Effective C++

目录

第一章：让自己习惯 C++

条款 1：视 C++ 为一个语言联邦

C++ 拥有多种不同的编程范式，而这些范式集成在一个语言中，使得 C++ 是一门即灵活又复杂的语言：

1. 传统的面向过程 C：区块，语句，预处理器，内置数据类型，数组，指针。
2. 面向对象的 C with Classes：类，封装，继承，多态，动态绑定。
3. 模板编程 Template C++ 和堪称黑魔法的模板元编程（TMP）。
4. C++ 标准库 STL。

条款 2：尽量以 const, enum, inline 替换 #define

在原书写成时 C++11 中的 constexpr 还未诞生，现在一般认为应当用 constexpr 定义编译期常量来替代大部分的#define 宏常量定义：

```
#define ASPECT_RATIO 1.653
```

替代为：

```
constexpr auto aspect_ratio = 1.653;
```

我们也可以将编译期常量定义为类的静态成员：

```
class GamePlayer {
public:
    static constexpr auto numTurns = 5;
};
```

enum 可以用于替代整型的常量，并且在模板元编程中应用广泛（见条款 48）：

```
class GamePlayer {
public:
    enum { numTurns = 5 };
};
```

大部分#define 宏常量应当用内联模板函数替代：

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

替代为：

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

需要注意的是，宏和函数的行为本身并不完全一致，宏只是简单的替换，并不涉及传参和复制。

条款 3：尽可能使用 const

若你想让一个常量只读，那你应该明确说出它是 const 常量，对于指针来说，更是如此：

```
char greeting[] = "Hello";
char* p = greeting;                 // 指针可修改，数据可修改
const char* p = greeting;           // 指针可修改，数据不可修改
char const* p = greeting;           // 指针可修改，数据不可修改
char* const p = greeting;           // 指针不可修改，数据可修改
const char* const p = greeting;     // 指针不可修改，数据不可修改
```

对于 STL 迭代器，分清使用 const 还是 const_iterator：

```
const std::vector<int>::iterator iter = vec.begin();
// 迭代器不可修改，数据可修改
std::vector<int>::const_iterator iter = vec.begin();
// 迭代器可修改，数据不可修改
```

面对函数声明时，如果你不想让一个函数的结果被无意义地当作左值，请使用 const 返回值：

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

const 成员函数：

const 成员函数允许我们操控 const 对象，这在传递常引用时显得尤为重要：

```
class TextBlock {
public:
    const char& operator[](std::size_t position) const {
    // const 对象使用的重载
        return text[position];
    }
```

```
    char& operator[](std::size_t position) {
    // non-const 对象使用的重载
        return text[position];
    }

private:
    std::string text;
};
```

这样，const 和 non-const 对象都有其各自的重载版本：

```
void Print(const Textblock& ctb) {
    std::cout << ctb[0];
    // 调用 const TextBlock::operator[]
}
```

编译器对待 const 对象的态度通常是位常量性，而我们在编写程序时通常采用逻辑常量性，这就意味着，在确保客户端不会察觉的情况下，我们认为 const 对象中的某些成员变量应当是允许被改变的，使用关键字 mutable 来标记这些成员变量：

```
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);  // 可以修改 mutable 成员变量
        lengthIsValid = true;             // 可以修改 mutable 成员变量
    }
    return textLength;
}
```

在重载 const 和 non-const 成员函数时，需要尽可能避免书写重复的内容，这促使我们去进行常量性转除。在大部分情况下，我们应当避免转型的出现，但在此处为了减少重复代码，转型是适当的：

```
class TextBlock {
```

```
public:
    const char& operator[](std::size_t position) const {

        // 假设这里有非常多的代码
        return text[position];
    }

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

private:
    std::string text;
};
```

需要注意的是，反向做法：令 const 版本调用 non-const 版本以避免重复——并不被建议，一般而言 const 版本的限制比 non-const 版本的限制更多，因此这样做会带来风险。

条款 4：确定对象在使用前已被初始化

无初值对象在 C/C++ 中广泛存在，因此这一条款就尤为重要。在定义完一个对象后需要尽快为它赋初值：

```
int x = 0;
const char* text = "A C-style string";

double d;
std::cin >> d;
```

对于类中的成员变量而言，我们有两种建议的方法完成初始化工作，一种是直接在定义处赋初值（since C++11）：

```
class CTextBlock {
private:
    std::size_t textLength{ 0 };
    bool lengthIsValid{ false };
};
```

