# Effective C++笔记

——参考笔记：arkingc/note/ Effective C++

# 1 自己习惯 C++

## 规则 1 将 C++视为语言联合体

1. C++不是单一的语言，包含了四种子语言：
   - C
   - 面向对象的 C++
   - 模板 C++：C++泛型编程以及模板元编程的基础
   - STL：是一个模板库，将容器、迭代器、算法和函数对象整合在一起

## 规则 2 用 const、enum 和 inline 取代#define

1. 通俗规则：用编译器取代预处理器
2. #define 定义的变量的缺点：
   #define ASPECT_RATIO 1.653
   - 在编译器使用 ASPECT_RATIO 之前，该名字就被预处理器消除，该名字可能没被加入符号表中，这对于调试有很大影响，因为错误信息中很可能用 1.653 代替 ASPECT_RATIO
   - #define 会将程序中所有 ASPECT_RATIO 替换成 1.653，这会产生很多拷贝
3. 使用 const 替换宏定义有两点注意：
   - 关于常量指针的定义，通常应该定义为 const 指针
   - 类内的常量：为了将一个常量的作用范围限制在一个类内，必须将它作为一个成员，为了确保它最多只有一个常量拷贝，必须把它声明为 static。只要不是取该常量的地址，可以只声明并使用而不提供定义，如果需要取该常量的地址，需要提供定义（如果有类内初始值，则类外可以不提供，注意类外定义时不需要 static）
4. 特殊：当在类的编译期需要类中的常量的值时，如果有的编译器禁止给类内的静态整型常量赋初值，可以使用 the enum hack，即构建一个不限定作用域的 enum。该情况更像#define，因为此时无法获取其引用或指针。该项技术也被用在模板元编程中
5. #define 函数的缺点：
   ```
   #define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
   int a = 5, b = 0;
   CALL_WITH_MAX(++a, b);      // a 被累加了两次
   CALL_WITH_MAX(++a, b+10);   // a 被累加了一次
   ```
   - 必须为宏函数中每一个参数加上括号
   - 前置递增/减和后置递增/减在使用中会存在问题
6. 使用内联函数的模板来替代#define，可解决上述问题

总结：

- 对于单纯常量，最好以 const 对象或 enums 替换＃define。
- 对于形似函数的宏 (macros) ，最好改用 inline 函数替换＃define。

## 规则 3 只要可能就用 const

1. const 的位置

- 可以在类型前也可以在类型后
  const Widget *pw;
  Widget const *pw;

2. 常见的 const 声明：
   - 将 operator*的返回值声明为 const
   - 将成员函数声明为 const 的优点
     - 可以让程序员知道哪个函数可以改变对象
     - 可以和 const 对象一起工作
3. const 成员函数
   - 二进制位常量性（bitwise constness）：一个成员函数，当且仅当它不改变对象的任何数据成员时（static 除外），则这个函数就是 const 的但：
   - 如果对于一个类来说，其存储的是一个指向某个区域的指针，即使将该类的对象声明为 const，其 const 成员函数返回的值也可能被改变，因为 const 只对该类的成员有效，而对所指向的某个区域无效
   - 逻辑常量性（logical constness）：一个 const 成员函数可能会改变调用它的对象中的一些数据，但是只能用客户无法察觉的方法，此时可以将某些成员声明为 mutable
4. 一般在 const 和非 const 的成员函数中，在非 const 成员函数中调用同名的 const 成员函数来完成相同工作(需要有两次强制转换,一次是 static_cast 将 this 转为 const，另一次是 const_cast 将 const 转回非 const)，而不会在 const 成员函数中调用非 const 来完成，因为 const 成员函数就是保证不改变数据

## 规则 4 确保对象在使用前被初始化

1. 原则：永远在使用对象之前先将它初始化
2. 对于类来说：
   - 确保每一个构造函数都将对象的每一个成员初始化，并且是在成员初始化了列表中进行初始化，而不是进行赋值
   - 如果成员变量是 const 或引用时，在成员初始化列表中一定要有初值
   - 如果一个类有多个构造函数，而成员变量又比较多时，可以将“赋值与初始化一样好”的成员变量（通常是内置类型）移到一个 private 函数完成，供所有构造函数调用
   - 类中的成员初始化按照类中声明的顺序而来
3. 非局部静态对象（全局对象，namespace 内的对象，类内的 static 或 file 作用域内的声明为 static 的对象）初始化次序是不确定的，如果在跨编译单元之间使用，例在一个文件中声明的 FileSystem 对象，需要在另一个类中使用，一般需要在该类中以函数构建该类的静态对象，然后返回该对象的引用，再在需要的地方调用该函数

```
class FileSystem{
public:
    std::size_t numDisks() const;
};

// 这不是类内定义的函数，是类外定义的专属函数
FileSyetem &tfs()
{
    static FileSystem fs;
    return fs;
}
extern FileSystem &tfs();

class Directory{ … };
Directory::Directory(params)
{
    std::size_t disks = tfs().numDisks();
}

// 这不是类内定义的函数，是类外定义的专属函数
Directory &tempDir()
```

```
{
    static Directory td;
    return td;
}
```

# 2 构造/析构/赋值

## 规则 5 了解 C++默默编写并调用哪些函数

1. 默认生成的函数都是 public 且 inline 的，注意编译器生成的析构函数是非虚函数，除非该类的基类的析构函数是虚函数
2. 一般情况下，编译器会为类合成下列函数：
   - **default 构造函数**
   - **copy 构造函数**：编译器生成的版本只是单纯地将来源对象的每一个 non-static 成员变量拷贝到目标对象
   - **copy assignment 操作符**：编译器生成的版本只是单纯地将来源对象的每一个 non-static 成员变量拷贝到目标对象
   - **析构函数**：编译器生成的版本是 non-virtual 的
3. 以下情况编译器不会合成 copy assignment 操作符：
   - **含有引用成员**：原因在于这种情况下，赋值的目的不明确。是修改引用还是修改引用的对象？如果是修改引用，这是被禁止的。因此编译器干脆拒绝这样的赋值行为
   - **含有 const 成员**：const 对象不应该修改
   - **父类的 copy assignment 操作符被声明为 private**：无法处理基类子对象，因此也就无法合成
4. 当类的基类的拷贝控制函数是 private 时，派生类不会合成拷贝控制函数，因为无法对派生类中的基类部分进行处理

## 规则 6 若不想使用编译器自动生成的函数，就该明确拒绝

1. 为了禁止使用编译器合成的版本，将合成的函数声明为 private 并且不进行实现即可
2. 为了拒绝对象拷贝，可以使用以下方法：
   - **将 copy 构造函数或 copy assignment 操作符声明为 private，并且不定义**
     - 这种情况下 member 函数和 friend 函数还是可以调用，如果 member 函数或 friend 函数中执行了复制，会引发链接错误
   - **使用一个基类，在基类中将 copy 构造函数或 copy assignment 操作符声明为 private，并且继承这个基类**
     - 这样可以**将链接错误移至编译期**，因为尝试拷贝时，编译器会试着生成一个 copy 构造函数和一个 copy assignment 操作符，这些函数的“编译器合成版”会尝试调用其基类的对应兄弟，而那些调用会被编译器拒绝，因为 private

     ```
     class Uncopyable{
     protected:
         Uncopyable() {};
         ~Uncopyable() {};
     private:
         Uncopyable(const Uncopyable&);              // 阻止拷贝
         Uncopyable &operator=(const Uncopyable&);
     };
     ```
   - 在 C++11 中，使用 **delete** 声明即可

## 规则 7 为多态基类声明 virtual 析构函数

1. 如果不声明：基类指针指向派生类时，如果使用 delete 时，只能释放派生类中**基类**

部分的成员，派生类自身的数据成员释放不了

2. 原则：只有当类内含至少一个虚函数时，才为它声明虚析构函数
3. class 不用作基类时，不要将析构函数声明为 virtual：virtual 会引入虚函数指针，这会增加空间开销，使得类无法被 C 函数使用，从而不再具有移植性
   - 实现细节：一旦在类中含有至少一个虚函数，该类就会建立一个虚函数表指针，该表中的指针指向类中的虚函数，这会增加类的内存大小
4. 纯虚函数：一般需要给纯虚的析构函数提供一个定义

## 规则 8 别让异常逃离析构函数

1. 析构函数绝对不要跳出异常，如果一个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（不是个好主意）或结束程序
   - 如果析构函数吐出异常，程序可能过早结束（比如某个函数调用发生异常，在回溯寻找 catch 过程中，每离开一个函数，这个函数内的局部对象会被析构，如果此时析构函数又抛出异常，前一个异常还没得到处理又来一个，因此一般会引起程序过早结束）。异常从析构函数中传播出去，可能会导致不明确的行为
2. 解决办法：
   - 在析构函数中 catch 异常，然后调用 abort 终止程序。通过 abort 抢先置“不明确行为”于死地
   - 在析构函数中 catch 异常，然后记录该失败，即吞掉异常（通常是个坏主意，因为这样压制了“某些动作失败”的重要信息。但是也比负担“草率结束程序”或“不明确行为带来的风险”好）
   - 重新设计接口，类中提供一个普通成员函数执行可能抛出异常的操作，让客户能够在析构前主动调用可能引起异常的函数，然后析构函数中使用一个 bool 变量，根据用户是否主动调用来决定析构函数中是否应该调用可能引起异常的函数，让客户拥有主动权（如果客户没有主动调用，那么当发生异常时也不应该抱怨，因为已经给出了客户自己处理异常的机会）

```
class DBConn {
public:
void close() // 供客户使用的新函数
{
    db.close();
    closed= true;
}

~DBConn()
{
    if (! closed) {
    try { // 关闭连接（如果客户不那么做的话）
        db.close();
    }
    catch (...) {
        制作运转记录,记下对 close 的调用失败; // 如果关闭动作失败，记录下来并结束程序或吞下异常
    }
}

private:
    DBConnection db;
    bool closed;
};
```

## 规则 9 绝不在构造函数和析构过程中调用 virtual 函数

1. 该条款与 Java 和 C#不同
2. 如果希望在继承体系中根据类型在构建对象时表现出不同行为，可以会想到在基类的构造函数中调用一个虚函数：

```
class Transaction {                    // 所有交易的基类
public:
```

```
    Transaction(){
        ...
        logTransaction();
    }
    virtual void logTransaction() const = 0; // 做出一份因类型不同而不同的日志记录
};

class BuyTransaction: public Transaction {       // 派生类
public:
    virtual void logTransaction() const;
};

class SellTransaction: public Transaction {      // 派生类
public:
    virtual void logTransaction() const;
};
```

但是最终调用的 virtual 函数都是基类的版本。

3. **在派生类对象的基类构造期间，对象的类型是基类而不是派生类**，在派生类的构造函数执行之前不会成为一个派生类对象
4. 同样的道理也适用于析构函数。**一旦派生类析构函数开始执行，对象内的派生类成员变量便呈现未定义值**。进入基类析构函数后，对象就成为一个基类对象。
5. 在构造函数和析构函数中调用的函数也不应该调用虚函数
6. 常用解决方法：在类内定义一个 private static 函数，在成员初始化列表中调用该函数将派生类信息传递给基类

## 规则 10 令 operator=返回一个*this 的引用

1. 这是为了实现“连锁赋值”。这个协议除了适用于 operator=，还适用于+=、-=、*=
2. 这只是个协议，并无强制性，如果不遵循，代码一样可通过编译

## 规则 11 在 operator=中处理“自赋值”

1. 常用处理方法：
   - 证同测试：该种方法可以处理自赋值的问题，但是不具备异常安全性
   - 拷贝备份：既可以处理自赋值也具有异常安全性
   - copy and swap 技术：最好的方法

2. 考虑如下 Widget 类：

```
class Bitmap {...};
class Widget{
    ...
private:
    Bitmap *pb;
};
```

下面的 operator=实现是一份不安全的实现，在自赋值时会出现问题：

```
Widget&
Widget::operator=(const Widget& rhs){
    delete pb;                  // stop using current bitmap
    pb = new Bitmap(*rhs.pb);   // start using a copy of rhs's bitmap
    return *this;               // see Item 10
}
```

要处理自赋值，可以有以下几种方式：

**1. 在开头添加“证同测试”**

```
1.  Widget& Widget::operator=(const Widget& rhs){
2.      if (this == &rhs) return *this;
3.      delete pb;                  // stop using current bitmap
4.      pb = new Bitmap(*rhs.pb); // start using a copy of rhs's bitmap
5.      return *this;               // see Item 10
6.  }
```

这样做虽然能处理自赋值，但不是异常安全的，如果 new 时发生异常，对象的 pb 将指向一块被删除的内存

**2. 通过确保异常安全来获得自赋值的回报**

```
1.  Widget& Widget::operator=(const Widget& rhs){
2.      Bitmap *pOrig = pb;             // remember original pb
3.      pb = new Bitmap(*rhs.pb);       // make pb point to a copy of *pb
4.      delete pOrig;                   // delete the original pb
5.      return *this;
6.  }
```

