

# Lecture Notes on C++ Multi-Paradigm Programming

*Bachelor of Software Engineering, Spring 2013*

**Wan Hai**

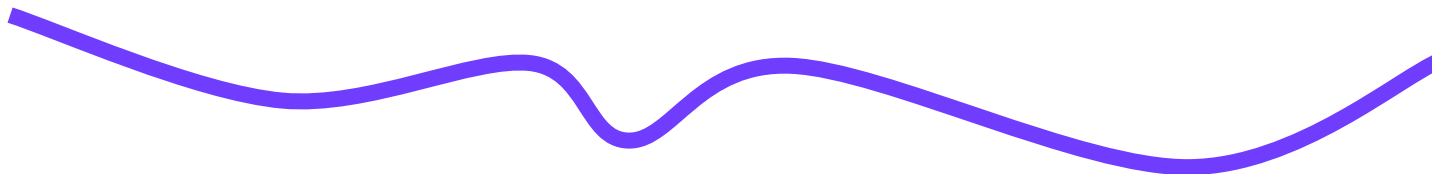
*whwanhai@163.com*

*13512768378*


**Software School, Sun Yat-sen University, GZ**



# Effective C++



# 导读(术语解释)

- **STL** 标准模板库(Standard Template Libry),是C++标准程序库的一个部分, 致力于容器(如 `vector`,`list`,`set`,`map`)、迭代器(如 `vector<int>::iterator`,`set<string>::iterator`等)、算法(如 `for_each`,`find`,`sort`等等)及相关机能。
  - **TR1** (“Technical Report 1”)是一份规范, 描述加入C++标准程序库的诸多新机能。所有的TR1组件都被置于命名空间`tr1`内, 后者嵌套于命名空间`std`内。
  - **Boost**是个组织, 也是一个网站(<http://boost.org>)。
  - **Explicit**用来阻止编译器执行隐式类型转换, 但可以进行显示类型转换。
- 

# 第一章：让自己习惯C++

- 条款01：视C++为一个语言集合(**View C++ as a federation of languages**)
  - C。说到底C++仍是以C为基础。区块(blocks)、语句(statements)、预处理器(preprocessor)、内置数据类型(built-in data types)、数组(arrays)、指针(pointers)等统统来自C。
  - Object-Oriented C++。Classes(包括构造函数和析构函数)，封装(encapsulation)、继承(inheritance)、多态(polymorphism)、virtual函数.....等等。
  - Template C++。泛型编程部分。
  - STL。
  - C++ 高效编程守则视状况而变化，取决于你使用C++的哪一部分。



- 条款02：尽量以**const,enum,inline**替换 **#define**
  - Prefer consts enums and inline to #defines
    - #define 可能并不进入符号表(symbol table)。
    - Const:
      - 常量指针(constant pointers)
        - 例：不变的char\* - based 字符串（最好使用string）
          - » char\* const authorName = "Scott Meyers";
      - Class专属常量
        - 为了确保此常量至多只有一份实体，必须让它成为一个static成员：
          - » Class GamePlayer {
          - » Private:
          - » static const int NumTurns = 5 ; //常量声明
          - » };
        - 注意：只要不取它的地址，你可以声明并使用它们而无需提供定义。  
如果你取某个class专属常量的地址或者编译器报错，你就必须提供定义。  
如：const int GamePlayer::NumTurns;  
由于class常量已在声明时获得初值(static)，因此定义时不用再设初值。  
如果编译器还不允许这样，可以使用在声明外面定义赋予初值。  
如：const int GamePlayer::NumTurns = 5;  
如果还是报错，可以选择用 enum 关键字。  
如：enum{ NumTurns = 5};...
    - Template Inline代替宏
    - 对于形似函数的宏(macros)，最好改用inline函数替换#defines.
    - 对于单纯常量，最好以const对象或enums替换#defines.

