More Effective C++

1. 区分指针和引用

   指针和引用的最实质的区别在于：不存在为空的引用，引用必须指代某个对象。

   如果需要能够在不同时候指向不同对象，或者有可能什么也不指向，这个时候应该用指针；

   如果不允许指代null和不会改变所指代的对象的时候，就应该使用引用。

   还有一点，指针是兼容C的，而引用为C++专用。

2. 优先使用C++风格的类型转换

   C++风格的类型转换比C风格的强制类型转换更佳，而且更好查找，这会使用用工具分析代码更方便。

   static_cast<type>(expression) 常用的类型转换

   const_cast<type>(expression) 为去除const特性的类型转换

   dynamic_cast<type>(expression) 操纵继承体系的类型转换

   reinterpret_cast<type>(expression) 最常见的用法是用于在函数指针之间进行类型转换，几乎是不可移植的，这个转换应该尽量不要用

3. 绝不要将多态应用于数组

   将多态运用于数组时，数组的遍历行为都将产生错误，编译器无法遍历派生类的数组。

4. 避免不必要的默认构造函数

   没有默认构造函数的类，通常以下三种情况容易出问题

   1. 在堆上创建数组时，通常没有很好的访求可以指定数组元素的构造函数的参数。

      需要使用placement new来进行无格式的内存分配才能做到，非常麻烦容易出错。析构时需要用手动完全delete做的两种工作：调用析构函数和调用operator delete[]来释放内存。

   2. 无法作为许多基于模板的容器类的类型参数使用。

   3. 虚基类，没有默认构造函数的虚基类使用起来很痛苦。因为虚基类的构造函数所要求的参数必须由被创建对象所属的最远的派生类所提供。这样，所有的派生类都必须理解虚基类构造函数的参数的含义并提供这些参数。

5. 小心用户定义的转换函数

   任何时候，都应该把单个参数的构造函数声明为explicit，除非你自己明确的想要这个隐式的转换，但是这样的隐式转换几乎总是不好的。相对于隐式的转换，宁愿使用一个显式的转换！

   而且到某某类型的类型转换更难以使用，往往使用一个asDouble()类似的函数是更好的选择。

   允许编译器进行隐式类型转换往往弊大于利的，所以除非确实需要，不要提供类型转换函数。

6. 理解new和delete在不同情形下的含义

   1. new operator通常做两件事情：申请内存和调用构造函数进行初始化。

string *ps = new string("Memory");

其实类似于以下代码

void *memory = operator new(sizeof(string));

call string::string("Memory") on *memory;

string *ps = static_cast< string* >(memory);

2. delete operator是对应的，delete是逆序做这些事情，先调用析构函数，再回收内存。

   delete ps;

   类似于以下代码

   ps->~string();

   operator delete(ps);

3. 如果想在一块已经有指针指向的内存里建立一个对象，就使用placement new

   new (buffer) Widget(widgetSize);

7. 使用析构函数防止资源泄漏

   #include <memory> 对于那些使用本地资源的指针，尽可能的使用auto_ptr，防止资源的泄露（永远永远都要记住，资源一定要被封装在对象里）

   写一个类似于auto_ptr的智能指针应该有以下的部分：

   - 构造 + 析构
   - 当这是一个智能类型而不是一个智能指针的时候，需要一个转换为原来类型的隐式类型转换，以确保在任何可以使用以前类型的地方都可以使用新的类型。
   - 明确禁止赋值和拷贝函数（声明为private）
   - 重载指针操作符&和.（如果是智能指针而不是类型的话）

8. 防止构造函数里的资源泄漏

   C++只销毁构造完全的对象，所谓构造完全的对象是指它的构造函数被完全执行的对象。因此：如果一个对象在其构造函数中抛出异常的话，它的析构函数是不会被调用的。

   而且C++中没有任何办法可以做到销毁未构造完全的对象，所以唯一的办法是手动去确保自己的构造函数设计成能够自己处理异常，在发生异常时能自己清理已经构造的部分对象。

   将自定义类的构造函数用一个try{} catch(...){}块包围起来。

   或者，简单的使用auto_ptr就可以了：把那些声明为指针的类成员替换成它们相应的auto_ptr对象。

9. 阻止异常传递到析构函数以外

   析构函数有两种可能被运行：一是对象的正常析构、二是异常传递过程中的栈解开。因此析构函数应该永远考虑在异常传递时被调用的情况。而如果析构函数内部抛出异常未被处理，当控制权离开析构函数的时候，碰巧另外一个异常也处于活动状态，被会导致C++立刻调用terminate函数，终止程序的运行，记住是立刻调用terminate而不进行栈解开。