现在，如果 new 失败，pb 会保持原状。同时也能处理自赋值。如果担心效率可以在开头加上“证同测试”。但是 if 判断也会引入开销，因此需要权衡自赋值发生的频率

**3. 使用 copy and swap 技术**

```
1. //参数为 pass by reference
2. Widget& Widget::operator=(const Widget &rhs){
3.     Widget temp(rhs);
4.     swap(temp);              // swap *this's data with
5.     return *this;            // the copy's
6. }
7. //参数为 pass by value
8. //这种方式的缺点是代码不够清晰，但是将“copying 动作”从函数本体内移至“函数参数构造阶段”
9. //却可令编译器有时生成更高效的代码
10.Widget& Widget::operator=(Widget rhs){
11.    swap(rhs);               // swap *this's data with
12.    return *this;            // the copy's
13.}
```

## 规则 12 复制对象时勿忘其每一个成分

1. 当编写拷贝构造函数和重载赋值运算符时，确保(1)**复制所有局部成员变量**(2)并且**调用所有的基类内的适当的拷贝函数**
2. 拷贝构造函数不应该调用赋值运算符函数，反之也不行。如果有相同的操作，最好是定义一个 private 成员函数函数供两个函数调用
3. 拷贝构造函数
   - **非继承中**：当为类添加一个新成员时，copy 构造函数也需要为新成员添加拷贝代码。否则会调用新成员的默认构造函数初始化新成员
   - **继承中**：在派生类的 copy 构造函数中，不要忘记调用基类的 copy 构造函数拷贝基类部分。否则会调用基类的默认构造函数初始化基类部分
4. 拷贝赋值运算符
   - **非继承中**：当为类添加一个新成员时，copy 赋值运算符中也需要为新成员添加赋值代码，否则新成员会保持不变
   - **继承中**：在派生类的 copy 赋值运算符中，不要忘记调用基类的 copy 赋值运算符，否则基类部分会保持不变

# 3 资源管理

## 规则 13 以对象管理资源

1. 手工 delete 一个是需要时刻记住 delete，增加编码负担，另一个是即使明确 delete，在 delete 之前控制流可能发生改变从而还是会造成资源泄露
2. 因此，一个好的办法是使用对象管理资源，包括下列两个关键想法：
   - **获得资源后立刻放进管理对象**：“以对象管理资源”的观念常被称为“资源取得

时机便是初始化时机(RAII)”

- **管理对象运用析构函数确保资源被释放**

3. 获得资源后立刻放进管理对象内，通常是使用 shared_ptr 智能指针
4. 对于对象管理资源，需要**注意对象的复制行为**：例如，复制逻辑可能是**多个对象管理相同的资源**，那么析构时就会重复 delete。因此，如果是这种复制逻辑，那么应该引入**引用计数**，析构时根据引用计数决定是否 delete；否则，一个资源就应该只由一个对象来管理，那么复制时就原来对象管理的资源就应该修改成 null，而复制所得的新对象将取得资源的唯一拥有权（如 auto_ptr）
5. 智能指针中的析构函数是使用 delete 来释放资源的，对于数组，标准库中没有特定的类来管理，但是 Boost 库中的 scoped_array 和 shared_array 类可以管理
6. C++没有特别针对”动态分配数组“而设计的类似 auto_ptr 或 tr1::shared_ptr 那样的东西，甚至 TR1 中也没有。那是因为 vector 和 string 几乎总是可以取代动态分配而得的数组。因此当需要动态分配数组时，提倡使用 vector

## 规则 14 在资源管理中小心 copying 行为

**并非所有资源都是动态内存**，除此之外还有锁等资源，也应该通过“对象管理资源”来确保获取资源后能够正确的释放，**根据资源的类型，和不同的需求，可能需要定义不同的 copy 行为**：

1. **禁止复制**：比方说锁资源，管理锁资源的对象复制通常并不合理。因此应该禁止这类对象的复制，可以通过继承一个 copying 操作被声明为 private 的基类来禁止复制，这点在条款 06 中有提到
2. **对底层资源使用“引用计数法”**：如果希望保有资源，直到它的最后一个使用者（某对象）被销毁。这种情况下复制 RAII 对象时，应该将资源的“被引用数”递增。tr1::shared_ptr 便是如此（当资源引用计数减为 0 时，如果不希望删除资源，比方说锁资源，可以使用 shared_ptr 的“删除器”，即自定义删除函数）
3. **复制底部资源**：这种情况下，希望在复制 RAII 对象时，同时复制其关联的底层资源。展现出一种“深拷贝”的行为，（例如 string 类）
4. **转移底部资源的拥有权**：如果希望任一时刻一个资源只由一个 RAII 对象管理，那么在复制 RAII 对象时，应该实现拥有权的“转移”，原 RAII 对象拥有的资源设为 null（如 auto_ptr）（unique_ptr）

## 规则 15 在资源管理类中提供对原始资源的访问

1. 资源管理的类中一般要求访问原始资源，所以一般需要提供一个“取得其所管理资源”的方法
2. 取得 RAII 对象所管理资源的办法可以通过显式转换或隐式转换：
   - 显式转换(比较安全，但不易用)：如 shared_ptr 的 get()方法
   - 隐式转换(比较易用，但不安全)：如 shared_ptr 的 operator*和 operator->

如果通过实现隐式转换（比如，实现 operator()）来提供对元素资源的访问，可能不安全：

```
//以下，Font 是一个 RAII 对象，FontHandle 是一个原始资源
Font f1(getFont());
...
FontHandle f2 = f1; //原意是想使用 Font，复制一个 RAII 对象
```

在上面的例子中，如果实现了隐式转换，底层资源会被复制，如果 f1 销毁，f2 会成为“虚吊的”（dangle）

是否该提供一个显式转换函数将 RAII 转换为其底层资源，或是应该提供隐式转换，答案主要取决于 RAII 被设计执行的特定工作，以及它被使用的情况

## 规则 16 成对使用 new 和 delete 时要采取相同形式

**如果在 new 表达式中使用[]，必须在相应的 delete 表达式中使用[]，如果在 new 表达式中不使用[]，则不要在相应的 delete 表达式中使用[]**

当使用 new 和 delete 时，发生 2 件事

- new
  1. 内存被分配出来（通过名为 operator new 的函数）
  2. 针对此内存会有一个（或更多）构造函数被调用
- delete
  1. 针对此内存会有一个（或更多）析构函数被调用
  2. 内存被释放（通过名为 operator delete 的函数）

单一对象的内存布局一般而言不同于数组的内存布局。更明确地说，数组所用的内存通常还包括“数组大小”的记录，以便 delete 知道需要调用多少次析构函数。

**当使用 delete 时，唯一能够让 delete 知道内存中是否存在一个“数组大小记录”的办法是，由你来告诉它。即加上[]，delete 便认为指针指向一个数组，否则它便认为指针指向单一对象**

因此，应该像这样使用 new 和 delete

```
std::string* stringPtr1 = new std::string;
std::string* stringPtr2 = new std::string[100];
...
delete stringPtr1;
delete [] stringPtr2;
```

- 如果对 stringPtr1 使用“delete[]”形式，结果未定义，但不太可能让人愉快。假设内存布局上，delete 会读取若干内存并将它解释为“数组大小”，然后开始多次调用析构函数，浑然不知它所处理的那块内存不但不是个数组，也或许并未持有它正忙着销毁的那种类型的对象
- 如果没有对 stringPtr2 使用“delete[]”形式，结果亦未定义，但可以猜想可能导致太少的析构函数被调用。犹有进者，这对内置类型如 int 者亦未定义，即使这类类型并没有析构函数

**因此，如果调用 new 时使用了[]，必须在对应调用 delete 时也使用[]；如果调用 new 时没使用[]，那么也不该在对应调用 delete 时使用[]**

这点在 typedef 中尤其需要注意：

```
typedef std::string AddressLines[4];//每个人的地址有 4 行，每行是一个 string

std::string *pa1 = new AddressLines;       //就像 new string[4]一样

delete pa1;            //错误！行为未定义
delete [] pa1;         //很好
```

为避免这类错误，最好尽量不要对数组形式做 typedef 动作

## 规则 17 以独立语句将 new 得到的对象存入智能指针

1. 对于 new 生成的指针一般需要存入 shared_ptr 中，但是有一个问题就是在将该智能指针传入函数实参的时候，一般是生成一个智能指针对象之后再传入

   int priority();

   void processWidget(std::shared_ptr<Widget> pw, int priority);

   错误调用：

   processWidget(std::shared_ptr<Widget>(new Widget), priority());

   该调用出现的问题是：由于 std::shared_ptr<Widget>(new Widget)和 priority()传递给实参时的顺序未知，如果是下列顺序：

   - new Widget
   - priority()
   - std::shared_ptr<Widget>()

   如果 priority()调用时出现错误，会导致 new Widget 生成的**内存泄漏**，也就是说，在“资源被创建”和“资源被转换为资源管理对象”两个时间点之间有可能发生异常干扰。

解决方法：**使用独立语句将 new 产生的对象存入智能指针**

```
std::shared_ptr<Widget> pw(new Widget);
processWidget(pw , priority());
```

# 4 设计与声明

## 规则 18 让接口容易被正确使用，不易被误用

1. **通过引入新类型来防止误用**

```
class Date{
public:
    Date(int month,int day,int year);
    ...
}
```

- 上面日期类的构造函数中，年月日都是 int，那么很容易传入顺序错误的参数。因此，可以因为 3 个表示年月日的新类：Year、Month、Day。从而防止这种问题。
- 更进一步，为了使得传入的数据有效，比如月份，可以设计生成 12 个月份对象的 static 成员函数，并将构造函数声明为 explicit 强制要求通过调用 static 成员函数得到月份对象。使用 enums 没有那么安全，enums 可被拿来当作一个 ints 使用

2. **尽量让你的 type 的行为与内置类型一致**：如 if(a*b=c)对内置类型来说不合法，那么你的 type 在实现 operator*时就应该返回一个 const 对象
3. **提供一致的接口**：如 C++STL 容器都提供 size()返回容器大小，但是 Java 和.Net 对于不同容器大小接口可能不同，这会增加使用负担
4. **返回“资源管理对象”而不是原始资源**：如用 shared_ptr 管理资源时，客户可能会忘记使用智能指针，从而开启了忘记释放和重复释放的大门。通过修改接口的返回类型为智能指针，从而确保元素资源处于“资源管理对象”的掌控之中

1. 促进正确使用的方法：接口的一致性，并且与内置类型兼容
2. 阻止误用的方法：建立新类型（例如 struct 类型和函数）、限制类型上的操作以约束对象的值
3. 智能指针 shared_ptr 可自定义删除器函数，可以防止 cross DLL 问题，能用于自动解锁互斥体

## 规则 19 视类设计为类型设计

1. 设计规范所要考虑的问题：
   - 新类型的对象应该如何创建和销毁？
     这会影响到类的构造函数和析构函数，以及内存分配和回收函数（operator new，operator new[]，operator delete，operator delete[]）
   - 对象的初始化和对象的赋值应该有什么不同？
     决定了构造函数和赋值运算符的行为和它们之间的不同
   - 以值传递对于你的新类型的对象意味着什么？
     拷贝构造函数定义了一个新类型的传值如何实现
   - 新类型的合法值的限定条件是什么？
     对于一个类的数据成员来说，仅有某些值的组合是合法的，这些组合决定了类必须维持的不变量，这些不变量决定了必须在成员函数内进行错误检查，也会影响函数抛出的异常
   - 新类型是否适合放进一个继承图表中？
     如果继承自其它类，注意基类的虚函数
     如果希望其它类继承自该类，需要考虑哪些类声明为虚函数

- 新类型允许哪种类型转换？
  T1 类型转换为 T2 类型：T1 内写类型转换函数或 T2 内写非显式的构造函数，而且是单一参数调用
- 新类型哪些运算符和函数有意义？
  应该为运算符和函数声明为成员函数或非成员函数
- 哪些标准函数不应该被接受？
  不被接受的需要声明为 private，在 C++11 中可以声明为=delete
- 新类型的哪些成员可以被访问？
  决定哪些成员是 public、protected、private
  决定哪些类或函数应该是友元
  决定是否应该嵌套一个类
- 新类型的未声明接口？
  对效率、异常安全性以及资源运用提供哪些保证
- 新类型有多大程度的通用性？
  考虑是否定义成一个类模板
- 新类型是否真的需要？
  考虑是否只使用非成员函数或模板就可以完成需要的工作而非一个新类型

