45 Copying and Copy Constructors in C++

拷贝以及C++拷贝构造函数

1. 拷贝

copy（拷贝）指的是要求复制数据，复制内存。
当我们想把一个对象或者*primitive*（原语）或者一段数据从一个地方复制到另一个地方时，我们实际上有两个副本。我们要避免不必要的复制，因为这样会浪费性能。

理解复制是如何在C++中工作的，如何避免让它工作，或者在你不想复制的时候避免复制，对于理解语言以及能高效正确地编写C++代码非常重要。

```C++
int a = 2;
int b = a;
```

这里实际上是创建了一个a的副本，a和b是两个独立的变量，它们有不同的内存地址。所以如果我们把b改为3，a仍然是2。

在类中，也是同样的道理。

复制指针就不一样了，指向的内容是同一个，修改b也会影响a。

```C++
struct Vector2
{
    float x, y;
};

int main()
{
    Vector2* a = new Vector2();
    Vector2* b = a;
    b->x = 5;
    std::cout << "x:" << a->x << "y:" << a->y << std::endl;  // x:5,y:0
}
```

所以除了引用外，每当你编写一个变量被赋值给另一个变量的代码时，你总是在复制。

```C++
#include <iostream>
#include <string>

class String
{
private:
    char* m_Buffer;  //指向字符缓冲区
    unsigned int m_Size;  //保存string的大小
public:
    String(const char* string)
    {
        m_Size = strlen(string);
        m_Buffer = new char[m_Size];  //需要给空终止符留个位置，这里故意少了
        /*for (int i = 0; i < m_Size; i++)
            m_Buffer[i] = string[i];*/
        //可以很简单地用for循环一个个填入，但这里用更简洁的方法
        memcpy(m_Buffer, string, m_Size);
    }
    friend std::ostream& operator<<(std::ostream& stream, const String& string);//友元，
让外部函数可以访问私有成员
};


std::ostream& operator<<(std::ostream& stream,const String& string )
{
    stream << string.m_Buffer;
    return stream;
}//重载输出运算符,将 `String` 对象输出到流 `std::ostream` 中。

int main()
{
    String string = "Cherno";
    std::cout << string << std::endl;

    std::cin.get();
}
```

可以看到输出结果有很多随机字符，正是因为没有空终止字符导致的。


```C++
m_Buffer = new char[m_Size + 1 ]; //增加空终止符的位置，也可以用以后会学到的strcpy
m_Buffer[m_Size] = 0; //添加空终止符
```

现在可以正常输出“Cherno”了。
再回去看下String类部分的代码，有什么尚存的问题吗？

答案是有一个内存泄漏。

```C++
m_Buffer = new char[m_Size + 1 ];
```

我们分配这个new char时，并没有配套的`delete`。当然如果你用智能指针或者vector是不需要delete的。
于是我们在String类的析构函数中修改：

```C++
~String()
{
    delete[] m_Buffer;
}
```

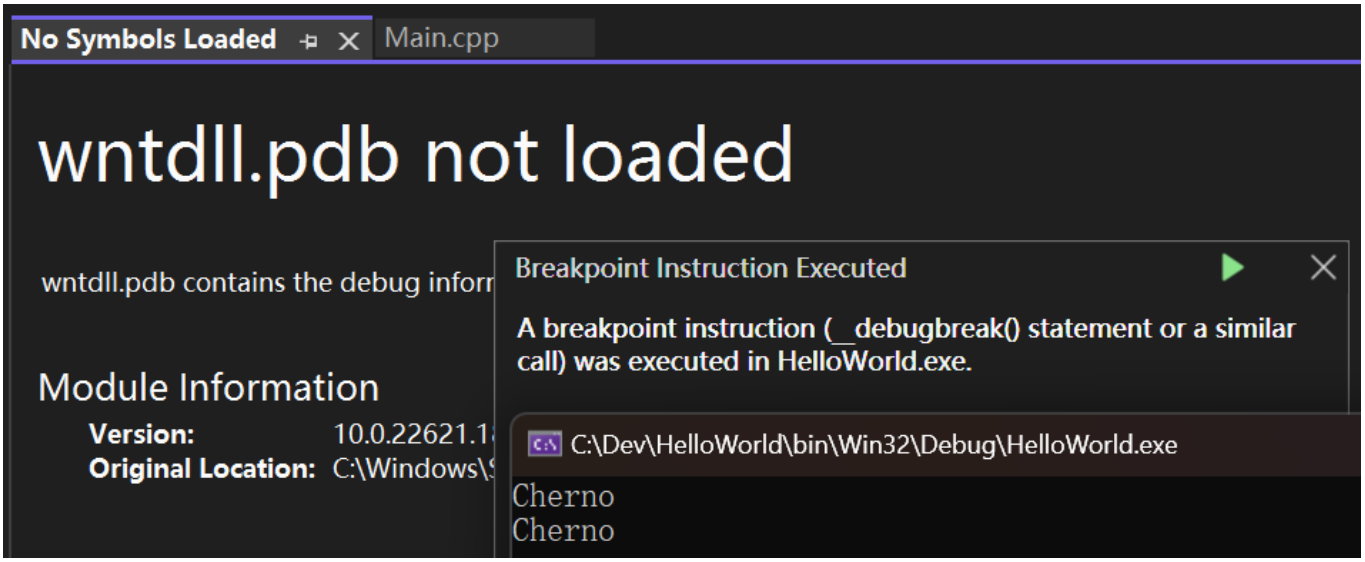
接下来我们复制这个string并且尝试打印：

```C++
int main()
{
    String string = "Cherno";
    String second = string;

    std::cout << string << std::endl;
    std::cout << second << std::endl;

    std::cin.get();
}
```

