Exceptional C++ 读书笔记

**1 泛型程序设计与C++标准库（没看）**

***Item 1 Iterator:***1)注意当前迭代器是否有效，如果无效则解引用产生程序错误；  
2)注意当前迭代器生命期，某些容器经过某些操作后将重新分配内部存储空间，则当前迭代器无效;  
3)有效范围, 类似find(first, last, value)时, 迭代器first 必须在last之前,必须保证指向同一个容器;  
***Item 2, 3: Case-Insensitive String***

1)关于std::string: std::string实际是  template<class charT, class traits = char\_traits<charT>, class Allocator = allocator<charT> >  class basic\_string;              typedef basic\_string<char> string;

要实现case-insensitive string需要改变string的compare方式,因为char\_traits<char>类提供compare等方式,所以就需要对char\_traits<char>作相应的变动：char\_traits::eq()和lt()提供相等和小于的对比方式,compare()和find()提供对比和搜索字符串序列的功能;更改这四个成员函数  就可以实现case-insensitive string;

***Item 4, 5 Max Reusable Generic Containers***

***Item 6, 7 Temporary Objects***1) a: 用传递const T& 代替传值;     b: 用预先创建的对象代替不需要的重复创建的对象,类似(container.end()每次被调用将返回一个临时的对象,这是不必要的);     c: 用++T 代替 T++ (因为T++ 一般先制造一个临时对象,然后更新本身,最后返回临时对象);         d: 用++T来实现T++;     e: 避免做无用的隐式转换;  
2) 注意对象生命期,绝不准让函数返回一个函数内部的auto变量的指针或者引用;

 **2 异常安全性的问题和技术**

***Item 8 -- 17 Writing Exception-Safe Code***

**Item 8::** 1)异常中立：如果函数不打算处理（转换或者吸收）所遇到的或者即将抛出的异常，应该他们抛给函数调用者去处理，而自身不会出现资源泄漏和不完整问题；  
2)异常安全：当异常出现时，让资源正确的被释放，数据处于一个稳定的状态；能够继续正确运行；

3)“通过异常退出构造函数”的时候，该对象就没有被构造出完整的对象，也就是说这个对象的生命周期没有出现过，所以也就不需要进行释放;

4)绝不能让异常从destructor或者被重载的operator delete[],operator delete()中抛出,即：声明并确保throw()

Void operator delete[](void\*) throw();              void operator delete[](void\*, size\_t) throw();

**Item 9::**1)copy constructor函数，不需要考虑rhs == \*this的问题，因为这时候this还不存在；

2)copy构造和copy赋值等相关的需要产生新资源或者修改状态的函数，应该遵循如下的copy-swap idiom（ecpp说过）：

**在每个函数中，把所有可能抛出异常的代码单独放到一起（或者成立一个新的函数），并对这些代码进行安全处理。然后，当你确认这些代码所进行的工作都已经完成的时候，才可以使用不会抛出异常的操作来修改（清理）程序的状态。**

**Item 10::**1)对于stack的push，使用copy-swap idiom就可以很好的实现功能；

2)对于pop，内部看起来没有问题，但是当客户调用赋值的时候，在从函数返回临时对象到生成一个新的对象过程中，可能会异常（比如，copy构造函数异常），这时候，stack的顶端元素内容会永远被丢失了

3)但是pop的根本原因在于一个函数实现了两个功能，所以让每一个代码片段（类，模块，函数）负责单一的任务，即：STLstack中的pop和top是分离的。 **High cohesion, low coupling**. 如果不分离，就很难做到强安全性。

**Item 11::**1)exception-safe基本准则:

A:基本保证: 当异常抛出时不产生资源泄漏；该对象是可析构和可用的；对象处于一个一致的状态，但这种状态并不一定是可预测的；B:强保证：如果某个操作因为异常的抛出而终止，则对象当前状态不应改变；commit-or-rollback；  
C:不抛出异常保证：约定并保证这个函数绝不抛出异常

**Item 12-> Item14::**1)将资源管理单独的封装在StackImpl中。**Pimpl**惯用法：如果创建“编译器防火墙”将调用代码和类的私有部分完全隔离是有意义的，就应该使用Pimpl惯用法：将它们隐藏在一个不透明的指针后面（和智能指针结合则更好），详情参见Ch4

2)用资源申请既初始化(Resource Acquisition Is Initialization (RAII))来隔离类的实现和资源的管理与所有权，智能指针  
**Item 15::**1)多用聚合,少用继承,包括这种pivate继承--因为这将导致在constructor中无法控制base class