- 条款03：尽可能使用**const**

- Use const whenever possible

- `std::vector<int> vec ;`

- `const std::vector<int>::iterator  
iter=vec.begin() ; // iter 的作用像个T* const`

- `++ iter ; //错误！ iter 是const`

- `std::vector<int>::const iterator  
clter=vec.begin() ;`

- `*clter = 10 ; // 错误！ *clter 是const`

- `++clter ; // 正确。`

- 条款03：尽可能使用**const**
  - Use const whenever possible
  - 如果关键字**const**出现在星号左边，表示被指类型是常量；如果出现在星号右边，表示指针自身是常量；如果在星号两边，表示被指类型和指针两者都是常量。
  - **Const**最具有代表性的是函数声明时的应用。**Const**可以和函数返回值、各参数、函数自身产生关联。
  - **Const** 成员函数
    - 1) 它们使**class**接口比较容易理解。
    - 2) 它们使“操作**const**对象”成为可能。
  - 如果函数的返回类型是个**内置类型**，那么改掉函数返回值就不可能合法。
  - **Bitwise constness(physical constness)**
    - 成员函数只有在**不更改对象的任何成员变量**(**static**除外)时才可以说是**const**。也就是说它不更改对象内任何一个位(bit)。
  - **Logical constness**
    - 一个**const**成员函数可以修改它所处理的对象内某些**bits**，但**只有在客户端侦测不出的情况下**才可以。
      - **Mutable**（可变的）关键字可以释放掉**non-static**成员变量的**bitwise constness**约束；

- 在const和non-const成员函数中避免重复
  - Const成员函数调用non-const成员函数是一种错误行为，因为对象有可能因此被改动。
  - **Const\_cast**
    - 用法: **const\_cast**<type\_id>(expression)
      - 该运算符用来修改类型的const或volatile属性。
      - 常量指针被转化成非常量指针，并且仍然指向原来的对象；
      - 常量引用被转换成非常量引用，并且仍然指向原来的对象；
      - 常量对象被转换成非常量对象；
  - **Static\_cast**
    - 用法: **static\_cast**<type\_id>(expression)
      - 该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。
      - C++ primer 里说明在进行隐式类型转换都用
        - » **Int** i = **Static\_cast**<int>f; // **float** f = 1.42f;



- 总结

- 将某些东西声明为`const`可帮助编译器侦测出错误的用法。`Const` 可被施加于任何作用域内的对象、函数参数、函数返回类型、成员函数本体。
- 编译器强制实施 `bitwise constness`，但你编写程序时应该使用“概念上的常量性”(`conceptual constness`)。
- 当`const` 和 `non-const` 成员函数有着实质等价的实现时，令 `non-const`版本调用 `const`版本可避免代码重复。



- 条款04：确定对象被使用前已被初始化
  - **Make sure that object are initialized before they're used.**
  - 读取未初始化的值会导致不明确的行为。
- 对象的初始化何时一定发生，何时不一定发生。
  - 对于无任何成员的内置类型，必须手工完成此事。
  - 对于内置类型以外的任何其他东西，初始化责任落在构造函数 (constructors)身上。确保每一个构造函数都将对象的每一个成员初始化。
  - **C++规定，对象的成员变量的初始化动作发生在进入构造函数本体之前。**
  - 构造函数的最佳写法是，使用 **member initialization list**（成员初始化表）如：
    - `ABEntry::ABEntry(char &name,char& address,list &phones)`
    - `: theName(name),theAddress(address),thePhones(phones)`
    - `{`
    - `– .....`
    - `}`

- 编译器会为用户自定义类型(**user-defined types**)之成员变量自动调用**default**构造函数 --- 如果那些成员变量在“成员初始化列表”中没有被指定初值的话。
- 成员变量是 **const** 或 **references**，它们就一定需要初值，不能被赋值。
- C++有着十分固定的“成员初始化次序”。**Base classes** 更早于其 **derived classes** 被初始化，而**class**的成员变量总是以其声明次序被初始化。
- **Static** 对象，其有效时间从被构造出来直到程序结束为止，因此 **stack**和**heap-based**对象被排除。
  - C++对于“定义于不同的编译单元内的**non-local static**对象”的初始化相对次序并无明确定义。
    - 小方法：将每一个**non-local static**对象放到自己的专属函数内（该对象在此函数内被声明为**static**），这些函数返回一个**reference**指向它所含的对象。

- 总结

- 为内置型对象进行手工初始化，因为C++不保证初始化它们。
- 构造函数最好使用成员初始化列表(member initialization list),而不要再构造函数本体内使用赋值操作(assignment).
- 为免除“跨编译单元之初始化次序”问题，请以local static对象替换non-local static对象。



## 第二章：构造、析构、赋值

- 条款05：了解C++默默编写并调用哪些函数

- Know what functions C++ silently writes and calls

- C++会为默认的空类(empty class)添加

- Default 默认构造函数
- Copy 构造函数
- 析构函数
- Copy assignment 复制赋值操作符

- 唯有这些函数被调用时，它们才会被编译器创建出来。

- `class Empty { };`

`class Empty {`

## 第二章：构造、析构、赋值

- 条款06：若不想使用编译器自动生成的函数，就该明确拒绝
  - **Explicitly disallow the use of compiler-generated functions you to not want.**
  - 方法：可以自定义编译器会默认生成的函数，手动定义成`private`;
    - 或者使用空基类(empty base class)声明空函数来继承。



