

89 Move Semantics in C++

85 左、右值后隔了几节，终于到移动语义了。

我们已经知道了右值是什么，以及右值引用是什么，现在可以看看它们最大的一个用处——移动语义了。

1. 移动语义

移动语义本质上允许我们移动对象，这在C++11之前是不可能的，因为C++11才引入了右值引用，这是移动语义所必需的。

基本思想是，当我们在写C++代码时有很多情况下，我们不需要或者不想把一个对象从一个地方复制到另一个地方，但又不得不复制，因为这是唯一达到目的的方式。例如要把一个对象传递给一个函数，那么它要获得那个对象的所有权，只能选择拷贝；想从函数中返回一个对象时也是一样的，仍然需要在函数中创建那个对象然后返回它（复制了数据）。

不过现在有“返回值优化”可以对返回值这部分做优化处理，但第一个例子中仍然需要在当前堆栈帧中构造一个一次性对象，然后复制到我们正在调用的函数中，这并不理想。

这正是移动语义的用武之地了：如果我们能只是移动对象而不是复制它，那么性能会更高。

```

#include <iostream>

class String
{
public:
    // 默认构造函数
    String() = default;

    // 从C字符串初始化的构造函数
    String(const char* string)
    {
        printf("Created!\n");
        // 计算字符串长度
        m_Size = strlen(string);
        // 动态分配内存来存储字符串内容
        m_Data = new char[m_Size];
        // 从输入字符串复制内容到新分配的内存
        memcpy(m_Data, string, m_Size);
    }

    // 拷贝构造函数
    String(const String& other)
    {
        printf("Copied!\n");
        m_Size = other.m_Size;
        m_Data = new char[m_Size];
        memcpy(m_Data, other.m_Data, m_Size);
    }

    // 析构函数, 释放动态分配的内存
    ~String()
    {
        delete[] m_Data;
    }

    // 打印字符串内容
    void Print()
    {
        for (uint32_t i = 0; i < m_Size; i++)
            printf("%c", m_Data[i]);

        printf("\n");
    }

private:
    char* m_Data;        // 存储字符串内容的指针
    uint32_t m_Size;     // 字符串的大小 (长度)
};

class Entity
{
public:
    // 构造函数, 接受一个String参数来初始化名字

```

```

Entity(const String& name)
    :m_Name(name) {}

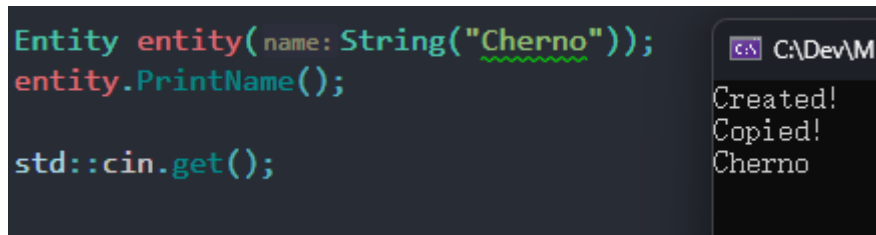
// 打印实体的名字
void PrintName()
{
    m_Name.Print();
}

private:
    String m_Name;    // 实体的名字
};

int main()
{
    // 创建一个名为"Cherno"的Entity对象
    Entity entity(String("Cherno"));
    // 打印这个Entity的名字
    entity.PrintName();

    std::cin.get();
}

```



```

Entity entity(name: String("Cherno"));
entity.PrintName();

std::cin.get();

```

C:\Dev\M
Created!
Copied!
Cherno

可以看到打印结果说明，我们的数据被复制了。

我们首先在main函数的作用域中创建它，然后把它传递给Entity的构造函数，再复制给m_Name，为什么我们不能直接把它分配到Entity的私有成员 `String m_Name` 里呢？我们无法这样做，除非能够访问这个字符串然后手动来操作。但我们可以再main函数中分配它，然后把它移动到这个空间中，这就是 *move statement*（移动语句）的用武之地了。