   所以就像Effective C++和More Effective C++中反复提到的一样：析构函数永远不应该抛出异

常！

10. 理解抛出异常与传递参数或者调用虚函数之间的不同

    传递一个对象到一个函数或者通过这个对象触发一个虚函数与把一个对象作为异常抛出之间有三个主要的不同之处：

    1. 异常对象总是要被copy的，这是对于异常对象的强制性拷贝（这是为了避免抛出一个过期的局部变量）！当以传值的方式被捕获时，它被拷贝了两次，以引用传递时被拷贝一次；而传递给函数参数的对象绝对没有强制性拷贝的要求。

       这里要注意一个问题是：拷贝永远是基于对象的静态类型的

       ```
       SpecialWidget Local_sw;
       Widget& w = Local_sw;
       throw w;  //先拷贝w的一份临时变量，再将那个临时变量抛出。
                 //被抛出的临时变量类型为Widget，而非 SpecialWidget。
       ```

    2. 作为异常抛出的对象不会进行隐式类型转换，catch(double)永远都不能捕获一个类型为int的异常的。但是基于继承体系的类型转换可以发生，为捕获父类设置的catch可以捕获子类类型的异常；还有就是可以从一种类型指针转换到无类型的指针（这种情况很多，可以视为void*为所有指针类型的父类吧）。
    3. 虚函数采用的是"最优匹配"、而异常处理采用的是"最先匹配"。这点非常容易理解！

11. 通过引用捕获异常

    给catch传递异常对象有有三种方案：by value, by reference, by point

    1. by value会遇到异常对象的切割问题。而且虚函数只有在引用和指针时才能正常工作，所以会造成只调用父类定义的虚函数的问题。还有，对象传递时需要拷贝两次！
    2. by point必须保证控制权离开抛出指针的那个函数以后这个对象仍存在，一般需要全局对象和静态对象完成这件事情。如果从返回从堆中new的对象时，还要遇到对象的删除问题。
    3. by reference可以避免所有的问题，而且异常对象只需要拷贝一次。在没有特殊理由的情况下，通过reference捕获异常是最佳选择！

12. 谨慎地使用异常规格

    如果异常规格被违反（实际上异常规格太容易就被违反，标准规定因为编译器只能做部分的检查），将调用unexpected->terminate->abort。活动栈桢中的局部变量不会被销毁、申请的锁资源不会被释放等等，因为abort终止程序时不会做清理工作。甚至，它们会阻止高层次的异常处理函数来处理所导致的unexpected异常，即使高层次的函数知道如何处理。

    而且有一点：异常规格有着与try块一样的运行开销，它不仅仅是规格说明的。

13. 了解临时对象的来源

    任何时候只要见到常量引用参数，就存在创建临时对象与这个参数相绑定的可能性。任何时候只要见到函数返回对象，就会有一个临时对象被创建（稍后被销毁）。

    C++禁止为非常量引用产生临时变量，很明显临时变量不能够被修改的！

14. 协助编译器实现返回值优化

    许多函数是必须以传值的方式返回对象的（eg:operator*），可以通过返回带参数的构造函数而不是局部对象的方法来实现返回值优化。

```
inline const Rational operator* ( const Rational &lhs, const Rational &rhs )
{
 //Rational result( lhs.numerator*rhs.numerator, lhs.denominator*rhs.denominator );
 //return result;
 return Rational( lhs.numerator*rhs.numerator, lhs.denominator*rhs.denominator );
}
```

    C++允许编译器消除operator*内部的临时变量以及所返回的临时变量，编译器可以把return表达式所定义的返回对象构造在分配给用于保存函数返回对象的内存上。