另一种是使用构造函数成员初始化列表：

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
```

```
    : theName(name),
      theAddress(address),
      thePhones(phones),
      numTimesConsulted(0) {}
```

成员初始化列表也可以留空用来执行默认构造函数：

```
ABEntry::ABEntry()
    : theName(),
      theAddress(),
      thePhones(),
      numTimesConsulted(0) {}
```

需要注意的是，类中成员的初始化具有次序性，而这次序与成员变量的声明次序一致，与成员初始化列表的次序无关。

类中成员的初始化是可选的，但是引用类型必须初始化。基于赋值的初始化会首先调用 default 构造函数设置初值，之后再立刻赋上新值。为了一致性，能使用初始化列表就使用。

静态对象的初始化：

C++ 对于定义于不同编译单元内的全局静态对象的初始化相对次序并无明确定义，因此，以下代码可能会出现使用未初始化静态对象的情况：

```
// File 1
extern FileSystem tfs;

// File 2
class Directory {
public:
    Directory() {
        FileSystem disk = tfs;
    }
};

Directory tempDir;
```

在上面这个例子中，你无法确保位于不同编译单元内的 tfs 一定在 tempDir 之前初始化完成。

这个问题的一个有效解决方案是采用 **Meyers' singleton**，将全局静态对象转化为局部静态对象：

```
FileSystem& tfs() {
```

```
    static FileSystem fs;
    return fs;
}

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

这个手法的基础在于：C++ 保证，函数内的局部静态对象会在**该函数被调用期间**和**首次遇上该对象之定义式**时被初始化。

当然，这种做法对于多线程来说并不具有优势，最好还是在单线程启动阶段手动调用函数完成初始化。

第二章：构造/析构/赋值运算

条款 5：了解 C++ 默默编写并调用哪些函数

C++ 中的空类并不是真正意义上的空类，编译器会为它预留以下内容：

```
class Empty {
public:
    Empty() { ... }                             // 默认构造函数（没有任何构造函数时）
    Empty(const Empty&) { ... }                 // 拷贝构造函数
    Empty(Empty&&) { ... }                      // 移动构造函数 (since C++11)
    ~Empty() { ... }                            // 析构函数

    Empty& operator=(const Empty&) { ... }      // 拷贝赋值运算符
    Empty& operator=(Empty&&) { ... }           // 移动赋值运算符
};
```

唯有当这些函数被调用时，它们才会真正被编译器创建出来，下面代码将造成上述每一个函数被创建：

```
Empty e1;                   // 默认构造函数 & 析构函数
Empty e2(e1);               // 拷贝构造函数
Empty e3 = std::move(e2);   // 移动构造函数 (since C++11)
e2 = e1;                    // 拷贝赋值运算符
e3 = std::move(e1);         // 移动赋值运算符 (since C++11)
```

需要注意的是，拷贝赋值运算符只有在允许存在时才会自动创建，比如以下情况：

```
class NamedObject {
private:
    std::string& nameValue;
};
```

在该类中，我们有一个 string 引用类型，然而引用无法指向不同对象，因此编译器会拒绝为该类创建一个默认的拷贝赋值运算符。

除此之外，以下情形也会导致拷贝赋值运算符不会自动创建：

1. 类中含有 const 成员。
2. 基类中含有 private 的拷贝赋值运算符。

条款 6：若不想使用编译器自动生成的函数，就该明确拒绝

原书中使用的做法是将不想使用的函数声明为 private，但在 C++11 后我们有了更好的做法：

```
class Uncopyable {
public:
    Uncopyable(const Uncopyable&) = delete;
    Uncopyable& operator=(const Uncopyable&) = delete;
};
```

条款 7：为多态基类声明虚析构函数

当派生类对象经由一个基类指针被删除，而该基类指针带着一个非虚析构函数，其结果是未定义的，可能会无法完全销毁派生类的成员，造成内存泄漏。消除这个问题的方法就是对基类使用虚析构函数：

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

如果你不想让一个类成为基类，那么在类中声明虚函数是是一个坏主意，因为额外存储的虚表指针会使类的体积变大。

只要基类的析构函数是虚函数，那么派生类的析构函数不论是否用 virtual 关键字声明，都自动成为虚析构函数。

虚析构函数的运作方式是，最深层派生的那个类的析构函数最先被调用，然后是其上的基类的析构函数被依次调用。

如果你想将基类作为抽象类使用，但手头上又没有别的虚函数，那么将它的析构函数设为纯虚函数是一个不错的想法。考虑以下情形：

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

但若此时从该基类中派生出新的类，会发生报错，这是因为编译器无法找到基类的析构函数的实现。因此，即使是纯虚析构函数，也需要一个函数体：

```
Base::~Base() {}
```

