Week16

a. Why is it okay to pass a const char array into a function that requires an std::string?

这是因为 C++ 允许**隐式转换(Implicit Conversion)**。`std::string` 类拥有一个接受 `const char*`(字符数组)作为参数的构造函数。当你传递一个字符数组(如 `"Cherno"`)给需要 `std::string` 的函数时，编译器会自动调用该构造函数，将字符数组隐式地转换为一个 `std::string` 对象。

b. How many implicit conversions of an object are allowed in C++?

C++ 编译器只允许对一个对象进行**一次**隐式转换。例如，如果有从 A 到 B 的转换和从 B 到 C 的转换，编译器不会自动执行 A 到 C 的两次连续转换。

c. When does a stack-based object get destroyed and when does a heap-based object get destroyed?

- **栈上对象(Stack-based object)**： 当对象超出其所在的**作用域(scope)**时(例如遇到右花括号 `}`)，该对象会被自动销毁，其内存会被释放。
- **堆上对象(Heap-based object)**： 使用 `new` 关键字创建的对象，即使超出作用域也不会自动销毁。它会一直存在，直到你显式调用 `delete` 或者程序终止。

d. What is a scoped pointer?

Scoped Pointer(作用域指针)是一个包装了指针的类(Wrapper class)，通常用于自动化内存管理。其核心原理是利用栈对象的生命周期特性：在构造函数中初始化或接收一个堆分配的指针，并在**析构函数**中自动调用 `delete` 来释放该指针。这样，当 Scoped Pointer 对象超出作用域时，它所管理的堆内存也会被自动释放。

e. How to implement a scoped pointer class?

实现一个 Scoped Pointer 类主要包含以下步骤：

1. 创建一个类，包含一个成员变量用于存储原始指针。
2. 编写**构造函数**，接收一个指针并将其赋值给成员变量。
3. 编写**析构函数**，在其中对成员指针调用 `delete` 。

简化的代码逻辑如下：

```
class ScopedPtr {
private:
    Entity* m_Ptr;
public:
    ScopedPtr(Entity* ptr) : m_Ptr(ptr) {}
    ~ScopedPtr() { delete m_Ptr; }
};
```

f. How to overload the arrow operator for a scoped pointer class?

为了让 Scoped Pointer 像普通指针一样使用(即通过 `->` 访问成员)，需要重载箭头运算符 `operator->`。

- 该函数应该返回被包装的**原始指针**。
- 为了支持常量对象，通常还需要提供一个 `const` 版本的重载。

实现示例：

```
Entity* operator->() {
    return m_Ptr; // 返回内部保存的指针
}

const Entity* operator->() const {
    return m_Ptr; // const 版本
}
```

g. List five member functions of std::vector.

1. `push_back`：向向量末尾添加元素。
2. `size`：获取向量当前的元素数量。
3. `clear`：清空向量中的所有元素，将大小重置为0。
4. `erase`：删除指定位置的元素，通常需要配合迭代器使用。
5. `begin`：返回指向向量第一个元素的迭代器。