- 条款07：为多态基类声明**Virtual**析构函数
  - **Declare destructors virtual in polymorphic base classes.**
  - C++明确指出，当**derived class**对象经由一个**base class**指针被删除，而该**base class**带着一个**non-virtual**析构函数，其结果未有定义---实际执行时通常发生的是对象的**derived**成分没被销毁。
    - 方法：给**base classes**定义一个 **virtual 析构函数**。
    - 任何**class**只要带有**virtual** 函数都几乎确定应该有一个**virtual**析构函数。
  - class AWOV {
  - public:
  - virtual ~AWOV() = 0;
  - };
  - AWOV::~~AWOV() { } //pure virtual析构函数的定义

- 条款07：为多态基类声明Virtual析构函数
  - **Declare destructors virtual in polymorphic base classes.**
  - 欲实现出virtual函数，对象必须携带某些信息，主要用来在运行期决定哪一个virtual函数被调用。这份信息通常由一个所谓vptr（virtual table pointer）指针指出。
  - 令class带一个pure virtual(纯虚)析构函数会导致abstract(抽象)classes ---也就是不能被实体化(instantiated)的class.
  - 总结
    - Polymorphic(带多态性质的)base classes 应该声明一个 virtual析构函数。如果 class带有任何virtual函数，它就应该拥有一个virtual析构函数。
    - Classes 的设计目的如果不是作为base classes使用，或不是为了具备多态性(polymorphic),就不该声明Virtual析构函数。



- 条款08：别让异常逃离析构函数

- Prevent exception from leaving destructors

- 当析构函数发生异常时，有以下2中解决办法

- 如果抛出异常就结束程序，通常通过调用 `abort`完成

- `DBConn::~DBConn()`

- {

- » `Try{ ;}`

- » `Catch(...){`

- » `std::abort();`

- » }

- }

- 吞下因调用析构函数而发生的异常；

- 总结

- 析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析构函数应该捕捉任何异常，然后吞下它们(不传播)或结束程序。

- 如果客户需要对某个操作函数运行期间抛出的异常做出反应，那么class应该提供一个普通函数(而非在析构函数中)执行该操作。

- 条款09：绝不在构造和析构过程中调用**Virtual**函数
  - **Never call virtual functions during construction or destruction**
- Base class构造期间 virtual 函数绝不会下降到 derived classes 阶层。（在base class构造期间，virtual函数不是virtual函数）



- 条款10: 令**operator =** 返回一个 **reference to \*this**
  - **Have assignment operator return a reference to \*this.**
- 关于赋值，你可以把它们写出连续形式：
  - `Int x,y,z;`
  - `X = y = z =15;`
- 同样的，赋值采用的是右结合方法，所有上述连续赋值解析为：
  - `X =(y = ( z = 15));`
- 为了实现连续赋值，赋值操作符必须返回一个**reference**指向操作符的左侧实参。

```
class Widget{
public:
    ...
    Widget & operator=(const Widget& rhs) //返回类型是个reference, 指向当前对象
    {
        ...
        return * this ; //返回左侧对象
    }
    ...
};
```

- 条款11：在**operator=** 中处理“自我赋值”
  - **Handle assignment to self in operator=.**
  - 确保当对象自我赋值时 **operator=** 有良好的行为。其中技术包括比较“来源对象”和“目标对象”的地址、精心周到的语句顺序、以及 **copy-and-swap**。
  - 确定任何函数如果操作一个以上的对象，而其中多个对象是同一个对象时，其行为仍然正确。

```
class Bitmap { ... };
class Widget {
    ...
private:
    Bitmap* pb; //指针，指向一个从heap分配而得的对象
};

Widget& Widget::operator=(const Widget& rhs) //一份不安全的operator=实现版本
{
    delete pb; //停止使用当前的bitmap,
    pb = new Bitmap(*rhs.pb); //使用rhs的bitmap副本（复件）。
    return *this; //见条款10。
}
```

- 条款11：在**operator=** 中处理“自我赋值”

```
Widget& Widget::operator=(const Widget& rhs) //一份不安全的operator=实现版本
{
    if( this == &rhs ) return *this; //证同测试 (identity test)。

    delete pb;
    pb = new Bitmap(*rhs.pb);
    return *this;
}
```

```
Widget& Widget::operator=(const Widget& rhs)
{
    Bitmap* pOrig = pb; //记住原先的pb
    pb = new Bitmap(*rhs.pb); //令pb指向*pb的一个复件 (副本)
    delete pOrig; //删除原先的pb
    return *this;
}
```