或者以下写法也被允许：

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

条款 8：别让异常逃离析构函数

在析构函数中吐出异常并不被禁止，但为了程序的可靠性，应当极力避免这种行为。

为了实现 RAII，我们通常会将对象的销毁方法封装在析构函数中，如下例子：

```
class DBConn {
public:
    ...
    ~DBConn() {
        db.close();     // 该函数可能会抛出异常
    }

private:
    DBConnection db;
};
```

但这样我们就需要在析构函数中完成对异常的处理，以下是几种常见的做法：

第一种：杀死程序：

```
DBConn::~DBConn() {
    try { db.close(); }
    catch (...) {
        // 记录运行日志，以便调试
        std::abort();
    }
}
```

第二种：直接吞下异常不做处理，但这种做法不被建议。

第三种：重新设计接口，将异常的处理交给客户端完成：

```
class DBConn {
public:
    ...
    void close() {
        db.close();
        closed = true;
    }

    ~DBConn() {
        if (!closed) {
            try {
                db.close();
            }
            catch(...) {
                // 处理异常
            }
        }
    }

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

在这个新设计的接口中，我们提供了 close 函数供客户手动调用，这样客户也可以根据自己的意愿处理异常；若客户忘记手动调用，析构函数才会自动调用 close 函数。

当一个操作可能会抛出需要客户处理的异常时，将其暴露在普通函数而非析构函数中是一个更好的选择。

条款 9：绝不在构造和析构过程中调用虚函数

在创建派生类对象时，基类的构造函数永远会早于派生类的构造函数被调用，而基类的析构函数永远会晚于派生类的析构函数被调用。

在派生类对象的基类构造和析构期间，对象的类型是基类而非派生类，因此此时调用虚函数会被编译器解析至基类的虚函数版本，通常不会得到我们想要的结果。

间接调用虚函数是一个比较难以发现的危险行为，需要尽量避免：

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

private:
    void Init(){
        ...
        LogTransaction();      // 此处间接调用了虚函数!
    }
};
```

如果想要基类在构造时就得知派生类的构造信息，推荐的做法是在派生类的构造函数中将必要的信息向上传递给基类的构造函数：

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

Transaction::Transaction(const std::string& logInfo) {
    LogTransaction(logInfo);
    // 更改为了非虚函数调用
}

class BuyTransaction : public Transaction {
public:
    BuyTransaction(...)
        : Transaction(CreateLogString(...)) { ... }
```

```
 // 将信息传递给基类构造函数
private:
    static std::string CreateLogString(...);
}
```

注意此处的 CreateLogString 是一个静态成员函数，这是很重要的，因为静态成员函数可以确保不会使用未完成初始化的成员变量。在 base class 构造和析构期间调用的 virtual 函数不可下降至 derived classes。

条款 10：令 operator= 返回一个指向 *this 的引用

虽然并不强制执行此条款，但为了实现连锁赋值，大部分时候应该这样做：

```
class Widget {
public:
    Widget& operator+=(const Widget& rhs) {
    // 这个条款适用于  +=, -=, *= 等等运算符
        return *this;
    }
    Widget& operator=(int rhs) {
     // 即使参数类型不是 Widget& 也适用
        return *this;
    }
};
```

条款 11：在 operator= 中处理“自我赋值”

自我赋值是合法的操作，但在一些情况下可能会导致意外的错误，例如在复制堆上的资源时：

```
Widget& operator+=(const Widget& rhs) {
    delete pRes;                              // 删除当前持有的资源
    pRes = new Resource(*rhs.pRes);           // 复制传入的资源
    return *this;
}
```

但若 rhs 和*this 指向的是相同的对象，就会导致访问到已删除的数据。

最简单的解决方法是在执行后续语句前先进行**证同测试（Identity test）**：

```
Widget& operator=(const Widget& rhs) {
    if (this == &rhs) return *this;
    // 若是自我赋值，则不做任何事
```

```
    delete pRes;
    pRes = new Resource(*rhs.pRes);
    return *this;
}
```

另一个常见的做法是只关注异常安全性，而不关注是否自我赋值：