    这是一种依赖编译器的优化，但是大多数的编译器都已经实现了这个优化。

15. 考虑使用op=来取代单独的op运算符

    由于单独形式的运算符（如：operator+)依赖于赋值版本的运算符（如：operator+=）所实现，所以赋值版本的运算符效率更高。

    还有一个：没有名字的临时变量的效率要比有名字的局部变量开销更小一点。

16. 理解虚函数、多重继承、虚基类以及RTTI所带来的开销

    1. 虚函数的实现机制其实很简单：virtual table + virtual table pointer

       每一种类都有一个virtual talbe，记录着当前类型所对应的虚函数的入口地址；然后每一个对象都有一个virtual table pointer（编译器实现为隐藏的数据成员），指向一个virtual table。这样，调用虚函数时进行查表就可以得到应该调用的函数的入口地址。
    2. 编译器一般总是忽略虚函数的inline指令。因为inline意味着"在编译时刻用被调用的函数体来代替被调用函数"而虚函数意味着"在运行时刻决定被调用的是哪一个函数"

17. 使构造函数和非成员函数具有虚函数的行为

    针对虚函数返回值的松散规则：派生类重新定义基类的虚函数时，不再被强制要求和基类函数一样的返回值，但是参数的类型必须要一致！

18. 限制类对象的个数

    当一个对象被实例化时，我们唯一可以确定的是：肯定会有一个构造函数被调用。

    1. 允许一个或0个对象：class Printer

```
{
public:
friend Printer& thePrinter(); //设置好友元,让thePrinter访问私用的构造方法
private:
Print(const Printer& rhs);    //构造函数私有,就保证不能再创建新对象了
};
```

```
Printer& therPrinter()        //注意这里不能声明为内联函数,因为只能有一个Printer
{
static Printer p;             //函数里的静态对象,非常常用和好用的技巧
return p;
}
```

2. 函数里的静态对象：
   - 基于种种原因，在某个类里声明的静态对象总是要被构造的（以及析构），即使它从来没有被用到。与此不同，在某个函数里声明的静态对象只在第一次执行到这个函数的时候才被创建，如果这个函数从来没有被调用，那么这个静态对象就不会被创建。（但是，每次这个函数被调用的时候，你要付出一定的代价来检测这个静态对象是否已经被创建）
   - 函数里的静态对象在这个函数第一次执行时一定被初始化了。但是对于类的静态对象（或者全局的静态对象），C++只保证在一个特定的编译单元内（指产生同一个目标文件obj的那些代码）的静态对象的初始化顺序，但是没有就不同编译单元的静态对象的初始化顺序做出任何规定。
3. 不要创建包含静态局部对象的非成员内联函数。（这里的therPrinter()）
4. 构造函数为private的类不能作为基类使用，也不能被嵌入到其它对象中。因为派生类的构造函数会调用基类的构造函数。
5. 构造函数被在三种情况下被调用：
   1：普通的调用，构造一个新对象；2：派生类的调用；3：包含类的调用
6. 匿名枚举类型：匿名枚举类型的一个常见应用是作为定义整数常量的另一种方式。例如：
   enum {feetPerYard = 3, inchesPerFoot = 12, yardsPerMiles= 1760 };
   因为以前C++中，静态成员要在类的外部指定初始值，这时就可以用匿名枚举这样的技巧来实现指定静态成员的初始值。新C++标准中已经可以为静态常量指定初始值。

19. 要求或者禁止对象分配在堆上
    1. 要求对象分配在堆上

```
class UPNumber
{
public:
void destroy() const {delete this}
private:
~UPNumber();
};
```

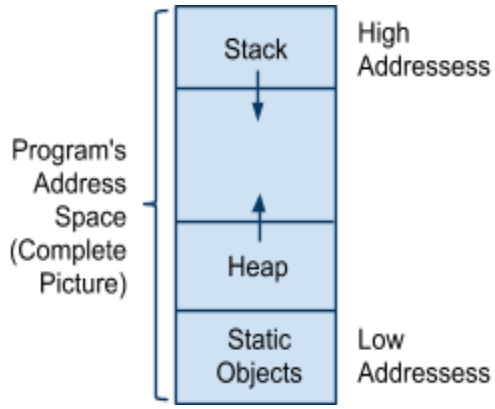
```
UPNumber n;                    //非法，析构函数为私有的
UPNumber* p = new UPNumber;
p->destroy();                  //通过一个公开方法来调用析构函数
```