- 条款11：在**operator=** 中处理“自我赋值”

```
class Widget{  
    ...  
    void swap(Widget& rhs); //交换*this和rhs的数据; 详见条款29  
    ...  
};  
Widget& Widget::operator=(const Widget& rhs)  
{  
    Widget temp(rhs); //为rhs数据制作一份复制 (副本)  
    swap(temp); //将*this数据和上述复件的数据交换。  
    return *this;  
}
```

- 条款12：复制对象时勿忘其每一个成分
  - **Copy all parts of an object**
  - 当你编写一个copying函数，请确保(1)复制所有local成员变量，(2)调用所有base classes内的适当的copying函数。
  - Copying函数应该确保复制“对象内的所有成员变量”及“所有base class成员”。
  - 不要尝试以某个copying函数实现另一个copying函数。应该将共同功能放进第三个函数中，并由两个copying函数共同调用。



# 第三章：资源管理

- 所谓资源就是，一旦用了它，将来必须还给系统。**C++**程序中最常使用的资源就是动态分配内存(如果你分配内存却从来不曾归还它，会导致内存泄露)，但内存只是你必须管理的众多资源之一。
- 条款**13**:以对象管理资源
  - **Use object to manage resources.**
  - 把资源放进对象内，我们便可依赖**C++**的“析构函数自动调用机制”确保资源被释放。
  - **STL**标准程序库提供的**auto\_ptr**正是针对这种形式而设计的特制产品。**Auto\_ptr**是个“类指针(**pointer-like**)对象”，也就是“智能指针”，其析构函数自动对其所指对象调用**delete**。
    - **Std::auto\_ptr<Investment>plnv(CreateInvesment());**



- 上面的例子示范“以对象管理资源”的两个关键想法：
  - 获得资源后立刻放进管理对象(**managing object**)内。
  - 管理对象(**managing object**)运用析构函数确保资源被释放。
- 使用**auto\_ptr**有一个性质：若通过**copy**构造函数或**copy assignment**操作符复制它们，它们会变成**NULL**，而复制所得的指针将取得资源的唯一拥有权！
- **Auto\_ptr**的替代方案是“引用计数型智慧指针”（**reference-counting smart pointer; RCSP**）。**RCSP**也是个智能指针，持续追踪共有多少对象指向某笔资源，并在无人指向它时自动删除该资源。
  - **TR1**的**tr1::shared\_ptr**就是个**RCSP**，你可以这么写
    - `Std::tr1::shared_ptr<Investment>plnv(CreateInvestment());`
- **Auto\_ptr** 和 **shared\_ptr** 都是在其析构函数上调用**delete** ,而不是**delete []**动作。那意味着在动态分配而得的**array**上使用，是错误的。
- 总结
  - 为防止资源泄露，请使用**RAII**对象，它们在构造函数中获得资源并在析构函数中释放资源。
  - 两个常被使用的**RAII classes** 分别是**tr1::shared\_ptr** 和 **auto\_ptr**。

- 条款14：在资源管理类中小心**copying**行为
  - **Think carefully about copying behavior in resource-managing classes.**
  - **RAII守则：**资源在构造期获得，在析构期被释放；
  - 类似Mutex的互斥对象(mutex object)时，因为有lock,unlock两种状态，可以采用以下方法，确保释放：
    - 禁止复制。Auto\_ptr创建
    - 引用计数(reference-count)。Shared\_ptr创建
  - 总结
  - 复制**RAII**对象必须一并复制它所管理的资源，所以资源的**copying**行为决定**RAII**对象的**copying**行为。
  - 普遍而常见的**RAII class copying**行为是：抑制**copying**、施行引用计数法(**reference counting**)。不过其他行为也都可能被实现。

- 条款15：在资源管理类中提供对原始资源的访问
  - **Provide access to raw resources in resource-managing classes**
  - 所有智能指针，如`tr1::shared_ptr` 和 `auto_ptr` 也重载了指针取值 (`pointer dereferencing`)操作符(`operator->`)和(`operator*`),它们允许隐式转换至底部原始指针；
  - 总结
    - API 往往要求访问原始资源(`raw resources`)，所有每一个RAII class应该提供一个“取得其所管理之资源”的办法。
    - 对原始资源的访问可能经由显示转换或隐式转换。一般而言显示转换比较安全，但隐式转换对客户比较方便。