```
Widget& operator=(const Widget& rhs) {
    Resource* pOrigin = pRes;              // 先记住原来的 pRes 指针
    pRes = new Resource(*rhs.pRes);        // 复制传入的资源
    delete pOrigin;                        // 删除原来的资源
    return *this;
}
```

仅仅是适当安排语句的顺序，就可以做到使整个过程具有异常安全性。

还有一种取巧的做法是使用 copy and swap 技术，这种技术聪明地利用了栈空间会自动释放的特性，这样就可以通过析构函数来实现资源的释放：

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

上述做法还可以写得更加巧妙，就是利用按值传参，自动调用构造函数：

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

条款 12：复制对象时勿忘其每一个成分

这个条款正如其字面意思，当你决定手动实现拷贝构造函数或拷贝赋值运算符时，忘记复制任何一个成员都可能会导致意外的错误。

当使用继承时，继承自基类的成员往往容易忘记在派生类中完成复制，如果你的基类拥有拷贝构造函数和拷贝赋值运算符，应该记得调用它们：

```
class PriorityCustomer : public Customer {
public:
    PriorityCustomer(const PriorityCustomer& rhs);
    PriorityCustomer& operator=(const PriorityCustomer& rhs);
```

```
private:
    int priority;
}

PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
    : Customer(rhs),                    // 调用基类的拷贝构造函数
      priority(rhs.priority) {
}

PriorityCustomer::PriorityCustomer& operator=(const PriorityCustomer&
rhs) {
    Customer::operator=(rhs);           // 调用基类的拷贝赋值运算符
    priority = rhs.priority;
    return *this;
}
```

注意，不要尝试在拷贝构造函数中调用拷贝赋值运算符，或在拷贝赋值运算符的实现中调用拷贝构造函数，一个在初始化时，一个在初始化后，它们的功用是不同的。

第三章：资源管理

条款 13：以对象管理资源

对于传统的堆资源管理，我们需要使用成对的 new 和 delete，这样若忘记 delete 就会造成内存泄露。因此，我们应尽可能以对象管理资源，并采用 RAII（Resource Acquisition Is Initialize，资源取得时机便是初始化时机），让析构函数负责资源的释放。

在 C++11 中，通过专一所有权来管理 RAII 对象可以使用 std::unique_ptr，通过引用计数来管理 RAII 对象可以使用 std::shared_ptr。

```
// Investment* CreateInvestment();

std::unique_ptr<Investment> pUniqueInv1(CreateInvestment());
std::unique_ptr<Investment> pUniqueInv2(std::move(pUniqueInv1));
// 转移资源所有权

std::shared_ptr<Investment> pSharedInv1(CreateInvestment());
```

```
std::shared_ptr<Investment> pSharedInv2(pSharedInv1);
// 引用计数+1
```

智能指针默认会自动 delete 所持有的对象，我们也可以为智能指针指定所管理对象的释放方式（删除器 deleter）

条款 14：在资源管理类中小心拷贝行为

我们应该永远保持这样的思考：当一个 RAII 对象被复制，会发生什么事？

选择一：禁止复制

许多时候允许 RAII 对象被复制并不合理，如果确是如此，那么就该明确禁止复制行为，条款 6 已经阐述了怎么做这件事。

选择二：对底层资源祭出“引用计数法”

正如 std::shared_ptr 所做的那样，每一次复制对象就使引用计数+1，每一个对象离开定义域就调用析构函数使引用计数-1，直到引用计数为 0 就彻底销毁资源。

选择三：复制底层资源

在复制对象的同时复制底层资源的行为又被称作**深拷贝（Deep copying）**，例如在一个对象中有一个指针，那么在复制这个对象时就不能只复制指针，也要复制指针所指向的数据。

选择四：转移底层资源的所有权

和 std::unique_ptr 的行为类似，永远保持只有一个对象拥有对资源的管理权，当需要复制对象时转移资源的管理权。

条款 15：在资源管理类中提供对原始资源的访问

和所有的智能指针一样，STL 中的智能指针也提供了对原始资源的隐式访问和显式访问：

```
Investment* pRaw = pSharedInv.get();     // 显式访问原始资源
Investment raw = *pSharedInv;            // 隐式访问原始资源
```

当我们在设计自己的资源管理类时，也要考虑在提供对原始资源的访问时，是使用显式访问还是隐式访问的方法，还是两者皆可。

```
class Font {
```

```
public:
    FontHandle Get() const { return handle; }       // 显式转换函数
    operator FontHandle() const { return handle; }  // 隐式转换函数

private:
    FontHandle handle;
};
```

一般而言显式转换比较安全，但隐式转换对客户比较方便。

条款 16：成对使用 new 和 delete 时要采用相同形式

使用 new 来分配单一对象，使用 new[]来分配对象数组，必须明确它们的行为并不一致，分配对象数组时会额外在内存中记录“数组大小”，而使用 delete[]会根据记录的数组大小多次调用析构函数，使用 delete 则仅仅只会调用一次析构函数。对于单一对象使用 delete[]其结果也是未定义的，程序可能会读取若干内存并将其错误地解释为数组大小。

