让自己习惯c++

条款01: 视c++ 为一个语言联邦

如今的 c++ 已经是一个 **多重泛型编程语言**，一个同时支持过程形式、面向对象形式、函数形式、泛型形式、元编程形式 的语言。将 c++ 看作一个联邦，主要包括四个部分：

- C: c++ 是 c 的继承，但是包含很多 c 语言以外的特性
- Object-Oriented c++: 这部分就是 c with class
- Template C++: 这是 c++ 的泛型编程部分
- STL: STL 是 template 程序库

条款02: 尽量以 const, enum, inline 替换掉 #define

- 对于单纯常量，最好使用 const 对象或者 enums 来替代 #define
- 对于形似函数的宏，最好使用 inline 函数来替代 #define

这条条款实际上意味着，“宁可用编译器替换预编译器”。

```
#define ASPECT_RATIO 1.6
```

上面的这种语句当发生错误时，编译器产生的错误信息可能提到的时 1.6 而非 ASPECT_RATIO。当你查找错误时，这个举动 会徒增难度。

解决的方法是 使用常量替代 上述的宏定义：

```
const double AspectRatio = 1.6;
```

这种方式不仅解决了 编译器无法找到对应的记号的问题，同时也解决了宏定义可能产生多份复制的情况。

常量替换 #define

使用常量替换 #define 还需要讨论两个小问题。

1. 常量指针

因为常量定义通常被放在头文件中，所以为了防止程序某处对其内容进行修改，需要将指针声明为 const，不仅是所指之物。

```
const char* const authorName = "hello world";
```

或者你也可以选择使用某些对象进行替换，比如上例就可以使用 string 对象来实现

```
const std::string authorName("hello world");
```

2. class 专属常量

为了将常量限制在 class scoop 中，且限制只存在一个常量。需要使用 static 关键字来实现。

```
class GamePlayer {
private:
    static const int NumTurns = 5; // 常量声明式
    int scores[NumTurns];
};

const int GamePlayer::NumTurns; // 常量定义式
```

c++ 要求你对你所使用的任何东西提供一个定义式。特别是 class 专属常量，又是 static 且为整数类型。这就需要特殊 处理。只要不取他们的地址，你可以只声明他们而不用定义。而如果你需要获取某个 class 专属常量的地址，或者编译器坚 持要看到一个定义式，你就必须提供上面最后这样类似的定义式。其中的定义式部分并没有提供特定的初值，因为在声明中已 经给出了初值，因此在定义式中不能再设置。

但是有时候，编译器可能不允许 “static 整型 class 常量完成 in class 初值设定”。而如果你又坚持要在编译期间知道 数组的大小，你可以改用所谓的 “the enum 可权充 ints 使用” 的技巧。

```
class GamePlayer {
private:
    enum { NumTurns = 5 };
    int scores[NumTurns];
};
```

这种 enum hack 方式值得学习，它具有几点好处：

1. enum hack 行为类似 #define，不能取用地址。当你不希望别人通过 pointer 或者 reference 来获取该常量可以使用。
2. 实用主义，很多代码使用了它。enum hack 实际上是 “模板元编程” 的基础技术。

inline 函数替换 #define

通常希望使用 宏定义 的方式来实现一个看起来像函数的东西，因为不会产生调用带来的额外开销。但是有时候会产生更加难 以控制的局面。

```
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))

int a = 5, b = 6;
CALL_WITH_MAX(++a, b); // 比较完成后，a = 7
CALL_WITH_MAX(++a, b + 10); // 比较完成后，a = 8
```

上面这段代码显然不会按照原有的设想运行，但是如果你使用 template inline 函数来书写则不会产生这种结果。

```
template <typename T>
inline void callWithMax(const T& a, const T& b) {
    f(a > b ? a : b);
}
```

除了上面这点保证，inline 也会保证遵守作用域原则，而 #define 则不会。

条款03: 尽可能使用 const

- 将某些东西声明为 const 可以帮助编译器侦测出错误用法。
- 编译器强制实施 bitwise constness，但你编写程序时应该使用 logical constness
- 当 const 和 non-const 成员函数有着实质等价的实现时，令 non-const 版本调用 const 版本可避免代码重复。

---

const 允许你指定一个语义约束，而编译器会强制实施这项约束。

关于指针的 const 使用：

- const 在 * 的左边，意味着被指物是常量
- const 在 * 的右边，意味着指针自身是常量

```
int a = 10;
int b = 11;
const int* p = &a;
int* const q = &a;
*p = 12; // 错误
p = &b;  // 正确
*q = 12; // 正确
q = &b;  // 错误
```

同样的,使用 STL 迭代器，也需要区分不同的 const：

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

const std::vector<int>::iterator iter = vec.begin();
std::vector<int>::const_iterator cIter = vec.begin();
iter++; // 错误
cIter++; // 正确
*iter = 12; // 正确
*cIter = 12; // 错误
```

上述代码中前者实际相当于 iter 是不可更改的，即 `int* const` 类型，而后者则是 `const int*` 类型，即被指物是 不可更改的。

在函数声明中使用 const 进行限制，包括函数返回值、参数、函数自身，可以降低因使用者错误使用而造成的意外。

```
class Rational{
public:
    const Rational operator*(const Rational& lhs, const Rational& rhs);
};

Rational a, b, c;
...
if (a * b = c) {...} // == 意外书写为了 =
```

在上述的代码中，使用 const 来限定返回值可以防止 `(a * b) = c` 这类代码通过编译，因为这显然是不合理的。

const 成员函数

将 const 实施于成员函数，是为了确认该成员函数可作用 const 对象。

1. const 成员函数使得 class 接口比较容易理解
2. 他们使得 “操作 const 对象” 成为可能。

> [!tip] 两个成员函数，如果只是常量性不同，可以被重载

```
class TextBlock {
public:
    const char& operator[](std::size_t position) const {
        return text[position];
    }
    char& operator[](std::size_t position) {
        return text[position];
    }
private:
    std::string text;
};

void print(const TextBlock& ctb) {
    std::cout << ctb[0];
}

TextBlock tb("hello");
TextBlock ctb("hello");
tb[0] = 'x';  // 正确
ctb[0] = 'x'; // 错误
```

成员函数是 const 的含义

对于这个观点，存在两个流行概念：bitwise constess 和 logical constness

bitwise const

此阵营的人认为，**成员函数只有不更改对象之任何成员变量时才可以说是 const 的。**但是这也存在漏洞，比如将数据 存储为 char* 而非 string 的话就可能会产生问题。

```
class CTextBlock {
public:
    char& operator[] (std::size_t position) const {
        return pText[position];
    }

private:
    char* pText;
};
```

尽管 operator[] 中并没有对任何数据进行更改，但是却返回了一个指向对象内部值的 reference。如果调用者在后续对返 回值进行更改，仍然是可行的，但是却违背了 const 的约定。

logical constness

这一派则主张，一个 const 成员函数可以修改它所处理的对象内的某些 bits，但是只有在用户侦测不出的情况下才可以。

```
class CTextBlock {
public:
    std::size_t length() const;
private:
    char* pText;
    mutable std::size_t textLength;
    mutable bool lengthIsValid;
};

std::size_t CTextBlock::length() const {
    if(!lengthIsValid) {
        textLength = std::strlen(pText);
        lengthIsValid = true;
    }
    return textLength;
}
```

上面这段代码显然不符合 bitwise const，因为 length 函数改变了其中的值，但是却符合 logical constness。此外， 其中使用 mutable 关键字进行限定，其目的是当编译器坚持 bitwise const 时，使用该关键字允许释放掉 non-static 成员变量的 bitwise constness 约束。

在 const 和 non-const 成员函数中避免重复

随着代码越来越复杂，const 与 non-const 代码之间可能出现大量的重复内容，比如：

```
class TextBlock {
public:
    const char& operator[] (std::size_t position) const {
        // 边界检查
        // 日志记录
        // 数据完整性校验
        return text[position];
    }

    char& operator[] (std::size_t position) {
        // 边界检查
        // 日志记录
        // 数据完整性校验
        return text[position];
    }
private:
    std::string text;
};
```

上面这段代码通过使用 non-const 版本调用 const 函数的方式来进行简化，能够大大降低代码的复杂度。

```
class TextBlock {
public:
    const char& operator[] (std::size_t position) const {
        // 边界检查
        // 日志记录
        // 数据完整性校验
        return text[position];
    }

    char& operator[] (std::size_t position) {
        return const_cast<char&>(
            static_cast<const TextBlock&> (*this)[position]
        );
    }
private:
    std::string text;
};
```

如果反过来使用 const 函数调用 non-const 函数，怎么样？不要这么做，因为 const 成员函数承诺不对其中的对象进行 逻辑状态的更改，如果反向调用则会产生修改的风险。

条款04: 确定对象被使用前已经被初始化

- 为内置型对象进行手工初始化，因为 c++ 不保证初始化他们。

- 构造函数最好使用成员初值列，而不要在构造函数中使用赋值操作。初始化顺序于声明次序相同，而与初值列顺序无关。
- 为免除"跨编译单元之初始化次序"问题，使用 local static 对象替换 non-local static 对象。

关于 "将对象初始化"，c++ 似乎反复无常。但是现在，我们有一些规则，描述"对象的初始化动作何时一定发生，何时不一定 发生"。

通常，如果使用 c part of c++ 而且初始化可能招致运行期成本，那么就不会保证发生初始化。但是当你进入 non-c part of c++，则具有保证。

这似乎是个无法决定的状态，最佳的处理方法就是永远在使用对象之前先将它初始化。

对于内置类型以外的其他东西，初始化责任则落在了构造函数身上。就需要确保每一个构造函数都将对象的每一个成员初始化。

```
class PhoneNumber {};
class ABEntry {
public:
    ABEntry(const string& name, const string& address, const list<PhoneNumber>&
phones);
private:
    string theName;
    string theAddress;
    list<PhoneNumber> thePhones;
    int numTimesConsulted;
};

ABEntry::ABEntry (const string& name, const string& address, const
list<PhoneNumber>& phones)
    : theName(name), theAddress(address), thePhones(phones), numTimesConsulted(0)
{}
```

上面使用 member initialization list 来实现初始化，而非在 constructor 内使用赋值来进行初始化操作。因为，前 者发生在 constructor 开始执行本体和 default constructor 之前。这种方式效率更高。

许多 classes 拥有多个构造函数，每个构造函数都有自己的成员初值列。如果这种 classes 存在许多成员变量，多份成员 初值列的存在就会导致重复。这种情况下，可以将部分 "赋值表现和初始化一样好" 的成员变量放置在 constructor 进行 进行操作。并且可以将重合部分提取为 private 的"伪初始化"函数。

关于 "成员初始化次序"，c++ 中有着严格的次序规定：base classes 更早于 derived classes 被初始化。成员变量总是 以声明次序进行初始化，无论成员初值列的次序如何。

不同编译单元内定义之 non-local static 对象

如果你能够按照上面的这些约束进行编码，那么只剩最后一件值得关心的事情 "不同编译单元内定义之 non-local static 对象" 的初始化次序。

static 对象是指 global 对象，定义于 namespace 作用域内的对象，在 class 内、函数内、以及在 file 作用域内被声 明为 static 的对象。他们的寿命从构造出来直至程序结束为止。其中在函数内的 static 对象被称为 local

static 对象。 而其他则都被称为 non-local static 对象。

编译单元是指产生单一目标文件的那些源码。

```
// 1.cpp
class FileSystem {
public:
    std::size_t numDisk() const;
};

extern FileSystem tfs;
```

```
// 2.cpp
class Director {
public:
    Directory(params) {
        std::size_t disks = tfs.numDisks();
    }
};

Directory tempDir(params);
```

上面这两段代码，其中 tfs 以及tempDir 是 non-local static 对象，而且存放在不同的源码文件中。从中可以看到二者 存在直接关系，即 tempDir 需要借助 tfs 完成初始化操作。但是 c++ 没有规定 non-local static 对象在不同编译单元 内定义初始化的顺序，因此很容易产生错误。

解决办法十分简单，通过 singleton 模式，将 non-local static 对象转换成 local static 对象，即将前者放入一个专属函数中实现。

```
// 1.cpp
class FileSystem {
public:
    std::size_t numDisk() const;
};

FileSystem& tfs() {
    static FileSystem fs;
    return fs;
}
```

```
// 2.cpp
class Director {
public:
    Directory(params) {
        std::size_t disks = tfs().numDisks();
    }
```

```
};

Directory& tempDir() {
    static Directory td;
    return td;
}
```

这种结构下，函数十分淡出，可以通过 inline 来实现，尤其是如果他们被频繁调用的话。

> [!note] 在 more effective c++ 中，作者在条款26中提到，不要产生内含 local static 对象的 inline non-member functions。 主要是因为 inline non-member functions 存在内部连接，可能会将 local static 对象复制多份。但是，其中注释 也解释道自 1996年7月，ISO/ANSI 委员会便将 inline 函数的默认连接由内部改为了外部。因此现在可以使用 inline 来标记存在 local static 对象的 non-member functions。

构造/析构/赋值运算

条款05: 了解 c++ 默默编写并调用哪些函数

- 编译器可以暗自为 class 创建 default 构造函数、copy 构造函数、copy assignment 操作符 以及 析构函数。

即使你写了一个空类，编译器仍然会为它声明一些方法：

```
class Empty {};
// 等价于
class Empty {
public:
    Empty() {}
    Empty(const Empty& rhs) {}
    ~Empty() {}

    Empty& operator=(const Empty& rhs) {}
};
```

编译器创建的这些函数中 default constructor/destructor 主要用来给编译器用来防止 “幕后” 代码，例如调用 base classes 和 non-static 成员变量的构造函数和析构函数。

至于 copy 构造函数 和 copy assignment 操作符，编译器创建的版本只是简单的将每一个 non-static 成员变量拷贝到 目标对象。

当编译器无法产生合适的 copy assignment 时，它会放弃产生。

```
template <class T>
class NamedObject {
public:
    NameObject(std::string& name, const T& value);
private:
    std::string& nameValue;
    const T objecteValue;
};
```

显然上面这个类无法生成 copy assignment，因为 nameValue 是一个 reference，而 reference 没有办法被重新赋值。

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

```
class Derive : public Base { }
```

上面的 Derive 类也不会产生 copy assignment，因为 base class 将 operator= 设置为了 private，因此并不可以使用赋值的方式处理 Base 对象，也就无法正确处理 Derive。

条款06: 若不想使用编译器自动生成的函数，就该明确拒绝

- 为驳回编译器自动提供的机能，可以将相应的成员函数声明为 private 并且不予实现。使用像 Uncopyable 这样的 base class 也是一种方法。

有时候，我们不希望某些编译器自动生成的函数，那么我们就需要用某种方法拒绝这些函数的生成与调用。

对于 default constructor，只需要自行编写其他 constructor 就会避免其被调用。

对于 copy constructor/assignment 则不一样，可以使用 private 来限制其调用。

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

用这种方式实现 class 可以当客户企图拷贝 HomeForSale 对象，编译器会阻挠他。如果你不慎在 member 函数或者 friend 函数中这么做，则连接器会发出报错。

使用继承一个 private copy base class 的方式可以将报错提前至编译期。

```
class Uncopyable {
protected:
    Uncopyable() {}
    ~Uncopyable() {}
private:
    Uncopyable(const Uncopyable&);
    Uncopyable& operator=(const Uncopyable&);
};

class HomeForSale : public Uncopyable {};
```

上面这种实现非常微妙，但是功能强大。除了这种方法，你可以使用(继承) Boost 提供的版本，类名为 noncopyable。

条款07: 为多态基类声明 virtual 析构函数

- polymorphic base classes 应该声明一个 virtual 析构函数。任何拥有 virtual 函数的 class 应该拥有一个 virtual destructor。
- classes 的设计目的如果不是作为 base classes 使用，或不是为了具备多态性，就不该声明 virtual 析构函数。

在继承体系中，可能出现的一个情况是使用 factory 函数返回一个指向 新生成的 derived class 对象的 base class 指 针。当需要进行清理，调用 delete 时则需要调用 dervied class 对象的 destructor 函数，否则很容易出现 “局部销毁” 现象。使用 virtual destructor 能够很好的避免这个问题。

```
class TimeKeeper {
public:
    TimeKeeper();
    virtual ~TimeKeeper();
};

class AtomicClock : public TimeKeeper {};
class WaterClock : public TimeKeeper {};
class WristWatch : public TimeKeeper {};