## 规则 20 传 const 引用取代传值

1. 尽量以传递 const 引用来代替传值，主要是以下优点：
   - **避免拷贝从而获得更高的效率**：尤其对自定义类型来说，不会有调用构造函数和析构函数的开销
   - **防止继承中的对象切割**：如果是 pass-by-value，并且传入一个子类对象时，传入的子类对象会被切割，只保有基类对象的部分，从而无法表现多态，但是传递 const 引用可以避免

```
class Window {
public:
    std::string name() const; // 返回窗口名称
    virtual void display () const; // 显示窗口和具内容
};

class WindowWithScrollBars: public Window {
public:
    virtual void display() const;
};

void printNameAndDisplay(Window w) // 不正确！参数可能被切割。
{
    std::cout << w.name();
    w.display() ;
}

void printNameAndDisplay(const Window& w) // 参数不会被切割
{
    std::cout << w.name();
    w.display();
}
```

   在 printNameAndDisplay 函数内不论传递过来的对象原本是什么类型，参数 w 就像一个 Window 对象（因为其类型是 Window）。因此在 printNameAndDisplay 内调用 display 调用的总是 Window::display，绝不会是 WindowWithScrollBars::display。
2. **对于内置类型、STL 的迭代器和函数对象，一般采用传值而不是传引用**
   - references 往往以指针实现出来，因此 pass by reference 通常意味真正传递的是指针。因此，对于内置类型，pass by value 往往比 pass by reference 的效率更高。pass by value 同样适用于 STL 的迭代器和函数对象
3. 并不是所有小型对象都是 pass-by-value 的合格候选者：
   - 对象小并不意味着 copy 构造函数不昂贵。许多对象——包括大多数 STL 容器

——内含的东西比一个指针多一些，但是复制这种对象却需承担“复制那些指针所指的每一样东西”。那将非常昂贵

- 即使 copy 构造函数不昂贵，还是可能有效率上的争议。某些编译器对待“内置类型”和“用户自定义类型”的态度截然不同，纵使两者拥有相同的底层表述，“用户自定义类型”也不会被编译器放入缓存器，因此传引用更适合

可以合理假设“传值并不昂贵”的唯一对象就是内置类型和 STL 的迭代器和函数对象。其它任何时候，宁以传 const 引用替换传值

## 规则 21 必须返回对象时，不要试图返回一个引用

1. **绝不要返回一个局部栈对象的指针或引用，绝不要返回一个被分配的堆对象的引用**，如果存在需要一个以上这样的对象的可能性时，绝不要返回一个局部 static 对象的指针或引用
2. 在必须返回对象时，不要企图返回 reference，可以通过反面来说，也就是如果返回 reference 会是什么情况？
   - 使用 stack 构造一个局部对象，返回局部对象的 reference

```
const Rational& operator*(const Rational &lhs,const Rational &rhs)
{
    Rational result(lhs.n * rhs.n,lhs.d * rhs.d);
    return result;
}
```

注意！使用 reference 的本意是避免构造新对象，但是一个新的对象 result 还是经由构造函数构造。更严重的是，这个局部对象在函数调用完成后就被销毁了，reference 将指向一个被销毁的对象

   - 使用 heap 构造一个局部对象，返回这个对象的 reference

```
const Rational& operator*(const Rational &lhs,const Rational &rhs)
{
    Rational *result = new Rational(lhs.n * rhs.n,lhs.d * rhs.d);
    return *result;
}
Rational w,x,y,z;
w = x * y *z;
```

这样虽然 reference 不再引用一个被销毁的对象，但是因为了动态内存分配的开销，而且谁该为 delete 负责也成为问题。同时，在上面的连乘例子中，会多次动态分配内存，但是只返回最后一次的指针，因此会造成资源泄露

   - 构造一个 static 局部对象，每次计算结果保存在这个对象中，返回其 reference

```
const Rational& operator*(const Rational &lhs,const Rational &rhs)
{
    static Rational result
    result = ...;
    return result;
}
Rational w,x,y,z;
if((w * x) == (y * z)){...}
```

首先，显而易见的问题是这个函数在多线程情况下是不安全的，多个线程会修改相同的 static 对象；除此之外，在上面的 if 判断中，不管传入的 w,x,y,z 是什么，由于 operator*传回的 reference 都指向同一个 static 对象，因此上面的判断永远都会为 true

## 规则 22 将数据成员声明为 private

1）为什么不能是 public

3 个原因：

1. **语法一致性**：如果成员变量和成员函数一样，都是 public，那么调用时会困惑于该不该使用括号。如果想获取大小时使用 size，但是这到底是一个成员变量还是一个成员函数？
2. **更精准的控制**：通过将成员变量声明为 private，通过成员函数提供访问，可以

实现更精准的访问控制

```
class AccessLevels{
public:
    ...
    int getReadOnly() const   {return readOnly;}
    void setReadWrite(int value)    {readWrite = value;}
    int getReadWrite() const    {return readWrite;}
    void setWriteOnly(int value)    {writeOnly = value;}
private:
    int noAccess;   //对此 int 无访问动作
    int readOnly;   //对此 int 做只读访问
    int readWrite;  //对此 int 做读写访问
    int writeOnly;  //对此 int 做只写访问
};
```

3. **封装(主要)**：private 将成员变量封装，如果通过 public 暴露，在需要改成员变量的大量实现代码中，会直接使用当这个成员变量被修改或删除时，所有直接访问该成员变量的代码将会变得不可用

2）那么 protected 行不行

protected 成员变量和 public 成员变量的论点十分相同。“语法一致性”和“细微划分的访问控制”等理由也适用于 protected 数据。同时，protected 也并不具备良好的封装性

假设有一个 public 成员变量，而我们最终取消了它。所以使用它的客户代码都会被破坏。因此，public 成员变量完全没有封装性。假设有一个 protected 变量，而我们最终取消了它，所有使用它的派生类都会被破坏。因此，protected 成员变量也缺乏封装性

因此，从封装的角度看，只有 private 能提供封装性

1. **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**
2. **protected 并不比 public 更具有封装性**

## 规则 23 用非成员非友元函数取代成员函数

假设有个浏览器类，包含一些功能用来清除下载元素高速缓冲区、清除访问过的 URLs 的历史记录、以及移除系统中的所有 cookies：

```
class WebBrowser{
public:
  ...
  void clearCache();
  void clearCookies();
  void clearHistory();
  ...
};
```

此时，如果想整个执行所有这些动作，那么有两种选择，一种实现成 member 函数，一种实现成 non-member 函数：

```
class WebBrowser{
public:
  //实现成成员函数，能访问 private 成员
  void clearEverything(){
    clearCache();
    clearCookies();
    clearHistory();
  }
}
//或者实现成非成员函数，不能访问 private 成员
void clearEverything(WebBrowser& wb){
  wb.clearCache();
  wb.clearCookies();
  wb.clearHistory();
}
```

问题是应该如何选择？这个问题主要在于**封装性**：

- 对于对象内的代码。越少代码可以看到数据(也就是访问它),越多的数据可被封装,我们也就越能自由地改变对象数据。作为一种粗糙的测量,越多函数可访问它,数据的封装性就越低
- 条款 22 所说,成员变量应该是 private。能够访问 private 成员变量的函数只有 class 的 member 函数加上 friend 函数而已。**如果要在一个 member 函数和一个 non-member,non-friend 函数之间做选择,而且两者提供相同机能,那么,导致较大封装性的是 non-member,non-friend 函数**

一个扩展性的问题是——这些 non-member,non-friend 函数应该实现于何处?

- 一个像 WebBrowser 这样的 class 可能拥有大量便利函数,某些与书签有关,某些与打印有关,还有一些与 cookie 的管理有关...通常客户只对其中某些感兴趣。没道理一个只对书签相关便利函数感兴趣的客户却与一个 cookie 相关便利函数发生编译相依关系。分离它们的最直接做法就是将书签相关便利函数声明于一个头文件,将 cookie 相关便利函数声明于另一个头文件,再将打印相关...以此类推:

```
// 头文件 "webbrowser.h" — 这个头文件针对 class WebBrowser 自身及 WebBrowser 核心机能。
namespace WebBrowserStuff {
class WebBrowser {...};
... // 核心机能,例如几乎所有客户都需要的 non-member 函数。
}

// 头文件 "webbrowserbookmarks.h"
namespace WebBrowserStuff {
... // 与书签相关的便利函数
}

// 头文件 "webbrowsercookies.h"
namespace WebBrowserStuff {
... // 与 cookie 相关的便利函数
}
```

- 这正是 C++标准库的组织方式。标准库并不是拥有单一、整体、庞大的<C++ Standard Library>头文件并在其中内含 std 命名空间内的每一样东西,而是有数十个头文件(<vector>,<algorithm>,...),每个头文件声明 std 的某些机能。客户可以根据需要使用的机能选择性的包含头文件

总结:

1. **采用非成员非友元函数可以增加封装性、包裹弹性和机能扩充性。**
2. 推崇封装的原因是:它使我们能够改变事物而只影响有限客户的,但是成员函数和友元函数都可以访问类的私有成员,这其实有较低的封装性
3. 目前一般的处理方法是:**在一个头文件中的命名空间中定义核心机能代码,其他使用该核心机能的代码完成的某些功能定义在其他头文件中的同名命名空间中,这些定义在其他头文件中的函数一般都是非成员非友元的函数,**这种就是 C++标准程序库的组织方式,即在命名空间 std 中,分布在多个头文件来实现不同的内容(<vector>,<memory>,<algorithm>)

## 规则 24 当所有参数均需类型转换时,声明为非成员函数

为 class 支持隐式类型转换不是个好主意,但是在数值类型之间颇为合理。考虑有理数和内置整形之间的相乘运算。具有如下有理数:

```
class Rational{
public:
    Rational(int n = 0, int d = 1); //构造函数刻意不为 explicit,提供了 Int-to-Rational 的隐式转换
    int numerator() const;      //分子的访问函数
    int denominator() const;    //分母的访问函数
private:
...
};
```

现在,有理数提供了 Int-to-Rational 的隐式转换方式,那么 operator*应该实现成 member,

还是 non-member?

```
class Rational{
public:
    //实现为 member
    const Rational operator*(const Rational& rhs) const;
}

//实现为 non-member
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

问题发生在混合运算上。如果实现成 member，那么下面的混合运算只有一半行得通：

```
result = oneHalf * 2;               // OK
result = 2 * oneHalf;               // Error
```

因为内置类型 int 并没有相应的 class，也就没有 operator*成员函数。所以后者会出错。但是当实现为 non-member 时，具有 2 个参数，都能通过 int 转换为 Rational，所以上面 2 行代码都能运行。因此，若所有参数皆需类型转换，请为此采用 non-member 函数

总结：

1. 重要原则：**与成员函数相对的是非成员函数，而不是友元函数，仅仅因为函数不应该作为成员并不意味着它应该作为友元**
2. **如果允许在某个函数的参数使用类型转换，该函数必须是非成员函数**。最典型的是向数值类型的类型转换，考虑：
   result = oneHalf * 2;
   result = 2 * oneHalf;
   如果是成员函数并且参数 2 可以转换到 oneHalf 的类型，则第一个能执行，主要是 oneHalf 会调用 oneHalf.operator*(2);但第二个不能执行，因为不存在 2.operator*(oneHalf);的调用方法

### 规则 25 考虑支持不抛出异常的 swap 函数

1. STL 容器中的 swap 处理方式：
   - 容器类提供 public 的 swap 成员函数
   - 特例化容器版本的 std::swap 来调用容器类中的 swap 函数
2. 注意：可以特例化 std 内的模板，但是不能添加新的模板到 std 命名空间中
3. 如果 swap 缺省实现的效率不足，则应该做以下事情：
   - 提供一个 public 的 swap 成员函数，高效的进行替换，并确定这个函数不抛出异常
   - 在自定义的类或模板的命名空间内提供一个非成员版本的 swap，并令其调用 swap 成员函数
   - 如果是一个自定义的 class（而非 class template），还需要为这个自定义的类特例化 std::swap，并令它调用类的 swap 成员函数
   - 如果在程序中用到了 swap，则需要添加 using std::swap，以便让 std::swap 在你的函数内曝光可见，然后不加任何 namespace 修饰符地调用 swap
4. 成员版本的 swap 不应该抛出异常

# 5 实现

### 规则 26 尽可能延后变量定义式的出现时间

1. 只要定义了一个变量而其类型带有一个构造函数或析构函数，那么
   - 当程序的控制流到达这个变量定义式时，你便得承受构造成本
   - 当这个变量离开作用域时，你便得承受析构成本
2. 应该尽可能延后变量的定义，甚至应该尝试延后这份定义直到能够给它初值实参为止，这样既可以避免构造和析构非必要对象，还可以避免无意义的默认构造行为

3. 对于循环来说，将变量定义在循环外和定义在循环内的比较：

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

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

- 循环外的开销：1 个构造+1 个析构+n 个赋值
- 循环内的开销：n 个构造+n 个析构

由于在循环外定义，该变量的作用域比在循环内定义大，增加程序的维护，所以除非（1）赋值成本比“构造+析构”成本低（2）正在处理的代码需要很高的性能，否则应该使用循环内的定义

## 规则 27 将强制转换减到最少

1. C++风格的强制类型转换：
   - const_cast：通常用来将对象的常量性移除，**const 转 non-const 必须使用 const_cast**。
   - dynamic_cast：主要用来执行“安全**向下**的类型转换”，即派生类转换到基类。用来决定某对象是否归属继承体系中的某个类型，它是唯一无法由旧式语法执行的动作，也是唯一可能耗费重大运行成本的类型转换
   - reinterpret_cast：执行低级转型，实际动作可能**取决于编译器**，因而其不可移植
   - static_cast：最常用的强制类型转换，例如将 non-const 对象转为 const 对象，或将 int 转为 double 等等。它也可以用来执行上述多种转换的反向转换，例如将 void*指针转为 typed 指针，将基类指针转为派生类指针（向上类型转换，没有动态类型检查，不安全）。但它**无法将 const 转为 non-const**——这个只有 const_cast 才办得到。
2. 建议使用新型类型转换的原因：
   - 很容易在代码中被辨识出来
   - 各个强制类型转换动作的目标愈窄化，编译器愈可能诊断出错误的运用
     - 例如，如果你打算将常量性(constness)去掉，除非使用新式转型中的 const_cast 否则无法通过编译。
3. 尽量少做转型的原因：

   1) **转型不只是告诉编译器把某种类型视为另一种类型这么简单。任何一个转型动作往往令编译器编译出运行期间执行的代码**

