《effective c++》

一、让自己习惯c++

- **c++ 联邦**

把c++ 看作是 c语言、STL、面向对象、模板 的联邦。例子：对内置对象（c语言），用值传递更高效；STL是基于指针的，所以值传递也更好；面向对象、模板部分，引用传递更好。

- **不滥用宏 #define**

宁可以编译器替代预处理器。#define并不重视作用域，不可取地址。const可以取地址，enum不可取地址（当不想让别人获得指向你的整数常量的指针时）。

对于单纯的常量，使用const、enum替代。宏在预处理时就被替代，编译器没看到不利于检错。

对于形似函数的宏，用inline函数代替#define。

```
#define PI 3.14
const double PI = 3.14;

class GamePlayer{
    private:
    static const int Num = 5; // 常量声明式，但凡需要取某个class的常量地址，则需要定义
    int scores[Num];
}
const int GamePlayer::Num; // 定义，但可以不用赋值.

// 但如果不支持在声明时赋值（in-class初值设定）
class GamePlayer{
    private:
    static const int Num ; // 常量声明式，但凡需要取某个class的常量地址，则需要定义
    int scores[Num];       // 这种情况将无法在编译阶段知道scores需要的大小。
}
const int GamePlayer::Num=5;
// 所以改成
class GamePlayer{
    private:
    enum{Num = 5};  // 令5成为一个记号，类似于#define的效果
    int scores[Num];
}
```

```
#define CALL_WITH_MAX(a,b) f((a) > (b) ? (a):(b))
int a=5, b=0;
CALL_WITH_MAX(++a, b);    // a被累加2次
CALL_WITH_MAX(++a, b+10);  // a被累加1次

// 改用 inline-- 这使得它符合函数的作用域和访问规则
template<typename T>
inline void callWithMax(const T& a, const T& b){
    f(a>b ? a : b);
}
```

- **尽可能使用const——如果出现在星号左边，表示被指物是常量，右边则表示指针自身是常量。**

1. 限定变量、对象、函数参数、函数返回类型、函数本体的修改权限，利于编译器侦测检错；
2. 限定成员函数，使接口含义更明确（哪个函数可以改动对象），使操作const对象成为可能；

```
class Text{
const char& operator[] (int pos) const{} // 给 const Text t; 调用
char& operator[] (int pos) {}// 给 Text t;
};
成员函数为const 意味着什么--bitwise const 和logical const
bitwise: bit意义上的，任何非non-static的成员变量都不可改。
logical: 不影响使用目的的情况下，可以修改对象内部分bit。--使用mutable。加了mutable的成员变量可以在const函数内进行修改。
```

3. 当const和non-const成员函数有着实质性等价的实现时，令non-const版本调用const版本可避免代码重复。

```
const char& operator[] (int pos) const{...}
char& operator[] (int pos) {return const_cast<char&>(
    static_cast<const Text&>(*this)[pos] // 为(*this)加上const后调用const op[]
);}
```

4. const修饰函数返回值

```
const Rational operator* (const Rational & lhs, const Rational & rhs);
可以避免
if( (a*b) =c ); // 这样的错误语法
```

5. 在STL的迭代器（iterator）中：

```
const std::vector<int>::iterator iter = vec.begin(); // 这个作用像 T* const
iter++; // 将会是错误的。

const std::vector<int>::const_iterator cter=vec.begin(); // 这个作用像 const T*
*cter = 10; // 将会是错误的
```

- **确保对象在使用之前被初始化**

1. 确保对内置对象（int, bool等）的手动初始化。
2. 区分构造函数时的赋值和**初始化列表**。赋值：等号赋值写在函数体内。初始化：跟在构造函数冒号后面，函数体是空的。尽可能使用初始化列表的方式，但并不是绝对的，如果构造函数过多，可以考虑将重复性的工作（赋值和初始化列表同样表现良好的成员变量的初始化）移入某个private的函数内。这种做法在"成员变量的初值是由文件或数据库读入"时特别好用。

```
ABEntry::ABEntry(const std::string &name, const std::list<Phone> &phone)
    :theName(name),
    thePhone(phones),
    numTimes(0)
{}
```

3. 上述办法已经能解决大部分问题了。但考虑一种情况，如果一份代码（编译单元）中的non-local-static对象使用了另一份代码中的non-local-static对象，将因为无法决定两者的初始化顺序而可能导致出错。解决办法：将需要初始化的对象，封装至函数内，返回该对象的引用。因此在后续使用中使用的是指向static对象的引用，而不再是对象本身。

二、构造、析构、赋值

- **了解c++默默编写并调用了哪些函数**

默认添加构造（如果没有声明构造函数的话）、拷贝构造、析构、=操作符（这些函数都是inline并且**public**的）。有需要时（**const**类型、**引用**类型变量）可以**自己定义**这些函数。

```
Class Base{
    string & str;
    const T value;
}
Base a("dog",2);
Base b("cat",1);
a = b // 编译器拒绝编译，需要自己定义=operate
```