TimeKeeper* ptk = getTimeKeeper();
delete ptk;
```

通常 base class 都会使用一个 virtual destructor，同时也可能使用其他 virtual function。而如果一个 class 不 企图当作 base class，而令其析构函数为 virtual 则是一个馊主意。这主要与 virtual 函数的实现细节有关系，它通过 vptr 和 vitual table 实现。这会使得对象的存储大小增加，同时也会降低可移植性。

最后得出的心得是：只有当 class 内含有至少一个 virtual 函数，才为它声明 virtual 析构函数。

还需要注意的是一些STL中不提供 virtual 的 class，当发生继承关系时要慎重考虑 delete。

```
class SpecialString : public std::string {};
SpecialString* pss = new SpecialString("Impending Doom");
std::string* ps;
ps = pss;
delete ps; // 错误，因为 string 并不提供 virtual destructor，因此会发生局部销毁而导致资源泄露
```

有时候希望拥有一个抽象 class，这就需要 pure virtual 函数。而有时没有办法找到一个好的 virtual 函数，可以使用 pure virtual destructor。但是一定要提供一份定义，因为后续继承会调用。

```
class AWOV {
public:
    virtual ~AWOV() = 0;
};

AWOV::~AWOV() {}
```

最后仍要强调 “virtual 析构函数” 只适用于带多态性质的 base classes 身上，如果没有多态性质，该操作只会徒增成本。

条款08: 别让异常逃离析构函数

- 析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析构函数应该捕捉任何异常，然后吞下他们或结束程序。
- 如果客户需要对某个操作函数运行期间抛出的异常做出反应，那么 class 应该提供一个普通函数（而非在析构函数中）执行该操作。

```
class Widget {
public:
    ~Widget();
};

void doSomething {
    std::vector<Widget> v;
}
```

上述代码中，在 doSomething 的结尾 vector v 会被销毁，它有责任销毁其中的所有 Widget。而销毁过程中，如果产生 两个异常，则程序会自动停止。

因此最佳方法是在 destructor 中就处理异常，并且避免在 catch 子句中产生新的异常，或者可以完全不做多余处理。

```
class DBConnection {
public:
    static DBConnection create();
    void close();
};

class DBConn {
public:
    ~DBConn() {
        try {
            db.close();
        } catch (...) {
            // std::abort();
        }
    }
private:
    DBConnection db;
};
```

条款09: 绝不再构造和析构过程中调用 virtual 函数

- 在构造和析构期间不要调用 virtual 函数，因为这类调用从不下降至 derived class

```
class Transaction {
public:
    Transaction();
    virtual void logTransaction() const = 0;
};

Transaction::Transaction() {
    logTransaction();
}

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

class SellTransaction : public Transaction {
public:
    virtual void logTransaction() const;
}

BuyTransaction b;
```

在创建 b 时，会先调用 base class 的 constructor，这也就会先调用 base class 的 virtual logTransaction。不 止 virtual function 会这样，使用 dynamic_cast 和 typeid 也会产生同样的结果。同时这种情况也出现在析构函数中。

大多数情况下，编译器会甄别这种错误，但是有时编译器也会失效。当存在大量初始化相同代码时，通常会使用一个 init 函 数来提取其中重复部分。这就导致了判别失误的情况。

```
class Transaction {
public:
    Transaction() {init();}
    virtual void logTransaction() const = 0;

private:
    void init() {
        logTransaction();
    }
};
```

上面这种情况，编译器可能并不会发现在 constructor 种调用了 pure virtual 函数，但是这可能导致最终程序崩溃。如 果是一个在 Transaction 种实现了的 virtual 函数，那么程序会正常执行下去，但是结果与预期不相同。

不在构造/析构函数中使用 virtual 函数可以避免错误，但是有时候可能希望在对象创建时，调用适当版本的 logTransaction。 这又该如何呢？可以在 class Transaction 中将 logTransaction 函数改为 non-virtual，并传递

必要信息来实现。

```
class Transaction {
public:
    explicit Transaction(const std::string& logInfo) {
        logTransaction(logInfo);
    }
    void logTransaction(const strd::string& logInfo) const;
};

class BuyTransaction : public Transaction {
public:
    BuyTransaction(param) : Transaction(createLogString(param)) {}
private:
    static std::string createLogString(param);
}
```

通过 string param 来控制 logTransaction 的执行。

条款10: 令 operator= 返回一个 referenc to *this

- 令 assignment 操作符返回一个 referenc to *this

赋值操作需要满足 “连锁赋值” 和 “右结合律” 的特点，因此需要将 =,+=,-=,*=,/=等等操作符返回 reference to *this。

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

条款11: 在 operator= 中处理自我赋值

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

“自我赋值” 发生在对象被赋值给自己时。以下均是自我赋值的可能情况：

```
a = a;
a[i] = a[j]; // i == j
*px = *py; // px py 指向同一内存空间
```

有时候当你尝试自行管理资源时，可能会产生 “在停止使用之前以外释放” 的问题。

```
class Bitmap {};
class Widget {
public:
    Widget& operator=(cosnt Widget& rhs) {
        delete bp;
        pb = new Bitmap(*rhs.pb);
        return *this;
    }
private:
    Bitmap* bp; // 指向一个从 heap 分配而来的对象。
};
```

上面这段代码提供了一个非自我赋值安全的 operator=。当执行 delete bp 操作时，如果 rhs 也指向 *this，那么相应 的 rhs.bp 也被删除，那么后续执行赋值时则会出现异常。

证同测试

通过证同测试可以避免自我赋值的发生，从而解决该问题。

```
Widget& Widget::operator=(const Widget& rhs) {
    if (this == &rhs) return *this;

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

异常安全性

但是上面这个方法不具备 “异常安全性”。通常，如果让 operator= 具备 “异常安全性” 往往自动获得 “自我赋值安全” 的 回报。

```
Widget& Widget::operator=(const Widget& rhs) {
    Bitmap* pOrigin = pb;
    pb = new Bitmap(*rhs.pb);
    delete pOrigin;
    return *this;
}
```

这种写法，保证了当 new Bitmap 发生异常时，不会删除原有的 pb 内容。同时，又保障了无论如何删除的对象和新建的对 象是分开的，因此是自我赋值安全的。

copy and swap

相较于前面的手工对语句顺序排序，另一个替代方案是使用 copy and swap 技术。

```
class Widget {
private:
    void swap(Widget& rhs); // 用于交换 *this 和 rhs 的数据
};

Widget& Widget::operator=(const Widget& rhs) {
    Widget tmp(rhs);
    swap(tmp);
    return *this;
}
```

条款12: 复制对象时勿忘其每一个成分

- Copying 函数应该确保复制“对象内的所有成员变量”以及“所有 base class 成分”。
- 不要尝试用某个 copying 函数实现一个 copying 函数。应该将共同技能放进第三个函数中，并由两个 copying 函数共同调用。

当自行设计 copy constructor 以及 copy assignment 时，编译器可能会无法发现其中的错误。

```
void logCall(const std::string& funcName);
class Customer {
public:
    Customer(const Customer& rhs) : name(rhs.name) {
        logCall("...");
    }
    Customer& operator=(const Customer& rhs) {
        loCall("...");
        name = rhs.name;
        return *this;
    }

private:
    std::string name;
};
```

上述代码中看起来完美无瑕，但是当你尝试添加一个新的成员变量时却忘记在 copy constructor/assignment 中添加就会 导致局部拷贝的发生。不幸的是编译器并不会对此做出提醒。

如果还存在继承关系，那么就会发生危险。

```
class PriorityCustomer : public Customer {
public:
    PriorityCustomer(const PriorityCustomer& rhs) : priority(rhs.priority) {
        logCall("***");
    }
    PriorityCustomer& operator=(const PriorityCustomer& rhs) {
        logCall("***");
        priority = rhs.priority;
        return *this;
    }
private:
    int priority;
};
```

上述代码似乎没有问题，但是却忘记对 Customer 部分进行了复制。而默认则调用了 default constructor，这显然与复制 操作大相径庭。因此，任何时候当年你主动为 derived class 编写 copy 函数时，就要注意将 base class 成分也要进行 复制。

```
class PriorityCustomer : public Customer {
public:
    PriorityCustomer(const PriorityCustomer& rhs) : Customer(rhs),
priority(rhs.priority) {
        logCall("***");
    }
    PriorityCustomer& operator=(const PriorityCustomer& rhs) {
        logCall("***");
        Customer::operator=(rhs);
        priority = rhs.priority;
        return *this;
    }
private:
    int priority;
};
```

除了上述问题外，还要注意当 copy assignment 和 copy constructor 存在大量相同代码时，最好的操作是提取出一个 独立函数，并调用。千万不要尝试 assignment 调用 constructor，或者 constructor 调用 assignment。

资源管理

在程序编写过程中需要面对大量资源的管理问题，包括内存、文件描述符、互斥锁、图形界面中的字型和笔刷、数据库连接、网络sockets。 经过训练后，基于对象的资源管理办法，几乎可以消除资源管理问题。

条款13: 以对象管理资源

- 为防止资源泄露，请使用 RAII 对象，他们在构造函数中获得资源并在析构函数中释放资源。
- std::shared_ptr 和 std::auto_ptr 是常用的 RAII class。前者是较佳选择，因为其 copy 行为比较直观，后者需要容忍复制行为。

```
class Investment {};
Investment* createInvestment();
void f() {
    Investment* pInv = createInvestment();

    delete pInv;
}
```

上面这段代码展示了通过工厂函数来提供某个特定的 Investment 对象的过程。看起来没有问题，但实际上当有人开始更改这段代 码时就会产生问题，比如有人在 delete 之前添加 当某些条件满足时 return。这将导致 delete 不会被执行，也就造成了资源 泄露的问题。

修改的方法：把资源放进对象内，依赖“析构函数自动调用机制”确保资源被释放。

```
void f() {
    std::auto_ptr<Investment> pInv(createInvestment());
    ...
}
```

这样在运行结束时，auto_ptr 的析构函数会自动删除 pInv。

这种例子展示了这个观点的两个关键想法：

- 获得资源后立即放进管理对象(RAII)
- 管理对象运用自购函数确保资源被释放

需要注意的是别让多个 auto_ptr 同时指向同一个对象。如果是那样，对象会被删除一次以上。而为了预防这个问题，auto_ptr 有 一个不寻常的性质：若通过 copy 构造函数 或 copy assignment 操作符来复制他们，他们会变成 null，而复制所得的指针将 取得资源的唯一拥有权。

```
std::auto_ptr<Investment> pInv1(createInvestment());
std::auto_ptr<Investment> pInv2(pInv1); // 现在 pInv1->nullptr
```

```
pInv1 = pInv2; // 现在 pInv2->nullptr
```

引用计数型智慧指针

有时候需要允许元素完成正常的复制行为，比如 STL 容器。在这种情况下使用 auto_ptr 就不是最佳选项了。使用 std::shared_ptr 可以允许正常的复制行为。

```
void f() {
    std::shared_ptr<Investment> pInv1(createInvestment());
    std::shared_ptr<Investment> pInv2(pInv1);
}
```

值得注意的是，两者在析构函数中做 delete 而不是 delete[] 操作。因此不要处理动态分配而得来的 array。使用 vector 或者 string 来代替他们。

本条建议不止包括使用 std::auto_ptr 和 std::shared_ptr 这些资源管理类来管理资源，还建议面向无法被这些管理类管理的 资源通过手动编写自己的资源管理类来进行管理。

条款14: 在资源管理类中小心 copying 行为

- 复制 RAII 对象必须一并复制它所管理的资源，所以资源的 copying 行为决定 RAII 对象的 copying 行为
- 普遍而常见的 RAII class copying 行为是：抑制 copying、引用计数。

在条款13中所使用的两个智能指针都旨在管理 heap-based 资源，而有时我们需要管理 non-heap-based 资源时就需要自行撰写 管理对象。

一个例子是为 C API 所提供的 Mutex 互斥对象提供资源管理对象，保证解锁的正常进行。

```
class Lock{
public:
    explicit Lock(Mutex* pm) : mutexPtr(pm) { lock(mutexPtr); }
    ~Lock() { unlock(mutexPtr); }
private:
    Mutex* mutexPtr;
};

Mutex m;
{
    Lock m1(&m); // 进入时加锁，离开此块时 m1 的析构函数自动调用解锁
}
```

看上去一起都好，但是当出现复制操作时问题就会变得复杂。

```
Lock m1(&m);
Lock m2(m1);
```

```

```

你可以有4种选择：

1. 禁止复制

许多时候对允许 RAII 对象被复制并不合理。这个时候就应该禁止这种行为的发生。条款6给出了方案，使用 private 限定 copy 或者继承含有 private copy 的对象：

```
class Lock : private Uncopyable {
public:
    ...
};
```

2. 对底层资源使用 “引用计数法”

有时候，我们希望保有资源直到它的最后一个使用者被销毁。通常使用 std::shared_ptr 来实现引用计数就可以完成这个任务。 需要注意 shared_ptr 的缺省行为是引用为0时删除对象，而我们则需要为 shared_ptr 指定一个删除器 unlock。

```
class Lock{
public:
    explicit Lock(Mutex* pm) : mutexPtr(pm, unlock) { lock(mutexPtr.get()); }
private:
    std::shared_ptr<Mutex> mutexPtr;
};
```

3. 复制底部资源

有时候需要针对一份资源拥有任意数量的附件。而资源管理类的唯一理由是，当不再需要某个附件时确保释放它。此时当发生复制时 应该使用深拷贝。

4. 转移底部资源的拥有权

某些罕见的场合你可能希望确保永远只有一个 RAII 对象指向一个未加工资源，即使 RAII 对象被复制依然如此。此时，资源的拥 有权会从被复制物转移到目标物。这是 auto_ptr 所奉行的复制意义。

条款15: 在资源管理类中提供对原始资源的访问

- APIs 往往要求访问原始资源，所以每一个 RAII class 都应该提供一个 “取得其所管理资源” 的办法。
- 对原始资源的访问可能经由 显式/隐式 转换。一般而言显式更加安全，隐式更加方便。

资源管理类能够很好的对抗资源泄露，但是有时你不得不直接访问原始资源。auto_ptr 和 shared_ptr 就提供了访问原始资源的 能力。

```

```

```
std::shared_ptr<Investment> pInv(createInvestment());
int daysHeld(const Investment* pi);
int days = daysHeld(pInv.get());
```

同时，两者也提供了指针取值操作符(-> *)。

除了这种显示转换方案，也可以通过提供隐式类型转换函数来简化。

```
FontHandle getFont();
void releaseFont(FontHandle fh);
void changeFontSize(FontHandle fh, int size);

class Font {
public:
    explicit Font(FontHandle fh) : f(fh) {}
    ~Font() { releaseFont(f); }
    operator FontHandle() const { return f; } // 隐式转换函数

private:
    FontHandle f;
};

Font f(getFont());
int newFontSize;
changeFontSize(f, newFontSize); // 这里直接使用隐式转换
```

需要注意，隐式转换会增加错误机会。这一点就要求在实现 资源管理对象 时要依据需要完成的工作，谨慎选择使用 显示 或者 隐式 的转换方案。

条款16: 成对使用 new 和 delete 时要采用相同形式

- 如果你在 new 表达式中使用 []，必须在对应的 delete 表达式中使用 []。如果 new 没有使用，delete 也一定不要使用。
- 尽量避免对数组形式做 typedef 动作。

这一点看似简单，但是也有可能出现问题。

```
typedef std::string AddressLines[4];
std::string* pal = new AddressLines;

delete pal; // 错误
delete [] pal; // 正确，因为AddressLines 等价于 string[4]，因此实际上是个数组
```