2)什么时候应当用private继承代替聚合：  
A:需要存取某个类的protected成员;           B:需要覆写虚函数;         C:base对象需要在其他类对象建构之前建构

**Item 16-> item17::**1)析构函数中决不允许抛出异常: A:对象本身所申请的的资源不能完全释放；  
b：考虑T t[] = new [20];的情况；析构某个T时抛出异常,则会产生未定义情况

***Item 18, 19 Code Complexities***

1)提供强异常安全需要损失一定程度的性能;    2)如果一个函数存在几个不相关的副作用,那么它不可能实现异常安全, 否则应当把这几个函数分割为不同的函数;        3)不是所有的函数都需要强异常安全;

 **3 类的设计与继承**

***Item 20 类的编写技巧***1) 设计时首先考虑标准库是否存在相关的类；

2) 用explicit constructor代替隐式转换。所以：构造函数应该避免隐式类型转换，会隐含产生临时变量；用explicit  
3)如果提供一个常规的operator op(类似+)，那么应该同样提供一个operator op=(+=)。operator op(+ - \* /)不应该改变对象值, 它应当返回一个 + 操作后的临时对象；operator op=返回对象的引用；  
4): 多用a op= b 代替 a = a op b； 一般 const T operator op(T, T)产生一个临时变量；  
**5) =，()，[]和->必须是成员函数，+=，-=，/=，\*=（等等）是成员函数，new、delete、delete[]是static 成员函数；流<<和 >>, 需要左参数进行类型转换的，可以是非成员函数或者friend函数；可以用公用接口实现的，用非成员函数**

6) ++T 应当返回一个引用， T++应当返回一个const T，以防止 T++++这种情况发生；T++应当由++T来实现:

**前缀： T& operator++();              后缀：Const T operator++(int);**7) <<和>>应该返回流对象的引用，以方便链式操作：cout << a << b;

8)避免使用前导下划线来命名，前导下划线是留给编译器使用的

***Item 21对虚函数 进行重载overload***1) 除非确定某个类不被派生,否则都应当提供一个虚函数;  
2) 关于 overload, override和 hide  
a: overload重载：在相同作用域中，提供相同名称但是不同参数的函数， 编译时由编译器决定哪个是最匹配的函数；仅仅返回值不一样的两个函数不算重载；  
b: override覆盖：在派生类提供相同名称且相同参数不同实现的虚函数，动态调用时将从派生类对象的基类指针调用这个函数；子类的函数仅仅返回值与基类不一样，编译错误；  
c: hide隐藏：内层作用域对外层作用域(基类，被嵌套类，名称空间等）的同名函数隐藏，无论返回值和参数是否一样  
**注意：在这三种情况中，都没有对返回值进行格外的控制和要求。**

3) 当派生类提供一个和基类的某个成员函数同名的函数时,如果你不想隐藏基类的同名函数,请在派生类声明基类的同名函数: using Base::f;  
4) 永远不要在派生类override virtual function中改变基类函数的默认参数;  
5) 要非常注意如下两个情况(Base\* p = new Derived();)：

|  |  |
| --- | --- |
| Class Base{  **//这是P的静态类型** | Class Derived: public Base{ **//这是P的动态类型** |
| Virtual void f(**double**) {…};            // item (a)  Virtual void g(int n = **10**) {…};          // item (b) | Void f(**OtherClass**){…}; // OtherClass can accept double  Void g(int n = **20**){…} |

item (a): p->f(1.0)会调用Base::f(),因为重载是用于静态类型范围内，不能在范围之外找；

item (b): p->g()会调用Derived::g(),默认参数为Base里面的10。虚函数找动态类型，默认值找静态类型；二者结合

***Item 22, 23 Class Relationships***1): 什么时候不应该选用public 继承：a)如果某个类未提供虚函数，这个类本身不打算实施多态；  
b)未提供protected 成员       c)类型不是IS-A, WORKS-LIKE-A or USABLE-AS-A;           d)派生类只用了基类的public 成员；

2):只有当需要存取某个类protected 成员或者override 虚函数的时候才应该选用private继承  
3): 避免提供public virtual function, 使用template method 设计模式代替:细节通过virtual function 留给派生类来实现。这个也叫NVI（Non-virtual-interface） idiom：public接口+private virtual  
4): 应用compiler-firewall 方法隐藏实现细节,使用一个不透明的指针来存private 成员，包括状态变量和成员函数：

