Week12

a. When will you probably use `using namespace`?

通常在以下情况可能会使用 `using namespace`：

- **局部作用域内**：如果只是在某个特定的函数内部(例如 `if` 语句块或函数体)频繁使用某个命名空间的内容，可以在该局部作用域内使用它，这样可以保持代码整洁且不污染全局作用域。
- **处理深层嵌套或冗长的命名空间**：当你处理非常长的命名空间，或者在你自己的项目中定义了深层嵌套的命名空间，且在特定实现文件中频繁访问其中的符号时，为了减少代码冗余可能会使用它。
- **个人小型项目或演示代码**：在非生产环境或小型学习项目中，为了方便演示有时会使用，但在严肃的工程中应避免对 `std` 全局使用。

b. Will you use `using namespace` in a header file?

绝对不会。

- 永远不要在头文件中使用 `using namespace`。
- 原因在于头文件会被其他文件包含。如果在头文件中声明了 `using namespace`，它会强制将该命名空间引入到所有包含了该头文件的源文件中。这会导致无法预料的命名冲突(Name Clashes)，并且随着项目规模变大，这种隐式的包含会导致难以追踪的编译错误。

c. Explain the difference between const int* and int* const.

- **`const int*`**(或 `int const*`)：指针指向的内容是常量。
  - 你**不能**修改该指针指向地址中的数据(即不能修改 `*ptr` 的内容)。
  - 但是，你**可以**修改指针本身，让它指向其他的内存地址。
- **`int* const`**：指针本身是常量。
  - 你**不能**修改指针本身指向的地址(一旦初始化，它就不能指向别处，也不能设为 `nullptr`)。
  - 但是，你**可以**修改该指针指向地址中的数据(即可以修改 `*ptr` 的内容)。

d. What is the difference between a const member function and a non-const member function of a class?

- **const 成员函数**：
  - 在函数声明的参数列表后加上 `const` 关键字。
  - 它承诺**不修改**类对象的任何成员变量(即它是只读的)。
  - 它可以被 `const` 对象(或 `const` 引用/指针)调用。
- **非 const 成员函数**：
  - 可以修改类的成员变量。

- **不能**被 `const` 对象调用。如果一个对象被声明为 `const`，它只能调用该类的 `const` 成员函数。

e. Why can't you declare a setter as const?

- 因为 Setter(设置器)的本质目的是**修改**(Write)类的成员变量。
- `const` 成员函数的定义就是承诺不修改类的任何成员变量。因此，在 Setter 中试图修改成员变量会违反 `const` 的契约，导致编译错误。

f. Interpret the restrictions on the getter const int* const GetX() const;

这个函数声明包含了三个层面的 `const` 限制：

1. **`const int*`**(返回类型部分)：函数返回的指针指向的数据是常量，**不能**通过该指针修改其指向的整数值。
2. **`* const`**(返回类型部分)：函数返回的指针本身是常量。这意味着返回的这个指针值(地址)是不应被修改的(尽管对于返回值而言，这通常意味着返回的是一个右值常量，实际意义在于强调其不可变性)。
3. **`GetX() const`**(函数尾部)：这是一个 `const` 成员函数。这承诺了 `GetX` 方法本身**不会修改**调用它的那个类对象(`Entity`)的任何成员变量。

g. What are the benefits of passing an object into a function by const reference?

- **性能优化(避免复制)**：通过引用传递(Reference)，避免了复制整个对象。如果对象很大，复制会造成显著的性能损耗(“浪费性能”)。
- **安全性(只读保证)**：通过 `const` 修饰，保证了函数内部不能修改传入的对象。这既享受了引用的速度，又拥有了传值的安全性(Read-Only)，防止数据被意外篡改。

h. Why is there a need for const member functions?

- **支持常对象的使用**：如果你的程序中创建了 `const` 对象，或者通过 `const` 引用传递对象，你只能调用该对象上的 `const` 成员函数。
- **接口契约**：如果不将不修改数据的方法标记为 `const`，编译器会阻止在 `const` 实例上调用它们。为了让只读操作(如 Getter)在任何情况下(包括 `const` 上下文)都能被调用，必须将它们声明为 `const` 。

i. What is the use the mutable keyword in a class?

- `mutable` 关键字允许成员变量即使在 `const` 成员函数中也能被修改。
- **用途**：它通常用于逻辑上是常量的操作，但内部实现需要修改某些状态的情况。最常见的例子是**调试**(例如在 `const` 的 Getter 中增加调用次数计数器)或**缓存**(内部数据结构变动但不影响外部观察状态)。

j. Why should you use member initializer lists?

- **避免双重初始化(性能)**：这是最重要的原因。如果在构造函数体内赋值，对于类类型的成员，它们会先调用默认构造函数进行初始化，然后在函数体内再次被赋值。使用成员初始化列表可以直接构造并初始化成员，避免了先构造再赋值的浪费。
- **代码整洁**：对于拥有大量成员变量的类，使用初始化列表可以让构造函数体保持整洁，专注于实际的逻辑代

码，而不是被一堆赋值语句堆满。

- **必要性**：对于 `const` 成员变量或引用成员变量，必须使用初始化列表，因为它们不能被赋值，只能被初始化(虽然源文件中未明确提及此点，属于补充知识，但源文件强调了性能差异)。

k. What happens if you don't initialize a class object in the member initializer list?

- **性能浪费**：如果不使用初始化列表，类类型的成员变量会首先调用其默认构造函数进行初始化。
- **双重工作**：接着，在构造函数体内进行赋值操作时，实际上是丢弃了刚才创建的初始值，并用新值覆盖它。这就相当于创建了两个对象(或者说进行了两次操作)，浪费了性能。