```
int* array = new int[10];
int* object = new int;

delete[] array;
delete object;
```

需要注意的是，使用 typedef 定义数组类型会带来额外的风险：

```
typedef std::string AddressLines[4];

std::string* pal = new AddressLines;
// pal 是一个对象数组，而非单一对象

delete pal;                                 // 行为未定义
delete[] pal;                               // 正确
```

条款 17：以独立语句将 newed 对象置入智能指针

原书此处所讲已过时，现在更好的做法是使用 std::make_unique 和 std::make_shared：

```
auto pUniqueInv = std::make_unique<Investment>();    // since C++14
auto pSharedInv = std::make_shared<Investment>();    // since C++11
```

std::make_shared 会分配一大块内存来同时持有 Widget 对象和控制块。这种优化减少了程序的静态尺寸，因为代码只需要调用一次内存分配函数，然后它增加了代码执行的速度，因为只需要分配一次内存（说明是分配内存这个函数开销略大）。而且，使用 std::make_shared 能避免了一些控制块的簿记信息，潜在地减少了程序占用的内存空间。

第四章：设计与声明

条款 18：让接口容易被正确使用，不易被误用

1. 好的接口很容易被正确使用，不易被误用。你应在在你的所有接口中努力达成这些性质。
2. “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。
3. “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。

```
// 三个参数类型相同的函数容易造成误用
Data::Data(int month, int day, int year) { ... }

// 通过适当定义新的类型加以限制，降低误用的可能性
Data::Data(const Month& m, const Day& d, const Year& y) { ... }
```

尽量使用智能指针，避免跨 DLL 的 new 和 delete，使用智能指针自定义删除器来解除互斥锁（mutexes）。

条款 19：设计 class 犹如设计 type

几乎在设计每一个 class 时，都要面对如下问题：

新 type 对象应该如何被创建和销毁？ 这会影响到类中构造函数、析构函数、内存分配和释放函数（operator new，operator new[]，operator delete，operator delete[]）的设计。

对象的初始化和赋值该有什么样的差别？ 这会影响到构造函数和拷贝赋值运算之间行为的差异。

新 type 的对象如果被按值传递，意味着什么？ 这会影响到拷贝构造函数的实现。

什么是新 type 的合法值？ 你的类中的成员函数必须对类中成员变量的值进行检查，如果不合法就要尽快解决或明确地抛出异常。

你的新 type 需要配合某个继承图系吗？ 你的类是否受到基类设计地束缚，是否拥有该覆写地虚函数，是否允许被继承（若不想要被继承，应该声明为 final）。

什么样的运算符和函数对此新 type 而言是合理的？ 这会影响到你将为你的类声明哪些函数和重载哪些运算符。

什么样的标准函数应该被驳回？ 这会影响到你将哪些标准函数声明为= delete。

谁该取用新 type 的成员？ 这会影响到你将类中哪些成员设为 public，private 或 protected，也将影响到友元类和友元函数的设置。

什么是新 type 的“未声明接口”？ 为未声明接口提供效率、异常安全性以及资源运用上的保证，并在实现代码中加上相应的约束条件。

你的新 type 有多么一般化？ 如果你想要一系列新 type 家族，应该优先考虑模板类。

条款 20：以按常引用传参替换按值传参

当使用按值传参时，程序会调用对象的拷贝构造函数构建一个在函数内作用的局部对象，这个过程的开销可能会较为昂贵。对于任何用户自定义类型，使用按常引用传参是较为推荐的：

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

因为没有任何新对象被创建，这种传参方式不会调用任何构造函数或析构函数，所以效率比按值传参高得多。

使用按引用传参也可以避免**对象切片（Object slicing）** 的问题，参考以下例子：

```
class Window {
public:
    ...
    std::string GetName() const;
    virtual void Display() const;
};
```

```
class WindowWithScrollBars : public Window {
public:
    virtual void Display() const override;
};
```

此处一个 WindowWithScrollBars 类继承自 Window 基类。

