1. Each **class** controls how its members are initialized
2. The **base class** is initialized first, and then the **members** of the **derived class** are initialized in the order in which they are **declared** in the class.
3. If a base class defines a static member, there is only one such member defined for the entire hierarchy.
4. we can prevent a class from being used as a base by following the class name with **final**.
5. We must define every virtual function, regardless of whether it is used, because the compiler has no way to determine whether a virtual function is used
6. A **derived-class** function that **overrides** an inherited **virtual function** must have exactly the same parameter type(s) as the **base-class** function that it overrides.
7. With one exception, the return type of a virtual in the derived class also must match the return type of the function from the base class. The exception applies to **virtuals** that return a reference (or pointer) to types that are themselves related by inheritance. Example
8. A function that is virtual in a base class is implicitly **virtual** in its derived classes. When a derived class **overrides** a **virtual**, the **parameters** in the base and derived classes **must match exactly**
9. We can designate a function as **final**. Any attempt to override a function that has been defined as **final** will be flagged as an error.
10. **final** and **override** specifiers appear after the parameter list (including any const or reference qualifiers) and after a trailing return.
11. If a class uses a default argument, the value that is used is the one defined by the **static type** through which the function is called.
12. **Virtual functions** that have default arguments should use the same argument values in the base and derived classes.
13. In some cases, we want to prevent dynamic binding of a call to a virtual function; We can use the scope operator to force that call to use a particular version of that virtual.