为了避免此类错误，应该尽量不要对数组形式做 typedefs 动作。

条款17: 以独立语句将 newed 对象置入智能指针

- 以独立语句将 newed 对象存储于智能指针中。如果不这样做，一旦异常被抛出，可能导致难以察觉的资源泄露。

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

processWidget(new Widget, priority());
```

上述代码的调用存在两个问题：

1. 禁止隐式转换

上述代码无法通过编译，因为 shared_ptr 构造函数是一个 explicit 的，无法进行隐式转换操作。因此上述代码 Widget* 无法 转换为 shared_ptr。

2. 可能出现资源泄露

c++ 的函数参数的调用次序不是固定的。如果顺序如下：

1. 执行 new Widget
2. 调用 priority()
3. 调用 std::shared_ptr()

且第二部产生异常，那么程序就不得不跳出。而此时，new 已经申请了资源，但并没有放入到 shared_ptr。因此不会自动回收，那 也就意味着资源泄露。

想要解决这两个问题，就直接将这两步进行拆分。

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

这样就避免了发生隐式转换与跨域语句的操作。

设计与声明

条款18: 让接口容易被正确使用，不易被误用

- 好的接口很容易被正确使用，不容易被误用。你应该在你的所有接口中努力达成这些性质
- “促进正确使用”的办法包括接口的一致性，以及与内置类型的行为兼容
- “阻止误用”的办法包括建立新类型、限制类型上的操作，束缚对象值，以及消除客户的资源管理责任
- shared_ptr 支持定制型删除器。这可以防范DLL问题，可被用来自动解除互斥锁等等。

设计接口的理想原则是，如果客户企图使用某个接口而没有获得他所预期的行为，这个代码就不应该通过编译；如果代码通过了编译， 它的作用就该是客户想要的。这就要求首先必须考虑客户可能做出的错误。

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

上面这段代码存在两个问题：

- 客户可能会按照错误的顺序传递参数
- 客户可能传递一个无效的月份或者天数

可以通过导入新类型而获得预防。

```
struct Day {
    explicit Day(int d) : val(d) {}
    int val;
};

struct Month {
    explicit Month(int m) : val(m) {}
    int val;
};

struct Year {
    explicit Year(int y) : val(y) {}
    int val;
};

class Date {
public:
    Date(const Month& month, const Day& day, const Year& year);
};

Date(Month(2), Day(15), Year(2022));
```

这样做就起到了警示作用，同时又限定了传递顺序。

限定了传递顺序后，我们可以进一步对使用的值进行限定。简单的方法是使用 enum 来限定，但是 enum 并不具备类型安全性，例如 enums 可以被用来当作一个 ints 使用。因此可以重新设计一个类。

```
class Month {
public:
    static Month Jan() { return Month(1); }
    static Month Feb() { return Month(2); }
    ...
    static Month Dec() { return Month(12); }
private:
    int val;
    explicit Month(int m) : val(m) {}
};
```

将 Month construct 设置为 private 防止产生新的月份。如果需要选择特定的月份，使用 static function 来完成。

预防客户错误的另一个方法是限制什么事可做，什么不能做。常见的方法是加入 const 限定。例如使用 const 修饰 operator* 的返回类型，可以阻止客户因 用户定义类型而犯错。

另一个一般性准则是："让 types 容易被正确使用，不容易被误用"。其表示形式是：除非有好理由，否则应该尽量令你的 types 行为与内置 types 一致。这条准则的内在理由是为了能够提供行为一致的接口。

任何要求客户必须记得做某件事情，就是有着 "不正确使用" 的倾向。例如条款 13 中所提供的 createInvestment 函数。如果 期望客户使用智能指针来接受，实际上是放纵客户产生资源泄露。

```
std::shared_ptr<Investment> createInvestment();
```

上面这种写法杜绝了客户使用过程中忘记 delete 的错误，因为客户必须将其存储在 std::shared_ptr 中。除此之外，也可以提 供特殊的 deleter 来防止客户错误的调用自行定义的 deleter。

```
std::shared_ptr<Investment> createInvestment() {
    std::shared_ptr<Investment> retVal(
        static_cast<Investment*>(0),
        getRidofInvestment()
    );
    retVal = ...; // 令 retVal 指向正确的对象
    return retVal;
}
```

上面这种写法，可以直接返回一个 "将getRidofInvestment" 绑定为删除器的 std::shared_ptr。这样做的另一个好处是它会 自动调用 deleter，因此可以消除潜在的客户错误: cross-DLL problem。这个错误发生于"在 DLL 中被

new 创建的对象，却 在另一个 DLL 内被 delete 释放”。这类行为，在众多平台上会导致运行期错误。而 std::shared_ptr 不会发生这个问题。

条款19: 设计 class 犹如设计 type

- class 的设计就是 type 的设计。在定义一个新的 type 之前，请确定你已经考虑过本条覆盖的所有讨论主题。

程序编码的大部分时间都在扩张类型系统。因此需要了解如何设计一个高效的 class (Type)。首先需要了解这需要面对哪些问题：

- 新 type 的对象应该如何被创建和销毁？
- 对象初始化和对象赋值该有什么样的差别？别混淆初始化和赋值，因为他们对应于不同的函数调用。
- 新 type 对象如果被 pass by value 意味着什么？
- 什么是新 type 的合法值？对 class 的成员变量而言，通常只有某些数值集是有效的。
- 你的新 type 需要配合某个继承图系吗？
- 你的新 type 需要什么样的转换？
- 什么样的操作符和函数对此新 type 而言是合理的？
- 什么样的标准函数应该驳回？
- 谁该取用新 type 的成员？
- 什么是新 type 的“未声明接口”？
- 你的新 type 有多么一般化？
- 你真的需要一个新的 type 吗？

条款20: 宁以 pass-by-reference-to-const 替换 pass-by-value

- 尽量以 pass-by-reference-to-const 替换 pass-by-value。前者通常比较高效，并可避免切割问题
- 以上规则并不适用于内置类型，以及 STL 的迭代器和函数对象。对他们而言，往往 pass-by-value 更合适

```
class Person {
public:
    Person();
    virtual ~Person();

private:
    std::string name;
    std::string address;
};

class Student : public Person {
public:
    Student();
    ~Student();

private:
```

```
    std::string schoolName;
    std::string schoolAddress;
};

bool validateStudent(Student s);
Student a;
bool aIsOk = validateStudent(a);
```

上述代码使用 by-value 的方式传值。在这个过程中，首先会调用 Student 的 copy constructor 来对 a 进行赋值，并且在 validateStudent 返回时调用 destructor。其中包含两个 string 的成员变量，因此也会发生对应的复制和销毁操作。同时， Student 继承自 Person，那么意味着 Person 也会发生复制和销毁，其中包含的两个成员变量也会发生复制和销毁。

可以看到这将经历 6 次构造和析构。使用 pass by reference to const 的方式传递则不会有任何进行调用。

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

by-reference 方式也可以避免 slicing 的发生。当一个 derived class 对象被以 by-value 的方式传入到 base class 中时会发生 slicing，即原有的 derived class 对象的特化性质全被切割掉了。因为这个过程调用的时 base class copy constructor。而不会将其余内容进行复制。

```
class Window {
public:
    std::string name() const;
    virtual void display() const;
};

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

void printNameAndDisplay(Window w) {
    std::cout << w.name();
    w.display();
}

WindowWithScrollBars w;
printNameAndDisplay(w);
```

上述代码只会调用 Window::display 而不会调用子类的 display 方法，因为发生了切割。解决 slicing 的方法就是用 by reference to const 的方式进行传递。

```
void printNameAndDisplay(const Window& w) {
    std::cout << w.name();
```

```
    w.display();
}
```

事实上，by-reference 的底层实现是 pointer 方式，因此如果有个内置类型的对象，by-value 的方式往往比 by-reference 的方式更加高效。对于 STL 中的迭代器和函数对象，也十分适用。

注意：小型的自定义类型并不意味着和内置类型对象的成本划等号。也就意味着，即使小也不一定适用于 by-value 方式。主要原因 包括两点：

- 编译器可能不这么认为，它可能将 double 放进缓存器中，却不愿意将你用 double 实现的对象放进缓存器。
- type 会变化，随着 type 的不断维护更新可能会变得越来越大。

所以 “pass-by-value 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**。

条款21: 必须返回对象时，别妄想返回其 reference

- 绝对不要返回 pointer 或 reference 指向一个 local stack 对象，或返回一个 reference 指向一个 heap-allocated 对象

```
class Rational {
public:
    Rational (int numerator = 0, int denominator = 1);

private:
    int n, d;
    friend const Rational operator* (const Rational& lhs, const Rational& rhs);
};
```

上述代码中，以 by value 的方式返回计算结果。而上述内容无法修改为返回 reference 因为会出现 意向不到的问题。

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

返回 stack 对象显然是不合理的，因为当脱离当前的 scoop 后，result 被释放，于是返回的 reference 指向了被销毁的地方。

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

那么返回一个 heap object 呢？这显然也是不合理的，因为你完全不知道何时去释放掉 new 出来的对 像，甚至有可能你都无法找到那个生成的对象，例如：

```
Rational w, x, y, z;
w = x * y * z; // 连续两次调用，而你无法获取第一次调用产生的指针
```

那么结果就是，当必须要返回新对象时，直接返回新对象就行了。如果成本过高，编译器会想办法进行优化。

条款22: 将成员变量声明为 private

- 切记将成员变量声明为 private。这可赋予客户访问数据的一致性、可细微划分访问权限、允许约束条件获得保障，为提供 class 作者以充分的实现弹性
- protected 并不比 public 更具封装性

首先需要了解为什么成员变量不应该是 public 和 protected 的，然后显然易见应当使用 private。

首先从语法的一致性开始。如果成员变量不是 public，客户就需要使用成员函数来访问成员变量。如此 以来就不再需要区分 成员变量 和 成员函数了，全都按照后者方式访问就行了。

其次，使用函数方式可以更加精确的控制成员变量。而如果成员变量是 public 的，这就意味着任何人都 有权限随时更改对象的成员变量。

```
class AccessLevels {
public:
    int getReadOnly() const { return readOnly; }
    void setReadWrite(int val) { readWrite = val; }
    int getReadWrite() const { return readWrite; }
    void setWriteOnly(int val) { writeOnly = val; }

private:
    int noAccess;
    int readOnly;
    int readWrite;
    int writeOnly;
};
```

通过精心设计，可以细微的划分访问控制权限。

最后就是封装，如果日后你需要改变某个值的计算方式，使用函数方式进行访问变量，用户完全不会知道 发生了什么改变。这种灵活性有时显得十分重要。

```
class SpeedDataCollection {
public:
    void addValue(int speed);
```

```
    double average() const;
};
```

上述代码可以有两个优化方向。一，在对象中维护一个平均速度，当每次调用 average 时直接返回对象。 二，每次被调用时重新计算平均值。

前者需要额外的存储，但是速度十分迅速。而后者计算缓慢却不需要额外的存储空间。因此在内存吃紧，但 很少需要平均值的机器上可以使用前者，内存宽裕但是频繁调用 average 的机器上则可以使用后者。

proteced 限定与 public 在封装上的影响基本类似，后者可以说完全没有封装性，但 protected 的 封装性也十分有限，因为这将影响所有 derived class。

从封装的角度看，其实只有两种访问权限: private(提供封装) 和 其他(不提供封装)。

条款23: 宁以 non-member、non-friend 替换 member 函数

- 宁可拿 non-member non-friend 函数替换 member 函数。这样做可以增加封装性、包裹弹性和机能扩充性。

```
class WebBrowser {
public:
    void clearCach();
    void clearHistory();
    void removeCoockies();

    void clearEverything();
};

void clearBrowser(WebBrowser& wb) {
    wb.clearCach();
    wb.clearHistory();
    wb.removeCoockies();
}
```

clearEverything 和 clearBrowser 相比之下，后者更好。这点与直观印象相反，面向对象的原则是 数据应该尽可能的被封装，然而与直观相反的是，member 函数带来的封装性要比 non-member 函数的 封装性更低。因为 non-member 函数能提供较大的"包裹弹性"。

对于封装而言，越多东西被封装，代码可改动的内容也就越大，而改动影响到的客户越少。而一个数据越多 函数可以访问，那它的封装性也就越差。

正如条款 22所言，我们要将成员变量声明为 private，而想要访问该变量就需要借助 成员函数 或者 友元函数。结合前一段所说，选择 non-member non-friend 函数，其封装性也就越好。

有两点内容需要注意：

1. 这个条款用于区分 member 和 non-member non-friend 函数，而非 non-member 函数。
2. 这里的 non-member non-firend 也可以是其他 class 的 member 函数，只要不是 friend

当随着类型的扩充，可能会提供大量的便利函数，而用户通常只需要其中的某一类。比如 WebBrowser 可 能提供了与书签相关的、与打印相关的、与cookie管理相关的内容。这时就可以使用不同的头文件进行管 理。

```
// webbrowser.h 关于 WebBrowser 的定义，以及核心机能
namespace WebBrowserStuff { // 放在一个命名空间中
class WebBrowser {};
// 核心机能函数
}

// webbrowserbookmarks.h
namespace WebBrowserStuff {
// 书签相关函数
}

// webbrowsercookies.h
namespace WebBrowserStuff {
// cookie相关函数
}
```

条款24: 若所有参数皆需要类型转换，请为此采用 non-member 函数

- 如果你需要为某个函数的所有参数进行类型转换，那么这个函数必须是个 non-member。

令 classes 支持饮食类型转换通常是一个糟糕的主意。但是也存在例外，例如设计一个 class 来表示 有理数，允许整数类型转换为有理数似乎颇为合理。

```
class Rational {
public:
    Rational (int numerator = 0, int denominator = 1);
    int numerator() const;
    int denominator() const;

private:
    int _numerator;
    int _denominator;
};
```

对于上面这个类，如果希望写一个 乘法 函数应该怎样操作呢?

```
class Rational {
public:
    const Rational operator*(const Rational& rhs) const;
};
```

最简单的方式肯定是写在 class 内，这显然是能正常运行的。但是如果你想实现混合运算又怎么办，例如：

```
Rational oneHalf(1, 2);
Rational result = oneHalf * 2; // 成功
Rational result = 2 * oneHalf; // 错误
```

此时就会出现问题，因为 2.operator*() 并不存在，因为 2 是常量。紧接着编译器会尝试寻找一个可 以被调用的 non-member operator*。此时仍然有问题，因为不存在一个接收 int 和 Rational 的 乘法。

而 oneHalf * 2 为什么能成功？首先其调用的是 oneHalf.operator*(Rational)，其次由于 Rational 的 constructor 没有使用 explict 修饰，因此是可以被隐式转换得来的。

想要完全支持混合运算，可以让 operator* 成为一个 non-member 函数，允许编译器在每一个实参身 上执行隐式类型转换。

```
class Rational {
public:
    Rational (int numerator = 0, int denominator = 1) : _numerator(numerator),
      _denominator(denominator) {}
    int numerator() const { return _numerator; }
    int denominator() const { return _denominator; }
private:
    int _numerator;
    int _denominator;
};

const Rational operator*(const Rational& lhs, const Rational& rhs) {
  return Rational(lhs.numerator() * rhs.numerator(),
    lhs.denominator() * rhs.denominator());
}
```

这种形式可以很好的实现混合算数。需要额外注意，既然可以通过 Rational 的 public 成员函数来完 成这种运算的实现，就不要将 operator* 标记为 friend 函数。

此外，这条的实现方式是在从 Object-Oriented C++ 的角度而选择的做法，当你跨进 Template C++ 时则会有新的争议、解法需要考虑。这点会在条款46中看到。

条款25: 考虑写出一个不抛出异常的 swap 函数

- 当 std::swap 对你的类型效率不高时，提供一个 swap 成员函数，并确定这个函数不抛出异常。
- 如果你提供一个 member swap，也该提供一个 non-member swap 用来调用前者。对于 classes (而非 templates) 也请特化 std::swap
- 调用 swap 时应针对 std::swap 使用 using 声明式，然后调用 swap 并且不带任何 “命名空间资格修饰”
- 为 “用户定义类型” 进行 std templates 全特化是好的，但千万不要尝试在 std 内加入某些对 std 而言全新的东西

---

swap是个有趣的函数，它是STL的一部分，后来称为 “异常安全性编程” 的脊柱。

```
namespace std {
    template<typename T>
    void swap(T& a, T& b) {
        T temp(a);
        a = b;
        b = temp;
    }
}
```

上面这种实现方法有时候显得有些慢，因为对于某些类型而言，这些复制动作无一必要。想要更加快速，最 主要的方式就是 “以指针指向一个对象，内含真正的数据”，这种设计就是所谓的 pimpl 手法(pointer to implementation)。

```
class WidgetImpl {
public:
private:
  int a, b, c;
  std::vector<double> v;
};

class Widget {
public:
  Widget(const Widget& rhs);
  Widget& operator=(const Widget& rhs) {
    *ipml = *(rhs.ipml);
  }

private:
  WidgetImpl* ipml;
}
```

上面就是一个使用 pimpl 手法实现的 Widget class。当使用 swap 时，直接调换 ipml 指针即可。 但是我们需要告知 std::swap 需要这么做，否则仍然是默认行为。一个做法是将 std::swap 针对 Widget 进行特化。