```
void PrintNameAndDisplay(Window w) {
// 按值传参，会发生对象切片
    std::cout << w.GetName();
    w.Display();
}
```

此处在传参时，调用了基类 Window 的拷贝构造函数而非派生类的拷贝构造函数，因此在函数种使用的是一个 Window 对象，调用虚函数时也只能调用到基类的虚函数 Window::Display。

由于按引用传递不会创建新对象，这个问题就能得到避免：

```
void PrintNameAndDisplay(const Window& w) { // 参数不会被切片
    std::cout << w.GetName();
    w.Display();
}
```

也并非永远都使用按引用传参，对于内置类型、STL 的迭代器和函数对象，我们认为使用按值传参是比较合适的。

条款 21：必须返回对象时，别妄想返回其引用

返回一个指向函数内部局部变量的引用是严重的错误，因为局部变量在离开函数时就被销毁了，除此之外，返回一个指向局部静态变量的引用也是不被推荐的。

尽管返回对象会调用拷贝构造函数产生开销，但这开销比起出错而言微不足道。

条款 22：将成员变量声明为 private

出于对封装性的考虑，应该尽可能地隐藏类中的成员变量，并通过对外暴露函数接口来实现对成员变量的访问：

```
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;
    int readOnly;
    int readWrite;
    int writeOnly;
};
```

通过为成员变量提供 getter 和 setter 函数，我们就能避免客户做出写入只读变量或读取只写变量这样不被允许的操作。

将成员变量隐藏在函数接口的背后，可以为“所有可能的实现”提供弹性。例如这可使得在成员变量被读或写时轻松通知其它对象，可以验证类的约束条件以及函数的提前和事后状态，可以在多线程环境中执行同步控制。

protected 和 public 一样，都不该被优先考虑。假设我们有一个 public 成员变量，最终取消了它，那么所有使用它的客户代码都将被破坏；假设我们有一个 protected 成员变量，最终取消了它，那么所有使用它的派生类都将被破坏。

在类中应当将成员变量优先声明为 private。

条款 23：宁以非成员、非友元函数替换成员函数

假设有这样一个类：

```
class WebBrowser {
public:
    void ClearCache();
    void ClearHistory();
    void RemoveCookies();
};
```

如果想要一次性调用这三个函数，那么需要额外提供一个新的函数：

```
void ClearEverything(WebBrowser& wb) {
    wb.ClearCache();
    wb.ClearHistory();
    wb.RemoveCookies();
}
```

注意，虽然成员函数和非成员函数都可以完成我们的目标，但此处更建议使用非成员函数，这是为了遵守一个原则：**越少的代码可以访问数据，数据的封装性就越强**。此处的 ClearEverything 函数仅仅是调用了 WebBrowser 的三个 public 成员函数，而并没有使用到 WebBrowser 内部的 private 成员，因此没有必要让其也拥有访问类中 private 成员的能力。

这个原则对于友元函数也是相同的，因为友元函数和成员函数拥有相同的权力，所以在能使用非成员函数完成任务的情况下，就不要使用友元函数和成员函数。

如果你觉得一个全局函数并不自然，也可以考虑将 ClearEverything 函数放在工具类中充当静态成员函数，或与 WebBrowser 放在同一个命名空间中：

```
namespace WebBrowserStuff {
    class WebBrowser { ... };
    void ClearEverything(WebBrowser& wb) { ... }
}
```

条款 24：若所有参数皆需类型转换，请为此采用非成员函数

现在我们手头上拥有一个 Rational 类，并且它可以和 int 隐式转换：

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

当然，我们需要重载乘法运算符来实现 Rational 对象之间的乘法：

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

将运算符重载放在类中是行得通的，至少对于 Rational 对象来说是如此。但当我们考虑混合运算时，就会出现一个问题：

```
Rational oneEight(1, 8);
Rational oneHalf(1, 2);
Rational result = oneHalf / oneEight;

result = oneHalf * 2;     // 正确
result = 2 * oneHalf;     // 报错
```

假如将乘法运算符写成函数形式，错误的原因就一目了然了：

