Effective C++

引言：

这是一本大师书籍，经典之作。阅读《Effective C++》使我真正开始接触并了解C++。作为引领我入门的一本好书籍，理应备受尊重，加以整理。

-- Brian Wu

PS：

笔记中每个条款不按照原书的条款来，而是我认为关键的、精华的部分，以我的思路整理而成。

一.让自己习惯C++

什么是C++？C++和C有什么区别？C++有什么最基础的准则？

1.

视C++为一个语言联邦：

- C98
- Object oriented （面向对象）
- Template and Generic Programming （模板与泛型编程）
- STL

2.

`const`

- 尽可能使用const: 可以避免变量（常用变量、指针、函数参数类型、类中的常函数）被意外修改
- 使用const 替换 #define: 减少宏函数的定义（尽管tensorflow等众多优秀的开源项目也大量使用宏函数）

3.

确定对象被使用前被初始化

- 最好是在构造函数的初始化列表中进行初始化操作：构造函数的内容是在初始化列表之后执行的，已经不算是初始化操作，并且会降低初始化的效率。由于构造函数中进行赋值的话，调用的是：默认构造函数+赋值函数，而在初始化列表中进行初始化，调用的是拷贝构造函数。**对于大多数类，默认构造函数+赋值函数的效率是小于只调用拷贝构造函数的。**
- 内置类型要手动初始化，不同的平台不能保证对内置类型进行初始化

二.构造/析构/赋值运算

4.

了解C++默默编写并调用了哪些函数

- 编译器默认实现的函数: 默认构造函数、析构函数、copy构造函数、赋值函数
- 对于拷贝构造函数，类内成员有**深拷贝的需求**/类内有**引用成员或`const`成员**，则需要自己编写拷贝构造函数/操作符，而不是把这件事情交给编译器来做。
- 对于析构函数，如果该类有多态需求，请主动将析构函数声明为`virtual`

5.

为多态基类声明`virtual`

- 带有**多态性质的基类**（普通的基类无需也不应该有虚析构函数，因为虚函数无论在时间还是空间上都会有代价）必须将**析构函数**声明为虚函数`virtual`，防止**指向子类的基类指针**（如果一个类有多态的内涵，那么几乎不可避免的会有基类的指针（或引用）指向子类对象）在被释放时只局部销毁了该对象。由于编译器自动生成的析构函数时非虚的，因此需要**显示的声明**。
- 如果一个类型没有被设计成基类，又有被误继承的风险，请在类中声明为`final`

6.

别让异常逃离析构函数

- 析构函数是一个对象生存期的最后一刻，负责许多重要的工作，如线程，连接和内存等各种资源所有权的归还。析构函数执行期间抛出异常是一个非常危险的举动，很有可能会让程序直接崩溃。因此，正确的做法是如果某些操作真的很容易抛出异

常，那么就请把这些操作移到析构函数之外，提供一个普通函数做类似的清理工作，在析构函数中只负责记录，我们需要时刻**保证析构函数能够执行到底**。

7.

绝不在构造和析构的过程中调用`virtual`函数

- **构造函数中虚函数并不是虚函数**，在不同的构造函数中，调用的虚函数版本并不同，因为随着不同层级的构造函数调用时，对象的类型在实时变化。同理，析构函数也一样。因此，不要在基类的构造函数中调用子类的虚函数
- 将构造函数的主要工作抽象成一个`init()`函数以防止不同构造函数的代码重复是一个很常见的做法，然而`init()`函数被构造函数调用时，也不应该包含`virtual`函数。

8.

赋值

- 令operator = 返回一个reference to `* this`：设计接口时一个重要的原则是，**让自己的接口和内置类型相同功能的接口尽可能相似**，所以如果没有特殊情况，就请让你的赋值操作符的返回类型为`ObjectClass&`类型并在代码中返回`*this`。
- 在operator = 中处理自我赋值: 自我赋值看似愚蠢无用但却在代码中出现次数比任何人想象的多得多的操作，这种操作常常需要假借指针来实现，例如：

```
*pa = *pb;                    //pa和pb指向同一对象，便是自我赋值。
arr[i] = arr[j];          //i和j相等，便是自我赋值
```

然而，operator = 在对象重载时管理一定的资源，因此无论是**深拷贝**还是**资源所有权的转移**，原先的内存或所有权一定会被**清空**才能被赋值，如果不加处理，这套逻辑被用在自我赋值上会发生先把自己的资源给释放掉了，然后又把以释放掉的资源赋给了自己的错误。

正确的做法是:

```
SomeClass& SomeClass::operator=(const SomeClass& rhs) {
  DataBlock* pOrg = ptr;
  ptr = new DataBlock(*rhs.ptr);                //如果此处抛出异常，ptr仍然指向之前的内存。
  delete pOrg;
  return *this;
}
```

或者使用**copy and swap** 技术。

9.