**箴言：大多数的问题都可以通过一个间接层来解决**

***Item 24 Uses and Abuses of Inheritance***1)多用聚合composition,当关系模型为 IS-IMPLEMENTED-IN-TERMS-OF时，请多考虑聚合，最好不用继承

2)使用private继承的条件:   a)当需要override virtual function时候;  
b): 需要存取基类protected成员时候         c)需要保证初始化顺序(基类必须建构在派生类之前,析构在派生类之后);  
d): 需要某种多态的访问控制时；                e)EBO(Empty Base Optimization)空基类优化，不占空间

***Item 25 Object-Oriented Programming***C++标准化工作对C++的最大贡献是：**强大的抽象机制降低了软件的复杂度**

**4 编译器防火墙和Pimpl惯用法（Compiler firewall and Pimpl idiom）**

***Item 26-28 Minimizing Compile-Time Dependencies***1)最小化编译依赖方法：a)决不#include无用的头文件；         b): 当前向声明某个stream时候用<iosfwd>足够了；

c)如果composition足够的话,决不能用继承;

2)PImpl惯用法，优点：a)私有部分所使用的类型定义，只有在类的实现中才会需要，在客户代码中不需要；

b)类的实现是可以修改的，修改之后，客户代码无需重新编译

5)PImpl的实现：   clas Map { private: struct MapImpl; MapImpl \*pimpl\_; };      // MapImpl的前置声明

struct Map::MapImpl{ … }       // MapImpl的真正定义

***Item 29 Compilation Firewalls(编译器防火墙)***1)**virtual、protected**（因为他们需要做多态和override）成员除外，其他的私有成员都放到Pimpl类中，因为客户不应当看到这些private成员；把那些只有私有成员才使用的非私有成员也放入PImpl中，对于这些非私有成员，需要在可见类中进行函数的转调；Pimpl类中某些函数可能需要可见类中某些成员的支持,所以需要回调指针指向可见类  
2) 或者，把可见类变成简单的转调类，这样避免了在Pimpl类中使用回调指针

***Item 30 The "Fast Pimpl" Idiom***  一般不使用

 **4 名字查找、命名空间和接口规则**

***Item 31 - 34 Name Lookup and Interface Principle*Item 31::**1)namespace A { class X {};  void f(X x) {} }  
namespace B { void f(A::X x) {} void g(A:: xx) { f(xx); } }  
这将导致B::g出现二义性而无法编译；原因在于g涉及A::X，所以编译器将把名称空间A和B内的函数都考虑进来，此时出现两个参数为A::X的函数f。如果希望成功，必须在调用前面加上合适的名字空间限定符：A::或者B::

2)**Koenig查找规则(简化版): 如果在声明函数的参数时使用了一个类(比如A::X)，那么查找正确匹配的函数的时候，编译器将会在包含参数类型的命名空间中也进行函数名字的匹配，即使没有使用using namespace A.**

3)简单的结论：名称空间不像想象的那样完全隔离函数声明，它只能属于部分独立。名称空间A中声明的函数将会影响到名称空间B中代码的含义，而A和B是完全独立的。虽然B中仅仅使用了A中的一个类型，连using也没使用。

**Item 32 -> 33:: 1)接口规则：对于类X来说，如果函数，包自由函数，能够满足如下条件：a)使用了X；b)与X同时定义(同一头文件或同一名称空间)；那么这些函数在逻辑上都是X的一部分，他们将构成X接口的一部分。**

2)不满足1)的条件的，比如不是在同一头文件或名称空间定义的，只能算作客户函数，不是接口一部分；

3)类：描述的是一组数据，以及操作这组数据的函数。所以X成员函数、与X同时定义的自由函数都是类的一部分

class X{...};                    ostream& operator<<(ostream&, const X&);      //<<与X在同一个头文件，属于X接口的一部分  
4)命名空间A中同时定义了类X和使用X的自由函数f，那么编译器把这二者建立”强关系”。不过，**成员函数与类的关系要强于非成员与类的关系**：

|  |  |
| --- | --- |
| 类与自由函数 | 类与成员函数 |
| Nameapsce A{ Class X{};    **void f(X);//f为新增** }  **Namespace** B { void f(A::X); void g(A::X p){ f(p);} } | Nameapsce A{ Class X{};    **void f(X);//f为新增** }  **Class** B { void f(A::X); void g(A::X p){ f(p);} } |
| g()调用产生二义性：A::f和B::f都是候选，且平等竞争 | g调用正常：B::f强于A::f |