```
namespace std {
  template <>
  void swap<Widget>(Widget& a, Widget& b) {
    swap(a.ipml, b.ipml);
  }
}
```

其中 template <> 用来表示这是 std::swap 的一个全特化版本，而后面的 表示这一特化 版本针对 Widget 设计。通常我们不能改变 std 命名空间中的任何东西，但是我们可以为标准 template 制作特化版本，使它专属于我们自己的 classes。

但是由于 ipml 是 private 的，因此需要这个特化版本声明为 friend，但是形式和以前的大不一样。

```
class Widget {
public:
  void swap(Widget& other) {
    using std::swap;
    swap(ipml, other.ipml);
  }

private:
  WidgetImpl* ipml;
};

namespace std {
  template <>
  void swap<Widget>(Widget& a, Widget& b) {
    a.swap(b);
  }
}
```

特化函数调用 Widget 的 swap 成员函数。

如果假设 Widget 和 WidgetImpl 都是 class templates 呢？也许会想到偏特化，但这是不合理的 因为 c++ 允许 class template 进行偏特化，而 function template 则不允许偏特化。如果打算 偏特化一个 function template, 那么惯用的手段实际上是重载。

```
namespace std {
  template <typename T>
  void swap(Widget<T>& a, Widget<T>& b) {
    a.swap(b);
  }
}
```

但是对于 std 这个特殊命名空间而言，客户可以对其中的 templates 进行特化，但不允许添加新的 templates。这也就意味着重载是不行的。一个好的方案是使用一个 non-member swap 来调用 member swap，但是不再将 non-member swap 声明为全特化或重载。

```
namespace WidgetStuff {

template <typename T>
class Widget {};

template <typename T>
void swap(Widget<T>& a, Widget<T>& b) {
    a.swap(b);
}
}
```

了解了所有的 swap 写法后，需要注意的一点是，最好能够同时提供 std::swap 的特化版本以及 non-member swap。这是为了允许客户有多种选择，当其中之一失效时仍然可以有正确的保证。

```
template <typename T>
void doSomething(T& obj1, T& obj2) {
    using std::swap;
    swap(obj1, obj2);
}
```

上述这个代码中，如果 T 对应的没专属 swap 存在则调用，否则会调用 std::swap。而这个过程中，编 译器仍然喜欢 T 的特化版本，因此会调用针对 T 的 std::swap 特化版本。但是千万要注意不要使用， std::swap 形式，而是使用 swap 形式，否则固定调用的就是 std::swap。

最后一点劝告，成员版的 swap 函数不要抛出异常，因为它帮助 classes 提供强烈的异常安全性保障。 这一约束仅适用于 成员版，而非适用于 non-member，因为 swap 缺省版本基于 copy，而一般情况下 两者都允许抛出异常。不抛出异常和高效置换是相等的，因为高效的基础在于置换几乎面向 内置类型 进行 操作(pimpl 手法的底层指针)，而内置类型绝对不会抛出异常。

实现

条款26: 尽可能延后变量定义式的出现时间

- 尽可能延后变量定义式的出现。这样做可以增加程序的清晰度并改善程序效率。

只要你定义一个变量，而其类型带有一个构造函数或者析构函数，那么当程序控制流到达这个位置时，就会 产生构造成本。离开作用域时你又需要承受析构成本。甚至你从未使用过这个变量。

```
std::string encryptPassword(const std::string& password) {
  using namespace std;
  string encrypted;
  if (password.length() < MinimumPassWordLength) {
    throw logic_error("Password is too short");
  }

  return encrypted;
}
```

上述代码就可能产生一个完全没有被使用的变量 encrypted。通过修改顺序，将 encrypted 的定义式 向后移动，可以避免这种事情的发生。

```
std::string encryptPassword(const std::string& password) {
  using namespace std;
  if (password.length() < MinimumPassWordLength) {
    throw logic_error("Password is too short");
  }
  string encrypted;

  return encrypted;
}
```

但是上述代码效率仍旧不高，因为 encrypted 并没有初始化，这意味着它会先调用 default constructor， 然后再调用 operator=。那前面 default constructor 的工作就前功尽弃了。

```
std::string encryptPassword(const std::string& password) {
  using namespace std;
  if (password.length() < MinimumPassWordLength) {
    throw logic_error("Password is too short");
  }
  string encrypted(password);
  encrypt(encrypted);
  return encrypted;
}
```

这种写法显然更好。这其中包含着延后变量定义式意味着直到能够给变量初值实参为止，再进行定义。但是，如果遇到循环应该怎么办呢？

```
Widget w;
for (int i = 0; i < n; ++i) {
  w = ...
}
```

```
for (int i = 0; i < n; ++i) {
  Widget w(...);
}
```

这两种方式哪个好。前者的成本更低，因为只需要 n 次赋值操作，但是不易理解和维护，因为 w 所在的 作用域更广，影响更大。而后者成本更高，但是更易维护。因此在选择时，除非 1. 知道赋值成本比构造+ 析构更低 2. 代码是效率高度敏感的部分，否则你应该选择下面的做法。

条款27: 尽量少做转型动作

- 如果可以，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方案。
- 如果转型是必要的，试着将它隐藏于某个函数背后。客户随后可以调用该函数，而不需要将转型放进他们自己的代码中。
- 宁可使用 c++ style 转型，不要使用旧式转型。前者很容易辨识出，而且也比较分门别类。

在 c++ 中转型会破坏类型系统，而不像 c 一样无可厚非。c++ 提供了 4 种新式的转型方式：

- const_cast(expression): 去除常量性
- dynamic_cast(expression): 安全向下转型，即用来决定某对象是否归属继承体系中的某个类型。(无法通过旧式转型实现)
- reinterpret_cast(expression): 执行低级转型，实际动作取决于编译器，即不可移植。pointer to int
- static_cast(expression): 强迫隐式转型。

推荐使用新式转型，因为很容易在代码中分辨，其次转型动作的目标更窄，更容易被编译器诊断出错误。 转型操作会被编译器编译出对应动作，而非直接更改类型。例如指针，当使用 base class pointer 指 向 derived class 时，指针地址可能并不相同。这种情况下会有偏移量在运行期被施行于 Derived* 身上。也因此不要做出 “对象在 C++ 中如何布局” 的假设。

错误转型的另外一个场景是，可能写出在其他语言正确在 c++ 中却错误的代码，例如：

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

```
};

class SpecialWindow : public Window {
public:
  virtual void onResize() {
    static_cast<Window>(*this).onResize();
    ...
  }
};
```

上述代码本意是，在子类中先调用父类的 onResize 方法，然后再在当前子类中进行更改。但是这完全不 正确，因为 static_cast 返回的是一个副本，而不是当前对象父类的引用。这意味这上述更改均更改在 了一个 Window 副本中，而当前对象中的 Window 对象却丝毫未动，正确的写法应该如下：

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

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

当调用转型时，你很有可能走上了不归路。dynamic_cast更是如此。dynamic_cast 在许多版本中的实 现方案效率都不是很高，因此在注重效率的代码中应该避免使用 dynamic_cast。

通常你使用 dynamic_cast 是因为你认定了该 base class pointer 指向的对象是一个 derived class 对象。那想要避免使用 dynamic_cast 有两个方法。

1. 使用容器存储对应的 derived class 指针

```
class Window {};
class SpecialWindow : public Window {
public:
  void blink();
};

typedef std::vector<std::shared_ptr<SpecialWindow>> VPSW;
VPSW winPtrs;

for (VPSW::iterator it = winPtrs.begin(); it != winPtrs.end(); ++it) {
  (*it)->blink();
}
```

这种做法不允许在一个容器内存储不同的 Window 派生类。

2. 通过 base class 接口来处理 “所有可能的各种 Window 派生类”，也就是使用 virtual。

```
class Window {
public:
  virtual void blink() {} // 缺省实现代码并不好
};

class SpecialWindow : public Window {
public:
  virtual void blink() {
    ...
  }
};

typedef std::vector<std::shared_ptr<Window>> VPW;
VPW winPtrs;

for (VPW::iterator it = winPtrs.begin(); it != winPtrs.end(); ++it) {
  (*it)->blink();
}
```

一定要避免的代码是，一连串的 dynamic_casts。

```
for (VPW::iterator it = winPtrs.begin(); it != winPtrs.end(); ++it) {
  (*it)->blink();
  if (SpecialWindow1* psw1 = dynamic_cast<SpecialWindow1*>(iter->get())) {
    ...
  } else if (SpecialWindow2* psw2 = dynamic_cast<SpecialWindow2*>(iter->get())) {
    ...
  } else if (SpecialWindow3* psw3 = dynamic_cast<SpecialWindow3*>(iter->get())) {
    ...
  }
}
```

这样的代码效率极低。总结来看，我们应该尽可能隔离转型动作，通常将它隐藏在某个函数内，函数的接口 会保护调用者不受内部任何动作的影响。

条款28: 避免返回 handles 指向对象内部成分

- 避免返回 handles(references, 指针，迭代器) 指向对象内部。遵守这条款可以增加封装性，帮助 const 成员函数的行为想个 const，并将发生 “虚吊 handle” 的可能性降至最低。

```
class Point {
public:
```

```
  Point(int x, int y);
  void setX(int newVal);
  void setY(int newVal);
};

struct RectData {
  Point ulhc; // upper left-hand corner
  Point lrhc; // lower right-hand corner
};

class Rectangle {
public:
  Point& upperLeft() const { return pData->ulhc; }
  Point& lowerRight() const { return pData->lrhc; }
private:
  std::shared_ptr<RectData> pData;
};
```

这样设计虽然能够通过编译，但是却是错误的，因为它违背了 reference constness 的原则。显然当你 使用一个 const Rectangle 对象调用 upperLeft() 时，你会获得一个指向内部数据 ulhc 的引用。 而这个引用却没有限定，因此可以被修改。而这一行为与 const 恰好冲突。

这就是 handles 的特点，它包括 reference、pointer 和 iterator。如果返回一个 “代表对象内部 数据” 的 handle，随之而来的就是 “降低对象封装性” 的风险。

使用 const 进行限定，可以实现由限度的松弛封装性，即可读但不可更改。

```
const Point& upperLeft() const {}
const Point& lowerRight() const {}
```

但是即使这样，还是会产生 dangling handles 的风险。这种风险往往来自于函数返回值。

```
class GUIObject {};
const Rectangle boundingBox(const GUIObject& obj);

GUIObject* pgo;
const Point* pUpperLeft = &(boundingBox(*pgo).upperLeft());
```

上述函数就会发生 dangling handles 的风险。因为实际上 pUpperLeft 指向的是一个临时变量的 ulhc， 即 temp.upperLeft()。而当这条语句运行结束之后，temp 会被销毁，那么此时 pUpperLeft 就变成 了空悬、虚吊。

条款29: 为“异常安全”而努力是值得的

- 异常安全函数即使发生异常也不会泄露资源或允许任何数据结构的败坏。这样的函数区分为三种可能的保证：基本型、强烈型、不抛异常型。

- “强烈保证”往往能够以 copy-and-swap 实现出来，但 “强烈保证” 并非对所有函数都可以实现或具备现实意义。
- 函数提供的 “异常安全保证” 通常最高只等于其所调用之各个函数的 “异常安全保证” 中的最弱者。

```
class PrettyMenu {
public:
  void changeBackground(std::istream& imgSrc);
private:
  Mutex mutex;
  Image* bgImage;
  int imageChange;
};

void PrettyMenu::changeBackground(std::istream& imgSrc) {
  lock(&mutex);
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
  unlock(&mutex);
}
```

上述代码不是 异常安全 的。异常安全需要满足两个条件，当异常被抛出时：

- 不泄露任何资源: 一旦 new Image 抛出异常，unlock 就不再执行，也就是说 mutex 永远被锁。
- 不允许数据败坏: 如果 new Image 抛出异常，bgImage 就指向了一个已经删除的对象，imageChanges 也被累加，而没有新的图像产生。

资源泄露的解决方案在 条款13 中已经指出可以使用对象来管理资源，条款14 中则给出了一个 Lock 类 的实现方案。因此可以改写为:

```
void PrettyMenu::changeBackground(std::istream& imgSrc) {
  Lock m(&mutex);
  delete bgImage;
  ++imageChanges;
  bgImage = new Image(imgSrc);
}
```

对于数据败坏的解决方案我们需要进行抉择，在此之前先了解一下选项的术语:

- 基本承诺: 如果异常抛出，程序内的任何事物仍然保持在有效状态下。然而此时程序的现实状态不可预料，可能是前一状态或是缺省状态。客户需要额外的函数来判断。
- 强烈保证: 如果异常抛出，程序状态不改变。成功就完全成功，否则就回复到调用之前的状态。这相较于前者调用起来更加简单，因为只有两个状态，前一状态和成功后状态。
- 不抛掷保证: 承诺不抛出异常，因为他们总是能够正确运行。比如内置类型就会提供 nothrow 保证。但是这并不是说绝不会抛出异常，而是如果产生异常那将是严重错误，而不可解决。

如果想要 异常安全性，那么就需要在上面三者中选择一个。选择的原则是，可能的话提供 nothrow 保证， 但是大多数条件下提供 基本保证 或者 强烈保证。

```
class PrettyMenu {
public:
  void changeBackground(std::istream& imgSrc);
private:
  Mutex mutex;
  std::shared_ptr<Image> bgImage;
  int imageChange;
};

void PrettyMenu::changeBackground(std::istream& imgSrc) {
  Lock m(&mutex);
  bgImage.reset(new Image(imgSrc));

  ++imageChanges;
}
```

这样改写代码，可以保证 基本承诺。首先是使用 std::shared_ptr 来管理 bgImage。这有两点好处 1. 强化资源管理 2. 使用内置的 reset 函数来实现替换操作。其次将 ++imageChanges 放到最后，保证 只有正确运行才计数。

但是上面只保证了 基本承诺，因为 imgSrc 的状态并不确定，它的读取记号可能已经被移走。但这种更改 很容易实现，因此不再进一步考虑。

下面需要了解另一个实现 强烈保证的手段—— copy and swap。即 swap 之前先进行 copy 操作。

```
struct PMImpl {
  std::shared_ptr<Image> bgImage;
  int imageChange;
};

class PrettyMenu {
public:
  void changeBackground(std::istream& imgSrc);
private:
  Mutex mutex;
  std::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc) {
  using std::swap;

  Lock m(&mutex);
  std::shared_ptr<PMImpl> pNew(new PMImpl(*pImpl));
  pNew->bgImage.reset(new Image(imgSrc));
  ++pNew->imageChanges;
```

```
  swap(pImpl, pNew);
}
```

使用 copy-and-swap 能够实现强烈保证，但是并不是总能成功。

```
void someFunc() {
  f1();
  f2();
}
```

例如上面这种形式的代码，即使 f1 成功调用，只要 f2 失败，那么就破坏了强烈保证的原则。这问题出 在连带影响上。如果函数只操作局部性状态，那就相对容易提供强烈保证，否则就不行。例如数据库的修改 动作往往就无法提供强烈保证。

除此之外，copy-and-swap 的效率问题也很严重。所以，当强烈保证可以实现时，你的确应该提供它，但是“强烈保证”并非在任何时刻都显得实际。而不切实际时，就应该提供“基本保证”。

条款30: 透彻了解 inlining 的里里外外

- 将大多数 inlining 限制在小型、被频繁调用的函数身上。这可以使得日后的调试过程和二进制升级更加容易，也可使潜在的代码膨胀问题最小化，使得程序的速度提升机会最大化。
- 不要只因为 function tempaltes 出现在头文件，就将他们声明为 inline。

如果 inline 函数的本体很小，编译器针对“函数本体”所产生的码可能比针对“函数调用”所产生的码更小。 将函数 inline 确实可以导致较小的目标码和较高的指令告诉缓存装置命中率。

inline 的定义可以分为隐式和显式:

```
// 隐式
class Person {
public:
  int age() const {return theAge;}
private:
  int theAge;
};
```