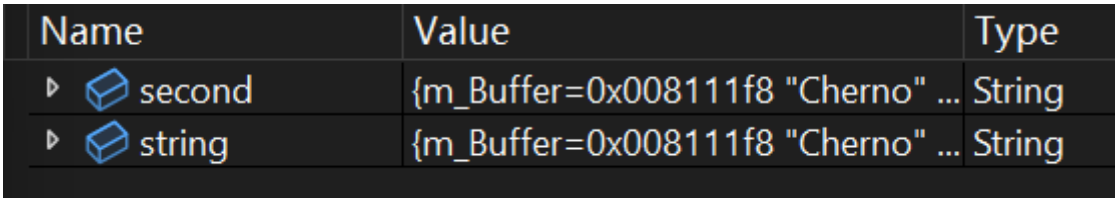
Opps，输出字符串执行到cin.get()按回车后程序崩溃了。调出callstack也看不懂，那到底是什么让我们的程序崩溃了？



我们复制这个String时，C++自动为我们做的是它将所有类成员变量，而这些成员变量组成了类（实例的内存空间），它是由一个char*和一个unsigned int组成的，它将这些值复制到了一个新的内存地址里面，这个新的内存地址包含了这个second字符串。

2. 浅拷贝

现在问题来了，内存中有两个String，因为它们直接进行了复制，这种复制被称为*shallow copy*（浅拷贝）。它所做的是复制这个char*，内存中的两个String对象有相同的char*的值，换句话说就是有相同的内存地址。这个m_Buffer的内存地址，对于这两个String对象来说是相同的，所以程序会崩溃的原因是当我们到达作用域的尽头时，这两个String都被销毁了，析构函数会被调用，然后执行`delete[] m_Buffer`两次，程序试图两次释放同一个内存块。这就是为什么程序会崩溃——因为内存已经释放了，不再是我们的了，我们无法再次释放它。

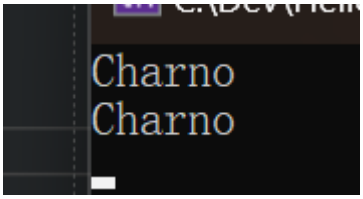


那假设我们想要修改second字符串，不完全修改，只是将字母e改成字母a，变成“Charno”

```C++
second[2] = 'a'; //还需要操作符重载

class String
{
.......
char& operator[](unsigned int index)
{
    return m_Buffer[index];
}
......
}
```

为啥会是俩Charno呢?



我们真正需要做的是，分配一个新的char数组，来存储复制的字符串。而我们现在做的是复制指针，两个字符串对象指向完全相同的内存缓冲区，它同时改变了它们，因为它们指向同一个内存块。或者当我们删除的时候它会把它们两个都删除，因为它们指向同一个内存块。

3. 深拷贝（用拷贝构造函数）

我们希望第二个字符串拥有自己的指针，以拥有自己唯一的内存块。
我们能做到这一点的方式是执行一种叫*deep copy*（深拷贝）的东西，也就是说我们实际复制了整个对象，不是我们上面提到的那种浅拷贝：这个对象是由什么构成的。浅拷贝不会去到指针的内容或者指针所指向的地方，也不会去复制它。