复制

- 复制对象时勿忘其每一个成分: 如果你的类有继承，那么在你为子类编写**拷贝构造函数**时一定要格外小心复制基类的每一个成分，这些成分往往是private的，所以你无法访问它们，你应该让子类使用子类的拷贝构造函数去调用相应基类的拷贝构造函数：

```
ChildClass::ChildClass(const ChildClass& rhs) :
BaseClass(rhs) {
    // ...
}
```

- copy构造函数和赋值操作符不要彼此调用: 拷贝构造函数在构造一个对象——这个对象在调用之前并不存在；而赋值操作符在改变一个对象——这个对象是已经构造好了的。因此前者调用后者是在给一个还未构造好的对象赋值；而后者调用前者就像是在构造一个已经存在了的对象

三、资源管理

常见的资源有哪些？如何对资源进行管理？资源管理的目的是什么？

- 常见的资源: **memory、mutual lock、dataset link、file descriptor**等
- 资源管理方法: 以**对象**管理资源（获取和释放）。当我们**构造对象**时资源自动获取，当我们不需要资源时，我们**析构对象**。此即RAII的思想（Resource Acquisition Is Initialization）。
- 资源管理目的: **优化异常处理、提升效率、减少开销**

四、设计与生命

10.

让接口容易被正确使用，不易被误用

- 一个合理的接口，应该尽可能的从**语法层面**并在**编译之时运行之前**，帮助接口的调用者规避可能的风险（限制调用者必须做的事和不能做的事）。
- 一个合理的借口应该表现出与内置类型的一致性

11.

设计class

- 对象该如何创建销毁：包括构造函数、析构函数以及new和delete操作符的重构需求。
- 对象的构造函数与赋值行为应有何区别：构造函数和赋值操作符的区别，重点在资源管理上。
- 对象被拷贝时应考虑的行为：拷贝构造函数。
- 对象的合法值是什么？最好在语法层面、至少在编译前应对用户做出监督。
- 新的类型是否应该复合某个继承体系，这就包含虚函数的覆盖问题。
- 新类型和已有类型之间的隐式转换问题，这意味着类型转换函数和非explicit函数之间的取舍。
- 新类型是否需要重载操作符。
- 什么样的接口应当暴露在外，而什么样的技术应当封装在内（public和private）
- 新类型的效率、资源获取归还、线程安全性和异常安全性如何保证。
- 这个类是否具备template的潜质，如果有的话，就应改为模板类。

12.

函数接口应该以`const`引用的形式传参，而不应该是按值传参

- 按值传参涉及**大量参数的复制**，这些副本大多是没有必要的，尤其是拷贝构造函数设计的是深拷贝而非浅拷贝时，那么拷贝的成本将远远大于拷贝某几个指针。
- 对于多态而言，将父类设计成按值传参，如果传入的是子类对象，仅会对子类对象的父类部分进行拷贝，即**部分拷贝**，而所有属于子类的特性将被丢弃，造成不可预知的错误，同时虚函数也不会被调用。
- **小的类型并不意味着按值传参的成本就会小**。首先，类型的大小与编译器的类型和版本有很大关系，某些类型在特定编译器上编译结果会比其他编译器大得多。小的类型也无法保证在日后代码复用和重构之后，其类型始终很小。

13.

必须返回对象时，不要把返回值写成引用类型

- 如果必须按值返回，那就让他返回值。多一次拷贝也是没办法的事，可以就是指望着编译器来优化。
- 对于C++11以上的编译器，我们可以采用给类型编写**“转移构造函数”**以及使用`std::move()`函数更加优雅地消除由于拷贝造成的时间和空间的浪费。

14.