```
// 显式
template<typename T>
inline const T& std::max(const T& a, const T& b) {
  return a < b ? b : a;
}
```

编译器通常会拒绝将太过复杂的函数 inlining，而所有对 virtual 函数的调用也都会导致 inlining 落空。总的来看，一个表面上看似 inline 的函数是否真的是 inline 主要取决于你的建置环境，主要 取决于编译器。有时候编译器可能还会为 inlining 生成一个函数本体，比如当你需要获取函数指针时编 译器就不得不生成一个 outlined 函数本体。

关于构造函数和析构函数，使用 inline 可能是一个糟糕的想法。例如 default constructor，看上 去似乎是一段空代码，但是实际上编译器会给你填充大量的内容，防止其产生异常。而这恰恰违背了 短小 的要求。

除此之外，将函数声明为 inline 所需要面对更多的是，随着代码的维护可能函数会进行扩充，这样就不 适用 inline 标记了。

所以选择是否适用 inline 的逻辑策略是，一开始先不要将任何函数声明为 inline，或至少将 inlining 施行范围局限在那些 “一定成为 inline” 或者 “十分平淡无奇” 的函数身上。

条款31: 将文件间的编译依存关系降至最低

- 支持 “编译依存性最小化” 的一般构想是：相依于声明式，不要相依于定义式。基于此构想的两个手段是 Handle classes 和 interface classes。
- 程序头文件应该以 “完全且仅有声明式” 的形式存在。这种做法不论是否涉及 templates 都适用。

假设你对 c++ 程序的某个 class 实现做出了轻微修改，当你进行编译时可能遇到大量代码重新编译的情 况。这个原因在于 c++ 没有把 "将接口从实现中分离" 这件事做好。

```
class Person {
public:
  Person(const std::string& name, const Date& birthday, const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;

private:
  std::string theName;
  Date theBirthDate;
  Address theAddress;
};
```

上述代码在进行编译时，需要引入其他定义文件。这样一来就形成了一种编译依存关系。这种依存关系之下，如果头文件或者头文件依赖的头文件发生改变时，Person class 以及任何使用 Person class 的文件 都必须重新编译。那么如何将接口分离呢？使用 “声明依存性” 替换 “定义依存性”。

```
#include <string>
#include <memory>

class PersonImpl; // Person 实现类的前置声明
class Date; // 接口用到的 class 前置声明
class Address;
```

```
class Person {
public:
  Person(const std::string& name, const Date& birthday, const Address& addr);
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;

private:
  std::shared_ptr<PersonImpl> pImpl; // pimpl idiom
};
```

上述代码的好处是，无论 PersonImpl 如何更改，或者其依赖 例如 Data，Address 如何更改，只需 重新编译到 PersonImpl 即可。因为 Person 中使用一个指针来指向 PersonImpl 对象，因此不需要 额外进行重新编译。这也就意味着，所有依赖 Person 的文件也不用重新更改了。这样一来，就完全分离 了他们之间的联系，也是真正的“接口与实现分离”。原则是：让头文件尽可能自我满足，万一做不到，则 让其他文件内的声明式相依。

其他的设计策略也都基于此:

- 如果使用 reference 或 pointers 可以完成任务，就不要使用 objects。你可以只靠一个类型声明式就定义出指向该类型的 references 和 pointers，但如果定义某类型的 objects， 就需要用到该类型的定义式。
- 如果能够，尽量用 class 声明式替换 定义。 当你声明一个函数而它用到某个 class 时，你并不需要该 class 的定义。
- 为声明式 和 定义式 提供不同的头文件。 为了遵守上述原则，需要两个头文件，一个用于声明式，一个用于定义式。引用时只引用声明式头文件。

```
/* date.h */
class Date {
  ...
};
```

```
/* datefwd.h */
class Date;
```

```
/* main.cpp */
#include "datefwd.h"
Date today();
void clearAppointments(Date d);
```

这种方式效仿了 c++ 标准程序库头文件的 。在其中包含了 iostream 各组件的声明式，其对 应定义则分布在若干不同的头文件内，包括，，，。它另一个特点是，如果建置环境允许 template 定义放在非头文件中，那么就可以通过声明式头文件来提 供 template。

另外一种方式时使用 c++ 提供的关键字 export。

回到 pimpl idiom，像 Person 这样使用 pimpl idiom 的 class 被称为 handle classes。他们 所有的函数都将任务转交给相应的实现类。

```
#include "Person.h"
#include "PersonImpl.h"

Person::Person(const std::string& name, const Date& birthday, const Address& addr)
:
  pImpl(new PersonImpl(name, birthday, addr)) {}
std::string Person::name() const {
  return pImpl->name();
}
```

实现 handle classes 的另一个方法时令 Person 称为 interface class。

```
class Person {
public:
  virtual ~Person();
  virtual std::string name() const;
  virtual std::string birthDate() const;
  virtual std::string address() const;

  static std::shared_ptr<Person> create(const std::string& name,
    const Date& date, const Address& address);
};

class RealPerson : public Person {
public:
  RealPerson(const std::string& name, const Date& birthday, const Address& addr);
  virtual ~RealPerson() {}
  std::string name() const;
  std::string birthDate() const;
  std::string address() const;

private:
  std::string theName;
  Date theBirthDate;
  Address theAddress;
};

std::shared_ptr<Person> Person::create(const std::string& name,
    const Date& date, const Address& address) {
  return std::shared_ptr<Person>(new RealPerson(name, date, addres));
}

std::shared_ptr<Person> pp(Person::create(name, date, address));
std::cout << pp->name()
          << pp->birthDate()
          << pp->address();
```

上面这段代码，将 Person 改为了 interface class，并提供了一个静态类方法用于实现 factory 函数。factory 函数的职责在于，根据条件生成一个实现类对象，这里就是 RealPerson。在使用过程 中，使用 Person pointer 进行操作，而实现则放在 RealPerson 中，这也同样实现了 pimpl idiom 类似的降低依赖的效果。

在成本角度，无论是 handle class 还是 interface class 都会带来额外的成本。handle class 通过 implementation pointer 进行对象访问，增加了一层间接访问。同时，每一个对象又会有额外的 开销。最后你要使用指针，那就可能发生动态内存分配的额外开销，以及 bad_alloc 的风险。而 interface class 则每次调用 virtual 都必须经过 vptr。同时 vptr 又增加了存储开销。

因此你需要在成本和耦合度之间做出抉择。

继承和面向对象设计

条款32: 确定你的 public 继承塑膜出 is-a 关系、

- public 继承意味着 is-a。适用于 base classes 身上的每一件事情一定也适用于 derived class 身上，因为每一个 derived class 对象也都是一个 base class 对象。

以 c++ 进行面向对象编程，最重要的一个规则是: public inheritance 意味着 is-a 的关系。也就 是说，当你令 class D public inheritance class B 时，D 的对象也是一个 B 的对象，但是 B 的对象不一定是个 D 的对象。

但是这种关系有时候会违反直觉，例如 企鹅 和 鸟 的关系:

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

class Penguin : public Bird {};
```

这种继承关系看起来没有问题，但是 Penguin 根本不会飞，这意味着 Bird 类在设计时发生了失误。因 为部分 fly 并不是 Bird 的共性。通常有两种修改方法:

```
void error(const std::string& msg);
class Penguin : public Bird {
public:
  virtual void fly() { error("..."); }
}
```

这种修改方式，当用户尝试调用 penguin.fly 时会产生错误。这并不是最佳的处理方案，因为这会在运 行期产生错误。正如 条款18 所建议的那样，应当将这种错误扼杀在 编译期 内。

```
class Bird {};
class FlyingBird : public Bird {
public:
  virtual void fly();
};

class Penguin : public Bird {};
```

第二种方法是可以按照这种结构修改继承关系。但是这种代码的问题在于，随着维护，未来可能对是否能够 fly 并不感兴趣，这也意味着这种结构不再适用。当需要进行更改时就会变得十分麻烦。

另外一个例子就是矩形和正方形的关系:

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);
  virtual int height() const;
  virtual int int width() const;
};

class Square : public Rectangle {};

void makeBigger(Rectangle& r) {
  int oldHeight = r.height();
  r.setWidth(r.width() + 10);
  assert(r.height() == oldHeight);
}

Square s;
assert(s.width() == s.height());
makeBigger(s);

assert(s.width() == s.height());
```

显然，直观的想法是两个 assert 结果应该相同，因为正方形总是长宽相等的。但是 makeBigger 后， 长宽不相等了，但是后面 assert 又为 true。这显然是不合理的。

is-a 并非唯一的 classes 之间的关系，除此之外还有 has-a 和 is-implemented-in-terms-of 两种关系。需要了解这些 class 相互关系之间的差异，才能设计好的继承类型。

条款33: 避免遮掩继承而来的名称

- derived classes 内的名称会遮掩 base classes 内的名称。在 public 继承下从来没有人希望如此。
- 为了让遮掩的名称再见天日，可使用 using 声明式或转交函数。

这个条款主要用来解释作用域的相关内容。

```
int x;
void func() {
  double x;
  std::cin >> x;
}
```

上面这段代码， global x 被 local x 名称遮掩了，因此实际上操作的是 double x。在继承关系里 也会发生这种事情。

```
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() {
    mf2();
  }
};
```

上述代码中，mf4 调用 mf2，编译器会先在 local 域内查找，没有的话查找 Drived 域，然后 Base 域，紧接着 namespace，最后 global。因此实际上调用的是 Base::mf2。如果修改一下上面的代码， 就可能发生 遮掩名称 的问题。

```
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() {
    mf2();
  }
};

Derived d;
int x;

d.mf1();  // 正确
d.mf1(x); // 错误
d.mf2();  // 正确
d.mf3();  // 正确
d.mf3(x); // 错误
```

上述代码在调用过程中，就会发生名称掩盖。可以看到，当调用 mf1 和 mf3 时都会选择 Derived:: 内 的内容，而没有选择 Base 内的方法。

想要正确运行，有两种方式完成:

1. 使用 using 声明式

```
class Derived : public Base {
public:
  using Base::mf1;
  using Base::mf3;
  virtual void mf1();
  void mf3();
  void mf4() {
    mf2();
  }
};

Derived d;
int x;

d.mf1();  // 正确
d.mf1(x); // 正确
d.mf2();  // 正确
d.mf3();  // 正确
d.mf3(x); // 正确
```

使用 using 声明式来重新声明那些被掩盖的名称，可以实现调用 base function 的功能。

2. 使用转交函数

但是有时候你又不想继承 Base 中的所有函数(这显然不能使用 public 继承)，而是想要继承部分函数。 那么就可以使用 转交函数 来实现。

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

这样做既不会像 using 那样将所有名称完全暴露，又能继承部分的函数。

条款34: 区分接口继承和实现继承

- 接口继承和实现继承不同。在 public 继承之下，derived class 总是继承 base class 的接口。
- pure virtual 函数只具体指定接口继承。
- 简朴的 impure virtual 函数具体指定接口继承和缺省实现继承

- non-virtual 函数具体指定接口继承以及强制性实现继承。

在 public 继承概念的基础上，可以发现它还由两部分组成: 函数接口继承 和 函数实现继承。身为 class 的设计者有时你会希望 derived class：1. 只继承函数接口，2. 同时继承接口和实现，但是希望能够 覆写，3. 同时继承接口和实现，并且不允许覆写。

```
class Shape {
public:
  virtual void draw() const = 0;
  virtual void error(const std::string& msg);
  int objectID() const;
};

class Rectangle : public Shape {};
class Ellipse : public Shape {};
```

- 成员接口总是会被继承。

Shape 对 derived class 的影响十分深远，正如 public 继承意味着 is-a 关系那样。所有能够施 用于 Shape 的方法，也同样适用 derived class。正是如此，成员接口必须被继承。

- 声明一个 pure virtual 函数的目的是为了让 derived classes 只继承函数接口

在 Shape 中，draw 是一个 pure virtual 函数，它具有两个突出特点: 1. 必须被 继承了他们 的 具象 class 重新声明，2。abstract class 中通常没有 pure virtual 的定义。这意味着所有继承 自 Shape 的 derived classes 都必须提供一个 draw 函数，但是 Shape 不干涉如何实现。

pure virtual 可以提供定义，但是需要指明 class 才可以调用，例如: Shape::draw()。它可以用 来实现一个机制，为简朴的 impure virtual 函数提供更平常更安全的缺省实现。

- impure virtual 函数的目的是，让 derived classes 继承该函数的接口和缺省实现。

适用 impure virtual 意味着，derived class 的设计者必须支持一个 error 函数，如果不想写， 可以适用 Shape class 的缺省版本。

但是这种 impure virtual 函数的写法很容易因为疏忽而产生问题，特别是在代码维护过程中。

```
class Airport {};
class Airplane {
public:
  virtual void fly(const Airport& destination);
};

void Airplane::fly(const Airport& destination) {
  缺省代码
}

class ModelA: public Airplane {};
class ModelB: public Airplane {};
```

```
class ModelC: public Airplane {};
```

上述代码的本意是 ModelA 和 ModelB 采用默认飞行方式，而 ModelC 则适用自己的飞行方式。但是在 程序维护过程中，忘记为 ModelC 实现了新的飞行方式，所以在调用时仍然采用了默认方案。这就产生了 错误。

为了防止这种事情的发生，必须时刻提醒 derived class 的设计者，在 缺省方案 和 自定义方案 中进 行选择。

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;
protected:
  virtual void defaultFly(const Airport& destination);
};
void Airplane::defaultFly(const Airport& destination) {
  ...
}


class ModelA : public Airplane {
public:
  virtual void fly(const Airport& destination) {
    defaultFly(destination);
  }
};

class ModelC : public Airplane {
public:
  virtual void fly(const Airport& destination);
};
void ModelC::fly(const Airport& destination) {
  ...
}
```

通过 pure virtual 来提醒 derived class 设计者必须进行选择，能够很好的防止疏忽产生问题。同 时又通过 defaultFly 来提供默认飞行方式。

上面的代码将接口和缺省实现分开实现了，但是很多人反对这样的事情发生。一种很好的手段就是为 pure virtual 函数定义自己的实现，来省去 defaultFly。

```
class Airplane {
public:
  virtual void fly(const Airport& destination) = 0;
};
void Airplane::fly(const Airport& destination) {
  ...
}

class ModelA : public Airplane {
```

```
public:
  virtual void fly(const Airport& destination) {
    Airplane::fly(destinatino);
  }
};

class ModelC : public Airplane {
public:
  virtual void fly(const Airport& destination);
};
void ModelC::fly(const Airport& destination) {
  ...
}
```

- 声明 non-virtual 函数的目的是为了令 derived classes 继承函数的接口以及一份强制性实现 non-virtual 函数意味着，不打算在 derived classes 中有不同的行为，它所表现出的不变性凌驾于 特异性之上。

如果你能够履行上面的这些差异，那么你应该能够避免两个错误:

1. 将所有函数声明为 non-virtual。这让 dervied classes 没有足够的空间进行特化工作。特别是 non-virtual 析构函数。实际上任何 class 如果打算被用来当作一个 base class，它就会拥有若干个 virtual 函数。
2. 将所有成员函数声明为 virtual 除了部分 interface classes，大多数 class 都会有某些函数就是不该在 derived class 中被重 新定义，那么你应该将这些函数声明为 non-virtual。

条款35: 考虑 virtual 函数以外的其他选择

- virtual 函数的替代方案包括 NVI 手法以及 Strategy 涉及模式的多种形式。NVI 手法自身一个特殊形式的 Template Method 设计模式。
- 将机能从成员函数转移到 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。
- tr1::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兼容” 的所有可调用物。

```
class GameCharacter {
public:
  virtual int healthValue() const;
};
```

上面是一个游戏的角色类型，他具有一个 non-pure virtual 函数，用来返回当前生命值。这意味着， 每个不同的角色可以实现不同的 healthValue 方案，同时还具有一个默认的方案。

如何替代 virtual 呢?