- **条款16：成对使用new 和 delete 时要采取相同形式**
  - **Use the same form in corresponding uses of new and delete.**
  - 当你使用new动态生成一个对象，有两件事发生：
    - 内存被分配
    - 针对此内存会有一个(或更多)构造函数被调用，然后内存才被释放(delete).
  - 尽量不要对数组形式做typedef动作。
  - 总结
    - 当你用new生成对象时，如果用new type-object[] ,则要使用 delete []type-object ， 否则使用 delete
- **条款17：以独立语句将newed对象置入智能指针**
  - **Store newed objects in smart pointers in standalone statements.**
  - 以独立语句将newed对象存储于（置入）智能指针内。如果不这样做，一旦异常被抛出，有可能导致难以察觉的资源泄露。

## 第四章：设计与声明

- 软件设计，是“令软件做出你希望它做的事情”的步骤和做法，通常以颇为一般性的构想开始，最终演变成十足的细节，以允许特殊接口(interface)的开发，这些接口而后必须转换为C++声明式。
- 条款18：让接口容易被正确使用，不易被误用
  - **Make interfaces easy to use correctly and hard to use incorrectly.**
  - 好的接口很容易被正确使用，不容易被误用。你应该在你的所有接口中努力达成这些性质。
  - “促进正确使用”的办法包括接口的一致性，以及与内置类型的行为兼容。
  - “阻止误用”的办法包括建立新类型、限制类型上的操作、束缚对象值，以及消除客户的资源管理责任。
  - `Tr1::shared_ptr`支持定制型删除器(custom deleter)。这可防范DLL问题，可被用来自动解除互斥锁(mutexes)

- 条款19: 设计 **class** 犹如设计 **type**
  - **Treat class design as type design.**
  - 设计高效类型(types)类(classes)的方法:
    - 新type的对象应该如何被创建和销毁?
      - 构造和析构函数以及内存分配和释放
    - 对象的初始化和对象的赋值该有什么样的差别?
      - 构造函数和赋值(**assignment**)操作符的行为
    - 新type的对象如果被passed by value (以值传递), 意味着什么?
      - **Copy**函数用来定义一个type的**passed by value**
    - 什么是新type 的“合法值”?
      - **<异常处理>**
    - 你的新type需要配合某个继承图系(inheritance graph)吗?
      - **<是否需要虚函数>**
    - 你的新type需要什么样的转换?
    - 什么样的操作符和函数对此新type而言是合理的?
    - 什么样的标准函数应该驳回?
    - 谁该取用新type的成员?
    - 什么是新type的“未声明接口”(undeclared interface)?
    - 你的新type有多么一般化?
    - 你真的需要一个新type吗?
  - **Class** 的设计就是**type**的设计。在定义一个新**type**之前, 请确定你已经考虑过本条款覆盖的所有主题。

- **条款20：宁以pass-by-reference-to-const替换pass-by-value**
  - **Perfer pass by reference to const to pass by value.**
  - 如果你有个对象属于内置类型(例如 `int`), `pass by value` 往往比 `pass by reference` 的效率高些。
  - 尽量以 `pass by reference` 替换 `pass by value` 。前者通常比较高效，并可避免切割问题(`slicing problem`)。
  - 不过并不适用于内置类型，以及STL的迭代器和函数对象。对它们而言 `pass by value` 往往比较适当。
- **条款21：必须返回对象时，别妄想返回其reference**
  - **Don't try to return a reference when you must return an object**
  - `Reference` 只是个名称，代表某个既有对象。
  - 绝不要返回pointer或reference指向一个local stack对象，或返回 `reference` 指向一个 `heap-allocated` 对象，或返回 `pointer` 或 `reference` 指向一个 `local static` 对象而有可能同时需要多个这样对象。

- 条款22：将成员变量声明为**private**
    - **Declare data members private**
    - 切记将成员变量声明为private.这可赋予客户访问数据的一致性、可细微划分访问控制、允诺约束条件获得保证，并提供class作者以充分的实现弹性。
    - Protected并不比public更具封装性。
  - 条款23：宁以**non-member**、**non-friend**替换**member**函数
    - **Prefer non-member non friend functions to member functions**
    - 宁可拿non-member non-friend函数替换member函数。这样做可以增加封装性、包裹弹性和技能扩充。
- 