让成员变量声明为`private`

- 把所有成员变量声明为private的好处有两点: 首先，所有的变量都是private了，那么所有的public和protected成员都是函数了，用户在使用的时候也就无需区分，这就是**语法一致性**；

  其次，对变量的封装意味着，可以尽量减小因**类型内部改变**造成的**类外外代码的必要改动**。(一个封装良好的类在内部产生改动后，对整个项目的影响只应是需要**重新编辑**而无需改动类外部的代码。)

- `public`和`protected`属性**从封装的角度来说**是等价的。一个`public`的成员说明了类的作者决定对类的**第一种客户**（使用类创建对象）不封装此成员，而一个`protected`的成员说明了类的作者对类的**第二种客户**（使用类的子类创建对象）不封装此成员。

15.

成员函数

- 使用`non-member,non-friend`函数替换高颗粒度的成员函数: 当member 函数是无需直接访问private成员，只是若干public函数集成而来的函数 （高颗粒度函数）时，应该尽可能放到类外。如果**高颗粒度函数**设置为类内的成员函数，那么一方面他会破坏类的封装性，另一方面降低了函数的包裹弹性。

```
class WebBrowser {                                  //  一个浏览器类
public:ß
    void clearCache();                          // 清理缓存，直接接触私有成员
    void clearHistory();                        // 清理历史记录，直接接触私有成员
    void clearCookies();                        // 清理cookies，直接接触私有成员

    void clear();                                       // 颗粒度较高的函数，在内部调用上边三个函数，不直接接触私有成员，本条款告诉我们这样的函数应该移至类外
}
```

- 使用操作符时希望操作符的**任意操作数**都可能发生**隐式类型转换**，那么应该把该操作符（加减乘除操作符）重载成non-member函数。因为如果一个操作符是成员函数，那么它的**第一个操作数**（即调用对象）不会发生隐式类型转换，第一个操作数是什么类型，它就调用那个类型对应的操作符。

E.g.: 当`Ratinoal`类的构造函数允许`int`类型隐式转换为`Rational`类型时，`Rational z = x + 2;`是可以通过编译的，但是`Rational z = 2 + x;`却会引发编译器报错，因为由于操作符的第一个操作数不会发生隐式类型转换，所以加号“+”实际上调用的是`2`——一个`int`类型的操作符，因此编译器会试图将`Rational`类型的`x`转为`int`，这样是行不通的。

PS: 如果你想**禁止隐式类型转换**的发生，请把你每一个单参数构造函数后加上关键字`explicit`。

16.

写一个不跑出异常的swap函数

- 一般写swap最普通的方法就是利用中间变量，这种方法在**内置类型**上的赋值绝对不会抛出异常，并且效率很高。但是如果a,b不是内置类型，就会调用类的copy构造函数和assign函数，并且必须是**深拷贝**。这样如果类的成员较多就会造成交换的效率很低。更好的做法就是**直接交换指针**就可以了，相当于交换了两个int(指针都是4字节的)，这就比拷贝这个指针指向的资源要快得多。
- 类内的swap交换内置类型时要调用std命名空间内的swap函数：`using std::swap`，否则就变成递归函数
- 在std命名空间内不能加入新东西，比如重载swap函数

五、实现

17.

尽可能延后变量定义时间

- copy construction的效率 > default construction +assign function，所以最好的做法是直接调用copy construction函数对变量直接进行初始化，而不是先定义，再赋值
- 假如你过早的定义变量，然后在你使用这个变量之前抛出了异常，那么这个变量的构造函数就没有意义而且降低效率。

18.

尽量避免转型

避免返回一个**指针、引用或者迭代器**指向类内的成员

`inline` 函数只是一种申请

- `inline`只是一种申请，编译器会根据具体情况来决定一个函数是否可以执行。比如**递归函数、`virtual`函数、代码较多的函数**，即使你声明了`inline`关键字，编译器也不会将此类函数视为`inline`的函数。

编写异常安全的函数

- 资源不泄漏
- 数据不丢失
- 不抛出异常

19.

将编译依存关系降至最低

- 对于C++类而言，如果它的**头文件**（.h）变了，那么所有这个类的对象所在的文件都要重编，但如果它的**实现文件**（.cpp）变了，而头文件没有变（对外的接口不变），那么所有这个类的对象所在的文件都不会因之而重编。
- 在头文件中用`class`声明外来类，用**指针或引用**代替变量的声明（1）；在源文件中包含外来类的头文件（2）。这样子由于（1），编译可以顺利通过；由于（2），当外来类的头文件改变时，只需要重新编译`#include`这个外来类的complex class的源文件，而无需重新编译所有包含complex class的文件。（外来类的源文件改变时，只需要重新编译外来类，甚至无需重新编译complex class 文件）
- 缺点：指针方式的实现要多分配指针大小的内存，并且每次访问都是间接访问。但一般这种实现对资源和效率的影响通常不是最关键的，因此可以放心的使用

https://www.cnblogs.com/jerry19880126/p/3551836.html

六、继承与面向对象设计

在设计一个与继承有关的类时，有很多事情需要提前考虑：

1. 什么类型的继承？各种类型的继承到底意味着什么？
2. 接口是虚函数还是非虚的？虚函数的本质需求是什么？虚函数是否是必须的呢？有哪些替代选择？
3. 缺省参数如何设计？

