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。

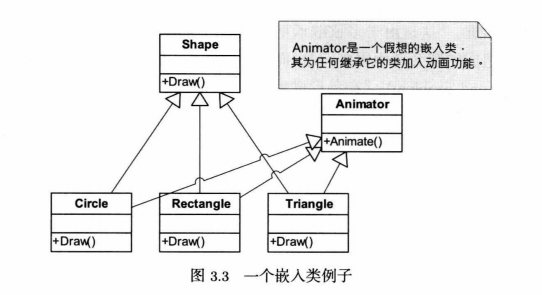
封装：封装(encapsulation)是指,对象向外只提供有限接口,隐藏对象的内部状态和实现细节。封装简化了类的使用方法,因为用户只需理解类的有限接口,而非类的内部实现细节,后者可能错综复杂。同时,程序员在编写类时,也可以通过封装使类的实体总是保持逻辑上的一致。

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。

强类型的enum：在C++03中, enum 会把其枚举项输出至该作用域,而枚举项的类型由编译器按枚举里的值决定。C++11引入了新的强类型枚举,它以关键字enum class来声明,其枚举项的作用域像类或结构体的成员,也让程序员可以指明底层的类型。

标准化的智能指针：在C++11中,std: : unique\_ptr , std: :shared\_ptr和 std: : weak\_ptr提供了一个扎实的智能指针设施,可满足我们一直期待的所有需求(如 Boost 库的智能指针系统)。当我们要让指向的对象维持具有独占的拥有权时,便可使用std: : unique\_ptr。若需要令多个指针指向单个对象,应该使用多个具有引用计数功能的std::shared\_ptr。另外,std: : week\_ptr的作用类似于std: : shared\_ptr,但它不影响指向对象的引用计数。因此, std: :week\_ptr常常用作“反向指针”,或用于其他指针“图”包含循环的情况。

lambda：lambda是匿名函数。它可用于任何需要函数指针、函子(functor)或std: : function的地方。术语lambda借用自Lisp和 Scheme等函数式语言。

可以用lambda内联实现一个函子,而无须在外部声明一个具名函数,然后把该函数作为参数传入。例如:

移动语义与右值引用：C++11提供了一个机制,允许我们纠正这类拷贝问题,但又无须改变函数签名去把输出对象以指针或引用参数方式传入函数。此机制称为移动语义(move semantics)，它依赖于可以区分拷贝左值(lvalue)对象和拷贝右值(rvalue)(临时)对象。

在C和C+＋中,左值表示在计算机寄存器或内存中的实际存储位置。而右值是临时数据对象,它只是逻辑上存在但非必须占用内存。当我们写int a = 7;时,变量a是左值,而字面值7是右值。你可以为左值赋值,但不能为右值赋值。

在C++03及之前,我们无法对右值和左值采用不同的拷贝策略。因此,拷贝构造函数和赋值运算符都假设了最坏的情况,把一切都当作左值。例如拷贝一个如 std: :vector 的容器,拷贝构造函数和赋值运算符都会进行深度拷贝(deep copy)——不单拷贝容器本身,还需对其存储的全部数据逐一拷贝。

在C++11中,我们可声名一个变量为右值引用(rvalue reference),方法是采用& &而不是&(如,int&& rvalueRef而不是int& lvalueRef)。然后我们便可以分别为拷贝构造函数和赋值运算符各写两个重载版本—-一个用于左值,一个用于右值。当拷贝一个右值时(如临时对象),不需要进行深度拷贝。取而代之,只需简单地“盗取”临时对象的内容,把它们直接移动至目标对象——因此它的术语为移动语义。

**3.2 C/C++的数据、代码及内存**

**数值表达形式：**

数值底数：计算机中常见的数值底数有2和16

有符号数及无符号数：在计算机科学中,我们同时使用有符号整数(signed integer)及无符号整数(unsigned in-teger)。其实,“无符号整数”有点用词不当。在数学上，自然数(natural number)'的范围是由0(或1)至正无穷,而整数的范围则是负无穷至正无穷。虽然如此,本书还是采用计算机术语,统一用“有符号整数”和“无符号整数”。

多数个人计算机和游戏主机能轻易处理32位或64位整数(虽然在游戏编程中也经常用到8位及16位的整数)。要表示一个32位无符号整数,只需把数值简单地编码为二进制记法。32位无符号整数的可表示数值范围从Ox00000000(0)至0xFFFFFFFF (4,294,967,295)。