```
result = oneHalf.operator*(2);     // 正确
result = 2.operator*(oneHalf);     // 报错
```

在调用 operator*时，int 类型的变量会隐式转换为 Rational 对象，因此用 Rational 对象乘以 int 对象是合法的，但反过来则不是如此。

所以，为了避免这个错误，我们应当将运算符重载放在类外，作为非成员函数：

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

条款 25：考虑写出一个不抛异常的 swap 函数

由于 std::swap 函数在 C++11 后改为了用 std::move 实现，因此几乎已经没有性能的缺陷，也不再有像原书中所说的为自定义类型去自己实现的必要。不过原书中透露的思想还是值得一学的。

如果想为自定义类型实现自己的 swap 方法，可以考虑使用模板全特化，并且这种做法是被 STL 允许的：

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

private:
    WidgetImpl* pImpl;
};

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

注意，由于外部函数并不能直接访问 Widget 的 private 成员变量，因此我们先是在类中定义了一个 public 成员函数，再由 std::swap 去调用这个成员函数。

然而若 Widget 和 WidgetImpl 是类模板，情况就没有这么简单了，因为 C++ 不支持函数模板偏特化，所以只能使用重载的方式：

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

但很抱歉，这种做法是被 STL 禁止的，因为这是在试图向 STL 中添加新的内容，所以我们只能退而求其次，在其它命名空间中定义新的 swap 函数：

```
namespace WidgetStuff {
    ...
    template<typename T>
    class Widget { ... };
    ...
    template<typename T>3
    void swap(Widget<T>& a, Widget<T>& b) {
        a.swap(b);
    }
}
```

我们希望在对自定义对象进行操作时找到正确的 swap 函数重载版本，这时候如果再写成 std::swap，就会强制使用 STL 中的 swap 函数，无法满足我们的需求，因此需要改写成：

```
using std::swap;
swap(obj1, obj2);
```

这样，C++ 名称查找法则能保证我们优先使用的是自定义的 swap 函数而非 STL 中的 swap 函数。

C++ 名称查找法则：编译器会从使用名字的地方开始向上查找，由内向外查找各级作用域（命名空间）直到全局作用域（命名空间），找到同名的声明即停止，若最终没找到则报错。

函数匹配优先级：普通函数 > 特化函数 > 模板函数

第五章：实现

条款 26：尽可能延后变量定义式出现的时间

当变量定义出现时，程序需要承受其构造成本；当变量离开其作用域时，程序需要承受其析构成本。因此，避免不必要的变量定义，以及延后变量定义式直到你确实需要它。

延后变量定义式还有一个意义，即“默认构造+赋值”效率低于“直接构造”：

```
// 效率低
std::string encrypted;
encrypted = password;

// 效率高
std::string encrypted(password);
```

对于循环中变量的定义，我们一般有两种做法：

A. 定义于循环外，在循环中赋值：

```
Widget w;
for (int i = 0; i < n; ++i) {
    w = 取决于 i 的某个值;
}
```

这种做法产生的开销：1 个构造函数 + 1 个析构函数 + n 个赋值操作

B. 定义于循环内：

```
for (int i = 0; i < n; ++i) {
    Widget w(取决于 i 的某个值);
}
```

这种做法产生的开销：n 个构造函数 + n 个析构函数

由于做法 A 会将变量的作用域扩大，因此除非知道该变量的赋值成本比“构造+析构”成本低，或者对这段程序的效率要求非常高，否则建议使用做法 B。

条款 27：少做转型动作

C 式转型：

```
(T)expression
T(expression)
```

C++ 式转型：

```
const_cast<T>(expression)
```

```
dynamic_cast<T>(expression)
reinterpret_cast<T>(expression)
static_cast<T>(expression)
```

- const_cast 用于常量性转除，这也是唯一一个有这个能力的 C++ 式转型。
- dynamic_cast 用于安全地向下转型，这也是唯一一个 C 式转型无法代替的转型操作，它会执行对继承体系的检查，因此会带来额外的开销。只有拥有虚函数的基类指针能进行 dynamic_cast。
- reinterpret_cast 用于在任意两个类型间进行低级转型，执行该转型可能会带来风险，也可能不具备移植性。
- static_cast 用于进行强制隐式转换，也是最常用的转型操作，可以将内置数据类型互相转换，也可以将 void*和 typed 指针，基类指针和派生类指针互相转换。

尽量在 C++ 程序中使用 C++ 式转型，因为 C++ 式转型操作功能更明确，可以避免不必要的错误。

唯一使用 C 式转型的时机可能是在调用 explicit 构造函数时：

```
class Widget {
public:
    explicit Widget(int size);
};
void DoSomeWork(const Widget& w);
DoSomeWork(Widget(15));
// 等价于 DoSomeWork(static_cast<Widget>(15));
```

需要注意的是，转型并非什么都没有做，而是可能会更改数据的底层表述，或者为指针附加偏移值，这和具体平台有关，因此不要妄图去揣测转型后对象的具体布局方式。

避免对*this 进行转型，参考以下例子：

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

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

这段代码试图通过转型*this 来调用基类的虚函数，然而这是严重错误的，这样做会得到一个新的 Window 副本并在该副本上调用函数，而非在原本的对象上调用函数。

正确的做法如下：

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

当你想知道一个基类指针是否指向一个派生类对象时，你需要用到 dynamic_cast，如果不满足，则会产生报错。但是对于继承体系的检查可能是非常慢的，所以在注重效率的程序中应当避免使用 dynamic_cast，改用 static_cast 或别的代替方法。

条款 28：避免返回 handles 指向对象的内部成分

考虑以下 Rectangle 类：