这些问题都需要缜密的考虑，避免类的设计错误导致的一连串的麻烦：

1. `public,private,protected`三种类型的继承对应不同的涵义：

a. `public`: 子类 is-a 父类 --> 子类必须涵盖父类的所有特点，必须无条件继承父类的所有特性和接口。
b. `private`: 本质是一种技术封装 --> 和public继承不同，private继承表达的是**“只有实现部分被继承，而接口部分应略去”**的思想。
c. `protected`:

2. 纯虚函数的意思是**“接口一定被继承，实现一定在子类更改”**；虚函数的意思是**“接口一定被继承，但实现可以在子类更改”**；非虚函数的意思是**“接口和实现都必须被继承”**。此即虚函数的本质需求，它告诉我们如何选取接口的类型。

   将纯虚函数、虚函数区分开的并不是在父类有没有实现（纯虚函数也可以有实现），二者本质区别在于父类对子类的要求不同：纯虚函数在于**从编译层面提醒子类主动实现接口**，虚函数则侧重于**给予子类自由度对接口做个性化适配**。

3.

20.

确定你的`public`继承保证了 is-a 关系

- public继承的意思是：**子类是一种特殊的父类**，这就是所谓的“is-a”关系。在使用public继承时，子类必须涵盖父类的所有特点，必须无条件继承父类的**所有特性和接口**。public继承关系不会使父类的特性或接口在子类中退化，只会使其扩充。

21.

避免遮掩继承而来的名称

- 名称在**作用域级别的遮盖是和参数类型以及是否虚函数无关的**，即使子类重载了父类的一个同名，**父类的所有同名函数在子类中都被遮盖**，无论这些重载函数的参数类型、属性类型是否不同（e.g. 父类中const属性重载，参数类型重载）
- 如果想要重启父类中的函数名称，需要在子类有此需求的作用域中（可能是某成员函数中，可能是public 或private内）加上`using Base::foo;`，即可把父类作用域汇总的同名函数（foo() const, foo(int)... ）拉到目标作用域中。

22.

区分接口继承和实现继承

- 在父类中声明纯虚函数，是为了**强制子类拥有一个接口**，并**强制子类提供一份实现**。
- 在父类中声明虚函数，是为了**强制子类拥有一个接口**，并**为其提供一份缺省实现**。

- 在父类中声明非虚函数，是为了**强制子类拥有一个接口以及规定好的实现**，并**不允许子类对其做任何更改**。

  绝不重新定义继承而来的非虚函数:

  - 如果你的函数有多态调用的需求，一定记得把它设为虚函数，**否则在动态调用**（基类指针指向子类对象）的时候是不会调用到子类重载过的函数的，很可能会出错。因为在多态的动态调用中，**只有虚函数是动态绑定，非虚函数是静态绑定的**。指针（或引用）的静态类型是什么，就调用那个类型的函数，和动态类型无关。因此在子类中的修改无法被多态的动态调用识别，造成出错。

  绝不重新定义继承而来的缺省参数值:

  - 若缺省参数值在非虚函数中，则根据上一条，不应该重新定义
  - 若缺省参数值在虚函数中，则为了避免子类用户犯错，设计时**虚函数要从始至终保持没有缺省参数值**。由于缺省参数值是属于**静态绑定**的（为了提高运行时效率），而虚函数属于动态绑定。虚函数在大多数情况是供动态调用，而在动态调用中，子类做出的缺省参数改变其实并没有生效，反而会引起误会，让调用者误以为生效了。

23.

通过**类的复合**塑膜出 has-a 关系

- 类的复合表示一个类的**对象**可以作为另一个类的**成员**
- 复合用于以下两种情况：
  a. 某一个类拥有另一个类对象作为一个属性时（E.g. `car` class has-a `turbo` class）
  b. 某一个类拥有另外一个类帮助自身实现（E.g. `queue` class has-a `stack` class）

24.