- 条款24：若所有参数皆需类型转换，请为此采用**non-member**函数
  - **Declare non-member functions when type conversions should apply to all parameters.**
  - 如果你需要为某个函数的所有参数(包括被this指针所指的那个隐喻参数)进行类型转换，那么这个函数必须是个non-member。
- 条款25：考虑写出一个不抛出异常的**swap**函数
  - **Consider support for a non-throwing swap.**
  - Swap（置换）两对象值，意思是将两对象的值彼此赋予对方。




## 第五章：实现

- 条款**26**：尽可能延后变量定义式出现的时间
  - **Postpone variable definitions as long as possible.**
  - 做法A: 1个构造函数 + 1 个析构函数 + n 个赋值操作
  - 做法B: n个构造函数 + n 个析构函数
  - 尽可能延后变量定义式的出现。这样做可增加程序的清晰度并改善程序效率。

```
//方法A: 定义于循环内
Widget w;
for(int i = 0; i < n; ++ i )
{
    w = 取决于i的某个值;
    ...
}
```

```
//方法B: 定义于循环外
for(int i = 0; i < n; ++ i )
{
    Widget w(取决于i的某个值);
    ...
}
```


## 第五章：实现

- 条款**27**：尽量少做转型动作
    - **Minimize casting.**
    - **Const\_cast** 通常被用来将对象的常量性转除(cast away the constness)。
    - **Dynamic\_cast**用来执行“安全向下转型”，用来决定某对象是否归属继承体系中的某个类型。
    - **Reinterpret\_cast**执行低级转型，取决于编译器，表示它不可移植。
    - **Static\_cast** 强迫隐式转换。
- 

- **条款28：避免返回handles指向对象内部成员**
  - **Avoid returning “handles”to object internals.**
  - 避免返回handles（包括reference、指针、迭代器）指向对象内部。遵守这个条款可增加封装性，帮助const成员函数的行为像个const，并将发生“虚吊号码牌”(dangling handles)的可能性降至最低。
- **条款29：为“异常安全”而努力是值得的**
  - **Strive for exception-safe code.**
  - 异常安全函数(exception-safe functions)即使发生异常也不会泄露资源或允许任何数据结构败坏。这样的函数区分为三种可能的保证：基本型、强烈型、不抛异常型。
  - “强烈保证”往往能够以copy-and-swap实现出来，但“强烈保证”并非对所有函数都可实现或具备现实意义。
  - 函数提供的“异常安全保证”通常最高只等于其所调用之各个函数的“异常安全保证”中的最弱者。

- 条款30：透彻了解inlining的里里外外
    - Understand the ins and outs of inlining.
    - 将大多数inlining限制在小型、被频繁调用的函数上。这可使日后的调试过程和二进制 升级(binary upgradability)更容易，也可使潜在的代码膨胀问题最小化，使程序的速度提升机会最大化。
    - 不要只因为function templates出现在头文件，就将它们声明为inline。
  - 条款31：将文件的编译依赖关系降至最低
    - Minimize compilation dependencies between files.
    - 支持“编译依存性最小化”的一般构想是：相依赖于声明式，不要相依赖于定义式。基于此构想的两个手段是Handle classes和Interface classes。
    - 程序库头文件应该以“完全且仅有声明式”(full and declearation-only forms)的形式存在。这种做法不论是否涉及templates都适用。
- 

## 第六章：继承与面向对象设计

- 条款32：确定你的public继承塑模出is-a关系
    - **Make sure public inheritance models “is-a”**
    - C++进行(OOP)面向对象编程,最重要的一个规则是：public inheritance（公开继承）意味“is - a”(是一种)关系。
    - 如果你令class D(“derived”)以public形式继承class B(“Base”),你便是告诉编译器：每一个类型为D的对象同时也是一个类型为B的对象，反之不成立。
    - “public继承”意味is-a。适用于Base classes身上的每一件事情一定也是用于 Derived classes 身上，因为每一个Derived class对象也都是一个Base Class对象。
- 

- 条款33：避免遮掩继承而来的名称
  - **Avoid hiding inherited names.**
  - **Derived classes**内的名称会隐藏**Base classes**内的名称。
  - 为了能让被隐藏的名称可以使用，可用**using**声明式或（转交函数）。

```
class Base{  
private:  
    int x;  
public:  
    virtual void mf1() = 0;  
    virtual void mf1(int);  
    virtual void mf2();  
    void mf3();  
    void mf3(double);  
    ...  
};
```

```
class Derived: public Base{  
public:  
    virtual void mf1();  
    void mf3();  
    void mf4();  
    ...  
};
```