要用32位表示有符号整数,需要一个方法去分辨正值和负值.最简单的方法之一就是原码法(sign-and-magnitude encoding),把最高有效位(most significant bit,MSB)用作符号位(sign bit)——符号位为О代表数值为正,1代表数值为负。余下的31位则是数值的模(magnitude),实际上即是把模的范围缩小至一半(但这样能表示每个模的正负两个版本,包括正负零)。

大多数微处理器采用更高效的技巧为负整数编码,此技巧称为二补数(two's complement)记法。对于数字零，二补数有唯一的表示方式,而简单使用符号位则会造出两个零的表示方式(正零及负零)。在32位二补数记法里,0xFFFFFFFF值代表-1,其他负值就从这个值倒数。任何最高有效位为1的值都代表负值。所以,从Ox00000000(0)至0x7FFFFFFF (2,147,483,647)的值代表正整数,从0x80000000 (-2,147,483,648)至0xFFFFFFFF(-1)代表负数。所谓的二补数其实就是补码。

定点数法：定点记法的缺点在于,它限制了可表示整数部分的范围及小数部分的精度。例如一个32位定点小数,其中有16位整数、15位小数、1位符号。此格式去除小数部分,其表示范围只是土65,535,并不一定足够。要解决此问题,可使用浮点记法。

浮点记法：在浮点(floating-point)记法里,小数点可以任意移动至不同位置,此位置仍是由指数(exponent)控制的。一个浮点数由3部分组成:尾数(mantissa)含有包括小数点前后的相关数字,指数(exponent)决定那串数字的小数点位于哪里,而符号位理所当然就是显示该值为正数或负数。虽然有不同的方式去安排这3部分在内存中的格式,但最流行的标准是IEEE-754。IEEE-754标准中定义的32位浮点数,其最高有效位是符号位,紧随的是8位指数和23位尾数。

浮点数的精度增加,可表示范围则缩小,反之亦然。这是因为,若使用固定数目的位去表示浮点数,尾数和指数所占的位数此消彼长。尾数位越多,精度越高;指数位越多,可表示范围越大。物理中常使用有效数字(significant figure)去描述此概念。

有限精度和机器epsilon 的概念对游戏软件有实质影响。例如,假设我们用浮点数去表示游戏从开始至今经过了多少秒,并称之为游戏绝对时间,那么,游戏要运行多久,才会导致加上1/30秒后,游戏的绝对时间维持不变?答案是大约12.9日。多数游戏不需要运行这么久,所以使用32位浮点表示以秒计算的游戏绝对时间还是可行的。然而,此例清楚地显示，我们必须了解浮点格式的限制,从而能预知潜在的问题,并按需采取措施加以防范。

基本数据类型：

1. char: char通常是8位的,足够存储一个ASCII或UTF-8字符(见5.4.4.1节)。有些编译器定义char为带符号的,有些编译器则预设char为无符号的。
2. int 、 short 、 long: int是有符号整数值,而其大小恰好是目标平台上最高效的运算单位。在32位CPU架构上, int通常被定义为32位,例如奔腾4(Pentium 4)和至强(Xeon);在64位架构上,int通常被定义为64位,例如英特尔Core i7;但 int的大小还受其他因素影响,如编译选项及目标操作系统。short本意是比int小的类型,short 在许多机器上为16位。而long 则等于或大于int,long 在一些平台上是32位或64位,甚至更宽,这同样取决于CPU架构、编译选项及目标操作系统。
3. float:在大部分现代编译器里,float是IEEE-754 的32位浮点数。double: double是IEEE-754的双精度(即64位)浮点数。
4. bool: bool保存真/假值。bool在不同编译器及硬件架构上会采用截然不同的大小。bool从不会实现为1位,有些编译器定义bool为8位,也有些将其定义为32位。

SIMD类型---许多电脑和游戏主机的CPU都有特殊的算术逻辑单元(arithmetic logicunit，ALU),称为矢量处理器(vector processor)或矢量单元(vector unit)。矢量处理器提供一种并行处理方式,名为单指令多数据(single instruction,multiple data,SIMD)。单个SIMD 指令可以并行地对多个数据进行运算。数据由矢量处理器处理,需先把数据以两个或更多个数值打包,存进64位或128位CPU寄存器。在游戏编程中,最常用的SIMD寄存器格式,是把4个32位.IEEE-754浮点数值打包,存进128位SIMD寄存器。此格式使一些计算,如矢量点积和矩阵乘法,比单指令单数据(single instruction, single data,SISD)算术逻辑单元更加高效。1

