**C++学习笔记——编程理念**

**1. 尽可能使用已有的STL模板数据结构**

常见STL模板数据结构包括array、vector、list、map等。只有模板数据结构确实不能满足需要时，才考虑自定义数据结构。

STL是大量优秀C++设计者长期工作的成果，在尽可能确保安全性的基础上，有着最高的执行效率。

**2. RAII (Resource Acquisition Is Initialization)**

描述：

资源（Resource），是编码中需要动态申请和回收的东西的总称。例如：堆空间。

RAII原则要求用对象来管理资源。为此需要声明资源管理类。具体来说：

(1) 在资源管理类中，封装原生资源。

(2) 用户创建资源管理对象时，对象自动申请原生资源。这一条是RAII得名由来。

(3) 将释放资源的代码写在类的析构函数当中。对象生命周期结束时，析构函数被自动调用，从而释放资源。用户无需（也不被允许）手动释放原生资源。

(4) 必要时，向用户提供访问封装的原生资源的接口函数。约定俗成的函数名为get()。

意义：

(1) 通过“对象生命周期结束时，必然调用析构函数”这一编程语言机制，确保资源一定会被释放，而不会因为用户忘记释放或代码执行出现异常，导致资源未被释放，出现内存泄漏的现象；也不会因为用户错误地多次释放同一资源导致程序崩溃。

(2) 便于用户使用。用户只需要创建资源管理对象，无需考虑申请和释放原生资源的细节。

示例：

智能指针（std::unique\_ptr、std::shared\_ptr）、STL容器类（std::vector）。

**3. 让接口容易被正确使用，同时不易被误用**

(1) 如果用户试图使用某个接口而却没有获得预期的行为，那么代码就不应该通过编译；如果代码通过了编译，它的行为就应该是用户想要的。

换言之，对于误用接口的行为，最好的应对措施是编译不通过，而非运行时出现错误提示或强制退出程序。

(2) 任何接口如果要求用户必须做某件事情，就是有着“不正确使用”的倾向，因为用户可能忘记做，或者以不正确的方式做那件事情。

例如：堆空间资源的释放。

(3) 自定义类型接口的行为应当尽可能与内置类型的行为保持一致。这种一致性是避免接口误用的重要前提。

(4) 具体方式：

- 封装内置类型：用自定义类型封装内置类型，从而应用更强的类型检查规则

- 限制类型上的操作：例如增加const限定

- 限定对象取值范围：例如将一个类的构造函数设为private，而后通过静态成员函数创建对象；或者使用enum class

- 避免用户承担资源管理责任：在第2节中已经提到

**4. 异常安全性保证**

(1) 除了极特殊需求之外，理论上，所有C++代码都应保证下列三种异常安全性之一：

**基本保证：**程序抛出异常时，程序的实际状态不可预知，但一定属于在预先定义的某种合法状态之一（类似“薛定谔的猫）。

**强烈保证：**程序抛出异常时，会完全回复到之前的状态。等价于原子性——程序状态要么完全更新，要么完全回退。

**不抛出异常（no-throw）保证：**保证代码不会抛出异常。在C++11中，可以用noexcept关键字声明此类保证，并且编译器在编译时还会对noexcept代码进行适当优化。

(2) 所有基于C++内置类型（基本类型和原生指针类型）的操作，都具备异常安全性。

(3) 具备异常安全性的函数，一定满足以下两者之一：

- 不抛出异常

- 抛出异常时，不泄漏任何资源（例如没有释放堆空间资源、永久占用互斥锁），也不破坏任何数据（例如令指针指向被销毁的对象）

(4) copy-and-swap是一种常用的确保异常安全性的策略，具体流程如下。注意，全程都需要使用智能指针，绝不允许使用原生指针。

假定智能指针pOld当前指向堆空间的一个对象obj。现在需要修改obj，全程对异常安全性提供**强烈保证**。

copy：用new深拷贝原始对象，生成副本，并用来初始化一个临时智能指针pTemp。

update：修改pTemp所指向的空间的内容。

swap：交换原始指针和pTemp。一般用std::swap即可；也可使用自定义swap函数，但自定义swap函数必须具备强烈异常安全性。

示例代码如下：

std::shared\_ptr<\_Tp> pNew = new \_Tp(\*pOld); // copy

pNew->someVar = newVal; // update

std::swap(pOld, pNew); // swap

copy-and-swap基于对象的深拷贝，因此时间和空间开销很大；并且，如果其他代码不具备强烈异常安全性，那么即使采用此策略，整体代码仍然不具备强烈异常安全性。

所以，并非所有场合都要采用copy-and-swap。

(5) 一个函数的异常安全性，最高只等于其所调用的各个函数的异常安全性的“最弱者”。

(6) 异常安全性的保证，需要技术和非技术因素的共同参与。

**5. public继承、组合、private继承的语义**

**(1) public继承**

如果类D对类B采用public继承，意味着二者构成”is-a”关系.

is-a：

等价于里氏替换原则（Liskov substitution principle）——即：D是B的一种；所有能够用到B的地方，也可以用D来取代B。换言之，D不应该比B具有更多的限制。

典型例子不胜枚举。这里说一下反例。

反例：正方形和矩形。在面向对象编程里，正方形与矩形不构成”is-a”关系，因为前者比后者具有更多的限制（要求邻边必须相等）。

**(2) 组合**

组合可用于实现两种语义：”has-a（包含）”、”is-implemented-in-terms-of（以某物实现出）”。

has-a：

非常直观。

示例：员工类包含各种个人信息，而每种个人信息都作为一个类而存在。

这个例子显然符合has-a，而不是is-a语义——毕竟，我们会说“员工具有住址/电话号码/公司名称”，而不能说“员工是住址/电话号码/公司名称的一种；所有用到住址/电话号码/公司名称的地方，都可以用员工来取代之”。

is-implemented-in-terms-of：

利用某一既有类的实现，来实现另一个类。经常用于数据结构当中。下文将这个被利用的类称为“**既有类**”。

使用组合表达此语义时，只能访问既有类的public接口，且无法修改既有类的实现。

示例：堆与数组、二叉排序树与链表。在这两个例子中，前者都是用后者来实现的。

**(3) private继承**

表达is-implemented-in-terms-of语义。与组合的第二种语义完全相同。

需要表达此语义时，应优先使用组合；但出现以下情形之一时，推荐使用private继承：

- 需要调用既有类的protected或private函数

- 需要修改既有类的既有实现，并且另一个类允许这样做（即既有类的相应函数为virtual且非final）

**特别提醒：**

对于std命名空间的所有类（例如STL模板类），只能使用组合，不能使用继承。因为采用继承意味着需要修改实现，而std命名空间的内容是不允许自行修改的。

**6. public继承中接口与实现的继承**

本节所述的理念，全部基于public继承，即派生类与基类构成”is-a”关系。

在面向对象编程中，基类的函数，按照接口和实现划分，可分为以下三类：

**纯虚函数：**只提供接口；一般不提供实现，即使提供实现也不能被直接继承。

**普通虚函数：**提供接口和默认实现，允许派生类重写。

**非虚函数：**提供接口和强制实现，原则上派生类不应该重写它。

注意事项：

(1) 基类的普通虚函数会提供默认实现，这可能导致一种情形：一个派生类本来不适用基类的默认实现，但开发者忘记重写它。这样的代码，编译并不会报错，但运行时会出现严重错误。这属于接口被误用，因此务必谨慎检查。

(2) 纯虚函数可以有实现版本，但这一特性很少被使用。除了纯虚析构函数必须要实现以外，其他时候，实现纯虚函数，更多是为了避免(1)中情形发生。