```
//示例一
int x,y;
...
double d = static_cast<double>(x)/y;
//示例二
class Base {...};
class Derived : public Base {...};
Derived d;
Base *pd = &d;  //隐式地将 Derived*转换为 Base*
```

   - 在示例一中：int 转型为 double 几乎肯定会产生一些代码，因为在大部分体系结构中，int 的底层表述不同于 double 的底层表述
   - 在示例二中：会有个偏移量在运行期被实施于 Derived*指针身上，用以取得正确的 Base*地址

   2) **很容易写出似是而非的代码**

```
class Window{
public:
    virtual void onResize() {...}
    ...
```

```
}
//错误的做法
class SpecialWindow: public Window{
public:
    virtual void onResize(){
        static_cast<Window>(*this).onResize();
        ...  //这里进行 SpecialWindow 专属行为
    }
    ...
}
//正确的做法
class SpecialWindow: public Window{
public:
    virtual void onResize(){
        Window::onResize();  //调用 Window::onResize 作用于*this 身上
        ...  //这里进行 SpecialWindow 专属行为
    }
    ...
}
```

上面的例子中，派生类的虚函数可能希望调用基类虚函数的版本执行一些固定操作，然后再执行一些专属行为。在前面的做法中，转型并非在当前对象身上调用 Window::onResize 之后又在该对象身上执行 SpecialWindow 专属动作。而是在”当前对象的 baseclass 成分“的副本上调用 Window::onResize，然后在当前对象身上执行 SpecialWindow 专属动作。如果 Window::onResize 修改了对象内容，当前对象其实没被改的，改的是副本。如果专属动作也修改对象，那么当前对象会进入一种”伤残“状态：其 baseclass 成分的更改没有落实，derivedclass 成分的更改倒是落实了

3) **继承中的类型转换效率低**

C++通过 dynamic_cast 实现继承中的类型转换，dynamic_cast 的大多数实现效率都是相当慢的。因此，应该避免继承中的类型转换。一般需要 dynamic_cast，通常是因为想在一个认定为 derivedclass 对象身上执行 derivedclass 操作，但是拥有的是一个”指向 base“的指针或引用。这种情况下有 2 种办法可以避免转型：

- **使用容器并在其中存储直接指向 derived class 对象的指针**：这种做法无法在同一个容器内存储指针”指向所有可能的各种派生类“。如果真要处理多种类型，可能需要多个容器，它们都必须具备类型安全性
- **将 derived class 中的操作上升到 base class 内，成为 virtual 函数，base class 提供一份缺省实现**：缺省实现代码可能是个馊主意，条款 34 中有分析，但是也比使用 dynamic_cast 来转型要好

4. 对于有继承体系的类来说：
   - 使用基类指针指向派生类，实际上此时会有一个偏移量在运行时施行在派生类的指针上（即单一对象可能有两个地址）
   - 在一个派生类的成员函数里，如果将 this 指针强制转换为基类指针，此时会生成一个副本，此时对其进行的操作并不会影响当前对象的内容
5. 避免使用 dynamic_cast 的方法：
   - 直接存储指向派生类对象的指针的容器
   - 基类定义虚函数，使用多态

总结：

1. 尽量避免转型，特别是在注重效率的代码中避免 dynamic_casts。如果有个设计需要转型动作，试着发展无需转型的替代设计。
2. 如果强制类型转换是必要的，尽量将其隐藏在某个函数背后，使得客户不需要使用强制类型转换的代码。
3. 尽量使用 C++ style（新式）转型，不要使用旧式转型。

## 规则 28 避免返回对象内部构件的“句柄”

1. **在类的成员函数中，避免使用句柄（指针、引用和迭代器）返回内部成员变量**，如果使用了则会降低封装性，因为很可能返回一个 private 的成员
2. 对于 const 成员函数，如果必须返回句柄，最好是返回 const 句柄

# 规则 29 争取异常安全的代码

考虑下面例子，有一个菜单类，changeBg 函数可以改变它的背景，切换背景计数，同时提供线程安全：

```
class Menu{
    Mutex mutex;        //提供多线程互斥访问
    Image *bg;          //背景图片
    int changeCount;    //切换背景计数
public:
    void changeBg(istream& sr);
};

void Menu::changeBg(istream& src){
    lock(&mutex);
    delete bg;
    ++changeCount;
    bg = new Image(src);
    unlock(&mutex); // 如果 newImage 发生异常，那么 unlock 就不会调用
}
```

1. 异常安全有 2 个条件：
   - **不泄露任何资源**：即发生异常时，异常发生之前获得的资源都应该释放，不会因为异常而泄露。在上面的例子中，如果 new Image 发生异常，那么 unlock 就不会调用，因此锁资源会泄露
   - **不允许数据败坏**：上面的例子也不符合，如果 new Image 异常，背景图片会被删除，计数也会改变。但是新背景并未设置成功

   对于资源泄露，条款 13 讨论过**以对象管理资源**。使用对象来管理锁可以保证锁的及时释放，异常发生时管理锁资源的对象被销毁后，会自动调用 unlock

   对于数据败坏：见下文
2. 异常安全的函数需要提供下述三种保证之一：
   - **提供基本保证**：允许抛出异常，但程序中剩下的每一件东西都处于合法状态
   - **提供强力保证**：允许抛出异常，如果异常抛出，程序的状态不会发生变化
   - **提供不抛出保证**：函数总是能完成它所承诺的事情，不抛出异常，所有对内置类型的操作都是不抛出异常的

对于前面的 PrettyMenu 对象，可以通过使用智能指针，以及重排 changeBg 的语句顺序来满足“强力保证”：

```
class Menu{
    shared_ptr<Image> bg;
    ...
};
void Menu::changeBg(istream& src){
    Lock m1(&mutex);    //Lock 以对象管理资源
    bg.reset(new Image(src));
    ++changeCount;
}
```

3. "copy and swap"设计策略通常能够**为对象**提供异常安全的“强力保证”。当我们要改变一个对象时，**先把它复制一份，然后去修改它的副本，改好了再与原对象交换。**

```
class Menu{
    ...
private:
    Mutex mutex;
    std::shared_ptr<MenuImpl> pImpl;
};
Menu::changeBg(std::istream& src){
    using std::swap;            // 见 Item 25
    Lock m1(&mutex);            // 获得 mutex 的副本数据

    std::shared_ptr<MenuImpl> copy(new MenuImpl(*pImpl));
    copy->bg.reset(new Image(src)); //修改副本数据
    ++copy->changeCount;
```

```
    swap(pImpl, copy);              //置换数据，释放 mutex
}
```

**为对象**提供异常安全的强力保证可以通过 copy and swap 来实现，但是并非对所有函数都可用，当函数中包含的其他函数不能提供强力保证时，该函数也无法提供强力保证

除此之外，copy and swap 必须为每一个即将被改动的对象作出一个副本，从而可能造成时间和空间上的问题

函数提供的”异常安全保证“通常最高只等于其所调用的各个函数的”异常安全保证“中的最弱者

总的来说就是，应该为自己的函数努力实现尽可能高级别的异常安全，但是由于种种原因并不是说一定需要实现最高级别的异常安全，而是应该以此为目标而努力

## 规则 30 透彻了解 inline 的里里外外

1. inline 提出方式包括 2 种：1）显式提出；2）隐式提出（类内实现成员函数）
2. inline 在大多数 C++程序中是**编译期**行为，内联是向编译器发出一个请求，而不是命令，编译器可以忽略
3. 声明方式：
   - 隐式声明：在类内部定义它
   - 显式声明：在类外部定义时添加 inline
4. inline 的优劣：
   **优**：较少函数调用的开销；由编译器对 inline 的优化
   **劣**：目标代码的增加，**程序体积增大**，导致额外的换页行为，降低指令高速缓存装置的命中率
5. 内联是编译时期的请求，所以不能与虚函数一起配合使用。**对 virtual 函数的调用会使 inlining 落空。因为 virtual 意味着“等待，直到运行期才确定调用哪个函数”，而 inline 意味着“执行前，先将调用动作替换为被调用函数的本体”**
6. 使用函数指针指向内联函数，然后使用函数指针调用该内联函数，此时可能不会进行内联
7. **即使是构造、析构函数内不含有任何代码，也不适合将其声明为内联的**。因为编译器会在安插包括异常处理在内的一系列代码，并且编译器会在构造函数中安插进行成员变量的构建以及其基类对象的构建相关的代码等，如果进行内联很容易**造成代码膨胀**

```
class Base{
public:
    ...
private:
    std::string bm1,bm2;    //base 成员 1 和 2
};

class Derived:public Base{
public:
    Derived()  {}
    ...
private:
    std::string dm1,dm2,dm3; //derived 成员 1-3
};
```

上面的代码中构造函数为空，但其会被编译器安插异常处理等代码：

```
Derived::Derived()
{
    Base::Base();
    try{dm1.std::string::string();}
    catch(...){
        Base::~Base();
        throw;
    }
    try{dm2.std::string::string();}
    catch(...){
```

```
        dm1.std::string::~string();
        Base::~Base();
        throw;
    }
    try{dm3.std::string::string();}
    catch(...){
        dm2.std::string::~string();
        dm1.std::string::~string();
        Base::~Base();
        throw;
    }
}
```

8. 内联函数改动之后必须全部重新进行编译，普通函数改动之后只需要编译改动的文件即可
9. 一般来说，调试器无法在内联函数中设立断点

## 规则 31 将文件间的编译依存关系降至最低

```
#include <string>
#include "date.h"
#include "address.h"
class Person{
public:
    ...
private:
    std::string theName;     //实现细目
    Date    theBirthDate;    //实现细目
    Address theAddress;      //实现细目
};
```

如果没有前面 3 行引入头文件，那么编译无法通过。但是如此却**会在 Person 定义文件和其含入文件之间形成了一种编译依存关系。如果这些头文件中有任何一个被改变，或这些文件所依赖的其它头文件有任何改变。那么每个含入 Person class 的文件就得重新编译，任何使用 Person class 的文件也必须重新编译**。这样的连串编译依存关系会对许多项目造成难以形容的灾难

你可能会想着将实现细目分开：

```
namespace std{
    class string;   //前置声明，但不正确
}
class Date;         //前置声明
class Address;      //前置声明

class Person{
public:
    ...
};
```

如果可以这么做，Person 的客户就只需要在 Person 接口被修改过时才重新编译。但是这种想法存在 2 个问题：

- string 并不是个 class，它是个 typedef，上述前置声明不正确，正确的前置声明比较复杂
  - 对于 std 命名空间的类，最好是提供#include 而不是提供前向声明，因为很多类是 typedef，例如 string
- 重点是，**编译器必须在编译期间知道对象的大小**，而编译器无法获取上面 Person 类的大小。

1. 常用的最小化编译的手段：
   - **声明两个类，一个类 A 用来完成实现，另一个类 B 内部使用一个指针指向类 A 来完成某些工作**

     可以把 Person 分割为两个类：1）一个只提供接口(Person)；2）一个负责实现接口(PersonImpl)。接口 class 中只包含一个负责实现接口的 class 的指针，因此任何改变都只是在负责实现接口的 class 中进行。那么 Person 的客户就完全与 Date,Address,以及 Person 的实现细目分离了。那些 classes 的任何实现修改都不

需要 Person 客户端重新编译。此外，由于客户无法看到 Person 的实现细目，也就不可能写出什么“取决于那些细目的代码”。这正是接口与实现分类。这种情况下，像 Person 这样使用 pimpl 的 classes 往往被称为 **handle classes**