- **若不想使用编译器自动生成的函数，就该明确拒绝**

如果不需要使用拷贝构造和=时（比如每个class实例都应是独一无二的），需要拒绝copying。

法1：将该函数定义为**private**且不实现它。

```
class Base{
private:
    Base(const Base&);
    Base& operator= (const Base&);
}
```

但member函数和friend函数可以调用它，所以并不是绝对安全。

法2：继承一个uncopyable基类，在这个基类中像上面一样拒绝该函数。法3：c++11特性，在该函数后添加 $= delete$

- **为多态基类声明virtual析构函数**

一般会用**父类的指针指向子类的对象**，所以如果析构父类指针时，如果不是虚析构将**无vptr指针**将所有内存释放干净（相当于只会局部释放内存——父类的那部分）。类带有任何virtual函数，都应该有一个虚析构函数。**任何不作为基类、不用于多态的均不要声明虚析构。**

```
class Base{
    Base();
    virtual ~Base();  // virtual的目的是运行派生类得以实现客制化
    int a,b;          // 并且在有virtual函数时，必然有vptr，因此内存大小不是64bit
};
```

另外有时候 纯虚析构很好用（实现了抽象、基类、虚析构的统一）。

补充：析构函数的运作方式——最深层的派生类的析构函数最先调用，而后依次往上。

- **别让异常逃离析构函数**

析构函数出现异常，使用abort程序或者try catch 记录异常。如果客户需要对异常做出反应，则提供一个普通函数执行改操作。

```
class DBConn{
    public:
    void close(){
        db.close();    // 关闭数据库连接
        closed = true;
    }
    ~DBConn(){
        if(!closed){
            try{
                db.close();
            }catch(...){}
        }
    }
}
```

- **绝不在构造、析构过程中调用virtual函数**

因为这类调用不会下降至derived class。根本原因是：在派生类的基类构造或析构时，该类为base类而不是derived类，这使得derived类的变量呈现未定义值。非要用，可以将下层的信息往上层传——取消virtual。

- **令operate= 返回一个 *reference to* *this**

为了能连锁赋值：x=y=z=15；

```
class Widget{
    Widget & operator=(const Widget& rhs){
        return * this;
    }
}
```

- **在operate=中处理自我赋值**

可能发生的情况a[i] = a[j] (当i==j时，便是自我赋值)等。这种带来的后果：可能在赋值前就删掉了该对象。

```
Widget& Widget::operator= (const Widget& rhs){
    delete pb; // 当pb == rhs 时，将会出错，可以加上证同测试
    pb = new Bitmap(*rhs.pb);
    return *this;
}
```

处理的传统做法：在operate=中添加一个"证同测试"$if(this == \&rhs)return * this;$；或者注意在赋值前暂存原值，赋值完成后再删除。最好的办法：copy and swap技术。

```
Widget& Widget::operator= (const Widget& rhs){
    Widget temp(rhs);
    swap(temp);         // 交换*this 和 temp 的数据
    return *this;
}
```

- **复制对象时勿忘其每一个成分**

编写copying函数时，确保复制**每一个local成员变量**，确保调用**所有的base class 内适当的copying函数**。ps：不要为了减少重复代码而让copying函数调用构造函数，反过来也不行，这没有可读性。可以创建private成员函数共两者共同调用相同部分的代码。

三、资源管理

- **以对象管理资源(RAII（资源取得时机便是初始化时机）)**

```
Base *b = createBase();
...
delete b;   /// 依赖于delete释放资源，但如果在delete前就return等操作，资源将泄露。
```

法1（使用c++预制的对象进行管理）：使用**auto_ptr智能指针**在构造函数中管理资源，在析构函数中释放，但不可复制。若通过copy构造或operator=赋值，它们会变成null。

```
void f(){
    std::auto_ptr<Base> p(createBase());
    p2 = p;  // p变成null，p2获得唯一资源管理权
}
```

或使用**shared_ptr，引用计数型智慧指针**，当计数为0时释放资源。

```
void f(){
std::tr1::shared_ptr<Base> p(createBase());
}
```

这两者均是在**析构函数内执行delete操作**，非delete[]，所以**不能用于数组释放。**

法2（自定义资源管理类）：把资源放进对象内，依赖C++的析构函数自动调用确保资源被释放。需要注意以下设计细节。

- **在资源管理类中小心copying行为**

当RAII对象被复制：

1. 禁止复制——继承一个uncopyable基类，其中copy构造和operator=被设为private。
2. 对底层资源使用引用计数法——使用shared_ptr

```
class Lock{
public:
    explicit Lock(Mutex* pm):mutexPtr(pm, unlock){ // 可以指定使用unlock函数解锁资源，而不是在引用计数为0时直接删除
        lock(mutexPtr.get());
    }
private:
    std::tr1::shared_ptr<Mutex> mutexPtr;
};
```