5)根据4)=>在名称空间A内增加函数，将会破坏名称空间外B的代码，即使A和B是完全独立的,连using也没使用

**Item 34::**1)关于名称隐藏：struct B {   int g(int);  };      struct D : public B    {  private:   int g(std::string, bool);  };

D d;  int i;  d.g(i); //错误，提示应当提供两个参数；  
a)派生类的函数g，会自动隐藏所有在直接基类和间接基类中名字相同的函数，不论参数是否一样；

b) 编译器查找匹配函数名称进行调用时，查找规则：**从最里层作用域开始(struct D)，在当前范围内的名称匹配函数，然后列出一个编译器所能找到的函数名称相同的函数的列表，无论存取权限和所携带参数是否正确；如果失败，将向外层继续查找，如果编译器找到一个或者多个候选的函数，编译器将停止查找；然后对函数进行重载和访问权限分析。**  
c)规则的好处：编译器会优先选择参数类型近似匹配的内层函数，而不是完全匹配的外层函数

d)这个问题的解决方法：  d.B::g(i);//编译器自然会考虑B的范围；或者，using B::g;  
2)派生类的名字隐藏问题，在嵌套namespace中也是同样存在的

3)koenig查找规则告诉我们：查找的时候会在当前作用域和参数类型所在的作用域中寻找，然后再向外层查找。得出推论：写一个类在名称空间N中，那么应将所有的辅助函数和操作符都写在N中；即：类和它的所有接口放在一起。

**6 内存管理**

***Item 35, 36 Memory Management*Item 35::**1)[常量数据（const data）区]存储字符串等在编译期间就确定的值。实例化的对象不能存在于这个区域中。

2)[栈（stack）区] 存储自动变量。分配操作要比动态存储区（堆或自由存储区）快，因为栈区的分配只涉及到一个指针的递增，而动态存储区的分配涉及到较为复杂的管理机制。栈区中，内存一  旦被分配，对象就立即被构造好了；对象一旦被销毁，分配的内存也立即被收回。

3)[自由存储区（free store）]是C++两个动态内存区域之一，使用new和delete来予以分配和释放。在自由存储区中，对象的生存周期可以比存放它的内存区的生存周期短；这也就是说，我们可以获得一片内存区而不用马上对其进行初始  化；同时，在对象被销毁之后，也不用马上收回其占用的内存区。在对象被销毁而其占用的内存区还未被收回的这段时间内，我们可以  通过void\*型的指针访问这片区域，但是其原始对象的非静态成员以及成员函数（即使我们知道了它们的地址）都不能被访问或者操纵。  
4)[堆（heap）区]是另一个动态存储区域，使用malloc、free以及这些函数的其他形式来进行分配和回收。虽然在特定的编译器里缺省的全局运算符new和delete也许会按照malloc和free的方式来被实现，但是堆与自由存储区是不同的——在某一个区域内被分配的内存不可能在另一个区域内被安全的回收。堆中被分配的内存一般用于存放在使用new的构造过程中和显式的析构过程中涉及到的类对象。堆中对象的生存周期与自由存储区中的类似。

5)[全局/静态区（Global/Static）] 全局的或静态的变量和对象所占用的内存区域在程序启动的时候才被分配，而且可能直到程序开始执行的时候才被初始化。比如，函数中的静态变量就是在程序第一次执行到定义该变量的代码时才被初始化的。对那些跨越了编译单元的全局变量进行初始化操作的顺序是没有被明确定义的，因而需要特别注意管理全局对象（包括静态类对象）之间的依赖关系。

6) [堆（heap）vs.自由存储区（free store）]  
在C++标准中，到底new和delete是按照malloc和free来实现的，或者反过来malloc和free是按照new和delete来实现的，并没有规定。简单地说，这两种内存区域运作方式不一样，访问方式也不一样！

**Item 36::** 1)delete和delete[]的标准形式如下。其中：size\_t可有可无；static可有可无，因为必须是static的。

static void operator delete(void\*) throw();                        static void operator delete[](void\*) throw();

static void operator delete(void\*, size\_t) throw();    static void operator delete[](void\*, size\_t) throw();

2)    class B{ static void operator delete(void\*) throw(){}; static void operator delete[](void\*) throw(){}; };

class D: B{static void operator delete(void\*) throw(); static void operator delete[](void\*) throw(); };