明智而审慎的使用`private`继承

- private继承表达**“通过某工具类实现另一个类”**。(和类的复合的第二条一致)。在private继承下，**父类的所有成员都转为子类私有变量**，不提供对外访问的权限（接口、变量都不对外暴露），外界也无需关心子类内有关父类的任何细节。
- 如何在类的复合与private继承中做选择:
  - 尽可能用复合，除非必要，不要采用private继承。
  - 当我们需要对工具类的某些方法（虚函数）做重载时，我们应选择private继承，这些方法一般都是工具类内专门为继承而设计的调用或回

调接口，需要用户自行定制实现（而复合不能提供这一编译器层面的约束，工具类只是复合类中的一个对象而已）。

25.

明智而审慎的使用多继承

- 原则上不提倡使用多继承：**多继承**（子类继承多个父类）可能会引起多父类共用父类（多个父类公用一个共同的祖父类），导致在底层子类中出现多余一份的共同祖先类的拷贝。

  为了避免这个问题C++引入了**虚继承**，但是虚继承会使子类对象变大，同时使成员数据访问速度变慢，这些都是虚继承应该付出的代价。因此尽可能地避免虚继承。虚基类中应尽可能避免存放数据。

七、模板与泛型编程

C++ templates的最初发展动机很直接：让我们得以建立“类型安全”的容器如 `vector`，`list`和`map`。然而当愈多人用上`templates`，他们发现`templates`有能力完成愈多可能的变化。容器当然很好，但泛型编程使得**写出的代码和其所处理的对象类型彼此独立**，是一种更好的选择。

C++ template机制自身是一部完整的图灵机 （Turing-complete）：它可以被用来计算任何可计算的值。于是导出了模板元编程（Template MetaProgramming），创造出**“在C++编译器内执行并于编译完成时停止执行”**的程序。TMP的作用有两个：

1. 它让某些事情更容易。如果没有它，那些事情将是困难的，甚至不可能的。例如可将工作由运行期移动到编译期完成，造成更高的执行效率（占用内存小，运行速度快）和更早的侦测错误。（缺点为编译时间长）
2. 第二，使用templates可以让代码更简洁，避免重复的书写一组重载函数。（缺点为代码不易理解）

26.

隐式接口与编译期多态

- `class`和`templates`都支持接口（interfaces）和多态（polymorphism）。
- 对`class`而言，**接口是显式的（explicit）**，以函数签名为中心。对 `template`参数而言，**接口**是隐式的（implicit），奠基于**有效表达式**。
- 对`class`而言，**多态**则是通过`virtual`函数发生于运行期。对 `template`参数而言，**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**。

27.

了解`typename`的双重意义

- 为了避免`class`在这两个地方的使用可能给人带来混淆，所以引入了 `typename` 这个关键字。在模板定义语法中关键字 `class`与 `typename`的作用完全一样，可以互换
- `typename`另外一个作用为：使用嵌套从属类型(nested depended name)时，告诉c++ 编译器，`typename`后面的字符串为一个类型名称，而不是成员函数或者成员变量。E.g.

```
template<class T>
void MyMethod( T myarr )
{
    typedef typename T::LengthType LengthType;
    LengthType length = myarr.GetLength;
}
```

这个时候如果前面没有 `typename`，编译器没有任何办法知道 `T::LengthType`是一个类型还是一个成员名称(静态数据成员或者静态函数)，所以编译不能够通过。因为C++有个规则：如果解析器在template中遭遇个嵌套从属名称，它便假设这名称不是个类型，除非你告诉它是。所以*缺省情况下嵌套从属名称不是类型*。

PS：嵌套从属名称

template内出现的名称如果相依于某个template参数，称之为**从属名称**。如果从属名称在class内呈嵌套状，我们称它为**嵌套从属名称**。

E.g.: `C::const_iterator`就是这样一个名称。实际上它还是个**嵌套从属类型**名称，也就是个嵌套从属名称并且指涉某类型。int是一个并不倚赖任何template参数的名称。这样的名称是所谓非从属名称。

28.