- **在资源管理类中提供对原始资源的访问**

```
std::tr1::shared_ptr<Investment> pInv(createInvestment());  // 有个智能指针

int daysHeld(const Investment* pi);// 返回投资天数
int day = daysHeld(pInv);  // 错误!

int day = daysHeld(pInv.get());  // 进行显示转换，获得内部Investment指针。

// 或者进行隐式转换，提供一个隐式转换函数。但两种方案的选择应该取决于实际需要。
```

- **成对使用new 和delete时要采取相同的形式**

```
typedef std::string AddressLines[4];
std::string *pal = new AddressLines;  // 像new string[4] 一样。
delete [] pal;  // 正确!
```

- **以独立语句将newed对象置入智能指针**

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
// 不要像上面这样写！上面在调用processWidget函数前执行三件事
// new Widget、调用priority函数、调用shared_ptr构造函数。但三者执行顺序不一定。如果是前面这种写法，当调用priority出错时，new Widget的指针将泄露。
// 正确写法如下
std::tr1::shared_ptr<Widget> pw(new Widget);
processWidget(pw, priority());
```

四、设计与声明

- **让接口容易被正确使用，不易被误用**

正确使用：接口的一致性，与内置操作的行为兼容等。

不被误用：1、借助const、类型系统等手段，在编译阶段检测提醒用户。2、强制设置返回值类型，消除客户的资源管理责任（条款14）

```
class Date{
public:
    Date(int month,int day,int year);
    // 改成以下
    Date(const Month& m, const Day& d, const Year& y); // 靠类型系统检查
}
Date d(30,3,2024);  // 这种是用户使用错误

std::tr1::shared_ptr<Investment> retVal(static_cast<Investment*>(0),
getRidOfInvestment); // 指定getRidOfInvestment为删除器
```

- **设计class时自问自己以下几个问题**

1. 新type的对象应该如何被创建和销毁？——涉及构造、析构、内存分配和释放
2. 对象初始化和对象赋值应该有什么样的差别?
3. 新type的对象如果被pass by value，意味着什么？——copy构造函数如何实现
4. 什么是新type的“合法值”？——错误检查、约束条件、构造、赋值等
5. 你的新type需要配合某个继承图系吗？——是否继承他人、是否会被继承（析构函数是否为virtual）
6. 你的新type需要什么样的转换?
7. 什么样的操作符和函数对此新type是合理的？——声明哪些函数，哪些为member函数

8. 什么的标准函数应该驳回？——哪些为private
9. 谁该取用新type的成员？——哪些成员为public，private，protected、friends
10. 什么是新type的“未声明接口”?
11. 你的新type有多么一般化?
12. 你真的需要一个新type吗?

- **宁以pass-by-reference-to-const替换pass-by-value**

```
bool validateStudent(Student s);
bool validateStudent(const Student& s);
```

但对内置类型和STL的迭代器和函数对象来说，pass-by-value更加高效。

- **必须返回对象时，别妄想返回其reference**

```
const Rational& operator* (const Rational& lhs, const Rational& rhs){
    Rational result(lhs*rhs); // 糟糕的代码！仍然调用了构造函数，并且使得返回值为一个local对象。
    Rational* result = new Rational(lhs*rhs); // heap内的对象，但仍然付出了构造函数的成本，以及谁来负责delete。
    return result;
}
// 正确写法
inline const Rational operator*(const Rational& lhs, const Rational& rhs){
    return Rational(lhs*rhs);
}
```

- **将成员变量声明为private**

好处：

1. 用户友好——反正一切都是要调用函数
2. 利于设计只读、不准访问、读写访问等对变量的操作权限函数。
3. 封装！！方便日后更改

- **宁以non-member, non-friend替换member函数——对于为实现便利的函数而言**

原因：获得更大的封装性、扩充性——member函数可以访问到的部分更多，所以封装性更差。此外，成为某个类的non-member函数并不意味着它不可以是另一个不使用其成员的类的member。（可以令clearBrowser成为某工具类的member函数，只要它不是WebBrowser的一部分。将所有这样的便利函数放在多个头文件但隶属于同一个命名空间。

- **若所有参数皆需类型转换，采用non-member函数**