2. 判断对象是否在堆上

   实际上，没有任何方法能够完全准确的判断一个对象是否在堆上。一般来说，不要试图去做这个努力，如果实在需要，再回来参考这一个条款吧。

3. 各种对象在内存中分配的位置

   静态对象不仅包括那些声明为static的对象，还包括在全局命名空间的对象。这些对象必定要分配在某个地方，而这个地方不是栈上也不是堆上。



4. 禁止对象分配在堆上

   简单的将operator new设置为私有，就可以应对大部分的情况，当然还有分配类数组的operator new[]

```
class UPNumber
{
private:
static void* operator new(size_t size);
static void* operator new[](size_t size);
static void operator delete(void* ptr);
static void operator delete[](void* ptr);
};
```

   但是，这种方法没办法禁止UPNumber作为基类被分配在堆上，正如无法判断一个对象是否在堆上一样，也没有办法完全的禁止对象分配在堆上。

20. 智能指针

1. auto_ptr在作用域结束时调用delete，所以一定不要用auto_ptr指向栈对象。
2. 两次删除对象的结果是未定义的（通常是灾难的）

3. 对于实现了operator()成员的对象来说：
   pt->displayEditDialog() 等价于 (pt.operator())->displayEditDialog();
4. 没有完美的方法来测试智能指针是否为空，折衷的办法是重载operator!
5. 如果允许客户直接对哑指针进行操作，几乎肯定会破坏引用计数的数据结构，从而导致计数失败。智能指针只能说是一种约定。而且，绝对不要提供智能指针到哑指针的隐式类型转换。
6. 编译器一次只能调用一个用户自定义的类型转换函数，这点很重要。
7. 成员函数模板：displayAndPlay(funMusic, 10);funMusic对象的类型是SmartPtr<Cassette>。函数displayAndPlay期望的参数是SmartPtr<MusicProduct>地对象。编译器侦测到类型不匹配，于是寻找把funMusic转换成SmartPtr<MusicProduct>对象的方法。它在SmartPtr<MusicProduct>类里寻找带有SmartPtr<Cassette>类型参数的单参数构造函数（参见条款M5），但是没有找到。然后它们又寻找成员函数模板，以实例化产生这样的函数。它们在SmartPtr<Cassette>发现了模板，把newType绑定到MusicProduct上，生成了所需的函数。实例化函数，生成这样的代码：
   SmartPtr<Cassette>:: operator SmartPtr<MusicProduct>()
   {
   return SmartPtr<MusicProduct>(pointee);
   }

   这种先后顺序相当麻烦，但是应该深入的了解，才能更好的掌握C++。
8. 智能指针与const
   对于智能指针来说，const只能放在一个地方，并只能对指针本身起作用，而不能对于指针所指向的对象起作用：const SmartPtr<CD> p = &goodCD; //p is a const smart ptr
   //to a non-const CD object
   不过，这很容易用指向const CD的智能指针来补救。
   SmartPtr<CD> p;
   SmartPtr<const CD> p;
   const SmartPtr<CD> p;
   const SmartPtr<const CD> p;
   但是，这样的non-const指针不能转化为const指针，这与普通指针不同。用成员函数模板的方法可以解决，如果需要的话，再回头来研究。

21. 引用计数
    1. C++标准库的string类型使用了引用计数功能。从非const版本的operator[]返回的引用，在调用下一个可能会对字符串进行改动的函数之前，都是有效的。在这之后，如果使用这个引用（或者它所指向的字符）产生的结果是不确定的。

2. 引用计数是一种优化技巧，这种技巧通常假设对象共享某些值。使用引用计数在很多时候都是行不通的，比如有些数据结构会自我引用或循环依赖。这样的数据结构很可能导致鼓励的自引用对象，即没有被别人引用，而其引用计数也不为0。（GC能处理这样的问题）