而深拷贝是根据定义复制整个对象。我们这里用*copy constructor*拷贝构造函数的方式。

拷贝构造函数是一个构造函数，当你复制第二个字符串时，它会被调用。当你把一个字符串赋值给一个也是字符串的对象时，（当你试图创建一个新的变量并给它分配另一个变量，而这个变量和你正在创建的变量类型相同时），你复制这个变量，也就是所谓的拷贝构造函数。

C++会自动为你提供一个拷贝构造函数：

```C++
class String
{
......
String(const String& other);
......
}
```

它所做的就是内存复制，将other对象的内存浅拷贝进这些成员变量。
所以如果我们自己写的话，它可能看起来像这样：

```C++
String(const String& other)
    : m_Buffer((other.m_Buffer)), m_Size(other.m_Size){};

// 更刺激的方式
String(const String& other)
{
    memcpy(this, &other, sizeof(String));
}
```

这就是C++默认提供的拷贝构造函数

如果我们决定不需要拷贝构造函数，不允许复制，我们可以将这个拷贝函数声明为`delete`：

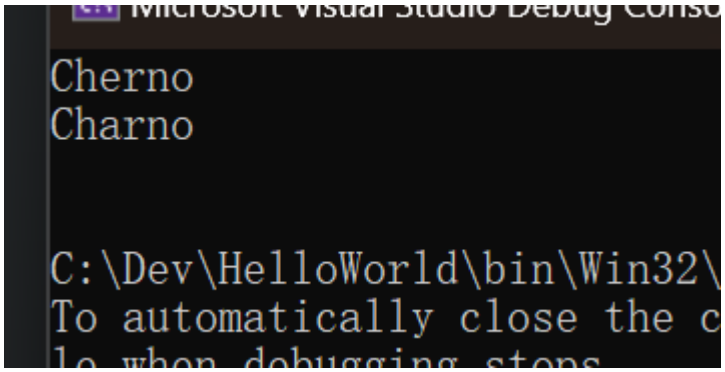
```C++
String(const String& other) = delete;
```

这就是unique_ptr所做的。

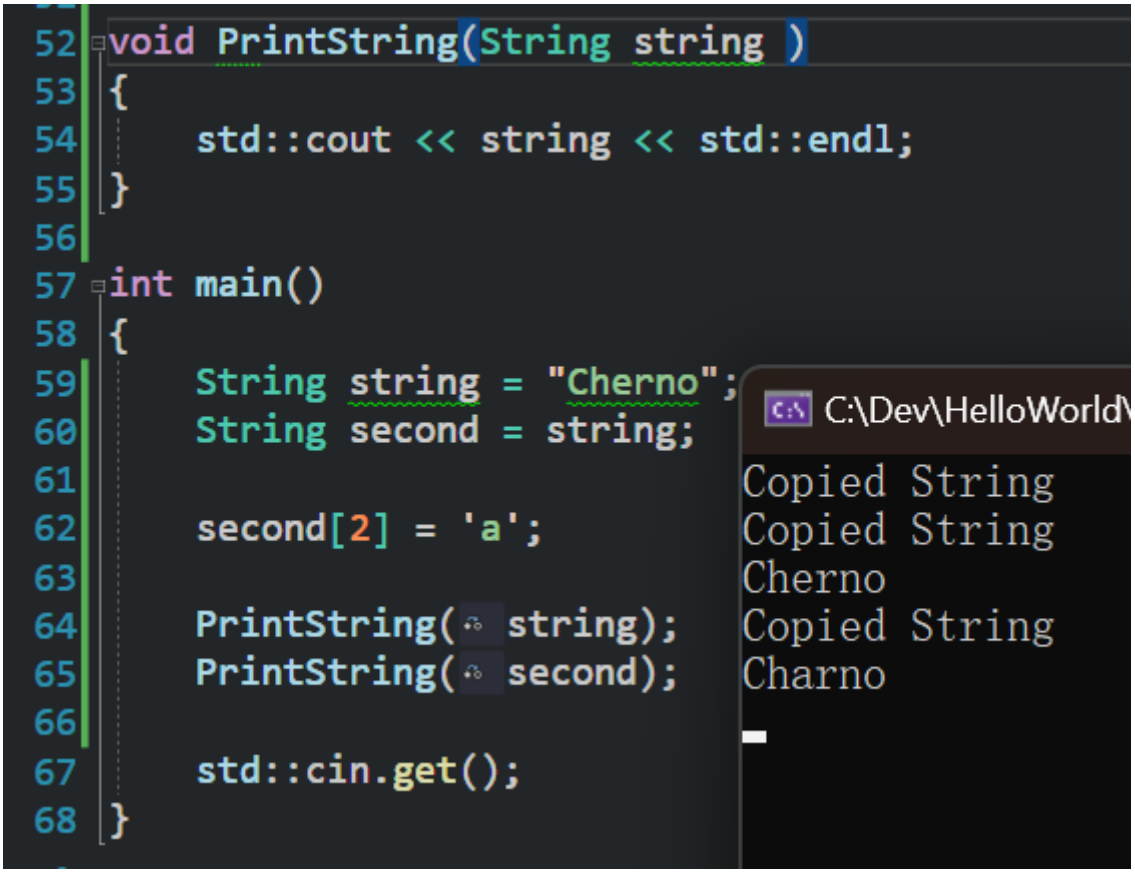
但我们这里想找到我们自己的拷贝构造函数：

```C++
String(const String& other)
    : m_Size((other.m_Size))
{
    m_Buffer = new char[m_Size+1];
    memcpy(m_Buffer, other.m_Buffer, m_Size + 1);
}
```