```
class Rational {
    public:
    const Rational operator* (const Rational& rhs, const Rational& lhs) const;
}
Rational oneHalf;
result = oneHalf * 2;  // 正确 因为发生了隐式类型转换
result = 2 * oneHale;  // 错误 参数列表为 int 和 Rational，并不会位于参数列，导致不进行参数列表转换。
// 解决办法：将operator*写到类外面，成为非成员函数
```

- **考虑写出一个不抛异常的swap函数**

如果内置版的swap效率不足，尝试一下步骤

1. 提供一个public成员函数swap（但不可抛出异常）
2. 在class或template的命名空间内，编写一个非成员函数调用上面这个函数。
3. 如果正在编写的是class而非模版类，则为class特化std::swap，并调用第一个swap。

```
// 传统的swap函数
void swap(T& a, T& b){
    T temp(a);
    a = b;
    b = temp;
}// 缺点：进行了一次拷贝，占用内存和时间。有些情况，只需要交换指针

// 声明一个名为swap的public成员函数做真正的置换工作，然后将std::swap特化，令它调用该成员函数。
class Widget{
public:
    void swap(Widget& other){
        using std::swap;
        swap(pImpl, other.pImpl);
    }
};
namespace std{
    template<>
    void swap<Widget>(Widget& a, Widget& b){a.swap(b);}
}

// 但当类成为一个模版类时，上述办法无法通过编译。（即不可偏特化一个function template
// 解决办法：1. 向std里面添加一个重载的swap（但这个办法不太好）
// 2.声明一个非成员函数swap，但不声明为特化版本或重载，让它调用public成员函数的swap。
template<typename T>
void swap(Widget<T>& a, Widget<T>& b){a.swap(b);} // 这里不属于std命名空间
```

五、实现

- **尽可能延后变量定义式出现时间**

尝试延后这份定义直到能够给它初值实参为止。举例：

```
std::string encryptPassword(const std::string& password){
    std::string encrypted;
    ...    // encrypt();
    return encrypted;
}
std::string encryptPassword(const std::string& password){
    std::string encrypted(password);
    encrypt(enctypted);
    return encrypted;
}
```

但如果在循环中

```
// 方法A
Widget w;
for(int i=0; i<n; i++) w=取决于i的某个值;

// 方法B
for(int i=0;i<n;i++) Widget w(取决于i的某个值);
```

方法A 包括 1次构造+1次析构+n次赋值； 方法B包括 n次构造+n次赋值。具体情况具体分析。

- **尽量少做转型**

c风格的转型和四种c++新式转型。新式的好处：1.易于辨识；2.编译器易于检错

```
(T) expression
const_cast<T> (expression)  // 将对象的常量性移除，唯一有此能力的操作办法
dynamic_cast<T>(expression)  // 主要用来执行“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<T> (expression) // 执行低级转型，实际动作可能取决于编译器
static_cast<T> (expression) // 强迫隐式转换，例如将non-const转为const，int转为double等
```

一些转型发生的情况

```
int x,y;
double d = static_cast<double>(x) /y;

Derived d;
Base* pb = &d;  // 隐喻将Derived* 转为 Base*

// static_cast
class Derived:public Base{
public:
    virtual void fun() {
        static_cast<Base>(*this).fun(); // 将*this转型为Base类调用其fun函数。这不可行
        ...
    }
}
// 因为它是在当前对象的base class成分的副本上调用的fun（函数是一样的，但this内的属性不一样），然后在当前对象上执行后续操作。所以导致可能base class成分的更改没有落实。
// 解决:去掉转型动作，直接写
virtual void fun() {
    Base::fun();
}

//dynamic_cast通常因为需要在派生类上执行派生类的函数，但只有基类的指针。
// 1. 使用容器，并直接向其中存储派生类对象的指针（通常是智能指针）——这种做法无法让你在一个容器中存储多种派生类的指针
typedef std::vector<std::tr1::shared_ptr<Derived>> VP;
// 2. 在基类中提供虚函数。
// 3. 绝对不要做连串的dynamic_casts
if (Derived1* d1 = dynamic_casts<Derived1*>(iter->get())) {...}
else if (Derived2* d2 = dynamic_casts<Derived2*>(iter->get())) {...}
```

- **避免返回handles指向对象内部成分**

不该令成员函数返回成员变量的handles（指针、引用、迭代器）。

1. 因为当handles指向的内容在该对象之外，则调用者无视const或private可以修改那部分数据，降低了封装性。
2. 导致空悬。即handles所指的东西不复存在。

```
struct RectData{
    Point ulhc;
    Ponit lrhc;
};
class Rectangle{
public:
    Point& upperLeft() const {return pData->ulhc;} // 尽管设置ulhc里的数据为private，但此时却能改变ulhc里的数据
private:
    std::tr1::shared_ptr<RectData> pData;
}
// 解决办法：返回值加上const
const Point& upperLeft() const {return pData->ulhc;}