因为operator delete是static，所以不可能是virtual。但是调用delete的时候，其行为却能够做到多态。

B\* pb1 = new D(); delete pb1;                   D\* pb2 = new D(); delete pb2;         D\* pb4 = new D[3]; delete pb4;

B\* pb3 = new D[3]; delete pb3;

前三种都能调用D::operator delete，并从D::~D开始析构; 第四种是行为未定义，因为不能以多态处理数组

3)typedef void (B::\*fpn)(void\*); fpn的要求是B::f，即参数是：(this, void\*)，所以不能用B::static函数。

4)定义new和new[]的时候一定要定义参数等完全匹配的delete和delete[]，否则很容易做出资源泄漏

***Item 37: auto\_ptr***1)在各种场合，传递数据的时候最好都使用auto\_ptr，而不要使用裸指针；

2)auto\_ptr的各个副本直接不是对等的，不能完全满足容器的模板参数类型需求，所以不能放到标准容器中；

3)由于2)等原因，请使用const auto\_ptr惯用法，禁止对auto\_ptr的内容进行修改和copy等；

**7 误区、陷阱和错误的惯用法**

***Item 38 Object Identity***

1)拷贝赋值操作符应当使用copy-and-swap  idiom，而无需做自身赋值检查；检查自身赋值是一个优化;

2)copy构造函数中进行自身赋值检查时没有意义的，因为这时候对象还没有生命。但是一些代码确实合法的：

T t;   new (&t) T(t);//placement new, t为copy构造的参数；      或者

T t = t;      //t没有调用默认构造函数，而是被一个没有初始化的另一个t拷贝构造了

***Item 39 Automatic Conversions***1)避免使用隐式转换操作符:  a: 隐式转换会影响重载决议;  b: 隐式转换会让错误的代码安静的编译通过;

***Item 40,41 Object Lifetimes***1): void f()  {         T t(1);       t.~T();//**t析构，内存不释放**        new (&t) T(2);          };//**t析构第二次，内存也释放**

这段代码是合理合法的，但是不安全和不推荐的。尤其是：**t被构造一次，被析构二次**，非常危险。   
2)如果T是一个基类，并有虚拟析构函数，那么会造成slice切片问题

**8 其他主题**

***Item 42: Variable Initialization -- Or is it?***

1) 关于 T u; T t = u;编译器会调用 copy构造，而不是调用默认构造之后调用operator=()，没有赋值操作

2) T t() 只是声明了一个函数，并没有创建t对象。          3) 用 T t(u);代替 T t = u;

***Item 43 Const-Correctiness;***

1)const T&  t2 = t;是const引用；           T& const t2 = t; 此处的const会被忽略，无效

2)void f(int);    void f(const int);            // 他们不是重载，只是一个函数

**void f(int&);    void f(const int&); // 他们是重载，不是一个函数**

3)避免在某个函数参数使用 const by value, 因为毕竟要生成临时对象；  
4)函数返回非内部类型的值,优先考虑const value；如果是int，考虑返回int，因为int类型值本身就是右值rvalue  
5) mutable类型的变量可以在const成员函数中修改

***Item 44 Casts***

1)避免用const\_cast消除const，请优先使用mutable。

2)Const\_cast只能处理指针和引用，不能处理对象；他能在const和非const之间转换，并非只能去掉const属性

3)如果在一个类结构里没有任何的virtual函数，dynamic\_cast不会生效。dynamic\_cast可以在平行的类间转化，比如：B:A; C:A; D:B,C. 可以在B和C直接转换。但不能在私有继承类之间转换；最好不要向下转型dynamic\_cast

***Item 45: bool***可以模拟bool的几种方法，但都有问题，所以必须有bool类型的出现  
1)typedef int bool;   问题：f(int)和f(bool)会认为是重复定义函数；

2)#define boo lint     问题：和typedef问题一样，甚至会更多；

3)enum bool{false, true};                   问题：不能处理bool b; b = ( I == j);

4)class bool;                       问题： operator int();    operator void\*(); 有问题

***Item 46 Forwarding Functions;***1): 多考虑传递对象的const &；2)如果不是性能上绝对需要，避免inline；  
***Item 47 Control Flow***1)避免使用全局或静态变量；它们的初始化顺序是未定义的；

2)基类的初始化顺序是依据类定义时从左到右顺序的，所以基类构造函数在初始化器列表中的顺序与类定义时一致；

3)将类定义中成员的顺序与初始化器列表中初始化的顺序一致

4)绝不能写依赖函数参数赋值顺序的代码