借由 non-virtual interface 实现 Template Method 模式

```
class GameCharacter {
public:
  int healthValue() const {
    ...
    int retVal = doHealthValue();
    ...
    return retVal;
  }
private:
  virtual int doHealthValue() const {
    缺省代码
  }
};
```

上面代码的思想是，virtual 函数应该几乎总是 private 的。而较好的设计是保留 healthValue 为 public 成员函数，但是让它成为 non-virtual，并调用一个 private virtual 函数。这种手法被 称为 non-virtual interface (NVI)。它是所谓 Template Method 设计模式的一个独特表现形式。

NVI 手法的优点在于，调用 doHealthValue 的前后可以添加部分处理工作，例如锁定/解锁 mutex，制 造运转日志记录项、验证 class 约束条件、验证函数先决/事后条件等等。

NVI 手法下其实没必要让 virtual 一定得是 private。例如某些函数可能 derived class 必须调用 base class 的函数，此时就可以适用 protected。而部分要求必须适用 public 修饰的函数，例如 具有多态性质的 base classes 的析构函数，这么依赖就不能使用 NVI 手法了。

借由 Function Pointers 实现 Strategy 模式

```
class GameCharacter;
int defaultHealthCalc(const GameCharacter& gc);
class GameCharacter {
public:
  typedef int (*HealthCalcFunc)(const GameCharacter&);
  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc) : healthFunc(hcf)
{}
  int healthValue() const {
    return healthFunc(*this);
  }
private:
  HealthCalcFunc healthFunc;
};

class EvilBadGuy : public GameCharacter {
public:
  explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc) : GameCharacter(hcf)
{}
};

int loseHealthQuickly(const GameCharacter&);
int loseHealthSlowly(const GameCharacter&);
```

```
EvilBadGuy ebg1(loseHealthQuickly);
EvilBadGuy ebg1(loseHealthSlowly);
```

通过上面这种方式，借助 Function Pointers 实现 strategy 模式，可以看到有两点好处:

- 同一类型的对象，可以使用不同的运算方法。
- 已知的某个对象，计算过程可以在运行期进行更改。

但是这样设计程序也会带来问题，因为你将一个 member function 替换成为了 non-member function， 这就意味着，如果这个函数需要用到 private member 相关的内容就没有办法了。除非，你尝试通过 firends 或者为其中一部分提供 public 访问方法，但是这些行为又会降低 GameCharacter 的封装性。因此， 你需要在 non-member funtion pointer 与 封装性 之间进行取舍。

借由 function 完成 Strategy

将上面的这种实现方式改为通过 std::function 对象来实现。

```
#include <functional>

class GameCharacter;
int defaultHealthCalc(const GameCharacter& gc);
class GameCharacter {
public:
  typedef std::function<int (const GameCharacter&)> HealthCalcFunc;
  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc) : healthFunc(hcf)
{}
  int healthValue() const {
    return healthFunc(*this);
  }
private:
  HealthCalcFunc healthFunc;
};
```

这种改变是为非常细小的，只是将原有的函数指针转化为了一个 std::function 对象，相当于指向函数 的泛化指针。但是有时可以提供更高的弹性。这主要来自于 std::function 可以封装普通函数、lambda 函数、仿函数(函数对象)以及成员函数。

```
// 普通函数，但是返回值为 short, std::function 允许隐式转型
short calcHealth(const GameCharacter& gc);

// 函数对象
struct HealthCalculator {
  int operator()(const GameCharacter& gc) const;
};

// 成员函数
class GameLevel {
public:
  float health(const GameCharacter& gc) const;
```

```
};

class EvilBadGuy : public GameCharacter {
public:
  EvilBadGuy(HealthCalcFunc hcf) : GameCharacter(hcf) {}
};

EvilBadGuy ebg1(calcHealth); // 封装普通函数
EvilBadGuy ebg2(HealthCalculator()); // 封装函数对象

GameLevel currentLevel;
EvilBadGuy ebg3(std::bind(
  &GameLevel::health,
  currentLevel,
  std::placeholders::_1
)); // 封装成员函数
```

上述代码为用户提供了更多的设计选择。

古典的 strategy 模式

传统的 Strategy 做法是采用一个分离的继承体系来完成的。其中包括两个根系，一个是用来表示角色的 GameCharacter(EvilBadGuy, EyeCandyCharacter)，另一个则是用来表示生命计算的 HealthCalcFunc (SlowHealthLoser, FastHealthLoser)

```
class GameCharacter;
class HealthCalcFunc {
public:
  virtual int calc(const GameCharacter& gc) const {}
};
class GameCharacter {
public:
  explicit GameCharacter(HealthCalcFunc* phcf = &defaultHealthCalc) :
pHealthCalc(phcf) {}
  int healthValue const {
    return pHealthCalc->calc(*this);
  }

private:
  HealthCalcFunc* pHealthCalc;
};
```

后续需要更更多的人物，以及生命值计算方法，只需要在这两个类的基础上进行继承就可以了。

条款36: 绝不重新定义继承而来的 non-virtual 函数

- 绝对不要重新定义继承而来的 non-virtual 函数。

加入存在下面的继承关系:

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

class D : public B {
public:
  void mf(void);
};

D x;
B* pb = &x;
pb->mf();
D* pd = &x;
pd->mf();
```

上面的代码中，两次调用结果不一样。这主要是因为 non-virtual 函数是 静态绑定 的，这也就意味着 使用 B pointer 来调用就会调用 B::mf，而用 D pointer 力调用则是 D::mf。另一方面 virtual 却是 动态绑定 的，所以 virtual function 并不会产生这个问题。

那么为什么不要重新定义继承而来的 non-virtual 函数呢？两点原因:

- 如果 D 需要重新定义 mf，而每个 B 对象又必须调用 B::mf，而非 D::mf，那么这就意味着这并非 public 继承关系。
- 如果 D 需要 public 继承 B，并且 D 需要实现不同的 mf，那么就应该使用 virtual 函数，而不应该反映出 “不变性凌驾特异性” 的性质

条款37: 绝不重新定义继承而来的缺省参数值

- 绝对不要重新定义一个继承而来的缺省参数值，因为缺省参数值都是静态绑定，而 virtual 函数——你唯一可以覆写的东西——确实动态绑定。

这条条款的范围可以缩小，因为条款36已经讨论了不应该重新定义 non-virtual 函数，那实际上这里强 调的就是 继承带有缺省数值的 virtual 函数。

本条条款成立的主要原因就是: virtual 函数是动态绑定的，而缺省参数值则是静态绑定的。

```
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;
};

Shape* p = new Rectangle();
p->draw();
```

像上面的 Rectangle 类，重定义了缺省参数值，这可能会带来与预期相违背的结果。正如上面的调用方 式，由于 p 的静态类型是 Shape*，而缺省参数是静态绑定的，因此实际上进行的是 p->draw(Red) 操 作而非 p->draw(Green)。

那如果并不重定义，只是将缺省传给 derived class 呢？

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

这种做法也是不可取的，因为代码重合。并且不好维护，比如后续将 Shape 的缺省参数改为 Green，那 就又会出现上面同样的问题。一个好的解决方案是，使用条款35中的手段来更改这种代码，比如 NVI。

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

class Rectangle {
private:
  virtual void doDraw(ShapeColor color) const {
    ...
  }
}
```

条款38: 通过复合塑模出 has-a 或 “根据某物实现出”

- 复合的意义和 public 继承完全不同。
- 在应用域，复合意味着 has-a。在实现域，复合意味 has-a。在实现域，复合意味着 is-implemented-in-terms-of。

复合关系是类型之间的一种关系，当某种类型的对象内含其他类型对象，就是这种关系。

复合关系也和 public 继承一样具有现实意义，它主要包括两种:

- has-a: 复合发生在应用域内的对象时，表现出 has-a 的关系。应用域就是为了塑造现实世界中的某些事物而设计的类。
- is-implemented-in-terms-of: 复合发生在实现域内，表现出这种关系。实现域是为了实现某些细节而设计的类，比如缓冲区、互斥器、查找树等等。

has-a

has-a 的关系很好理解，比如下面的代码就是 “人有一个地址” 这样的关系。

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

is-implemented-in-terms-of

这种 根据某物实现出 的意义则不太好理解。

这里以 set 为例，假如你希望设计自己的 set，并且你知道可以使用底层的 linked lists 来实现。 于是你开始尝试让 set 继承 list。

```
template<typename T>
class Set : public std::list<T> {};
```

看上去很完美，但是存在重大问题。我们已经知道了 public 是 is-a 的关系，那也就意味着 Set 一定 是一个 list。但是 list 允许多个重复数据，而 Set 不允许。那显然两者并不是 is-a 的关系。正确 的做法是，将 list 应用于 Set。

```
template<class T>
class Set {
public:
  bool member(const T& item) const {
    return std::find(rep.begin(), rep.end(), item) != rep.end();
  }
  bool insert(const T& item) {
    if (!member(item)) rep.push_back(item);
```

```
  }
  bool remove(const T& item) {
    typename std::list<T>::iterator it = std::find(rep.begin(), rep.end(), item);
    if (it != rep.end()) rep.erase(it);
  }
private:
  std::list<T> rep;
}
```

条款39: 明智而审慎的使用 private 继承

- private 继承意味着 is-implemented-in-terms of。它通常比复合级别耕地。但是当 derived class 需要访问 protected base class 的成员，或者需要重新定义继承而来的 virtual 函数时，这么设计是合理的。
- 和复合不同，private 继承可以造成 empty base 最优化。这对致力于 “对象尺寸最小化” 的程序开发者而言，可能很重要。

---

private 继承有两条规则:

1. private 继承关系，编译器不会自动将一个 derived class 对象转换为 base class。
2. private 继承将从 base class 继承而来的所有成员在 derived class 中变成 private 属性。

private 继承的含义是 implemented-in-terms-of。private 继承也意味着只有 实现部分 被继承， 接口部分 被自动略去。而 private 仅是一种实现计数，没有设计意义。

和前面提出的复合相比，应该尽可能的使用复合，必要时才使用 private 继承。这种必要源自当 protected 成员或者 virtual 函数被牵扯进来时。

一个例子是使用 timer。下面是一个 Timer 对象，它用来对时间进行计时操作。

```
class Timer {
public:
  explicit Timer(int tickFrequency);
  virtual void onTick() const;
};
```

现在我们需要为 Widget 对象实现一个功能，让它记录每个成员函数被调用的次数。我们可以使用 private 继承来实现，因为只有这样才能够重新定义 Timer::onTick 方法。

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

但是使用 private 继承并不是必要的，我们可以使用复合来重构上面的代码。

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

使用下面这种复合方式来撰写代码有两点好处:

- 如果你希望 Widget 可以拥有子类，但是又向阻止子类重新定义 onTick，就需要使用复用。因为继承无法实现这种手段。
- 如果你希望将 Widget 的编译依存性降至最低，就需要使用复用。因为这种形式不需要 include 任何东西。

空白基类优化

Empty Base Optimization(EOB)。这种行为源自对空白类大小的期待。通常，空白类的独立对象大小 会不为0, 因为 c++ 可能会安插一个 char 到空对象中。有的时候为了位对齐的需求，可能会更大。但是， 非独立对象却不是，非独立对象则很有可能大小为 0。

```
class Empty {};
class HoldsAnInt {
private:
  int x;
  Empty x;
};

// sizeof(HoldsAnInt) > sizeof(int)
```

```
class Empty {}
class HoldsAnInt : private Empty {
private:
  int x;
};

// sizeof(HoldsAnInt) == sizeof(int)
```

可以看到使用继承方案，可以消除空白类的体积负担。而这里的 empty 类，可能并不是 empty 的，他们 往往内涵 typedefs, enums, static 成员变量，或者 non-virtual 函数。这里的 empty 实际上 是指，不含 non-static 成员变量。

总结

总的来说当你面对 “并不存在 is-a 关系” 的两个 classes，其中一个需要访问另一个的 protected 成员时，或者重新定义 virtual 函数，private 继承极有可能称为正统设计。当然，你也可以使用 public 继承和复合技术来替代这个过程，尽管有更高的复杂度。

条款40: 明智而审慎的使用多重继承

- 多重继承比单一继承更复杂。它可能导致新的歧义性，以及对 virtual 继承的需要。
- virtual 继承会增加大小、速度、初始化复杂度等等成本。如果 virtual base class 不带任何数据，僵尸最具实用价值的情况。
- 多重继承的确有正当用途。其中一个情节涉及 “public 继承某个 interface class” 和 “private 继承某个协助实现的 class” 的两相组合。

语义歧义

当涉及多重继承，程序可能从一个以上的 base classes 继承相同的名称，这会导致歧义。

```
class BorrowableItem {
public:
  void checkOut(); // 离开时进行检查
};
class ElectronicGadget {
private:
  bool checkOut() const; // 执行自我检测，返回测试是否通过
};

class MP3Player : public BorrowableItem, public ElectronicGadget {};

MP3Player mp3;
mp3.checkOut(); // 究竟调用的哪个 checkOut。
```

虽然上面的 ElectronicGadget::checkOut 是 private 的，但是编译器在开始阶段只在乎调用的名 称是否解析正确。而在此时会发生歧义，因为按照最佳匹配，两者具有相同的匹配程度。

正确的做法是:

```
mp3.BorrowableItem::checkOut()
```

钻石型多重继承

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

上面就是一个钻石型的多重继承，其中存在的问题在于当 File 中有一个对象 A 时，IOFile 中应该有 几分 A 对象。c++ 支持两种选择。

- 多份 这是一个缺省做法，就是从 InputFile 和 OutputFile 中都复制对象 A。上面的代码就是这样的效果。
- 一份 如果你希望只存在一份，那么你就需要令那个 base class 成为一个 virtual base class，也就是你 必须使用 virtual 继承。

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

从正确行为的观点来看，public 继承总应该时 virtual 的。但是这又涉及到成本问题，编译器为了避免 成员变量重复，它必须在幕后做一些工作，这就导致几个结果: 1. virtual public 继承的体积更大，2. virtual public 继承的访问速度更慢，3. classes 若派生自 virtual base，当需要初始化时，必 须认知 virtual bases，不论距离 base 多远，4. 当一个新的 derived class 加入继承体系中，它 必须承担其 virtual bases 的初始化责任。

因此关于 virtual bases classes 的忠告很简单:

1. 非必须使用 virtual base，就使用 non-virtual 继承。
2. 如果必须使用 virtual base，那就尽量避免在其中放置数据。

public 继承 interface，private 继承协助类

多重继承也有很多正常的实用场景。这里举例一个 public 和 private 继承并用的场景。

```
class IPerson {
public:
  virtual ~IPerson();
  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};
```

IPerson 是一个接口，并且使用 工厂函数 产生 Person 对象。这也意味着，想要产生 Person 对象， 首先需要设计一个类继承并实现 IPerson。

另外一个工具类 PersonInfo，用来实现数据库相关操作，它为后续对象提供了一些好用的方法。

```
class PersonInfo {
public:
  explicit PersonInfo(DatabaseId id);
  virtual ~PersonInfo();
  virtual const char* theName() const;
  virtual const char* theBirthDate() const;
private:
  virtual const char* valueDelimOpen() const;
```

```
  virtual const char* valueDElimClose() const;
};
```

现在希望实现一个类，它既可以被生产为 IPerson，也可以依靠 PersonInfo 来实现 IPerson 其中的 操作。

```
class CPerson: public IPerson, private PersonInfo {
public:
  explicit CPerson(DatabaseId id): PersonInfo(id) {}
  virtual std::string name() const { // 借用 PersonInfo 方法实现 IPerson 的方法
    return PersonInfo::theName();
  }
  virtual std::string birthDate const {
    return PersonInfo::theBirthDate();
  }
private:
  virtual const char* valueDelimOpen() const { return "{"; } // 重写从 PersonInfo
继承来的 virtual 限界字符函数
  virtual const char* valueDElimClose() const { return "}"; }
}
```

模板与泛型编程

条款41: 了解隐式接口和编译期多态

- classes 和 templates 都支持接口和多态
- 对 classes 而言接口是显式的，以函数签名为中心。多态则是通过 virtual 函数发生于运行期。
- 对 template 参数而言，接口是隐式的，奠基于有效表达式。多态则是通过 template 具现化和函数重载解析发生于编译期。

面向对象编程世界总是以显式接口和运行期多态解决问题。

```
class Widget {
public:
  Widget();
  virtual ~Widget();
  virtual std::size_t size() const;
  virtual void normalize();
  void swap(Widget& other);
};
void doProcessing(Widget& w) {
  if (w.size() > 10 && w != someNastyWidget) {
    Widget temp(w);
    temp.normalize();
    temp.swap(w);
  }
}
```

上述代码中的 doProcessing 就体现了这两点：

- Widget 是一个显式接口
- Widget 中的 virtual 成员函数，在 doSomething 中依据动态类型而变化是运行期多态的体现。

而在 Templates 和 泛型编程的世界中，这两者重要性降低，反而 隐式接口和编译器多态 重要性升高。

```
template<typename T>
void doProcessing(T& w) {
  if (w.size() > 10 && w != someNastyWidget) {
    T temp(w);
    temp.normalize();
    temp.swap(w);
  }
}
```