// 空悬例子
const Rectangle boundingBox(const GUIObject& obj);
GUIObject* pgo;
const Point* p = &(boundingBox(*pgo).upperLeft()); // boundingBox(*pgo) 创造了一个匿名对象，而后被销毁
```

- **为"异常安全"而努力**

当异常被抛出时，带有异常安全性的函数应该满足：

1. 不泄露任何资源
2. 不允许数据败坏

异常安全性函数提供以下三个保证之一：

1. 基本承诺：没有数据败坏，所有对象都处于内部前后一致性中，但程序实际状态不可预料。
2. 强烈保证：如果异常抛出，程序状态不改变。即如果函数成功，就是完全成功，一旦失败，程序会回复到"调用函数之前的"状态。

一般化策略——copy and swap：在副本上进行修改，修改成功则swap，否则原对象仍未改变。但这种办法并不是一劳永逸，如果函数内部调用的其他函数的安全保证更低，则还需要对调用的函数进行恢复状态。另一个问题是效率。

3. 不抛掷（nothrow）保证：承诺绝不抛出异常（不会发生异常）。

- **inlining**

原理：在编译阶段，以函数本体替换函数调用并进行优化。inline只是对编译器的一个申请，而不是强制命令——所以会拒绝太过复杂的函数（递归或循环）以及virtual函数。包括隐喻方式和显示指出。

好处：免除函数调用成本。

坏处：增加目标码（object code）的大小，可能导致额外的换页，降低高速缓存的命中率，以及伴随的效率损失。（当然，如果函数本体比函数调用的产出码还小，就当这句话没说吧）

```
class Person{
public:
    int age() const {return theAge;}  // 定义于class定义式内的inline隐喻方式
private:
    int theAge;
};
// 明确声明
inline const T& std::max(const T& a, const T& b){return a<b? b:a;}

```

使用：

1. 将大多数inlining限制在小型、被频繁调用的函数身上。
2. 如果正在写一个模版，而你认为根据此模版具现出来的函数都应该inlined，请将此模板声明为inline（如上std::max）。否则避免这件事。
3. inline无法随着程序库升级而升级，如果决定修改inline函数，则整个客户端需要重新编译。
4. 大部分调试器对inline函数束手无策。

- **将文件间的编译依存关系降至最低**

1. 如果使用引用和指针可以完成任务，就不要用objects。
2. 尽量以class替换class定义式。
3. 为声明式和定义式提供不同的头文件。
4. 设计pimpl idiom（pointer to implementation）这样的类——**Handle classes**，其真正实现是交给PersonImpl这样的实现类。坏处：为每次访问增加了一层间接性，每一个对象额外增加了一个指针大小的内存，需要初始化（由此可能带来一些动态分配的开销和问题）

```
#include <string>
...
class PersonImpl; // 前置声明
class Date;
class Address;

class Person{
public:
    Person(); // 构造函数
    std::string name() const;
    ...
private:
    std::tr1::shared_ptr<PersonImpl> pImpl;  // 指针，指向实物
}
```

5. 设计为Interface class。即一种特殊的抽象基类——只有virtual析构函数和一堆纯虚函数，没有成员变量、构造函数。Interface class通常使用工厂函数（factory）或虚构造函数为其创建对象。坏处：为每次函数调用付出一个间接跳跃，内存增加一个vptr。

```
static:
class Person{
public:
    static std::tr1::shared_ptr<Person> create(...)  // 工厂函数
};

// 使用
std::tr1::shared_ptr<Person> pp(Person::create(...));
```

六、继承与面向对象设计

- **确定你public继承塑膜出is-a关系**

简单来说就是，public的继承，派生类是基类的特殊化。适用于基类的每一个事情应该适用于派生类。

- **避免遮掩继承而来的名称**

```
class Base{
public:
    virtual void f1() = 0;
    virtual void f1(int);
    virtual void f2();
    void f2();
    void f3(int);
};

class Derived: public Base{
public:
    // 要想调用到基类被遮掩的函数.1.使用using声明式(添加到Derived的public区域)
    using Base::f1;
    using Base::f3;
    virtual void f1();  // 这个会遮掩 Base::f1(int)
    void f3();    // 这个会遮掩 Base::f3(int)
    void f4();
}
```

- **区分接口继承和实现继承**

成员函数的接口总是会被继承。（三种函数接口）

```
virtual void f1() const = 0;  // 纯虚函数--继承者必须重新声明--目的是为了让派生类只继承函数接口。
virtual void f2();  // 虚函数--让派生类继承函数接口和缺省实现
// 1. 为了避免用户继承接口但是忘了实现，导致调用的基类实现，可以考虑将该虚函数变为纯虚函数，同时提供protected的non-virtual member函数，强制实现f2的同时，调用protected函数减少代码重复。2.考虑将该虚函数变为纯虚函数，同时实现它。
void f3();  // 普通函数--为了让派生类继承函数的接口及一份强制性实现。
```

- **考虑virtual函数以外的其他选择**（这个条款内的看不太明白，Strategy的什么鬼东西）

1. NVI手法：令客户通过public non-virtual成员函数间接调用private virtual函数。前者称为后者的外覆器。外覆器的好处在于，在真正调用private 虚函数前后可以进行一些操作，确保函数调用之前设定好了适当的环境等。
2. 藉由函数指针实现**Strategy**模式：将虚函数替换为“函数指针成员变量”
3. 藉由tr1::function完成**Strategy**模式：将虚函数替换为“tr1::function”
4. 将继承体系内的虚函数替换为另一个继承体系内的虚函数。

- **绝不重新定义继承而来的non-virtual函数**