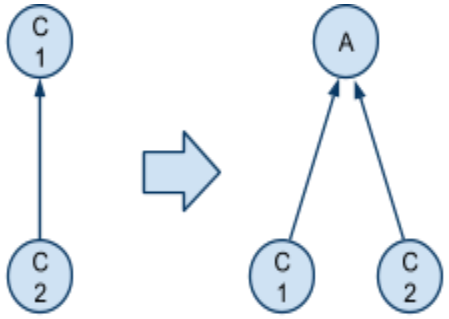
22. 代理类
    1. 代理类一般在其它的类中定义。
    2. 左值出现在赋值运算的左边，右值出现在赋值运算的右边。（这不是精确的定义，有待扩展）
    3. 代理类的一个非常重要的用途就是区分operator[]的读与写操作。

23. 基于多个对象的虚函数
    1. 处理这种问题，使用非成员函数是一个简单且大部分时候可行的方案。
    2. 无名命名空间中所有的东西都是当前编译单元（本质上说是当前文件）私有的--有点像在文件范围内被声明为static的函数一样。有了命名空间，文件范围的static已经不造成使用了，只要编译器支持，应该尽快转习惯用无名的命名空间。

24. 在将来时态下开发程序
    1. 一种好的作法是：用C++语言自己来表达软件设计上的约束条件，而不是用注释或文档。
    2. 处理每一个类的赋值和拷贝构造函数！
    3. 尽量与内建类型保持一致，参考int类型。
    4. 尽可能的使用无名的命名空间和文件内的静态对象或函数。
    5. 基类应该都有虚析构函数。如果一个公有基类没有虚析构函数，所有的派生类及其成员都不应该有析构函数；如果多重继承体系中某个类有析构函数，那么每一个基类都应该有虚析构函数。

25. 将非尾端类设计为抽象类
    1. 具体类继承自具体类会有指针赋值的问题！这点很重要，要永远记住：将非尾端类设计为抽象类
    2. 抽象类大部分时候是有一个纯虚的析构函数。

       绝大部分纯虚函数都没有实现，但是纯虚析构函数是个特例。它们必须被实现，因为它们在派生类析构函数调用时也被调用。实现纯虚析构函数是必须的！
    3. 提取公共的抽象基类



修改继承体制以消除具体类继承自具体类

4. 总结：一般的规则是：非尾端类应该是抽象类。在处理外来的类库时，你可能需要违背这个规则；但对于你能控制的代码，遵守它可以提高程序的可靠性、健壮性、可读性、可扩展性。

26. 理解如何在同一程序中混合使用C++和C

1. 重载不兼容于绝大部分链接程序，因为链接程序通常无法分辨同名的函数。名称改编是对链接程序的妥协，尤其是，链接程序通常要求函数名必须独一无二。
2. extern "C" 指示是完整意思是：禁用名称改编！

   extern "C"{

   ...

   }
3. 对于C++来说，在main函数执行前和执行后都有大量的代码要执行。尤其是，静态的类对象和定义在全局范围的，某个命名空间中的或文件范围内的对象的构造函数通常在main被执行之前就被调用，这个过程称为静态变量初始化。类似地，通过静态初始化产生的对象也要在静态析构函数过程中调用其析构函数，这个过程通过发生在main执行结束之后。

   所以，只要你的程序有部分是C++写的，你就尽量来用C++写程序的main函数。否则，就没有可移植的办法确保静态对象的构造函数和析构函数一定会被调用。
4. 总用delete释放new分配的内存；总用free释放malloc分配的内存。
5. 将在两种语言间传递的东西限制在用C编译的数据结构的范围内；这些结构的C++版本可以包含非虚成员函数。

27. 让自己熟悉C++语言标准

- 标准库中的string类，实现上没有这个类，只不过有一个叫作basic_string的模板类来代表字符序列，这个模板接受组成这个序列的字符的类型作为参数。

  通过认为string类是由basic_string<char>实例化而来的

  typedef basic_string<char> string;

  这个模板能够表示由char、wide char、Unicode char等组成的字符串。

完结