int main(){
    CustomString mystr("this is e CustomString class for testing!");
    cout << mystr[8] << endl;
    mystr[8] = 'a';
    cout << mystr <<endl;
    CustomString mystr2 = mystr;
    cout << mystr2 << endl;
    CustomString mystr3;
    mystr3 = mystr + mystr2;
    cout << mystr + mystr2 << endl;
    mystr3 += mystr;
    cout << mystr3 << endl;
    cout << (mystr != mystr2) << endl;
    cout << (mystr == mystr3) << endl;
    CustomString mystr4;
    cout << "Input any string to test the overloaded input operator >>: " <<
endl;
    cin >> mystr4;
    cout << mystr4 << endl;
    cout << "Congratulations! testing passed!" << endl;
    return 0;
}
```