- 条款33：避免遮掩继承而来的名称
  - **Avoid hiding inherited names.**
  - **Derived classes**内的名称会隐藏**Base classes**内的名称。
  - 为了能让被隐藏的名称可以使用，可用**using**声明式或（转交函数）。

```
class Base{
private:
    int x;
public:
    virtual void mf1() = 0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
    ...
};


class Derived: public Base{
public:
    using Base::mf1; //让Base class内名为mf1和mf3的所有东西
    using Base::mf3; //在Derived作用域内都可见（并且public）
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};
```



- 条款34：区分接口继承和实现继承
  - Differentiate between inheritance of interface and inheritance of implementation
  - Pure virtual函数有两个最突出的特性：它们必须被任何“继承它们”的具象class重新声明，而且它们在抽象class(基类)中通常没有定义。
    - 声明一个pure virtual函数的目的是为了derived classes只继承函数接口。
    - 声明一个impure virtual函数的目的，是让derived classes继承该函数的接口和缺省实现。
    - 声明non-virtual函数的目的是为了令derived classes继承函数的接口及一份强制性实现。
  - 接口继承和实现继承不同。在public继承下，derived classes总是继承Base class 的接口。
  - Pure virtual函数只具体指定接口继承。
  - Impure virtual 函数具体指定接口继承及缺省实现继承。
  - Non-virtual函数具体指定接口继承以及强制性实现继承。

- 条款35：考虑virtual函数以外的其他选择
  - Consider alternative to virtual functions.
  - 使用non-virtual interface(NVI)手法，那是Template Method设计模式的一种特殊形式。它以public non-virtual成员函数包裹较低访问性(private 或 protected)的virtual函数。
  - 将virtual函数替换为“函数指针成员变量”。将功能从成员函数移到class外部函数，带来一个缺点是，非成员函数无法访问class的non-public成员。



- 条款36：绝不重新定义继承而来的**non-virtual**函数
    - **Never redefine an inherited non-virtual function.**
    - 绝对不要重新定义继承而来的**non-virtual**函数。
  - 条款37：绝不重新定义继承而来的缺省参数值
    - **Nerver redefine a function's inherited default prameter value.**
    - 绝对不要重新定义一个继承而来的缺省参数值，因为缺省参数值都是静态绑定，而**virtual**函数-----你唯一应该复写的东西-----却是动态绑定。
  - 条款38：通过复合塑模出**has-a**或“根据某物实现出”
    - **Model “has-a” or “is-implemented-in-terms-of”through composition.**
    - 复合(**composition**)是类型之间的一种关系，当某种类型的对象内含它种类型的对象，便是这种关系。
- 

- 复合(**composition**)的意义和public继承完全不同。
- 在应用域(**application domain**),复合意味 **has-a** (有一个)。在实现域(**implementation domain**),复合意味**is-implemented-in-terms-of** (根据某物实现出)。

- **条款39：明智而审慎地使用private继承**

- **Use private inheritance judiciously.**
- 如果**classes**之间的继承关系是**private**，编译器不会自动将一个**derived class**对象转换为一个**Base class**对象。
- **Private base class**继承而来的所有成员，在**derived class**中都会变成**private**属性。


- **条款40：明智而审慎的使用多重继承**

- **Use multiple inheritance judiciously**
- 多重继承比单一继承复杂。它可能导致新的歧义性，以及对**virtual**继承的需要。
- **Virtual**继承会增加大小、速度、初始化（及赋值）复杂度等等成本。

- 多重继承的确有正当用途。其中一个情节涉及“**public**继承某个 **Interface class**”和“**private**继承某个协助实现的**class**”的两相结合



## 第七章：模板与泛型编程

- 条款41：了解隐式接口和编译期多态
    - Understand implicit interfaces and compile-time polymorphism.
    - 通常显示接口由函数的签名式(函数名称、参数类型、返回类型)构成。
    - 隐式接口并不基于函数签名式，而是由有效表达式(valid expressions)组成。
    - Classes 和 template 都支持接口(interfaces)和多态(polymorphism)。
    - 对classes而言接口是显示的(explicit),以函数签名为中心。多态则是通过virtual函数发生于运行期。
    - 对template参数而言，接口是隐式的(implicit)，莫基于有效表达式。多态则是通过template具现化和函数重载解析(function overloading resolution)发生于编译期。
- 