在这个例子中，我们需要写一个move构造函数，它和复制构造函数很像，除了它接受参数是一个右值（临时值）外。使用后，我们在main中调用时，`Entity entity(String("Cherno"));` 中的参数并不是一个左值，它没有赋值给任何东西，只是作为Entity构造函数的一个参数。

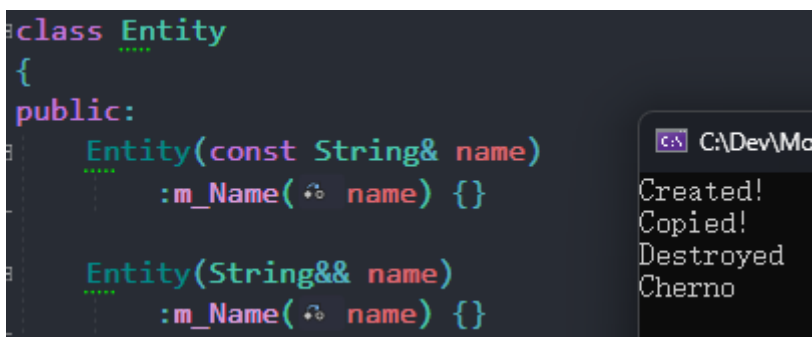
```

// 移动构造函数
String(String&& other) noexcept // 保证这个构造函数不会抛出任何异常
{
    printf("Moved!\n");
    // 从源对象拷贝大小和数据指针
    m_Size = other.m_Size;
    m_Data = other.m_Data; // 把源对象的数据指针直接赋给当前对象

    // 将源对象置于一个有效但不确定的状态
    other.m_Size = 0; // 设置源对象的大小为0
    other.m_Data = nullptr; // 设置源对象的数据指针为nullptr, 确保源对象不会删除此数据块
}

// Entity类的构造函数, 接受一个右值引用的String对象
Entity(String&& name)
    : m_Name(name) {} // 直接将右值引用的name传递给m_Name, 这里由于m_Name的构造函数也支持右值引用, 所以会调用String的移动构造函数

```



```

class Entity
{
public:
    Entity(const String& name)
        : m_Name(name) {}

    Entity(String&& name)
        : m_Name(name) {}

```

Output: Created!
Copied!
Destroyed
Cherno

确实调用了新写的Entity构造函数并输出了结果, 但还是发生了赋值, 说明调用的仍然是String的复制构造函数而非移动构造函数。

为了让它使用move构造函数, 我们必须显式地将它转换为一个临时对象:

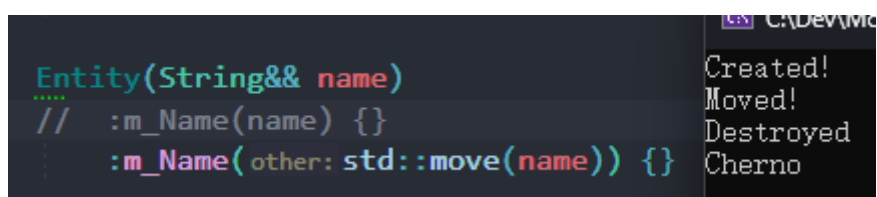
```

Entity(String&& name)
    : m_Name((String&&)name) {}

// 在实践中, 应该用更优雅的std::move来实现:

Entity(String&& name)
    : m_Name(std::move(name)) {} // 本质上和上面作用相同, 下节会详细讲述

```



```

Entity(String&& name)
// : m_Name(name) {}
    : m_Name(std::move(name)) {}

```

Output: Created!
Moved!
Destroyed
Cherno

现在我们成功地只分配了一次内存，并设法将字符串移动到了Entity类的成员中。没有使用复制构造函数来分配一个新内存块然后复制，只是移动了它，这太赞了。

2. “偷窃”内存

(85 左右值 一节中提到过)

这里移动构造函数确实“偷取”了 `other` 对象的内存。它直接接管了 `other.m_Data` 指向的内存，而不是分配新的内存并复制内容。这样，移动操作通常比复制操作更快、更有效率，因为它避免了资源的额外分配和复制。

之后，原始对象 `other` 的成员被设置为默认值（例如，`nullptr`），确保其析构函数不会释放已经转移出去的资源。这是移动语义的典型实现。