例如上述代码：

- 上述的执行过程要求 T 类型必须支持 size, normalize, swap, copy, inequality comparison 等行为，这就是一组隐式接口
- 所有涉及到 w 的任何调用都有可能让 template 具象化，这种行为发生在编译期，会导致编译期多态。

条款42: 了解 typename 的双重意义

- 声明 template 参数时，前缀关键字 class 和 typename 可互换。
- 请使用关键字 typename 标识嵌套从属类型名称，但不得在 base class lists 和 member initialization list 内用它作为 base class 修饰符。

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

上述代码中 class 与 typename 对于 c++ 而言，意义完全相同。但是推荐，当按时参数并非一定是个 class 类型时使用 typename。

但是并非任何时候 class 总是与 typename 等价。

嵌套从属类型名

```
template <typename C>
void print2nd(const C& container) {
  if (container.size() >= 2) {
    C::const_iterator iter(container.begin());
    ++iter;
    int val = *iter;
    std::cout << value;
  }
}
```

上面这段代码接收一个 STL container，并且输出第二号元素。这里面有几个概念：

- 从属名称: template 中出现的名称如果相依于某个 template 参数，就称为从属名称，例如 C 就依赖于 C 类型。
- 嵌套从属名称: 如果丛书名称在 class 内呈现嵌套状，就称为嵌套从属名称，例如 C::const_iterator。
- 非从属名称: 不依赖任何 template 参数的名称，例如 int。

嵌套从属名称可能会导致解析困难的问题，例如

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

上面这段代码如果 C::const_iterator 是一个类，那就没有什么问题。而如果不幸的是传入的类型 C 恰好声明了一个 static 变量，变量名称叫做 const_iterator。那这就变成了一个相乘行为。这显然 和预期不符。同样的，上面的 `C::const_iterator iter(container.begin())` 也会是一个非法 语句。

想要矫正这个行为可以通过 typename 来限制它只能是一个类型。

```
template <typename C>
void print2nd(const C& container) {
  if (container.size() >= 2) {
    typename C::const_iterator iter(container.begin());
  }
}
```

一般性规则是：任何时候当你想要在 template 中指涉一个嵌套从属类型名称，它就必须在紧邻它的前一 个位置放上关键字 typename。

> [!warning] 这一一般性规则的例外是，typename 不剋出现在 base classes list 内的嵌套从属类型之前，也 不可以出现在 member initialization list 中作为 base class 修饰符。

```
template <typename T>
class Derived : public Base<T>::Nested { // base class list 中不允许 typename
public:
  explicit Derived(int x) : Base<T>::Nested(x) { // initialization list 中不允许修饰 base class
    typename Base<T>::Nested temp;
  }
};
```

最后一个例子是 typedef typename 连用。

```
tempalte <typename IterT>
void workWithIterator(IterT iter) {
  typedef typename std::iterator_traits<IterT>::value_type value_type; // IterT 如果是 vector<string>::iterator，那么 value_type 就是 string
  value_type temp(*iter);
}
```

条款43: 学习处理模板化基类内名称

- 可在 derived class templates 内通过 “this->” 指涉 base class templates 内的成员名称，或借由一个明白写出的 "base class 资格修饰符" 完成。

如果编译期我们有足够的信息来决定哪些信息会如何处理，就可以采用基于 template 的方法。下面是一 个例子：

```
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) {}
}
```

在上面这段代码的基础上，如果还需要扩展功能，例如每次送出信息时都进行日志，则很容易在该代码上进 行扩充。

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    sendClear(info); // 无法通过编译
    // 传送后写 log
  }
};
```

虽然这样写非常合理，但是 sendClear 无法通过编译。因为在编译期，编译器并不清楚继承的 MsgSender 中 Company 是 CompanyA 还是 CompanyB。

另一个问题在于，如果进行了全特化，那这类继承又会产生错误。例如，CompanyZ 只能发送加密数据，于 是设计 特化版 的 MsgSender。

```
template <>
class MsgSender<CompanyZ> { // 删除了 sendClear 方法，添加了 sendSecret 方法
public:
  void sendSecret(const MsgInfo& info) {}
};
```

在具有全特化的情况下，再次考察 LoggingMsgSender 类，你会发现如果传入 CompanyZ，那么上面这 段代码注定失败，因为 MsgSender 不具备 sendClear 方法。这也是为什么编译器不允许 上述代码通过编译的原因。

为了编译成功，可以使用下列手段。

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    this->sendClear(info); // 假设 sendClear 被继承，这就会成功通过编译
    // 传送后写 log
  }
};
```

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  using MsgSender<Company>::sendClear;
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    sendClear(info); // 可以通过，直接告诉编译器我们要调用的函数，而不让编译器自己去
base class 中寻找
    // 传送后写 log
  }
};
```

```
template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
  void sendClearMsg(const MsgInfo& info) {
    // 传送前写 log
    MsgSender<Company>::sendClear(info); // 可以通过
    // 传送后写 log
  }
};
```

第三种实现方式通常是最不恰当的实现方式，因为如果被 virtual 修饰，以明确的名称调用会关闭 “virtual 绑定行为”。

上述这三种方法实际上都是 对编译器承诺 “base class template 的任何特化版本都将支持其泛化版本 所提供的接口”。但是面对 CampanyZ 这种违背承诺的行为，编译器仍会编译失败。

条款44: 将与参数无关的代码抽离 templates

- templates 生成多个 classes 和 多个函数，所以任何 template 代码都不应该于某个造成膨胀的 template 参数产生相依关系。
- 因非类型模板参数而造成的代码膨胀，往往可以消除，做法是以函数参数或 class 成员变量替换 template 参数。
- 因类型参数而造成的代码膨胀，往往可降低，做法是让带有完全相同二进制表述的具现类型共享实现码。

尽管 template 可以节省时间且避免代码重复，但是有时候也可能会导致代码膨胀。想要防止这种问题， 可以进行共性与变性分析。

在编写 template 时，也要进行重复代码的判断，这种判断并不像 non-template 这样明确。

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

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

上面这段代码便是存在重复，对于常量 5 和 10 来说，实际只需要一个传入参数的函数来实现即可。

```
template<typename T>
class SquareMatrixBase {
protected:
  void insert(std::size_t matrixSize);
};

template<typename T, std::size_t n>
class SquareMatrix : private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::insert;
public:
  void insert() { this->insert(n); };
};
```

使用这种结构进行设计可以尽量避免 derived classes 代码重复。这里使用 this->insert(n) 是为 了解决 模板化基类内的函数名称会被 derived class 掩盖的问题。但是上述结构仍然具有一些棘手的 问题而没有解决。那就是 SquareMatrixBase 如何修改矩阵数据呢？这部分内容只有 derived class 知道。

一个办法是，为 SquareMatrixBase::insert 添加新的指针参数，但是如果有其他函数你也要这样做。 另外一种办法是在 SquareMatrixBase 存储一个指针，指针指向矩阵所在的内存。

```
template<typename T>
class SquareMatrixBase {
protected:
  SquareMatrixBase(std::size_t n, T* pMem) : size(n), pData(pMem) {}
  void setDataPtr(T* ptr) { pData = ptr; }
  void insert(std::size_t matrixSize);
private:
  std::size_t size;
  T* pData;
};

template<typename T, std::size_t n>
class SquareMatrix : private SquareMatrixBase<T> {
private:
  using SquareMatrixBase<T>::insert;
public:
  SquareMatrix() : SquareMatrixBase<T>(n, data) {}
  void insert() { this->insert(n); }
private:
  T data[n * n];
};
```

条款45: 运用成员函数模板接受所有兼容类型

- 请使用 member function templates 生成 “可接受所有兼容类型” 的函数。
- 如果你声明 member template 用于“泛化 copy 构造”或者 “泛化 assignment 操作”，你还是需要声明正常的 copy 构造函数和 copy assignment 操作符。

在使用指针时，推荐使用智能指针，它能保障自动删除 heap-based 资源。但是有时候，智能指针并不会 像真实指针那样完成工作，例如 ++ 操作。因为真实指针支持隐式转换，特别是 derived class 转换为 base class。

```
class Top {};
class Middle : public Top {};
class Bottom : public Middle {};
Top* pt1 = new Middle;
Top* pt2 = new Bottom;
const Top* pct2 = pt1;
```

```
template<typename T>
class SmartPtr {
public:
  explicit SmartPtr(T* realPtr);
};

SmartPtr<Top> pt1 = SmartPtr<Middle>(new Middle);
```

```
SmartPtr<Top> pt2 = SmartPtr<Bottom>(new Bottom);
SmartPtr<const Top> pct1 = pt1;
```

上面这两段代码的对比就有这种情况，下面这段代码是不成立的。因为同一个 template 的不同具现体之 间并没有与生俱来的关系，即，不像继承关系那样。为了能够获得我们希望的 SmartPtr class 之间的 转换能力，我们必须将他们明确编写出来。

Templates 和 泛型编程

如果仅仅通过单一的构造函数来实现，那工程量显然是巨大的。任何新增的一种类型都会需要你通过扩充构 造函数来实现。

我们实际需要的是一个构造模板，这种模板是所谓的 member function templates，其作用是为了 class 生成 copy 构造函数。

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other);
};
```

上面这段代码被称为泛化 copy 构造函数。其中的参数没有声明为 explicit，是因为原始指针类型之间 的转换是隐式转换，无需明白写出的转型动作。

但是，上面这段代码的功能与我们所需的功能有部分冲突，因此我们需要扩充并限制。首先是我们希望安全 的隐式转换发生，而非任意的转换，例如 Base class 转向 derived class 或者 int* 转向 double*。其次是，我们希望能够提供和 std::auto_ptr 等标准的智能指针相同的方法，例如 get。

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

上面这段既提供了通用方法，又保证了合理的隐式转换的发生。

另外需要注意的一点是，在 class 内声明泛化的 copy 构造函数并不会阻止编译期生成自己的 copy 构 造函数。所以如果你想自己控制 copy 构造的方方面面，就必须同时声明泛化 copy 构造函数和 “正常” 的 copy 构造函数。

条款46: 需要类型转换时请为模板定义非成员函数

- 当我们编写一个 class template，而它所提供之 “与此 template 相关的” 函数支持 “所有参数之隐式类型转换” 时，请将那些函数定义为 “class template 内部的 friend 函数”。

在条款24中讨论了为什么只有 non-member 函数才有能力 在所有实参身上实施隐式类型转换。但是在模 板化过程中，条款24似乎就不再适用了。

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

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

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

上述代码无法通过编译，它并没有像非模板那样按照预期运行。这主要是因为，模板的运行过程是先根据传 入的参数进行对模板类型 T 的推算，而在这个过程中并不会进行隐式类型转换。

一种解决这种问题的思路是，利用 friend 来将 non-member 函数声明在 Rational 内，这样就 不再需要根据后面的参数进行推算了，这样也就支持了混合式调用了。

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

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

主义上面并没有将 operator* 定义在 Rational 外部，因为那会导致链接失败，这种情况只会发生在 template 领域中，c 和 OOC 中不会因此而链接失败。为了解决这个问题，我们选在将函数声明和定义 合并在 Rational 内部。这种 friend 的使用方式和往常不同，这里是为了在 class 内部声明一个 non-member 函数而可选的唯一办法。

写在内部，则意味着也就成为了一个 inline 函数。按照 条款30 所说的那样，如果函数内部过于复杂， 那可以通过调用辅助函数来实现（本案例中函数已经非常简单了，其实不用辅助函数也可以）。

```
template <typename T> class Rational;

template <typename T>
const Rational<T> doMultiply(const Rational<T>& lhs, const Rational<T>& rhs) {
  return Rational<T>(
    lhs.numerator() * rhs.numerator(),
    lhs.denominator() * rhs.denominator()
  );
}

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

作为一个 template，doMultiply 就不再支持混合式乘法了，但是 friend operator* 已经支持混合 式调用了，那也就无需担心 doMultiply 是否支持混合式乘法的问题了。

条款47: 请用 traits classes 表现类型信息

- traits classes 使得 “类型相关信息” 在编译期可用。它们以 templates 和 “template 特化” 来实现。
- 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

使用 traits class 是一种非常常见的手段，以 STL 中的 工具性 templates——advance 为例。

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d); // 将迭代器像前移动 d 单位。
```

在 advance 的实现上，需要区别 random access 迭代器，或者其他类型迭代器。因为只有前者可以 进行 += 操作。

STL 迭代器分类

- input 迭代器：只能向前移动，一次一步，可读但只可读一次。代表：istream_iterators。
- output 迭代器：只能向前移动，一次一步，可写但只可写一次。代表：ostream_iterators。

- forward 迭代器：可以同时完成 input/output 迭代器工作，且可读可写多次。
- Bidirectional 迭代器：除了向前移动还可以向后移动。
- random access 迭代器：在 Bidirectional 迭代器的基础上，可以执行“迭代器算术”。

这五类迭代器提供了专属的卷标结构(tag struct)，并且之间的继承关系是：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};
```

现在回到 advance 之上，我们已经清楚的知道了 random access 迭代器可以直接进行运算，因此不再 需要像其他类型迭代器一样进行多遍递增递减操作。那么如何来判断一个 iter 是否为 random access 迭代器呢？使用 traits。

traits 并不是 c++ 关键字或者一个预先定义好的构件，而是一个技术。按照这种技术写成的 templates 在标准库中有若干个，其中针对迭代器的被命名为 iterator_traits。

```
template <typename IterT>
struct iterator_traits;
```

如何让自己所实现的迭代器能够运用这个 traits 呢？只需如下编写代码。

```
template <>
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
  };
};
```

然后，iterator_traits 会将 iterator class 中嵌入的 typedef 进行读取。

```
template <typename IterT>
struct iterator_traits {
  typedef typename IterT::iterator::iterator_category iterator_category;
};
```

虽然上面这种方式对于用户自定义的 iterator 是成立的，但是对指针却行不通。为此，iterator_traits 特别针对指针提供了一个偏特化版本。

```

```

```
template<typename IterT>
struct iterator_traits<IterT*> {
  typedef random_access_iterator_tag iterator_category; // 指针 和 random access
的行为类似
};
```

如何实现一个 traits class

- 确认若干希望将来可以取得的类型相关信息，例如对 iterator 而言，希望将来可以取得分类。
- 为该信息选择一个名称。
- 提供一个 template 和一组特化版本，内涵你希望支持的类型和相关信息。

有了 iterator_traits 如何实现一个 advance。可能你会选择使用 if-else 来进行逐一判断。但是 这种效率并不高。更好的方法是使用和 template 一样在编译期进行判断的方法来实现。一种可行的做法 是重载。

```
template <typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::random_access_iterator_tag /*忽略变量名，
因为后续用不到*/) {
  iter += d;
};
template <typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::bidirectional_iterator_tag /*忽略变量名，
因为后续用不到*/) {
  if (d >= 0) { while(d--) ++iter; }
  else { while(d++) --iter; }
};
template <typename IterT, typename DistT>
void doAdvance(IterT& iter, DistT d, std::input_iterator_tag /*忽略变量名，因为后续
用不到*/) {
  if (d < 0)
    throw std::out_of_range("Negative distance");
  while(d--) ++iter;
};


template <typename IterT, typename DistT>
void advance(IterT& iter, DistT d) {
  doAdvance(iter, d, typename std::iterator_traits<IterT>::iterator_category());
}
```

现在，我们了解了如何使用一个 traits class。

- 建立一组重载函数或者函数模板，彼此之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。
- 建立一个控制函数或者函数模板，它调用上述这些"劳工函数"并传递 traits class 所提供的信息。

Traits 广泛用于标注程序库。包括 iterator_traits，它不仅提供了上述类型功能，还提供了四份迭 代器相关的信息，最有用的是 value_type。此外还有 char_traits 用来保存字符类型的相关信息，以 及 numeric_limits 用来保存数值类型的相关信息。

条款48: 认识 template 元编程

- TMP 可将工作由运行期移往编译期，因而得以实现早期错误侦测和更高的执行效率。
- TMP 可被用来生成 “基于政策选择组合” 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。

模板元编程 (Template metaprogramming, TMP) 是编写 template-based c++ 程序并执行于编译 期的过程。它是以 c++ 写成、执行于 c++ 编译器内的程序。

TMP有两个伟大的效力：

1. 它让事情变得更容易。
2. 由于 TMP 执行于 c++ 编译期，因此可将工作从运行期转移到编译期。

一个好的例子是上一条款所设计的 advance 函数，使用 TMP 将原本通过运行时 if-else 的方法，通过 重载的方法来让其在编译期实现。

针对 TMP 而设计的程序库 (Boosts's MPL) 提供更高层级的语法。

再从循环角度来看看 TMP 如何运作。TMP 并不提供真正的循环，而是使用递归方案来实现。TMP 的递归 甚至不是正常种类，因为 TMP 循环并不涉及递归函数调用，而是涉及 “递归模板具现化”。

使用 TMP 实现阶乘