```
class B{
public:
    void fm();
}
class D:public B{
public :
    void fm();
}
D x;
B* px = &x;
px->mf(); // 调用的是B的。因为是静态链接，而virtual函数才是动态链接。
```

- **绝不重新定义继承而来的缺省参数值**

```
class Base{
    virtual void f(int value = 0) const = 0;
}
class Derived :public Base{
    virtual void f(int value = 2) const;
}

Base* p = new Derived;
p->f(3); // 调用Derived的
p->f();  // 调用Base的
```

上面这种奇怪的现象在于：virtual决定了函数调用是动态绑定的，而缺省参数值却是静态绑定的。

那如何在派生类中实现特化的带有缺省参数值的虚函数呢?

1. 不改它的缺省参数值——导致代码重复，并且一旦修改Base的缺省值，所有的派生类都要跟着修改。
2. 虚函数的替代设计——条款35（对没错就是看不懂的那个NVI，Strategy）

- **通过复合塑膜出has-a或"根据某物实现出"**

复合是类型中的一种关系，形如以下。意味着**has-a**（有一个）或者**is-implemented-in-terms-of**（根据某物实现出）。

对于形如：车、人、图片等对象，它们属于应用域。在应用域的复合应该表现为has-a的关系。

对于形如：互斥锁、缓冲区等，它们属于实现域。在实现域的复合表现为is-implemented-in-terms-of。（与is-a的区别在于，一个是public继承，一个是作为成员变量复用）

```
class Address{...}
class PhoneNumber{...}
class Person{
public:
    Address address;
    PhoneNumber pn;
};
```

- **private继承使用**

1. private继承时，编译器不会自动将一个派生类对象转换为一个基类对象。
2. 继承得来的所有成员（public和protected）在派生类中均变成private属性。
3. private继承意味着**is-implemented-in-terms-of**；D private继承 B意味着D借由B实现，纯粹的实现技术。（但还是尽可能使用复合，必要时使用private继承——当protected成员或virtual函数

牵扯进来时）

```
class Widget{
private:
    class WidgetTimer : public Time{  // 为了重新定义、复用Time里的虚函数onTick()
    public:
        virtual void onTick() const;
    };
    WidgetTimer timer;
};
// 为什么直接private继承而是如此public继承+复合？
// 1. 在不想要派生类能再次定义onTick()函数。2. 将Widget的编译依存性将至最低。
```

4. 处理的class不带任何数据时，使用private继承它会减少内存开销。（空白基类最优化）

```
class Empty{};
class MyInt{  // sizeof(MyInt) > sizeof(int) 因为c++官方会插一个char到空对象中，再考虑对齐需求，可能加上padding
private:
    int x;
    Empty e;
}
class MyInt : private Empty{  //  sizeof(MyInt) == sizeof(int)
private:
    int x;
};
```

- **多重继承**

多重继承可能导致：

1. 函数调用歧义——基类们有相同的成员函数（尽管可能访问权限不同，但c++先考虑匹配性，再考虑可用性）

```
derived.Base1::f();  // 解决：指明调用的是谁的
```

2. “菱形继承”

解决方案：采用虚继承——但是要为此付出代价（时间和空间）

```
class A{...};
class B : virtual public A{...};
class C : virtual public A{...};
class D : public B, public C{..};
```

3. 但它确有正当用途，涉及“public继承某个接口类”和“private继承某个协助实现的类”的两相结合。

七、模板与泛型编程

- **了解隐式接口和编译期多态**

编译期多态和运行时多态类似于“哪一个重载函数被调用”和“哪一个虚函数被绑定”。

显示接口由函数签名式（函数名称、参数类型、返回类型）构成。

隐式接口有有效表达式组成。如下：关于此函数，对传入的参数w的约束为——size(), operator>, operator&&,operator!=。等

```
template<typename T>
void doProcessing(T& w){
    if(w.size() > 10 && w!=songNastyWidget){...}
}
```

- **了解$typename$的双重含义**

声template参数时，前缀关键字class和typename可互换。用typename标识**嵌套**从属类型名称；但不得在基类列或成员初值列内以它作为基类修饰符。