```
class Person{
public:
    Person(string& name);
    string name() const;
private:
    shared_ptr<PersonImpl> pImpl;
};
Person::Person(string& name): pImpl(new PersonImpl(name)){}
string Person::name(){
    return pImpl->name();
}
```

- **声明一个抽象基类，该类只包含虚析构函数和一组纯虚函数，并且该类中包含一个工厂函数，该函数被定义为 static，返回一个指向该抽象基类的智能指针**

另一种制作 Handle class 的办法是，令 Person 成为一种特殊的 abstract base class，称为 **interface class**。其目的是详细描述 derived classes 的接口，因此它通常不带成员变量，也没有构造函数，只有一个 virtual 析构函数以及一组 pure virtual 函数，用来叙述整个接口

```
class Person{
public:
    virtual ~Person();
    virtual string name() const = 0;
    virtual string birthday() const = 0;
};
```

客户不能实例化它，只能使用它的引用和指针。然而客户一定需要某种方法来获得一个实例，比如工厂方法：

```
class Person{
public:
    static shared_ptr<Person> create(string& name);
};
shared_ptr<Person> Person::create(string& name){
    return shared_ptr<Person>(new RealPerson(name));
}
...
shared_ptr<Person> p(Person::create("alice"));
```

Handle classes 和 Interface classes 解除了接口和实现之间的耦合关系，从而降低文件间的编译依存性。

2. 最小化编译的精髓：只要能实现，就**让头文件独立自足**，如果不能，就依赖**其他文件中的声明而不是定义**，即：
   - **当对象的引用和指针可以做到时就避免使用对象**
   - 只要能做到，**尽量以 class 声明式替换 class 定义式**，此时可以将声明的类作为函数的值参数或返回值参数

     ```
     class Date;
     Date today();
     void clearAppointments(Date d);
     ```

   - **为声明和定义提供不同的头文件**：标准库 std 中将一个类所要包含的声明通常放在<xxxfwd>中，所以一般我们也应将声明放在“datefwd.h”中

     ```
     #include "datefwd.h"    //包含了 class Date 的声明
     Date today();
     void clearAppointments(Date d);
     ```

3. 程序库文件应该以“完全且仅有声明式”的形式存在，以最大限度的减少编译之间的依存

# 6 继承与面向对象设计

## 规则 32 确保 public 继承模拟“is-a”关系

1. 公开继承意味“is-a”（是一种）的关系，即**适用于基类身上的每一件事情一定也适用于派生类身上，因为每一个派生类对象也都是一个基类对象**
2. **public 隐含的寓意**：每个派生类对象同时也是一个基类对象（反之不成立），只不过基类比派生类表现出更一般化的概念，派生类比基类表现出更特殊化的概念。
   可以举一个例子验证一下上面的说法。例如：
   ```
   class Person {...};
   class Student : public Person {...};
   ```
   显然，每个学生都是人，但并非每个人都是学生。对人可以成立的每一件事对学生也都成立（例如每个人都有生日），但对学生都成立的每件事并不一定对每个人也成立（例如注册于某个学校）

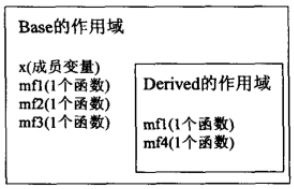
   因此，C++中，任何函数如果期望获得一个类型为基类的实参，都也愿意接收一个派生类对象。但是反之不成立：
   ```
   void eat(const Person &p);
   void study(const Student &s);
   Person p;
   Student s;
   eat(p);     //正确
   eat(s);     //正确
   study(s);   //正确
   study(p);   //错误
   ```
   谨记这种 is-a 关系以及背后隐藏的规则可以防止因为“经验主义”而使用不合理的继承：
   - 从“经验主义”上看，企鹅也是鸟，如果为鸟定义了虚拟(virtual)的飞的方法，然后企鹅类以 public 继承鸟类，那么是不合理的方式。这种情况下，可以设计一个会飞的鸟的类，虚拟(virtual)的飞的方法声明在这个类中，而鸟类里面没有声明飞的方法。或者根本不声明飞的方法
   - 从“经验主义”上看，正方形也是长方形，如果长方形有成员方法会修改长或宽，那么正方形以 public 继承长方形就显得不合理
3. 软件世界不同于现实世界。对于上面的鸟的设计，某些软件系统可能不需要区分会飞的鸟和不会飞的鸟。那么即使鸟类声明了飞的方法，然后企鹅类以 public 继承，也不会有多大问题。也就是说，不存在一个适用于所有软件的设计。最佳的设计取决于系统希望做什么事。如果程序对飞行一无所知，而且也不打算未来对飞行有所知，那么不去区分会飞的鸟和不会飞的鸟不失为一个完美而有效的设计。实际上可能比对两者做出隔离的设计更受欢迎，因为这样的区隔在你企图塑模的世界中并不存在。因此，**应该根据实际软件需求，合理使用 public**
4. 除了“is-a”关系外还有“has-a（有一个）”关系和“is-implemented-in-term-of（根据某物实现出）”这些关系

## 规则 33 避免覆盖继承而来的名字

1. **派生类作用域被嵌套在基类作用域内**

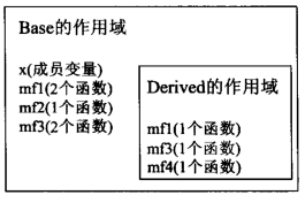
```
class Base {
private:
   int x;
public:
   virtual void mf1() = 0;
   virtual void mf2();
   void mf3();
   ...
};

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




   位于一个派生类成员函数内指涉（refer to）基类内的某物（也许是个成员函数、typedef 或成员变量）时，编译器可以找出我们所指涉的东西，因为派生类继承了声明于基类内的所有东西。
2. **名称遮掩会遮掩基类所有重载版本**

```
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();
   ...
};
```

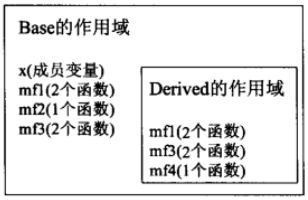



**派生类中同名的名称会遮掩基类中相同的名称，如果基类包含重载函数，所有重载函数都会被遮掩**。base class 内所有名为 mfl 和 mf3 的函数都被 derived class 内的 mfl 和 mf3 函数遮掩掉了。从名称查找观点来看，Base::mfl 和 Base::mf3 不再被 Derived 继承！

```
Derived d;
int x;
...
d.mfl(); // 没问题，调用 Derived::mfl
d.mfl(x); // 错误！因为 Derived::mfl 遮掩了 Base::mfl
d.mf2(); // 没问题，调用 Base::mf2
d.mf3(); // 没问题，调用 Derived::mf3
d.mf3(x); // 错误！因为 Derived::mf3 遮掩了 Base::mf3
```

3. **解决办法**：如果派生类继承基类并重载某些函数，而又希望重新定义或重载其中一部分，那么必须为那些原本会被遮掩的每个名称**引入一个 using 声明**，否则会隐藏掉希望继承的某些名字

```
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();
   ...
};
```




```
Derived d;
int x;
...
d.mfl(); // 仍然没问题，调用 Derived::mfl
d.mfl(x); // 现在没问题了，调用 Base::mf1
d.mf2(); // 仍然没问题，调用 Base::mf2
d.mf3(); // 仍然没问题，调用 Derived::mf3
d.mf3(x); // 现在没问题了，调用 Base::mf3
```

4. 提供的 using 声明一般放在 public 处，基类的 public 名字在 public 继承的派生类中应该也是 public 的
5. 如果说派生类并不需要继承所有的名字，只想引入基类被遮盖函数中的某个版本（注意，这种需求一般只在 private 继承中出现，因为如果只继承基类的部分操作，违背了条款 32），可以直接定义一个同名同参的函数，即**转交函数**（即 Base::func();），然后在这个函数内调用基类的版本，来引入基类（Base）中的 func 函数

```
class Base {
public:
   virtual void mf1() = 0;
   virtual void mf1(int);
   ...                    //与前同
};
class Derived: private Base {
public:
   virtual void mf1()  //转交函数（forwarding function）;
   { Base::mf1( ); }   //暗自成为 inline（见条款 30）
   ...
};
```

## 规则 34 区分接口继承和实现继承

1. 在继承体系中，函数分为：
   - 函数接口继承
   - 函数实现继承
2. 函数/虚函数的功能：
   - 声明 pure virtual（纯虚）函数的目的是为了**让派生类只继承函数接口**
   - 声明 impure virtual（非纯虚）函数的目的，是**让派生类继承该函数的接口和缺省实现**
   - 声明 non-virtual（非虚）函数的目的是为了**让派生类继承函数的接口及一份强制性实现**
3. 纯虚函数（pure virtual）：
   - 说明：
     - 纯虚函数一般作为接口，基类一般不提供定义，但是**基类可以为纯虚函数提供定义。**
     - **派生类必须声明纯虚函数，如果想要使用纯虚函数，派生类必须提供一份定义，即使基类已经为该纯虚函数提供了定义。**
     - **如果派生类不提供定义，仍然是一个抽象基类**
   - 目的：为了让派生类只继承函数接口
   - 使用场景：如果某个操作不同派生类应该表现出不同行为，并且没有相同的缺省实现，那么应该使用纯虚函数，此时派生类只继承接口
   - 注意：可以为纯虚函数提供定义，但是调用它的唯一途径是“调用时明确指出基类的名字”，即 Shape::draw();
4. 虚函数（impure virtual）：
   - 目的：为了让派生类继承该函数的接口和缺省实现
   - 说明：一般来说，基类提供一份缺省实现，派生类也可以进行自定义，也可以使用基类提供的缺省实现
   - 使用场景：如果某个操作不同派生类应该表现出不同行为，并且具有相同的缺省实现，那么应该使用非纯虚函数，此时派生类继承接口和缺省实现
   - 注意：假设引入了一个新的派生类，但是缺省行为并不适用于新的派生类，而新的派生类忘记重新定义新的行为，那么调用该操作将表现出缺省行为，这是不合理的。

     例如，某个航空公司有 A,B 两种类型的飞机，他们有相同的 fly 行为，这个 fly 行为在基类 Airplane 中声明为 impure virtual 函数，并且具有缺省的飞行实现。现在引入了一种新机型 C，但是这个缺省的 fly 行为并不适合 C，如果 C 忘记重新定义 fly，那么它将按照 A,B 缺省的行为飞行

```
class Airplane{
public:
    virtual void fly(){
        // 缺省的 fly 代码
    }
};
class ModelA: public Airplane{...};
class ModelB: public Airplane{...};
class ModelC: public Airplane{...};
Airplane* p = new ModelC;
p->fly();   //调用 Airplane::fly
```

此时的解决方法有两种：

- **定义一个 protected 的非虚函数来完成实现，将虚函数声明为纯虚函数，在派生类的该纯虚函数的实现中调用 protected 的函数来实现缺省的功能**

在上面的例子中，将 fly 改为纯虚函数，并且将缺省的飞行行为实现为一个 protected 函数：

```
class Airplane{
public:
    virtual void fly() = 0;
//这是合理的，因为它是 Airplane 及其 derived classes 的实现细目。乘客应该只在意飞机能不能飞，不在意它怎么飞
protected:
    //non-virtual 函数，因为没有任何一个派生类应该重新定义缺省行为
    void defaultFly(){...}
}
class ModelA: public Airplane{
public:
    virtual void fly(){defaultFly();}
}
class ModelB: public Airplane{
public:
    virtual void fly(){defaultFly();}
}
```

此时，fly 变成了纯虚函数，首先，飞机 C 必须声明 fly 函数，如果需要使用，必须为其定义。那么就可以防止因为忘记重新定义而引起的错误

- **将虚函数声明为纯虚函数，并在抽象基类中定义该纯虚函数，在派生类中的纯虚函数定义处调用基类中的纯虚函数实现以完成缺省行为**

上面的例子中，可以将缺省的行为定义在 fly 中，即为 fly 实现一份缺省的定义：

```
class Airplane{
public:
    virtual void fly() = 0;
};
void Airplane::fly(){
    // 缺省的 fly 代码
}

class ModelA: public Airplane{
public:
    virtual void fly(){
        Airplane::fly();
    }
};
```

由于任何派生类想要使用纯虚函数都必须提供一份定义，那么如果想要使用缺省行为，可以直接在定义中转调用基类的实现。否则，可以定制特殊的行为。因为是纯虚函数，只要不定义就无法使用，因此也可以避免前面的问题