模板特化与偏特化

- Template Specialization: 相对于模板泛化而言（Template Generalization），通过"给模板中的**所有**模板参数一个**具体的类**"的方式来实现的。E.g.

```
template<class Window, class Controller>
    class Widget
    {
      ... 泛化实现代码 ...
    };

template<>     //注意:template后面的尖括号中不带任何内容;
    class Widget<ModalDialog, MyController>
    {
      ... 特化实现代码 ...
    };
```

以后定义了`Widget<ModalDialog, MyController>`对象时,编译器就会使用上述的特化定义,如果定义了其它泛型对象（E.g. `Widget<A, B>`）,那么编译器就是用原本的泛化定义。这就是模板的特化。注意此时`template`后面的尖括号中不带任何内容。

- Partial Template Specialization: 模板偏特化则是通过"给模板中的**部分模板参数**以具体的类,而留下剩余的模板参数仍然使用原来的泛化定义"的方式来实现的。E.g.

```
template<class Window>
    class Widget<Window, MyController>
 //MyController是具体的类,是特化定义;
    {
      ... 偏特化实现代码 ...
    };
```

模板的偏特化能力很强大：当你实例化一个模板时,编译器会把目前存在的偏特化模板和全特化模板做比较，并找出其中最合适、最匹配的实现。这样，提高了模板的灵活性。但是模板的偏特化机制不能用在函数身上（不论成员函数还是非成员函数）。由于函数偏特化可以通过函数重载替换实现。

29.

学习处理模板化基类内的名称

- C++拒绝寻找模板化基类（base class templates）内的名称（函数名，成员变量名...）。因为它知道`base class templates`有可能被特化，而那个特化版本可能不提供和一般性`template`相同的接口。E.g.

```
class CompanyA
```

```
{
    public:
        void sendCleartext(const std::string& msg);
        void sendEncrypted(const std::string& msg);
};

class CompanyB
{
    public:
        void sendCleartext(const std::string& msg);
        void sendEncrypted(const std::string& msg);
};

class MsgInfo {  ... };

template<typename Company>
class MsgSender
{
    public:
        void sendClear(const MsgInfo& info)
        {
            std::string msg;
            Company c;
            c.sendCleartext(msg);
        }
        void sendSecret(const MsgInfo& info)
        { ... }   // 调用sendEncrypted
};

template<typename Company>
class LoggingMsgSender : public MsgSender<Company>
{
    public:
        void sendClearMsg(const MsgInfo& info)
        {
            sendClear(info);
        }
};
```

这里`MsgSender`为模板化基类，因此在`LoggingMsgSender` class中`sendClearMsg()`函数会编译出错，因为此时并不知道模板化基类中是否有此函数。在本例中是有的，然而在下面这个模板特化的例子中是没有的：

```
class CompanyZ
{
    public:
        // 不提供sendCleartext函数
        void sendEncrypted(const std::string& msg);
};

template<>      // 一个全特化
class MsgSender <CompanyZ>
{
    public:
        void sendSecret(const MsgInfo& info)
        { ... }   // 调用sendEncrypted
};
```

- 以下方法可以使得C++寻找模板化基类（base class templates）内的名称：

  a. 使用`using`声明式

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company>
{
    public:
        using MsgSender<Company>::sendClear;  // 告诉编译器，请假设sendClear位于base class内
        void sendClearMsg(const MsgInfo& info)
        {
            sendClear(info);
        }
};
```

  b. 明确指出被调用的函数位于`base class`内

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company>
{
    public:
        void sendClearMsg(const MsgInfo& info)
        {
            MsgSender<Company>::sendClear(info);
        }
        ...
};
```

https://www.cnblogs.com/yyxt/p/4821359.html

30.

将与`template`无关的代码抽离到模板外

- 如果将与模板无关的代码也放入模板函数或者类中，那么在模板具象化时就会生成重复的代码，导致代码膨胀的问题
- 正确的做法是：
    - **函数模板**中与参数无关的代码包装成单独的函数
    - **类模板**中与参数无关的代码放到父类中

31.