```
template <unsigned n>
struct Factorial {
  enum { value = n * Factorial<n - 1>::value };
};

template <>
struct Factorial<0> { // 全特化 0! = 1
  enum { value = 1 };
};

int main() {
  std::cout << "5! = " << Factorial<5>::value << std::endl;
  std::cout << "10! = " << Factorial<10>::value << std::endl;
}
```

使用 Factorial::value 就可以直接得到 n 阶阶乘。

下面给出三个通过 TMP 能实现的目标实例:

- 确保量度单位正确。例如质量、距离、时间、速度的关系。将一个质量赋值给速度是不正确的，但是距离除以时间赋值给速度则是正确的。使用 TMP 可以在编译期保证这种约束。
- 优化矩阵运算。原有的矩阵乘法使用 operator* 来执行，必须返回新对象。而在多个矩阵连乘的过程中则会创建多个临时对象。通过 TMP 来实现，就有可能消除这些临时对象，并合并循环。

- 可以生成客户定制之设计模式实现品。设计模式例如 Strategy, Observer, Visitor 等等都有多种实现方式。运用所谓的 policy-based design 之 TMP-based 技术，有可能产生一些 templates 用来表述独立的设计选项，然后可以任意结合它们，导致模式实现品带着客户指定的行为。这项技术已经被用来让若干 templates 实现出只能指针的行为策略，用以编译期生成数以百计不同的只能指针类型。

定制的 new 和 delete

条款49: 了解 new-handleer 的行为

- set_new_handler 允许客户指定一个函数，在内存分配无法获得满足时被调用。
- Nothrow new 是一个颇为局限的工具，因为它只适用于内存分配；后继的构造函数调用还是可能抛出异常。

当 operator new 无法满足某一内存分配需求时，它会抛出异常。对于旧式操作，该行为会返回一个 null 指针。我们可以通过修改错误处理函数来改变这种默认行为，从而获得旧式操作体验。

当 operator new 的行为无法被满足时，它会先调用一个客户指定的错误处理函数，所谓的 new-handler。客户可以通过调用 set_new_handler 来指定函数。

```
namespace std {
  typedef void (*new_handler) ();
  new_handler set_new_handler(new_handler p) throw(); // 不抛出任何异常
}
```

你可以这样设置 new-handler。

```
#include <new>

void outOfMem() {
  std::cerr << "Unable to satisfy request for memory\n";
  std::abort(); // 终结
}

int main() {
  std::set_new_handler(outOfMem);
  int* pBigDataArray = new int[10000000000L];
}
```

当 operator new 无法满足内存申请时，它会不断调用 new-handler 函数，直到找到足够的内存。可 以将上面的 std::abort 注释掉尝试一下。

因此一个结论是，一个设计良好的 new-handler 函数必须做到以下的事情:

- 让更多内存可以被使用: 这个策略的一个做法是，程序一开始执行就分配一大块内存，然后再 new-handler 第一次被调用时，将它们释放给程序使用。
- 安装另一个 new-handler: 如果当前这个 new-handler 无法获取更多的可用内存，或许它直到另外哪个 new-handler 有此能力，如果是这样可以对 new-handler 进行替换。
- 卸除 new-handler: 将 null 指针传给 set_new_handler，一旦没有安装任何 new-handler，operator new 会在内存分配不成功时抛出异常。

- 不返回: 直接调用 abort 或者 exit。

有时，我们希望能够根据不同的 class 来执行专属的内存分配失败处理函数。c++ 并不支持 class 专 属的 new-handlers，但是可以为每一个 class 提供自己的 set_new_handler 和 operator new。

```
class NewHandlerHolder { // RAII
public:
  explicit NewHandlerHolder (std::new_handler nh) : handler(nh) {}
  ~NewHandlerHolder() {
    std::set_new_handler(handler);
  }
private:
  std::new_handler handler;
  NewHandlerHolder(const NewHandlerHolder&); // 阻止 copying 发生
  NewHandlerHolder& operator=(const NewHandlerHolder&);
};


class Widget {
public:
  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void* operator new(std::size_t size) throw(std::bad_alloc);
private:
  static std::new_handler currentHandler;
};

std::new_handler currentHandler = 0;
std::new_handler set_new_handler(std::new_handler p) throw() {
  std::new_handler oldHandler = currentHandler;
  crrentHandler = p;
  return oldHandler;
}
void* Widget::operator new(std::size_t size) throw(std::bad_alloc) {
  NewHandlerHolder h(std::set_new_handler(currentHandler)); // 返回时自动销毁，并在
析构函数中重新绑定默认的 new_handler
  return ::operator new(size); // 调用 global operator new
}
```

上述代码可以进行一般化处理，采用复合的方式来构建。简单的做法是简历一个 “mixin” 风格的 base class，这种 base class 用来允许 derived class 继承单一特定的能力。

```
template <typename T>
class NewHandlerSupport {
public:
  static std::new_handler set_new_handler(std::new_handler p) throw();
  static void* operator new(std::size_t size) throw(std::bad_alloc);
private:
  static std::new_handler currentHandler;
};
template <typename T>
```

```
std::new_handler NewHandlerSupport<T>::set_new_handler(std::new_handler p) throw()
{
  std::new_handler oldHandler = currentHandler;
  crrentHandler = p;
  return oldHandler;
}
template <typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size) throw(std::bad_alloc) {
  NewHandlerHolder h(std::set_new_handler(currentHandler)); // 返回时自动销毁，并在析构函数中重新绑定默认的 new_handler
  return ::operator new(size); // 调用 global operator new
}
template <typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = 0;

class Widget : public NewHandlerSupport<Widget> { // 不再需要提供 set_new_handler 和 operator new
  ...
};
```

上面的 T 只是为了区分类型，而没有特殊含义。这种行为叫做 curiously recurring template pattern; CRTP。

现在大多数的 operator new 实现，如果空间不足都会抛出 bad_alloc 异常。但是也可以支持返回 null 的方法。

```
class Widget {};
Widget* pw1 = new Widget(); // 如果分配失败，抛出 bad_alloc
if (pw1 == 0) ... // 没必要，因为一定 pw1 不可能为0

Widget* pw2 = new (std::nothrow) Widget(); // 失败，返回0
if (pw2 == 0) ... // 可能成功
```

然而这种方式完全没必要，因为 nothrow 之后限制 operator new 抛出异常，而当 constructor 抛 出异常仍然会传播。

条款50: 了解 new 和 delete 的合理替换时机

- 有许多理由需要写个自定的 new 和 delete，包括改善效能、对 heap 运用错误进行调试、收集 heap 使用信息

为什么会想要替换编译器提供的 operator new 或者 operator delete ?

- 用来检测运用上的错误。
- 为了强化效能。
- 为了收集使用上的统计数据。

一个定制 operator new 的例子，用于促进并协助检查 "overruns"(写入点再分配区块之后) 或 "underruns"(写入点再分配区块之前)。

```
static const int signature = 0xDEADBEEF;
typedef unsigned char Byte;
// 下列代码仍然存在错误
void* operator new(std::size_t size) throw (std::bad_alloc) {
  using namespace std;
  size_t realSize = size + 2 * sizeof(int); // 增加两个空间用来存放 两个 signature

  void* pMem = malloc(realSize); // 调用 malloc 取得内存
  if (!pMem) throw bad_alloc(); // 如果为空就抛出 bad_alloc

  // 将 signature 写入内存的最前和最后
  *(static_cast<int*>(pMem)) = signature;
  *(reinterpret_cast<int*>(static_cast<Byte*>(pMem) + realSize - sizeof(int))) =
signature;

  return static_cast<Byte*>(pMem) + sizeof(int); // 返回 signature 之后的内存位置
}
```

对齐

这里强调一下 对齐 问题。在计算机体系结构中，如果对齐条件满足，通常是效率较高的。比如一个 double 如果是 8-byte，当它正好与地址的 8-byte 位置对齐时效率时最高的。通常使用 malloc 获得的地址 是满足对齐的。但是上述代码中，我们显然返回的不是原始 pointer，而是一个后移了 4-byte 的指针。 这时就可能获得的是一个没有适当对齐的指针，那么可能会造成程序崩溃或者执行速度缓慢。

有时好的效率是必要的，但是很多时候这也没有太大影响。

那么究竟何时需要替换缺省的 new 和 delete 呢？

- 为了检测运用错误。
- 为了收集动态分配内存之使用统计信息。
- 为了增加分配和归还的速度。
- 为了降低缺省内存管理器带来的空间额外开销。
- 为了弥补缺省分配器中的非最佳对齐。
- 为了将相关对象成簇集中。
- 为了获得非传统行为。

条款51: 编写 new 和 delete 时需要固守常规

- 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。它在应该有能力处理 0 bytes 申请。Class 专属版本则还应该处理 “比正确大小更大的申请”。
- operator delete 应该在收到 null 指针时不做任何事。Class 专属版本则还应该处理 “比正确大小更大的申请”。

编写自己的 operator new 和 operator delete 时应该遵守哪些规矩。

operator new

实现一致性的 operator new 1. 必须得返回正确的值，2. 内存不足时必须得调用 new-handling 函 数，3. 必须有能够对付零内存需求的准备，4. 避免不慎掩盖正常形式的 new。

operator new 的返回值非常简单，如果有能力就返回内存指针，没有则抛出 bad_alloc 异常。

一个 operator new 的伪代码:

```
void* operator new(std::size_t size) throw(std::bad_alloc) {
  using namespace std;
  if (size == 0) { // 用来处理 申请 0-bytes 内存的行为
    size = 1;
  }
  while (true) {
    尝试分配 size bytpes;
    if (成功)
      return pointer;

    // 分配失败
    new_handler globalHandler = set_new_handler(0); // 获取当前的 new-handling 函数
    set_new_handler(globalHandler);

    if (globalHandler) // 如果 globalHandler 不为空就执行
      (*gloablHandler)();
    else // 否则默认抛出异常
      throw std::bad_alloc();
  }
}
```

但是上述代码有一个小缺点，就是会影响 derived class。而通常我们重写 operator new 实际上是 为了能够针对某个特定 class 进行优化，而非其它 derived class。一种好的方法是，在operator new 中进行判断大小。

```
void* Base::operator new(std::size_t size) throw(std::bad_alloc) {
  if(size != sizeof(Base))
    return ::operator new(size);
}
```

这不仅包括了对类型的判断，也包括了对 size == 0 的判断，因为 c++ 中独立对象必须有大小，因此 sizeof(Base) 不可能为0。

对于 operator new[] 来分配数组，唯一需要做的就是分配一块未加工的内存，其他计算空间大小等工 作都不应该在此处进行。

operator delete

operator delete 情况十分简单，只需要记住的唯一一件事情，就是保证 “删除 null 指针永远安全”。

```
void operator delete(void* rawMemory) throw() {
  if (rawMemory == 0) return;
```

```
  // 归还 rawMemory 内存。
}
```

member 版本也十分简单，和 operator new 一样进行类型判断即可。

```
void* Base::operator delete(void* rawMemory, std::size_t size) throw() {
  if (rawMemory == 0) return;
  if(size != sizeof(Base)) {
    ::operator delete(rawMemory);
    return;
  }
  ...
}
```

另外，如果即将被删除的对象派生自某个 base class 而后者欠缺 virtual 析构函数，那么 c++ 传给 operator delete 的 size_t 数值可能不正确。

条款52: 写了 placement new 也要写 placement delete

- 当你写一个 placement operator new，请确定也写出了对应的 placement operator delete。如果没有这样做，你的程序可能会发生时断时续的内存泄露。
- 当你声明 placement new 和 placement delet，请确定不要无疑是的遮掩他们的正常版本。

---

new operator 总是分为两步进行的，首先调用 operator new 进行空间申请，而第二步则是执行 constructor。如果后者出错，那么 operator new 所分配的地址空间应该怎样做？答案必然是被释放以防止内存泄露。 但是有个问题，就是编译器怎么找到对应的 operator delete 呢？

```
class Widget {
public:
  static void* operator new(std::size_t size, std::ostream& logStream); // 非正常形式的 new
  static void operator delete(void* pMemory); // 正常形式的 new
};

{
  Widget* pw = new (std::cerr) Widget;
}
```

实际上编译器会选择同调用的 operator new 最相像的 operator delete 进行调用，也就是说寻找 参数个数和类型都与 operator new 相同的某个 operator delete。例如上面这段代码，当 Widget constructor 发生异常后就会调用 operator delete(std::size_t size, std::ostream& logStream); 而上面没有定义就会产生另一个异常最终导致 abort。

而在正常的主动删除情况下，可以直接 delete 即可，而此时并不会去调用与 operator new 对应版本 的 operator delete。

所以为了保证内存安全，你应该随时虽可提供相应形式的 delete。这和 placement new/delete 成对 设计的道理相同。

```
class Widget {
public:
  static void* operator new(std::size_t size, std::ostream& logStream); // 非正常形式的 new
  static void operator delete(void* pMemory); // 正常形式的 new
  static void operator delete(void* pMemory, std::ostream& logStream);
};

{
  Widget* pw = new (std::cerr) Widget;
}
```

需要注意的一点是，重新定义 operator new/delete 是会进行名称掩盖的。

```
class Base {
public:
  static void* operator new(std::size_t size, std::ostream& logStream)
throw(std::bad_alloc);
};
class Derived : public Base {
public:
  static void* operator new(std::size_t size) throw(std::bad_alloc);
};

Base* pb = new Base; // 错误，global 被掩盖了
Base* pb = new (std::cerr) Base; // 正确

Derived* pd = new (std::clog) Derived; // 错误，Base 被掩盖掉了
Derived* pd = new Derived; // 正确
```

如果你不希望这些默认行为发生掩盖呢？你可以按照下面这样进行设计。

```
class StandardNewDeleteForms {
public:
  // normal new/delete
  static void* operator new(std::size_t size) throw(std::bad_alloc) {
    return ::operator new(size);
  }
  static void operator delete(void* pMemory) throw() {
    ::operator delete(pMemory);
  }
```

```
  // placement new/delete
  static void* operator new(std::size_t size, void* ptr) throw(std::bad_alloc) {
    return ::operator new(size, ptr);
  }
  static void operator delete(void* pMemory, void* ptr) throw() {
    ::operator delete(pMemory, ptr);
  }

  // nothrow new/delete
  static void* operator new(std::size_t size, const std::nothrow_t& nt) throw() {
    return ::operator new(size, nt);
  }
  static void operator delete(void* pMemory, const std::nothrow& nt) throw() {
    ::operator delete(pMemory);
  }
};


class Widget : public StandardNewDeleteForms {
public:
  using StandardNewDeleteForms::operator new;
  using StandardNewDeleteForms::operator delete;
  static void* operator new(std::size_t size, std::ostream& logStream)
throw(std::bad_alloc);
  static void operator delete(void* pMemory, std::ostream& logStream) throw();
};
```

利用继承机制和 using 声明式来取得标准形式。

杂项讨论

条款53: 不要轻视编译器的警告

- 严肃对待编译器发出的警告信息。努力在你得编译器的最高警告级别下争取 “无任何警告” 的荣誉。
- 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，你原本依赖的警告信息有可能消失。

许多程序员会忽略编译器警告，但是有时候警告会产生运行时异常。例如下面这段代码：

```
class B {
public:
  virtual void f() const {
    std::cout << "i'm B" << std::endl;
  }
};

class D : public B {
public:
  virtual void f() {
    std::cout << "i'm D" << std::endl;
  }
};

B* pb = new D;
  pb->f();
```

编译器可能会说 warning: D::f() hides virtual B::f()，看上去这是理所应当的事情，但是实际 上它想告诉你，这并非是重写操作，这种名称掩盖会导致最终结果与期望不一的情况。

尽管写出一个完全没有警告的代码是十分困难的，但是你应该去理解这些警告的含义，并在完全理解后来判 断是否应该忽略这些警告，否则造成错误后很有可能就是警告造成的。

条款54: 让自己熟悉包括 TR1 在内的标准程序库

条款55: 让自己熟悉 Boost