```
template<typename T>
viod print2nd(const C& container){ // 但是这里不可typename const C& container
    if(container.size() > 2){
        typename C:: const_iterator iter(container.begin()); //用typename标识嵌套从属类型名称
    }
}

template <typename T>
class Derived: public Base<T>::Nested{ // 基类列中不允许typename
public:
    explicit Derived(int x)
        :Base<T>::Nested(x){  // mem.init.list中不允许typename
            typename Base<T>::Nested temp;  // 这里需要typename
        }
};

```

- **处理模版化基类的名称**

与上面类似，typename 在进行template时，无法在编译阶段就知道，后面的东西到底是个什么、包括了什么函数和变量，甚至是否拥有你想要调用的模板函数（全特化后的类可以不定义那个函数），所以编译会无法通过。

三个解决办法：

1. 在基类函数调用动作前加上 "$this->$"
2. 使用using声明式。
3. 明白指出调用的函数位于基类内。

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company>{
public:
    void sendClearMsg(const MsgInfo& info){
        sendClear();  // 调用基类的指针；这段代码无法通过编译。
    }
}
1. this-> sendClear();
2. using MsgSender<Company>::sendClear();
3. MsgSender<Company>::sendClear();
```

- **将与参数无关的代码抽离templates**

```
template<typename T, std::size_t n>
class SquareMatrix{
    classSquareMatrix{
    public:
        void invert();
    }
};

SquareMatrix<double, 5> sm1;
sm1.invert();
SquareMatrix<double,10> sm2;
sm2.invert();
```

这样子会具象化两份invert，而这两份函数代码，除了常量5和10没有什么不同。如何抽离相同部分呢?

方法：数据存在SquareMatrix中，让另一个类使用SquareMatrix，并传递参数n给SquareMatrix，使得调用的invert永远是基类中的那一个。坏处是对象本身可能非常大（考虑到这点可以将数据放进heap内）。

当然这种抽离也不是在所有情况下都是最好的——working set、对象大小等。

- **运用成员函数模板接受所有兼容类型**

如果以带有Base-Derived关系的B、D类具现化某个模板，产生出来的两个具现体不带有Base-Derived关系。所以为了使得具现体们能有所联系（能进行隐式转换），需要对template编写一个**泛化copy构造函数**（member templates）。

```
template<typename T>
class SmartPtr{
public:
    template<typename U>
    SmartPtr(const SmartPtr<U>& other); // 为任何类型U都可以生成一个类型T的SmartPtr；不能是explicit
};

SmartPtr<Base> ptr1 = SmartPtr<Derived>(new Derived);  // 现在可以进行隐式转换了
```

但这还有问题，这个泛化copy构造函数的功能远超我们需要，它也会将 $SmartPtr<Derived> p = SmartPtr<Base>$ 认为是正确的。所以我们需要为这个函数增加约束——用**成员初始化列表**，约束当只有某个隐式转换存在时，才能通过编译。

```
template<typename U>
    SmartPtr(const SmartPtr<U>& other):heldPtr(other.get()){...};
    T* get() const {return heldPtr;}
private:
    T* heldPtr;
```

泛化的构造函数、copy构造函数，均不影响编译器为你再自动声明一份默认的构造函数。所以如果要控制好构造函数，则建议声明一份正常的构造函数和拷贝构造。

- **需要类型转换时为模板定义非成员函数**

和之前讨论过的类似（条款24，，，我没标编号），反正是，在考虑函数的隐式转换的问题。现在是当函数为模板函数式，考虑隐式转换的问题。

```cpp
template<typename T>
const Rational<T> operator* (const Rational<T>& lhs, const Rational<T>& rhs);
// 尽管作为了non-member函数

Rational<int> oneHalf(1,2);
Rational<int> result = oneHalf * 2; // 当Rational为模板类时，这个没法通过编译。
```

解决办法：为与此template相关的函数定义为“class template”内部的友元函数。

```cpp
template<typename T>
class Rational{
public:
    friend  const Rational<T> operator* (const Rational<T>& lhs, const Rational<T>& rhs){
        return Rational(lhs*rhs);
    }
}
```

- **请使用traits classes表现类型信息**

不想看这个了

- **认识template元编程**

C++模板元编程详细教程（之一） c++ 模板元编程-CSDN博客

```cpp
// 用于引导模板全局常量的模板类(用于判断一个类型的长度是否大于指针)
template <typename T>
struct IsMoreThanPtr {
  static bool value = sizeof(T) > sizeof(void *);
};

// 全局模板常量
template <typename T>
constexpr inline bool IsMoreThanPtr_v = IsMoreThanPtr<T>::value;

```

八、定制new 和delete

- **了解new-handler的行为**

当operator new 未获得满足需求的内存之前，会调用new-handler（一个用户指定的错误处理函数），使用方式如下：

```cpp
void outOfMem(){ std::abort();}

int main(){
    std::set_new_handler(outOfMem);
    int* p = new int[1000000000];
}
```