得到了我们想要的结果，程序也能成功终止。

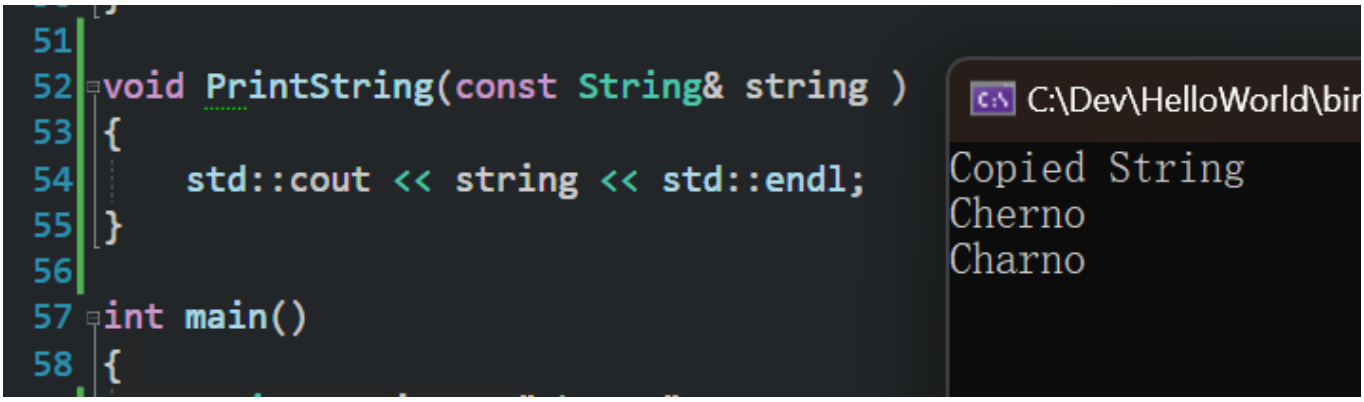


增加一个用来打印的函数，结果正常，但是实际上我们不必要进行这个复制。
在拷贝构造函数中加入一条指示（输出某字符串）：



我们看上去只做了一次复制，但是当我们把它传递给这个函数时，我们实际上做了2次额外的复制操作。每次调用析构函数删除内存，实际上是完全没有必要的。
我们真正想做的是将现有的字符串直接进入这个PrintString函数，我们知道自己不需要复制得到另一个副本，我们可以直接引用现有的string。

我们可以通过*referece*（引用）来实现。



只有在我们给第二个对象赋值为string时有一次复制。

如果我们突然决定在PrintString函数中还是要复制，只需要：

```C++
void PrintString(const String& string )
{
    String copy = string;
    std::cout << string << std::endl;
}
```

想告诉你的是，应该总是要通过const引用去传递对象，以后会深入地探讨它的优化（因为在某些情况下复制可能会更快）。但无论如何，在基础使用中用const引用更好。
总是要用const引用传递对象，因为你可以在你写的的函数的内部决定是否要复制，但是你没有理由到处复制，会拖慢你的程序。
重要的事情说三遍，当你传递字符串的时候，不管这个字符串是你自己的String类还是标准库里的String（std::string），总是要通过const引用来传递。除了减少拷贝外，这样还有其它好处，后续再谈。