1. **游戏软件工程基础**

**3.1 重温C++及最佳实践**

**扼要重温面向对象编程：**

类和对象：类(class)是属性(数据)和行为(代码)的集合,共同组成既有用又有意义的整体。可将类视为规格(specification),这些规格描述类的个别实例(instance)——又称为对象(object)

的构造方法。例如,一只叫阿旺的狗是“dog”类的一个实例。因此,类和其实例之间存有一对多的关系。

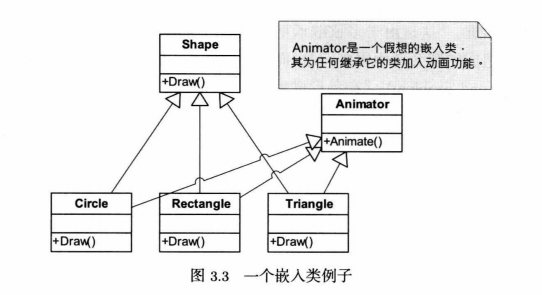
封装：封装(encapsulation)是指,对象向外只提供有限接口,隐藏对象的内部状态和实现细节。封装简化了类的使用方法,因为用户只需理解类的有限接口,而非类的内部实现细节,后者可能错综复杂。同时,程序员在编写类时,也可以通过封装使类的实体总是保持逻辑上的一致。

继承：继承(inheritance)能借着延伸现有的类去定义新的类。新类可修改或延伸现有类的数据、接口和行为。若一个名为child 的类延伸名为Parent 的类，可以说child继承自或派生自Parent。在此关系中, Parent称为基类(base class)或超类(superclass)，而 child则称为派生类(derived class）或子类(subclass)。显然,继承会产生类的层次(树状结构)关系。

继承使类之间产生“是一个(is-a)”关系。例如,圆形是一个图形种类。若要编写二维绘图应用软件,从基类shape 派生出circle类,是很合乎情理的做法。

我们可采用统一建模语言(Unified Modeling Language,UML)定义的表示法,描述类的层次结构图。在UML表示法里,长方形代表类,空心三角形箭头代表继承。继承箭头由子类指向父类。

------多重继承：多重继承是指一个类有一个以上的父类。理论上，多重继承是颇优雅的，但在实际应用中，这种设计通常会产生很多混淆和技术困难。这是由于多重继承把由类组成的简单的树(tree)变成可能很复杂的图(graph)。由类组成的图会产生很多问题,而这些问题不会在简单的树上发生,例如致命的菱形继承问题(diamond problem)。在菱形继承问题中，一个派生类最终包含了两份祖父类。(C++可以使用虚继承去掉重复祖父类的数据。)多重继承也会令类型转换变得更复杂,因为指针的实际地址会随转换的目标基类而改变。之所以出现这种情况，是由于对象存在多个vtable指针。大多数C++软件开发者都会完全避免使用多重继承,或只允许有限制地使用。常见的惯例是，只允许从一个单继承层次结构中多重继承一些简单且无父的类。这些类有时被称为嵌入类(mix-in class),因为它可在类树中的任何位置加入新功能。



多态：多态(polymorphism)是一种语言特征,允许采用单一共同接口操作一组不同类型的对象。共同接口能使异质的(heterogeneous)对象集合从使用接口的代码来看显得是同质的(homogeneous)。

例如,一个二维绘图程序要把一个形状列表绘于屏幕上,列表里有不同的形状。绘出这个异质形状集合的一种方法是,按不同形状类型,用switch语句区分并执行不同的绘制指令。

这种方式的问题是,drawShapes()函数需要“知悉”所有可以绘制的形状类型。在简单例子里还好,但随着代码量的增加和代码复杂度的提高,在系统里新增形状类型变得越来越困难。每加入一个新形状类型,必须搜索所有“知悉”各种形状类型的代码,如在例子里的switch语句中添加新的case去处理该新类型。

解决方法是把类型的内容从大部分代码中隔离出来。为了实施这个隔离,可以把每种要支持的形状定义为类。而所有这些类,都继承自一个共同的基类shape。在基类shape 中可以定义名为Draw()的虚函数(virtual function)(C++语言的主要多态机制),并在每个不同形状的类中,以不同方式实现这个函数。绘制时不需“知悉”给予的是何种形状,只需逐一简单地调用形状对象的 Draw()函数便可。

合成及聚合：合成(composition)是指,使用一组互动的对象去完成高阶任务。合成在类之间建立“有一个(has-a)”和“用一个(uses-a)”的关系。(从技术上说,“有一个”的关系称为合成，“用一个”的关系称为聚合, aggregation)。例如,一艘太空船有一台引擎,引擎又有一个燃料缸。使用合成/聚合常常使各个类变得更简单、更专注。缺乏面向对象经验的程序员常会过分依赖继承,而忽视合成及聚合。