5. 非虚函数（non-virtual）

如果某个操作在整个体系中，应该表现出一致的行为，那么应该使用非虚函数。此时派生类**继承接口和一份强制性实现**。

# 规则 35 考虑虚函数以外的其他选择

在面向对象中，如果希望某个操作存在缺省算法，并且各派生类可以定制适合自己的操作。可以使用 public virtual 函数，这是最简单直白且容易想到的方法，但是除此之外，也存在其它可替代的方案。它们有各自的优缺点，应该将所有方案全部列入考虑
以一个例子来介绍其它几种可替代方案。在一个游戏人物的类中，存在一个健康值计算的函数，不同的角色可以提供不同的健康值计算方法，并且存在一个缺省实现。以传统的 public virtual 函数实现如下：

```
class GameCharacter{
public:
    virtual int healthValue() const;    //健康值计算函数，派生类可以重新定义
};
```

1. **non-virtual interface（NVI）手法，是 Template Method 设计模式的独特表现形式**
   保留 healthValue 为 public 成员，但是让其成为**非虚函数**，并调用一个 private（也可以是 protected）**虚函数**进行实际工作：

```
class GameCharacter{
public:
    //non-virtual 函数，virtual 函数的包裹器(wrapper)
    int healthValue() const
    {
        ...                                 //做一些事前工作
        int retVal = doHealthValue();   //负责真正的健康值计算
        ...                                 //做一些事后工作
        return retVal;
    }
    ...
private:
    virtual int doHealthValue() const   //派生类可以重新定义
    {
        ...     //缺省的健康值计算方法
    }
};
```

   - 在类中的具体体现为：
     - 在类的 private 处定义一个虚函数完成某种操作
     - 在类的 public 处声明一个非虚函数的接口调用上述虚函数完成某些功能
   - 优点：在非虚函数调用虚函数之前和之后可以添加一些代码来设置和恢复环境
   - 注意：虚函数的定义有时候也可能是 protected，当派生类的虚函数需要调用基类的虚函数时，此时就必须是 protected 的

上面的方案本质还是使用 virtual 函数，人物的健康值计算(操作)还是与人物(类)相关。后面这几种方案，都是将任务的健康值计算(操作)与具体的每个人(对象)相关，并且每个人(对象)的健康值计算(操作)可以修改

2. **Function Pointers 手法，是 Strategy 模式的简单应用**
   每个人物(类)包含一个计算健康值的函数指针，每创建一个人(对象)时，可以为其指定不同的健康值计算函数。因此将**操作和类分离**。同时，如果提供修改函数指针成员的方法，每个对象还能使用不同的计算方法

```
class GameCharacter;    //前置声明
//健康值计算的缺省函数
int defaultHealthCalc(const GameCharacter &gc);
class GameCharacter{
public:
    typedef int (*HealthCalcFunc)(const GameCharacter&);
    explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc) :
healthFunc(hcf) {}
    //non-virtual 函数，virtual 函数的包裹器(wrapper)
    int healthValue() const
    {
        return healthFunc(*this);
    }
    ...
```

```
private:
    HealthCalcFunc healthFunc;
};
```

- 常用方法：类的**构造函数接受一个函数指针**，该函数用以完成某些功能
- 优点：
  - 同一类型的每个对象可以有不同的函数
  - 同一对象的函数指针的值可以在运行期间进行更改
- 缺点：由于不可访问 private 成员，所以有时候会弱化 class 的封装

3. **Function 手法，也是 Strategy 模式的应用**

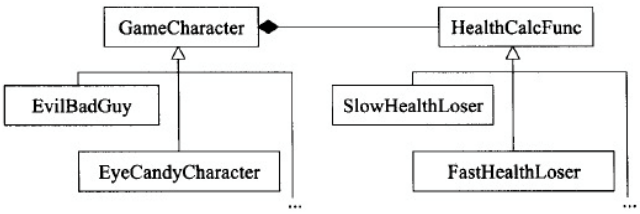
这种方案是前一种的加强，将函数指针改成任何可调用对象。因此允许任何与可调用声明相兼容(即可以通过类型转换与声明相符)的可调用物

```
class GameCharacter;    //前置声明
//健康值计算的缺省函数
int defaultHealthCalc(const GameCharacter &gc);
class GameCharacter{
public:
    //现在，类型 HealthCalcFunc 从函数指针变成了可调用物
    typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;
    explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc) :
healthFunc(hcf) {}
    //non-virtual 函数，virtual 函数的包裹器(wrapper)
    int healthValue() const
    {
        return healthFunc(*this);
    }
    ...
private:
    HealthCalcFunc healthFunc;
};
```

- 优点：可以使用任何可调用的对象而不单单是函数指针，这扩展了 Function Pointer 手法

4. **传统的 Strategy 模式**

传统的 Stategy 模式做法会将健康计算函数做成一个分离的继承体系中的 virtual 成员函数，设计结果看起来像这样:




```
class GameCharacter;    //前置声明
class HealthCalcFunc{
public:
    ...
    virtual int cacl(const GameCharacter &gc) const {...}
    ...
};
//创建一个 HealthCalcFunc 对象，可以通过它调用缺省的健康值计算方法
HealthCalcFunc defaultHealthCalc;

class GameCharacter{
public:
    explicit GameCharacter(HealthCalcFunc *phcf = &defaultHealthCalc) :
pHealthCalc(phcf) {}
    //non-virtual 函数，virtual 函数的包裹器(wrapper)
    int healthValue() const
    {
        return pHealthCalc->cacl(*this);
    }
    ...
private:
```

```
    HealthCalcFunc *pHealthCalc;
};
```

- 将继承体系内的 virtual 函数替换为另一个继承体系内的 virtual 函数

总结：
替换虚函数的四种选择：

- 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”，这是 Strategy 设计模式的一种分解表现形式。
- 以 trl::function 成员变量替换 virtual 函数，因而允许使用任何可调用物 (callable entity) 搭配一个兼容于需求的签名式。这也是 Strategy 设计模式的某种形式。
- 将继承体系内的 virtual 函数替换为另一个继承体系内的 virtual 函数。这是 Strategy 设计模式的传统实现手法。

- virtual 函数的替代方案包括 NVI 手法及 Strategy 设计模式的多种形式。NVI 手法自身是一个特殊形式的 Template Method 设计模式。
- 将机能从成员函数移到 class 外部函数，带来的一个缺点是，非成员函数无法访问 class 的 non-public 成员。
- trl::function 对象的行为就像一般函数指针。这样的对象可接纳”与给定之目标签名式(target signature)兼容”的所有可调用物(callable entities)。

## 规则 36 绝不重新定义继承而来的非虚函数

1. 非虚函数都是静态绑定的，即使使用指针和引用，调用的是指针和引用所指向的类型中定义的函数，而**不会发生动态绑定**，所以绝不要重新定义继承而来的非虚函数
2. 从规范上说，条款 34 提到，如果**某个操作在整个继承体系应该是不变的**，那么使用非虚函数，此时派生类从基类继承接口以及一份强制实现。**如果派生类希望表现出不同行为，那么应该使用虚函数**
3. 假设真的重新定义了继承而来的非虚函数，会表现出下列令人困惑的情况：

```
class B{
public:
    void mf();
    ...
};

class D : public B{
public:
    void mf();  //重新定义了继承而来的 non-virtual 函数
};

D x;
B *pB = &x;
D *pD = &x;

pB->mf();       //调用 B::mf
pD->mf();       //调用 D::mf
```

你可能会觉得因为 pB 和 pD 指向的是相同的对象，因此调用的非虚函数也应该相同，但是事实并非如此。因为**非虚函数是静态绑定**，因此实际上调用的函数由指针或引用的类型决定，调用指针指向的类中定义的函数。

## 规则 37 绝不重新定义继承而来的缺省参数值

1. **虚函数是动态绑定，而缺省参数值是静态绑定的**，所以当使用动态绑定调用虚函数时，很可能该虚函数的缺省参数值不是期望的那个（即派生类中的那个），反而获得

的缺省参数值是静态类型决定的（可能是基类的）。

```
class Shape{
public:
    enum ShapeColor {Red,Green,Blue};
    virtual void draw(ShapeColor color = Red) const = 0;
    ...
};

class Rectangle : public Shape {
public:
    virtual void draw(ShapeColor color = Green) const;
    ...
};

class Circle : public Shape {
public:
    virtual void draw(ShapeColor color) const;
    ...
};

Rectangle r;
Circle c;

r.draw();           //调用 Rectangle::draw，静态类型为 Rectangle，所以缺省参数为
Shape::Green
//c.draw();         //调用 Circle::draw，静态类型为 Circle，没有缺省参数，因此错
误，必须显式指定！

Shape *pr = &r;
Shape *pc = &c;

//以下为容易引起困惑的地方，函数与参数不一致
pr->draw();         //调用 Rectangle::draw，但是静态类型为 Shape，所以缺省参数
Shape::Red
pc->draw();         //调用 Circle::draw，但是静态类型为 Shape，所以缺省参数
Shape::Red
```

2. 若同时提供缺省参数值给基类和派生类的用户，无疑会导致没有必要的代码重复，并且当基类的缺省参数发生变化时，派生类的所有缺省参数也需要跟着修改。因此，本质在于，**不应该在 virtual 函数中使用缺省参数**

```
class Shape {
public:
enum ShapeColor { Red, Green, Blue};
virtual void draw(ShapeColor color = Red) const;
...
};

class Rectangle: public Shape {
public:
virtual void draw(ShapeColor color = Red) const;  // 重复指定缺省值
...
};
```

3. 如果说继承体系中确实每个派生类的某个虚函数都有相同的缺省值，则可以**使用 NVI 手法**，即
   - 在类的 **private 处定义一个虚函数**完成某种操作
   - 在类的 **public 处声明一个带缺省参数的非虚接口函数**调用上述虚函数完成某些功能

```
class Shape{
public:
    enum ShapeColor {Red,Green,Blue};
    //此时，带缺省参数的已经不是 virtual 函数
    void draw(ShapeColor color = Red) const // 带缺省参数的非虚接口函数
    {
        doDraw(color);  //调用一个 virtual
    }
```

```
    ...
private:
    //而完成真正工作的 virtual 函数已经不带缺省参数
    virtual void doDraw(ShapeColor color) const = 0;  //完成真正的工作
};

class Rectangle : public Shape {
public:
    ...
private:
    //而完成真正工作的 virtual 函数已经不带缺省参数
    virtual void doDraw(ShapeColor color) const;
    ...
};

class Circle : public Shape {
public:
    ...
private:
    //而完成真正工作的 virtual 函数已经不带缺省参数
    virtual void doDraw(ShapeColor color) const;
    ...
};
```

总结：

**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，而虚函数却是动态绑定。**

## 规则 38 通过复合模拟“has - a”或“根据某物实现出”

1. 复合是类型间的一种关系，当**某种类型的对象含有另一种类型的对象**，便是这种关系
2. **复合的意义和 public 继承完全不同**
3. **在应用域，复合意味 has-a，在实现域，复合意味 is-implemented-in-term-of（根据某物实现出）**

   has-a：

   ```
   class Address {...};
   class PhoneNumber {...};
   class Person{
   public:
       ...
   private:
       std::string name;
       Address address;
       PhoneNumber voiceNumber;
       PhoneNumber faxNumber;
   };
   ```

   根据某物实现出：

   ```
   template <class T, class Sequence = deque<T> >
   class stack {
   ...
   protected:
     Sequence c;   //底层容器
   ...
   };
   ```

   上面两者情况都应该使用复合，而不是 public 继承。在 has-a 中，每个人肯定不是一个地址，或者电话。显然不能是 is-a 的关系。而对于后者，由于每个栈只能从栈顶压入弹出元素，而队列不同，is-a 的性质是所有对基类为 true 的操作，对派生类也应该为 true。所以 stack 也不应该通过 public 继承 deque 来实现，因此使用复合
4. 要知道区分 is-a 和 is-implemented-in-term-of，is-a 表明的是对基类为真的事对其派生类也为真

# 规则 39 谨慎使用私有继承

1. private 继承意味着
   - 不意味着“is-a”关系，而是“implemented-in-term-of（根据某物实现出）”，所以**派生类不能自动转换到基类**，即使是引用和指针

```
class Person { ... };
class Student: private Person { ... };     // private 继承
void eat(const Person& p);                 // 任何人都会吃
Person p;                                  // p 是人
Student s;                                 // s 是学生
eat(p);                                    // 没问题，p 是人，会吃
eat(s);                                    // 错误！难道学生不是人？！
```

   如果使用 public 继承，编译器在必要的时候可以将 Student 隐式转换成 Person，但是 private 继承时不会，所以 eat(s)调用失败。从这个例子中表达了，private 继承并不表现出 is-a 的关系。实际上 private 表现出的是"is-implemented-in-terms-of"的关系

   - **基类中的所有成员（即使是 public、protected）在派生类中都是 private 属性**
2. private 继承和复合比较像，但是尽可能使用复合，必要时才使用 private 继承

■ **使用 private 继承的例子：**

假设 Widget 类需要执行周期性任务，于是希望继承 Timer 的实现。因为 Widget 不是一个 Timer，所以选择了 private 继承：

```
class Timer {
public:
   explicit Timer(int tickFrequency);
   virtual void onTick() const;       // 每滴答一次，该函数就被自动调用一次
};
class Widget: private Timer {
private:
  virtual void onTick() const;         // 查看 Widget 的数据...等等
};
```