需要类型转换时请为模板定义非成员函数

- 模板的运行需要进行模板推算得到模板参数类型，而在**模板参数推导过程中从不将隐式类型转换函数考虑在内**，这也是合理的因为你没法根据参数类型推导出模板参数的类型。E.g.

```
template <typename T>
const Rational<T> operator*(const Rational<T>& lhs, const
Rational<T>& rhs) {……}

Rational<int>   oneHalf(1, 2);
Rational<int>    ret = oneHalf*2;
```

`operator*`函数的两个模板参数T的类型要根据传入的参数类型进行确认。第一个参数因为是`oneHalf`，其本身就是`Rational<int>`类型，因此第一个参数的类型中的`T`很容易进行推理，但是第二个传入的参数是`int`，编译器无法根据这个`int`参数推导出第二个参数的类型`T`。

- 解决方法为在类中声明一个友元函数（在类中定义的友元函数都被视为非成员函数），先具象化这个友元函数（确定`T`），然后就可以进行隐式类型转换了：

```
friend const Rational operator*(const Rational& lhs, const
Rational& rhs) {
   return
(lhs.numrator()*rhs.numrator()/lhs.denominator()*rhs.denomin
ator());
}
```

由于此函数是在模板类的内部，因此当`oneHalf`对象生成之后，`T`就被确定为`int`，那么`operator*`函数的参数和返回值中的`T`也均是确定的了，因此此时第二个参数会发生隐式类型转换（int --> Rational），成功实现编译。

32.

请使用`traits classes`表现类型信息

- Traits 广泛用于标准程序库。**TR1**导入许多新的`traits classes`用以提供类型信息，包括`is_fundamental<T>`（判断T是否为内置类型），`is_array<T>`（判断T是否为数组类型），以及`is_base_of<T1, T2>`（T1和T2相同，抑或T1是T2的base class）。总计TR1一共为标准C++添加了50个以上的traits classes。
- `traits classes`使得“类型相关信息”在编译期可用。它们以templates（针对用户自定义类型）和“templates特化”（针对指针）完成实现。E.g.

```
template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d
std::random_access_iterator_tag)
{
    iter += d;     // random access 迭代器
}

template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d

 std::bidirectional_access_iterator_tag)
{
```

```
    .....
}
template<typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d
                        std::input_iterator_tag)
{
     .....
}
// 有了这些doAdvance重载版本，advance需要做的只是调用它们并额外传递一个对象，后者必须带有适当的迭代器分类。于是编译器运用“重载解析机制”（发生于编译期间）调用适当的实现代码：
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    doAdvance(iter, d,typename
std::iterator_traits<IterT>::iterator_category()
    );
}
```

首先建立一组重载函数或函数模板（`doAdvance`），彼此间的差异只在于各自的traits参数。令每个函数实现码与其接受之traits信息相应和。此后，建立一个控制函数或函数模板（`advance`），它调用上述那些`doAdvance`函数并传递traits class所提供的信息。

- 整合重载技术后，`traits classes`有可能在编译期对类型执行`if....else`测试。(利用编译期完成的“重载解析机制”)。当你重载某个函数f，你必须详细叙述各个重载件的参数类型，那么编译器便会触动重载解析机制，在编译期间匹配最适当的那个重载函数。通过此你可以获得类型相关的信息。

八、定制 new 和 delete（条款49~52）

33.

malloc, new, delete

- `malloc` 和 `new`:

    a. 一般的g++编译器实现的`new`的调用过程如下：**new operator->operator new->malloc**。即new 通过 调用malloc实现

    malloc分配的内存位于**堆**上，new分配的内存位于‘自由存储区’，自由存储区是C++中一个抽象的概念，有别于堆。（operator new可以被重载，所以通过operator new分配的内存未必都在堆上）

    b. new和delete会调用类的`constructor`和`destructor`，malloc则不会
    c. malloc分配内存失败会返回`nullptr`，而new则会直接抛出异常。另外new还可以通过`set_new_handler`设置分配失败时执行的逻辑，malloc则没有提供这样的使用方式
    d. malloc返回的指针类型是`void*`，需要手动强转为需要的类型，而new不需要如此，new的返回类型是**类型安全的**

- `free`和`delete`：delete时会先调用类的destructor，再调用free释放这个对象。
- malloc分配的时候要指定分配内存的大小，利用free释放的时候却不需要指定大小，那么free是怎么知道该释放多大内存呢？在实现上，malloc分配的内存大小会比传入的值稍大，**多余的内存**用来存储分配的内存大小等额外的关于这段内存的信息（用户是无权访问的，应该是由操作系统管理），然后在free的时候找到这个内存的大小，就可以释放对应的内存了。另外因为delete内部实现也是调用free，因此delete也不用传入内存大小。

PS：

此在C++11之后大部分以及解决，变得不再重要。有关于`new`,`delete`,`malloc`等区别见：https://www.cnblogs.com/deepllz/p/9927807.html

九、杂项讨论（条款53~55）