一个良好的new-handler函数必须选择做以下事情：

1. 让更多的内存可被使用——为了使得下一次new 的动作可能成功。做法：程序执行一开始就预留空间，new-handler被调用时释放这个空间。
2. 安装另一个new-handler。让下次的new动作在别的new-handler里面找有没有可用空间。

3. 卸除new-handler。std::set_new_handler(null);
4. 抛出bad_alloc的异常。
5. 不返回。调用abort或exit直接退出。

可以为每个类写自己的new-handler行为，但是要注意调用失败和调用成功时的恢复。

- **了解new和delete的合理替换时机**

为什么要替换编译器提供的new和delete呢?

1. 用来检测运用上的错误。
2. 为了强化效能。编译器提供的new和delete是一个中庸版本，因为考虑到了方方面面。
    1. 为了增加分配和归还速度。
    2. 为了降低缺省内存管理器带来的空间额外开销
    3. 为了弥补缺省分配器中非最佳对齐。
3. 为了收集使用上的统计数据。new和delete对内存的分配方式、顺序、大小、分布。
4. 为了将相关对象成簇集中
5. 为了获得非传统行为。

```
void* operator new (std::size_t size) throw(std::bad_alloc){
    void* p = malloc(size + 2*sizeof(int));
    if(!p) throw bad_alloc();
    ...
    return static_cast<Byte*>(p) + sizeof(int); // 可能导致齐位问题
}

```

齐位问题：c++要求所有operator new返回的指针有适当的对齐（取决于数据类型）

- **编写new和delete**

new包括：无穷循环尝试分配，分配失败则调用new-handler，handler失败则抛出异常bad_alloc。

```
void* operator new (std::size_t size) throw (std::bad_alloc){
    using namespace std;
    if(size == 0) size=1;
    while(true){
        尝试分配size bytes;
        if(分配成功)
            return 一个指针指向分配的内存;
        // 分配失败
        new_handler globalHandle = set_new_handler(0);
        set_new_handler(globalHandle);

        if(globalHandle) (*globalHandler)();
        else throw std::bad_alloc();
    }
}
```

如果写一个类的专属operator new，记得进行size判断，因为可能new的对象是它的派生类，所以可能大小并不是原本期望的Base。

delete要保证删除null指针是安全的。

```
class Base{
public:
    static void operator delete(void* rawMem, std::size_t size) throw();

}
void operator delete(void* rawMem, std::size_t size) throw(){
    if(rawMem == 0) return;
    if(size != sizeof(Base)) {::operator delete(rawMem); return;}
    归还内存。
    return;
}
```

- **写了placement new 也要写placement delete**

operator new 接受的参数除了size_t 外还有其他，则称为placement new。所以placement new 和 placement delete要成对出现，这样子编译器方便调用正确的delete对new的内存进行释放。

```
class Widget{
public:
    // placement new
    static void* operator new(std::size_t size, std::ostream& logStream)
throw(std::bad_alloc);

    static void operator delete(void* pMem) throw();
    static void operator delete(void* pMem, std::ostream& logStream)
throw(); // 对应的placement delete
}

Widget* pw = new (std::cerr) Widget; // 如果new成功而构造失败，则会调用placement
delete正确释放了内存。不会造成泄漏。
delete pw; // 这个调用正常版本
```

考虑名称遮掩问题——class内的new和delete会遮掩掉global的new和delete。

```
// global new
void* operator new(std::size_t) throw(std::bad_alloc);  // normal new
void* operator new(std::size_t, void*) throw();  // placement new
void* operator new(std::size_t, const std::nothrow_t&) throw();  // nothrow
new
```

解决办法：利用继承机制和using声明式。

```
class Widget:public StandardNewDeleteForms{// StandardNewDeleteForms类里包含所
有正常形式的new和delete
public:
    using StandardNewDeleteForms::operator new;
    using StandardNewDeleteForms::operator delete;
    // 写自定义的placement new和delete
    static void* operator new(std::size_t size, std::ostream& logStream)
throw(std::bad_alloc);
    static void operator delete(void* pMem, std::ostream& logStream) throw();
}
```

杂项讨论

- **不要轻忽编译器警告**
- **熟悉TR1在内的标准程序库**

TR1代表"Technical Report 1，宣示着$StandardC++1.1$ 的到来。

C++98标准程序库有：STL、Iostreams、国际化支持、数值处理、异常阶层体系、C89标准程序库。

TR1详细叙述了14个新组件（在std::tr1::下），但其本身只是个文档，为取得它所规范的机能，需要实现这些代码。TR1的14个组件中的10个基于免费的Boost程序库。

- **熟悉Boost**

http://boost.org