在 Widget 中重写虚函数 onTick，使得 Widget 可以周期性地执行某个任务

通过 private 继承来表现"is-implemented-in-terms-of"关系实现非常简单，而且下列情况也只能使用这种方式：

- 当 Widget 需要访问 Timer 的 protected 成员时。因为对象组合后只能访问 public 成员，而 **private 继承后可以访问 protected 成员**。
- 当 Widget 需要重写 Timer 的虚函数时。比如上面的例子中，需要重写 onTick。单纯的复合是做不到的

3. private 继承可以使用复合取代，具体方法**内嵌一个类 public 继承基类，将该内嵌类放在 private 下**，这样的优点有：
   - 该类的派生类不能重新定义内嵌类中的成员函数
   - 该类的编译依存性被降至最低，因为可以在该类中声明一个指向内嵌类的指针，这样内嵌类中用到的一些类成员就不会与该类有编译依存

■ **使用复合的例子：**

```
class Widget {
private:
    class WidgetTimer: public Timer {
    public:
        virtual void onTick() const;
    };
    WidgetTimer timer;
};
```

通过复合来表现"is-implemented-in-terms-of"关系，实现较为复杂，但是具有下列优点：

- 如果希望禁止 Widget 的子类重定义 onTick。因为派生类无法访问私有的 WidgetTimer 类
- 可以减小 Widget 和 Timer 的编译依赖。如果是 private 继承，在定义 Widget 的文件中势必需要引入#include"timer.h"。但如果采用复合的方式，可以把 WidgetTimer 放到另一个文件中，在 Widget 中使用 WidgetTimer*并声明 WidgetTimer 即可

4. 使用 private 继承的地方：
   - 当一个类是空类，即不带任何数据，没有任何非 static 变量，没有虚函数也没有虚基类时，这种类的对象一般不使用任何空间，因为没有任何隶属于对象的数据需要存储，但是往往有 typedef，enums，static 成员变量。如果有一个类继承单继承该空类，此时不会占用内存，但是以该类为成员是会占用内存的，这叫“空白基类最优化”
   - 当两个类不是“is-a”关系，其中一个类需要访问另一个类的 protected 成员或需要重新定义其虚函数，则可以使用 private 继承

总的来说，在需要表现"is-implemented-in-terms-of（根据某物实现出）"关系时。如果一个类**需要访问基类的 protected 成员，或需要重新定义其一个或多个 virtual 函数**，那么使用 private 继承。否则，在考虑过所有其它方案后，仍然认为 private 继承是最佳办法，才使用它

## 规则 40 谨慎使用多重继承

使用多继承时，一个问题是不同基类可能具有相同名称，产生歧义（即使一个名字可访问，另一个不可访问）

```
class Borrowableitem { // 图书馆允许你借某些东西
public:
void checkout(); // 离开时进行检查
  ...
) ;
class ElectronicGadget {
private:
bool checkout() const; // 执行自我检测，返回是否测试成功
  ...
) ;
class MP3Player: // 注意这里的多重继承
public Borrowableltem, //（某些图书馆愿意借出 MP3 播放器）
public ElectronicGadget
{...};
MP3Player mp;
mp.checkout(); // 歧义！调用的是哪个 checkOut?
```

为了解决这个歧义，你必须明白指出你要调用哪一个 base class 内的函数：

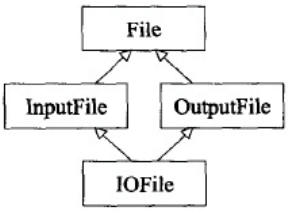
```
mp.Borrowableitem::checkout(); // 歧义！调用的是哪个 checkOut?
```

一般有两种方式使用多继承：

■ 一般的多重继承

如果某个基类到派生类之间存在多条路径，则**派生类会包含重复的基类成员**

```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```

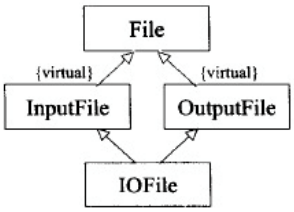



■ 虚继承（此时基类是虚基类）

如果某个基类到派生类之间存在多条路径，**派生类只包含一份基类成员**，但是这会带来额外开销

- 为避免重复，编译器必须提供一些机制，后果就是 virtual 继承的那些 classes 所产生的对象往往比 non-virtual 继承的体积大，访问 virtual basec lasses 的成员变量时，速度也更慢
- virtual base 的初始化由继承体系中的最底层 class 负责，这会带来开销
  - ◆ classes 若派生自 virtual bases 而需要初始化，必须认知其 virtual bases——无论那些 bases 距离多远
  - ◆ 当一个新 derived class 加入继承体系中，它必须承担其 virtual bases 的初始化责任

```
class File { ... };
class InputFile: virtual public File { ... };
class OutputFile: virtual public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```




如果你有一个单一继承的设计方案，而它大约等价于一个多重继承方案，那么单一继承设计方案几乎一定比较受欢迎。如果你唯一能够提出的设计方案涉及多重继承，你应该更努力想一想——几乎可以说一定会有某些方案让单一继承行得通。然而多重继承有时候是完成任务的最简洁、最易维护、最合理的做法，果真如此就别害怕使用它。只要确定，你的确是在明智而审慎的情况下使用它

1. 多重继承的缺点：
   - 如果派生类继承的两个基类中有相同的成员名，则会出现歧义，而不管 private 和 public。这主要是 C++解析重载函数调用的规则规定先找最佳匹配然后才检验可取用性
   - 对于钻石型继承，需要使用虚基类，但是虚继承会有很高成本
2. 虚继承的成本：
   - 虚基类的初始化复杂且不直观，虚基类的初始化由最低层的类来负责
   - 使用虚继承会增加大小、速度、初始化复杂度等等成本，除非是虚基类不带任何数据
3. 多重继承的适用场景：
   - public 继承某个接口类和 private 继承某个协助实现的类

# 7 模板与泛型编程

## 规则 41 了解隐式接口和编译期多态

1. 面向对象设计中的类（class）考虑的是**显式接口**（explicit interface）和**运行时多态**，而模板编程中的模板（template）考虑的是**隐式接口**（implicit interface）和**编译期多态**。
2. 对类而言，显式接口是由**函数签名**表征的，运行时多态由**虚函数**实现
3. 对模板而言，隐式接口是由**表达式的合法性**表征的，编译期多态由**模板初始化和函数重载**的解析实现

4. 隐式接口：每一个类模板规定了作为模板 T 的类必须支持哪些接口，这组接口是隐式接口
5. 编译期多态：以不同的模板参数实例化的模板函数会导致调用不同的函数，这是编译期多态

## 规则 42 了解 typename 的双重意义

1. 声明模板参数时，前缀关键字 class 和 typename 可互换

```
template<class T> class Widget;
template<typename T> class Widget;
```

2. 模板内出现的名称如何相依于某个模板参数，则称之为从属名称，如果从属名称在类内呈嵌套状，则称之为嵌套从属名称，如下列的 C::const_iterator，该名称还是一个嵌套从属类型名称；如果不依存模板参数，则称之为非从属名称，如下列的 int

```
template<typename C>
void print2nd(const C& container)
{
    C::const_iterator *x;
    int val = *x;
}
```

我们认为 C::const_iterator 表示容器 C 的迭代器类型，因此上述代码定义一个该迭代器类型的指针。但是这是一种先入为主的思想。如果 C::const_iterator 不是一个类型呢？比如恰巧有个 static 成员变量被命名为 const_iterator，或如果 x 碰巧是个 global 变量名称？那样的话上述代码就不再是声明一个 local 变量，而是一个相乘动作

3. 嵌套从属名称有可能导致解析困难，因为无法知道 C::const_iterator 是指代的类型还是指代的类内的 static 成员变量，为了区分这种情况，在 C::const_iterator 为类型时，在其前面添加 typename，即**任何时候当想要在模板中指涉一个嵌套从属名称，就必须在紧邻它的前一个位置放上关键字 typename**

```
template<typename C>
void print2nd(const C& container)
{
    typename C::const_iterator *x;
    ...
}
```

4. 除了下面 2 个例外，任何时候当你想要在 template 中指涉一个嵌套从属类型名称，就必须在紧临它的前一个位置放上关键字 typename：
   - typename **不可出现在 base classes list 内的嵌套从属名称之前**
   - typename 也**不可出现在成员初始值列表中**作为 base class 修饰符

```
template<typename T>
class Derived : public Base<T>::Nested { //typename 不可出现在此
public:
    explict Derived(int x) : Base<T>::Nested(x) //typename 也不可出现在此
    {
        typename Base<T>::Nested temp;  //这里必须使用 typename
    }
};
```

typename 相关规则在不同的编译器上有不同的实践。某些编译器接收的代码原本该有 typename 却遗漏了；原本不该有 typename 却出现了；还有少数编译器（通常是较旧版本）根本就拒绝 typename。这意味 typename 和“嵌套从属名称”之间的互动，也会在移植性方面给你带来一些麻烦

## 规则 43 学习处理模板化基类内的名称

1. **对于模板基类中声明的函数，其派生类不能直接进行调用**，主要是**因为如果对基类进行了模板特例化，此特例化版本可能不提供和一般性模板相同的接口**（即可能不含一般性模板中声明的某些函数），此时派生类直接调用该函数是无法找到的

假设以下 MsgSender 类可以通过两种方式发送信息到各个公司：

```
template<typename Company>
class MsgSender{
public:
    ...
    //1.发送原始文本
    void sendClear(...)
    {
        ...
        Company c;
        c.sendCleartext(...);
    }
    //2.发送加密后的文本
    void sendSecret(...) {...}
    ...
};
```

假设我们有时候想要在每次送出信息时志记(log)某些信息。因此有了以下派生类：

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company>{
public:
    ...
    void sendClearMsg(...)
    {
        //将“传送前“的信息写至 log;
        sendClear(...);             //调用 base class 函数，无法通过编译
        //将”传送后“的信息写至 log;
```

```
    }
    ...
};
```

现在问题是，如果有一个公司 CompanyZ 只支持加密传送，那么泛化的 MsgSender 就不适合，因此需要为其产生一个特化版的 MsgSender：

```
template<> // 特化
class MsgSender<CompanyZ>{ // 针对 CompanyZ 全特化
public:
    ...
    //只支持发送加密后的文本
    void sendSecret(...) {...}
    ...
};
```

因此，当 base class 被指定为 MsgSender 时，其内不包含 sendClear 方法，那么 derived class LoggingMsgSender 的 sendClearMsg 方法就会调用不存在的 sendClear

2. 解决方法，它们会通知编译器进入 base class 作用域查找继承而来的名称：
   - 在派生类中需要调用的函数之前**使用 this->**

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company>{
public:
    ...
    void sendClearMsg(...)
    {
        //将“传送前“的信息写至 log;
        this->sendClear(...);     //成立，假设 sendClear 将被继承
        //将”传送后“的信息写至 log;
    }
    ...
};
```

   - **使用 using** 声明引入基类中需要调用的函数

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company>{
public:
    //告诉编译器，请它假设 sendClear 位于 base class 内
    using MsgSender<Company>::sendClear;
    ...
    void sendClearMsg(...)
    {
        //将“传送前“的信息写至 log;
        sendClear(...);     //成立，假设 sendClear 将被继承
        //将”传送后“的信息写至 log;
    }
    ...
};
```

   - **使用作用域符**明确指出被调用的函数位于基类内

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company>{
public:
    ...
    void sendClearMsg(...)
    {
        //将“传送前“的信息写至 log;
        MsgSender<Company>::sendClear(...); //成立，假设 sendClear 将被继承
        //将”传送后“的信息写至 log;
    }
    ...
};
```

   这种方法往往最不让人满意，因为如果被调用的是 virtual 函数，这样会关闭“virtual 绑定行为”

3. 要注意的是，它们只是通知编译器进去查找。如同上面的 CompanyZ，**如果基类不存在相应名称，编译器还是会报错**

## 规则 44 将与参数无关的代码抽离模板

1. 模板提供的是编译期的多态，即使你的代码看起来非常简洁短小，生成的二进制文

件也可能包含大量的冗余代码。因为模板每次实例化都会生成一个完整的副本，所以其中与模板参数无关的部分会造成代码膨胀

2. 模板生成多个类和多个函数，所以任何模板代码都不该与某个造成膨胀的模板参数产生相依关系，**把模板中参数无关的代码重构到模板外便可以有效地控制模板产生的代码膨胀**
3. 因**非类型**模板参数而造成的代码膨胀，往往**可消除**，做法是**以函数参数或类成员变量替换非类型模板参数**，形成一个只带有模板参数的基类，派生类再指定非类型模板参数，这样对同一类型就会只有一份共享的基类

```
//非类型模板参数造成代码膨胀
template<typename T, int n>
class Square{
public:
    void invert();  //求逆矩阵
};
//以下会实例化两个类：Square<double, 5>和 Square<double, 10>
//会具现化两份 invert。除了常量 5 和 10，两个函数的其它部分完全相同
Square<double, 5> s1;
Square<double, 10> s2;
s1.invert();
s2.invert();
//以下，使用函数参数消除重复
template<typename T>
class SquareBase{
protected:
    //以下函数只是作为避免代码重复的方法，并不应该被外界调用，
    //同时，该函数希望被子类调用，因此使用 protected
    void invert(int size);
};
template<typename T, int n>
class Square:private SquareBase<T>{//只要 T 相同，都会使用同一份父类实例，
private:                           //因此，只有一份 invert(int size)
    using SquareBase<T>::invert;
public:
    //调用父类 invert 的代价为零，因为 Square::invert 是隐式的 inline 函数
    void invert(){ this->invert(n); }
}
```