- 条款42：了解typename的双重意义
    - Understand the two meanings of typename.
    - 声明template参数时，不论使用关键字class或typename，意义完全相同。
    - 请使用关键字typename标识嵌套从属类型名称；但不得在base class lists（基类列）或member initialization list（成员初始列表）内以它作为Base class修饰符。
  - 条款43：学习处理模板化基类内的名称
    - Know how to access names in templated base classes.
- 可在derived class templates 内通过“this->”指向base class templates 内的成员名称，或藉由一个明白写出的“base class资格修饰符”完成。

- **条款44：将与参数无关的代码抽离templates**
  - **Factor parameter-independent code out of templates.**
  - **Templates** 生成多个 **classes** 和多个函数，所以任何**template**代码都不该与某个造成膨胀的**template**参数产生相依关系。
  - 因**非类型模板参数(non-type template parameters)**而造成的代码膨胀，往往可消除，做法是**以函数参数或class成员变量替换template参数**。
  - 因**类型参数(type parameters)**而造成的代码膨胀，往往可降低，做法是让带有完全相同二进制表述(**binary representations**)的具现类型(**instantiation types**)共享实现码。
- **条款45：运用成员函数模板接受所有兼容类型**
  - **Use member function templates to accept “all compatible types.”**
  - 使用**member function templates**（成员函数模板）生成“**可接受所有兼容类型**”的函数。
  - 如果你声明 **member template** 用于“**泛化copy构造**”或“**泛化assignment操作**”，你**还是需要声明正常的copy构造函数和copy assignment操作符**。



- 条款**46**: 需要类型转换时请为模板定义非成员函数
  - Define non-member functions inside templates when type conversions are desired.
  - 当我们编写一个class template，而它所提供的“与此template相关的”函数支持“所有参数之隐式类型转换”时，请将那些函数定义为“class template内部的friend函数”。
- 条款**47**: 请使用traits classes表现类型信息
  - Use traits classes for information about types.
  - -----



- 条款48：认识**template**元编程
  - **Be aware of template metaprogramming.**
  - **Template metaprogramming**(**TMP**,模板元编程)可将工作由运行期移往编译器，因而得以实现早期错误侦测和更高的执行效率。
  - **TMP**可被用来生成“基于政策选择组合”（**based on combinations of policy choices**）的客户定制代码，也可用来避免生成对某些特殊类型并不适合的代码。



# 第八章：定制new和delete

- 条款49：了解new-handler的行为
  - Understand the behavior of the new-handler.
  - 当operator new无法满足某一个内存分配需求是，它会抛出异常。
    - 在反映一个未获满足的内存需求之前，它会调用一个客户指定的错误处理函数，一个所谓的新-handler。
      - 客户必须调用set\_new\_handler，那是声明于<new>的标准程序库；
        - » Namespace std {
        - » typedef void(\*new\_handler)();
        - » new\_handler set\_new\_handler(new\_handler p) throw();
        - » }
      - 可以这么使用new\_handler。
        - » Void OutOfMem()
        - » {
        - »     std::cerr<<“Unable to satisfy request for memory\n”;
        - »     std::abort();
        - » }
        - » Int main()
        - » {
        - »     std::set\_new\_handler(OutOfMem);
        - »     int \*pBigDataArray = new int [100000000L];
        - »     .....
        - » }

- **Set\_new\_handler** 允许客户指定一个函数，在内存分配无法满足时被调用。
  - **Nothrow new** 是一个颇为局限的工具，因为它只适用于内存分配；后继的构造函数调用还是可能抛出异常。
  - **条款50：了解new 和 delete 的合理替换时机**
    - **Understand when it makes sense to replace new and delete.**
    - 有许多理由需要写个自定的**new**和**delete**，包括改善效能、对**heap**运用错误进行调试、收集**heap**使用信息。
  - **条款51：编写new 和 delete时需固守常规**
    - **Adhere to convention when writing new and delete.**
    - **Operator new** 应该内含一个无穷循环，并在其中尝试分配内存，如果它无法满足内存需求，就该调用**new-handler**。它也应该有能力处理 0 bytes 申请。**Class** 专属版本则还应该处理“比正确大小更大的（错误）申请”。
    - **Operator delete** 应该在收到**null**指针时不做任何事，**Class** 专属版本则还应该处理“比正确大小更大的（错误）申请”。
- 

- **条款52: 写了 placement new 也要写 placement delete**
  - **Write placement delete if you write placement new.**
  - 当你写一个 placement operator new, 请确定也写出对应的 placement operator delete。如果没有这样做, 你的程序可能会发生隐微而时断时续的内存泄露。
  - 当你声明 placement new 和 placement delete, 请确定不要无意识（非故意）地遮掩了它们的正常版本。