设计模式：当同一类型的问题反复出现,而不同的程序员却采用相似的方案去解决这些问题时,就可以说,该问题引发了一个设计模式(design pattern)。在面向对象编程中,已经有很多常见的设计模式获得识别及描述。其中最知名的,是“四人组(Gang of Four, GoF)”著作内的23个设计模式。以下是几个常见的通用设计模式。

1.单例(singleton):此模式确保某个特殊类只有一个实例(这个就是单例实例, singleton in-stance),并提供这个单例的全局存取方法。

2.迭代器(iterator):迭代器提供高效存取一个集合的方法,同时不需要暴露该集合之下的实现。

3.抽象工厂(abstract factory):抽象工厂提供一个接口,创造一组相关或互相依赖的类,而不需要指明那些类的具体类(concrete class)。

游戏工业有自己的一套设计模式，以应对渲染、碰撞、动画、音频等各领域的问题。从某种意义上来说,本书所有内容都是关于现在三维游戏引擎设计中流行的高阶设计模式的。

**编码标准：为什么及需要什么**

工程师之间讨论编码约定(coding convention)时,经常能引起热烈的“宗教”辩论。笔者不希望在此引发那种辩论,但我提议,应该至少遵循以下这些编码标准(coding standard)的最小集合。编码标准之所以存在,有两个主因。

1.一些标准使代码更易读、更易理解、更易维护。

2.另一些约定能预防程序员做蠢事,自找麻烦。例如,某编码标准可能会促使程序员只使用编程语言中更易测试、更不易出错的一小部分功能。由于C++语言充满滥用的可能性,所以这类编码标准对使用C++来说特别重要。

编码约定中最需要做到的事情如下。

1. 接口为王:保持接口(.h 文件)整洁、简单、极小、易于理解,并有良好的注释。
2. 好名字便于理解及避免混淆:坚持使用能直接反映类、函数、变量用途的直观名字,应花些时间确定合适的名字。如果有一种命名方法,需要程序员查表才能理解代码的意义，就要避免使用这种命名方法。谨记，像C++这样的高级编程语言是为了供人阅读而设计的(若读者不同意,就问一问自己为何不直接用机器语言来编写你的全部软件)。
3. 不要给命名空间添乱:使用C++命名空间或统一的名字前缀,以确保自己的符号(symbol)不会和其他库的符号冲突。(但慎防过度使用命名空间或嵌套过深。)为宏命名时更要小心,因为C++预处理器的宏只是文本替换,所以宏会跨越全部C/C++作用域及命名空间范围。
4. 遵从最好的C++实践:一些书籍提供了卓越的指导方针，可使程序员避开麻烦，例如,Scott Meyers的 Effective C++系列(见参考文献[31]、[32])、Effective STL (见参考文献[33]),以及John Lakos的 Large-Scale C++ Software Design。
5. 始终如一:笔者会使用以下的规则,若从零开始写代码,可以自由地创造你的编码约定,然后坚持遵守约定。当编辑一些已有的代码时,无论那里有什么约定,都请尝试遵从。
6. 显露错误:Joel Spolsky写了一篇关于编码约定的出色文章《让错误代码显得错误》(Making Wrong Code Look Wrong)。文中提出,所谓最“整洁”的代码,并不需要是表面看来简洁整齐的代码,而更重要的是,代码的编写方法能容易显露常见的编程错误。Joel 的文章有趣且富有教育意义,笔者极力推荐此文。

**C++11**

auto:auto并不是C+＋语言的新关键字,但在C++11中改变了它的语义。在C++03中, auto是一种存储类指定符(storage class specifier),这些修饰符还有static、register和externo每个变量只能使用这四者其中之一,而默认的存储类指定符是auto，它表示变量为局部作用域,应分配于寄存器(如有足够的寄存器)或程序堆栈中。在C++11中, auto关键字被用作变量类型推导,即它可用于代替类型指定符,编译器从变量初始化表达式的右侧推导出变量的类型。

nullptr:在之前版本的C和C++中,NULL 指针使用字面量o来表示,有时候会转型为(void\*)或(char\*)。这种做法欠缺类型安全性，并且可能因为C/C++的隐式整数转换而造成问题。C++11中加入了类型安全的显式常数值nullptr表示空指针,它是类型std::nullptr\_t的实例。

基于范围的for循环：C++11扩展了for语句去支持一种简写的“foreach”循环风格。这个新扩展可让你迭代C风格的数组,以至任何数据结构,只需数据结构提供非成员函数begin ()和end ()。

override及final：C++中的virtual 关键字可导致令人迷惑和错误的代码,因为语言没有把这个关键字的多个用途分开:

1. 在类中加入一个新虚函数;
2. 覆盖(override)一个继承而来的虚函数,及
3. 实现一个叶子(leaf)虚函数,该虚函数不计划被子类覆盖。

而且,C++根本不需要程序员用virtual关键字去覆盖虚函数。为了纠正部分现状,C++11引人了两个新指定符,放置于虚函数声明的最后,从而把程序员的意图告诉编译器和代码的其他读者。override指定符表示该函数会覆盖一个基类现有的虚函数。而final指定符标记该虚函数不能再被子类覆盖。