上面代码还需要解决的问题是父类如何访问矩阵数据。原本这些数据在派生类中，但是因为 invert 核心代码转移到了父类，那么父类必须有办法访问这些数据。可以在调用 SquareBase::invert 时把内存地址也一起告知父类，但如果矩阵类中有很多函数都需要这些信息就需要为每个函数添加一个这样的参数。因此，可以把数据地址直接放在父类中

4. 因**类型**参数而造成的代码膨胀，往往**可降低**，做法是**让带有完全相同的二进制表述的具体类型共享实现码**，例如
   - int 和 long 具有相同的二进制表述，则 vector<int>和 vector<long>的成员函数可能完全相同，然而有些连接器会合并完全相同的实现代码，但有些不会，后者意味某些 templates 被具现化为 int 和 long 两个版本，并因此造成代码膨胀。
   - 在大部分平台上，所有指针类型都有相同的二进制表述。List<int*>，List<const int*>，List<double*>的底层实现也是一样的。但因为指针类型不同，也会实例化为多份模板类如果某些成员函数操作强型指针(T*)，应该令它们调用另一个操作无类型指针(void*)的函数，后者完成实际工作

## 规则 45 运用成员函数模板接受所有兼容类型

1. 需要使用成员函数模板的一个例子是**构造函数和拷贝赋值运算符**。例如，假设 SmartPtr 是一种智能指针，并且它是一个 template class。现在有一个继承体系：

```
class Top {...};
class Middle : public Top {...};
class Bottom : public Middle {...};
```

现在希望通过一个 SmartPtr<Bottom>或 SmartPtr<Middle>来初始化一个 SmartPtr<Top>。如果是指针，即 Middle*和 Bottom*可以隐式转换成 Top*，问题是：**同一个 template 的不同具现体之间不存在什么与生俱来的固有关系**，即使具现体之

间具有继承关系。因此，SmartPtr<Bottom>或 SmartPtr<Middle>并不能隐式转化成 SmartPtr<Top>。因此，我们需要一个构造函数模板，来实现这种转换：

```
template<typename T>
class SmartPtr{
public:
    //构造函数模板
    //意思是：对任何类型 T 和 U，可根据 SmartPtr<U>生成一个 SmartPtr<T>
    template<typename U>
    SmartPtr(const SmartPtr<U> &other)
     : heldPtr(other.get()) {...}
    //原始指针为 private 成员，需要一个接口来获取
    T* get() const {return heldPtr;}
    ...
private:
    T* heldPtr;   //智能指针所持有的原始指针
};
```

我们当然不希望一个 SmartPtr<Top>可以转化成 SmartPtr<Bottom>或 SmartPtr<Middle>，heldPtr(other.get())为此提供了保证。这个行为只有当“存在某个隐式转换可将一个 U*指针转为一个 T*指针”时才能通过编译

最后需要指明的是：**成员函数模板并不改变语言规则**，而语言规则说，如果程序需要一个 copy 构造函数，你却没声明它，编译器会为你暗自生成一个。因此，使用 member templates 实现一个泛化版的 copy 构造函数时，编译器也会合成一个“正常的”copy 构造函数

2. 使用成员函数模板生成“可接受所有兼容类型”的函数，在智能指针中使用的最为频繁
3. 如果声明了成员函数模板用于“泛化复制构造”或“泛化赋值运算符”，依旧需要声明正常的复制构造函数和重载赋值运算符

## 规则 46 需要类型转换时请为模板定义非成员函数

1. **在模板的实参推导过程中从不将隐式类型转换函数纳入考虑。**下列将条款 24 中的 Rational 和 operator*改成了 template，混合运算会编译错误：

```
template<typename T>
class Rational{
public:
    Rational(const T &numerator = 0, const T &denominator = 1);
    const T numerator() const;
    const T denominator() const;
    ...
};

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

Rational<int> oneHalf(1,2);
Rational<int> result = oneHalf * 2   //编译错误
```

将 oneHalf 传递给 operator*时，它将 T 推断为 int，因此期待第二个参数也为 Rational，但是第二个参数为 int，前面我们说了，template 实参推导过程中从不将隐式类型转换函数纳入考虑。因此编译错误

那么解决办法是什么？**在 class template 将其声明为 friend，从而具现化一个 operator***，具现化后就可以不受 template 的限制了：

```
template<typename T>
class Rational{
public:
    ...
    //也可以是 Rational<T>，但是省去<T>更简洁
    friend const Rational operator*(const Rational &lhs,const Rational &rhs)
    {
        return Rational(lhs.numerator() * rhs.numerator,
                        lhs.denominator() * rhs.denominator());
```

```
    }
};
```

如果上面只有函数声明，而函数定义在类外，那么会报链接错误。当传入第一个参数 oneHalf 时，会具现化 Rational<int>，编译器也就知道了我们要调用传入两个 Rational<int>的版本，但是那个函数只在类中进行了声明，并没有定义，不能依赖类外的 operator*template 提供定义，我们必须自己定义，所以会出现链接错误。解决方法就是像上面一样**在类内定义看起来有点像是 member 的友元函数，但是因为 friend 关键字**，所以实际是 non-member 函数，如果去掉 friend 关键字，就成了 member 函数，但是此时参数也只能有 1 个，就不能实现所有参数的隐式转换

上面的代码可能还有一个问题，虽然有 friend，上述函数仍是隐式的 inline。如果函数实体代码量较大，可以令 operator*不做任何事，只调用一个定义与 class 外部的辅助函数（当然这里没必要，因为本身只有 1 行）

```
template<typename T> class Rational;

//helper template
template<typename T>
const Rational<T> doMultiply(const Rational<T>& lhs, const Rational<T>& rhs);

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

2. 在编写一个类模板时，而该类模板需要定义某些**支持模板类型隐式转换的函数**，此时需要将这些函数定义为“**类模板内部的友元函数，并且需要在类内定义或者在类内定义部分调用外部辅助函数**”

## 规则 47 使用特征类（traits classes）表现类型信息

1. traits 并不是 C++关键字而是一种技术，traits 信息必须位于类型自身之外，标准技术是把它放进一个模板及其一或多个特化版本中
2. 例如 iterator_traits，其针对每一个类型 IterT，在 struct iterator_traits<IterT>内一定声明某个 typedef 名为 iterator_category，这个 typedef 用来确认 IterT 的迭代器分类，即：
   - 要求每一个“用户自定义的迭代器类型”必须嵌套一个 typedef，名为 iterator_category，用来确认迭代器类型
   - 在 iterator_traits 内直接使用 iterator_category 即可

   template<typename IterT>
   struct iterator_traits{
       typedef typename IterT:: iterator_category iterator_category;
   }
   但对指针类型需要进行偏特例化
   template<typename IterT>
   struct iterator_traits<IterT*>{
       typedef random_access_iterator_tag iterator_category;
   }
3. 使用 traits 类
   - 建立一组重载函数或函数模板，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的 traits 信息想应和
   - 建议一组控制函数或函数模板，调用上述函数并传递 traits 类所提供的信息
4. Traits 类使得“类型相关信息”在编译器可用，它们以模板和模板特例化完成实现，

在使用是需要整合重载技术，即可在编译期对类型执行 if else 测试

## 规则 48 认识模板元编程

1. 概念：模板元编程是编写基于 C++的模板程序并执行于编译期的过程
2. 模板元编程有两大优点：
   - 让某些事情更容易
   - 可将工作从运行期转移到编译期
3. 模板元编程中的循环通过递归模板具现化来实现
4. 模板元编程可被用来生成“基于政策选择组合”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

## 规则 49 了解 new-handler 的行为

1. 说明：当 operator new 无法满足某一内存分配需求时，它会抛出异常，当 operator new 抛出异常以反映一个未获满足的内存需求之前，它会先调用一个客户指定的错误处理函数，一个所谓的 new-handler，为了指定该函数，客户必须调用 set_new_handler 函数，该函数声明于<new>的标准程序库函数

```
namespace std{
    typedef void (*new_handler)();
    new_handler set_new_handler(new_handler p) throw();
}
```

2. set_new_handler 的参数是个指针，指向 operator new 无法分配足够内存时该被调用的函数，返回值也是个指针，指向 set_new_handler 被调用前正在执行的那个 new_handler 函数
3. 设计良好 new_handler 函数必须做下列事情：
   - 让更多内存可被使用：这可以造成 operator new 内的下一次内存分配动作可能成功
   - 安装另一个 new handler：如果目前这个 new handler 无法取得更多可用内存，或许它知道另外哪个 new handler 有此能力，果真如此，目前这个 new handler 就可以安装另外哪个 new handler 以替换自己，下次当 operator new 调用 new handler，调用的将是最新安装的那个
   - 卸除 new handler：将 null 指针传给 set_new_handler，如果没有安装任何 new handler，则 operator new 会在内存分配不成功时抛出异常
   - 抛出 bad_alloc（或派生自 bad_alloc）的异常：这样的异常不会被 operator new 捕获
   - 不返回：通常调用 abort 或 exit
4. 通常会写一个模板类完成自定义的 set_new_handler 及其管理，然后需要使用的类继承该类
5. 旧版本的 new 不抛出异常，当分配失败时会返回 null，为了支持旧版本，可以给 new 传递参数 std::nothrow，此时如果分配失败则会返回 null，但是这只能保证 new 不抛出异常，后续的构造函数调用还是可能抛出异常

## 规则 50 了解 new 和 delete 的合理替换时机

1. 替换 new 和 delete 的理由：
   - 用来检测运用上的错误：自定义的 operator new 可以分配额外的空间来放置签名，这样 operator delete 可以通过签名来记录是否该指针有错误
   - 为了强化效能：通常定制版的 operator new 和 operator delete 性能胜过缺省版本
   - 为了收集使用上的统计数据
   - 为了增加分配和归还的速度：泛用型分配器往往比定制型分配器慢，特别是当定制型分配器专门针对某特定类型的对象设计时
   - 为了降低缺省内存管理器带来的空间额外开销：泛用型内存管理器往往不只比

定制型慢，往往还使用更多内存
- 为了弥补缺省分配器中的非最佳齐位：
- 为了将相关对象成簇集中
- 为了获得非传统的行为：

2. 替换 new 和 delete 的时候需要考虑字节对齐的问题

## 规则 51 编写 new 和 delete 时需固守常规

1. 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，类专属版本则还应处理“比正确大小更大的错误申请”，一般是使用标准 operator new 来处理该错误申请，operator new 伪代码

```
void* operator new(std::size_t size) throw(bad_alloc)
{
    using namespace std;

    if(size == 0){              // 处理 0byte 申请
        size = 1;
    }

    while(true){                // 尝试分配 size bytes
        if(分配成功)
        return (一个指针，指向分配得来的内存);

        // 分配失败，找出目前的 new handling 函数
        new_handler globalHandler = set_new_handler(0);
        set_new_handler(globalHandler);

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

2. operator delete 应该在收到 null 指针时不做任何事，类专属版本则还应处理“比正确大小更大的错误申请”，一般是使用标准 operator delete 来处理该错误申请

## 规则 52 写了定位 new 也要写定位 delete

1. 当写了一个定位 new 时，同时也要提供对应参数的定位 delete，当定位 new 发生异常时 C++运行机制会调用对应的定位 delete，但如果是正常 delete，则使用的是标准 delete
2. 当声明定位 new 和定位 delete 时，注意不要遮掩它们的正常版本

```
void* operator new(std::size_t) throw(std::bad_alloc);              // 正常版本
void* operator new(std::size_t, void*) throw();                     // 定位 new
void* operator new(std::size_t, const std::nothrow_t&) throw();     // 不抛出异常版本
```

## 规则 53 不要轻忽编译器的警告

1. 严肃对待编译器发出的警告消息，努力在你的编译器的最高警告级别下争取“无任何警告”的荣誉
2. 不要过度依赖编译器的报警能力，因为不同的编译器对待事情的态度并不相同，一旦移植到另一个编译器上，原本依赖的警告信息可能消失