每个微处理器的 SIMD指令集名字各有不同,而且编译器对不同目标微处理器有特定的SIMD变量声明语法。例如,奔腾系列CPU的指令集称为单指令多数据流扩展(streamingSIMD extensions, SSE)，而微软Visual Studio编译器则提供内建数据类型\_m128,表示4个浮点数的SIMD 数值。在PS3和 Xbox 360 的PowerPC系列CPU中,其SIMD 的指令集称为Altivec，而 GNU C+＋编译器采用vector float语法去声明打包4个浮点数的SIMD变量。在4.7节会更详尽地讨论SIMD编程。

多字节值及字节序：大于8位(1字节)的值称为多字节量(multibyte quantity)。在使用16位或以上的整数/浮点数数值的软件中,多字节量非常普遍。例如,整数值4660 = 0x1234可由两个字节0x12和0x34表示。0x12称为最高有效字节(most significant byte，MSB),0x34称为最低有效字节(least significant byte,LSB1)。在32位的值中,例如0xABCD1234,最高有效字节是OxAB,最低有效字节是0x34。同样的概念也应用于64位整数及32/64位浮点数中。在内存中存储多字节整数有两种方式,不同的微处理器的选择有所不同

小端(little-endian):若微处理器将多字节值的最低有效字节存储于较低的内存位置,则该微处理器就是小端处理器。在小端的机器上，数字OxABCD1234 在内存中存储为连续字节0x34、Ox12、0xCD 、OxAB。

大端(big-endian):若微处理器将多字节值的最高有效字节存储于较低的内存位置,则该微处理器就是大端处理器。在大端的机器上，数字0xABCD1234在内存中存储为连续字节OxAB、OxCD、0x12、0x34。

多数程序员不需要顾虑字节序(endianness2)。然而,字节序对游戏程序员来说可能是一根刺。因为游戏通常是在英特尔 CPU(小端)的PC上开发的,而游戏可能执行于游戏主机,如Wii 、Xbox 360 、PlayStation 3—这3台主机皆使用PowerPC处理器的变种(可设置使用任意字节序,但预设是大端)。如果产生一个数据文件,供在英特尔处理器上运行的游戏引擎读取,之后再由 PowerPC上运行的引擎读取,会发生什么事情呢?任何写到数据文件中的多字节值都是小端格式,但在 PowerPC上运行的游戏引擎期望从数据文件中读取大端格式的数据。结果是,若写的是OxABCD1234,就会读到0x3412CDAB,显然非我们所要。

此字节序问题最少有两种解决办法。

1. 所有数据以文字方式写入文件。多字节数值以一串十进制数字或十六进制数字,每数字一个字节写入。此方法会浪费磁盘空间,但却可行。
2. 先用工具转换数据字节序,然后再把转换后的数据写进二进制文件。也就是说,即使执行工具的机器字节序与目标机器相反,也可确保存储的数据为目标机器的字节序。

整数字节序转换整数字节序转换的概念并不复杂。首先把该值的最高有效字节和最低有效字节交换,再继续交换直到该值的中间点。例如,OxA7891023会变成0x231089A7.

唯一的难点在于要知道哪些字节序要转换。例如,把内存中的C struct或C++ class写入文件时,要正确地转换字节，便需要知道struct里每个数据成员的位置及大小,并基于每个成员的大小逐一进行适当转换。

浮点字节序转换 让我们看看浮点字节序转换和整数字节序转换的差异。之前提及,IEEE-754标准中的浮点数有详细的内部结构,其中某些位作为尾数,某些位作为指数,并有一个符号位。虽然其结构比较复杂,但仍然可以把浮点数当作整数转换字节序,因为字节始终是字节。可以使用C++的reinterpret\_cast操作把浮点数诠释为整数,这又称为类型双关(typepunning)。但是,当使用严格别名(strict aliasing)时,类型双关可能会导致优化 bug。(有一篇文章对此问题做了非常好的描述1。)取而代之,一个简便的方法是使用union:;

**声明、定义及链接规范**