```
struct RectData {
    Point ulhc;
    Point lrhc;
};

class Rectangle {
public:
    Point& UpperLeft() const { return pData->ulhc; }
    Point& LowerRight() const { return pData->lrhc; }

private:
    std::shared_ptr<RectData> pData;
};
```

这段代码看起来没有任何问题，但其实是在做自我矛盾的事情：我们通过 const 成员函数返回了一个指向成员变量的引用，这使得成员变量可以在外部被修改，而这是违反 logical constness 的原则的。换句话说，你绝对不应该令成员函数返回一个指针指向“访问级别较低”的成员函数。

改成返回常引用可以避免对成员变量的修改：

```
const Point& UpperLeft() const { return pData->ulhc; }
```

```
const Point& LowerRight() const { return pData->lrhc; }
```

但是这样依然会带来一个称作 **dangling handles（空悬句柄）** 的问题，当对象不复存在时，你将无法通过引用获取到返回的数据。

采用最保守的做法，返回一个成员变量的副本：

```
Point UpperLeft() const { return pData->ulhc; }
Point LowerRight() const { return pData->lrhc; }
```

条款 29：为“异常安全”而努力是值得的

异常安全函数提供以下三个保证之一：

基本承诺： 如果异常被抛出，程序内的任何事物任然保持在有效状态下，没有任何对象或数据结构会因此败坏，所有对象都处于一种内部前后一致的状态，然而程序的真实状态是不可知的，也就是说客户需要额外检查程序处于哪种状态并作出对应的处理。

强烈保证： 如果异常被抛出，程序状态完全不改变，换句话说，程序会回复到“调用函数之前”的状态。

不抛掷（nothrow）保证： 承诺绝不抛出异常，因为程序总是能完成原先承诺的功能。作用于内置类型身上的所有操作都提供 nothrow 保证。

原书中实现 nothrow 的方法是 throw()，不过这套异常规范在 C++11 中已经被弃用，取而代之的是 noexcept 关键字：

```
int DoSomething() noexcept;
```

注意，使用 noexcept 并不代表函数绝对不会抛出异常，而是在抛出异常时，将代表出现严重错误，会有意想不到的函数被调用（可以通过 set_unexpected 设置），接着程序会直接崩溃。

当异常被抛出时，带有异常安全性的函数会：

1. 不泄漏任何资源。
2. 不允许数据败坏。

考虑以下 PrettyMenu 的 ChangeBackground 函数：

```
class PrettyMenu {
public:
    void ChangeBackground(std::vector<uint8_t>& imgSrc);
private:
```

```
    Mutex mutex;          // 互斥锁
    Image* bgImage;       // 目前的背景图像
    int imageChanges;     // 背景图像被改变的次数
};

void PrettyMenu::ChangeBackground(std::vector<uint8_t>& imgSrc) {
    lock(&mutex);
    delete bgImage;
    ++imageChanges;
    bgImage = new Image(imgSrc);
    unlock(&mutex);
}
```

很明显这个函数不满足我们所说的具有异常安全性的任何一个条件，若在函数中抛出异常，mutex 会发生资源泄漏，bgImage 和 imageChanges 也会发生数据败坏。

通过以对象管理资源，使用智能指针和调换代码顺序，我们能将其变成一个具有强烈保证的异常安全函数：

```
void PrettyMenu::ChangeBackground(std::vector<uint8_t>& imgSrc) {
    Lock ml(&mutex);
    bgImage.reset(std::make_shared<Image>(imgSrc));
    ++imageChanges;
}
```

另一个常用于提供强烈保证的方法是我们所提到过的 copy and swap，为你打算修改的对象做出一份副本，对副本执行修改，并在所有修改都成功执行后，用一个不会抛出异常的 swap 方法将原件和副本交换：

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

class PrettyMenu {
private:
    Mutex mutex;
    std::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::ChangeBackground(std::vector<uint8_t>& imgSrc) {
    Lock ml(&mutex);
    auto pNew = std::make_shared<PMImpl>(*pImpl);    // 获取副本
    pNew->bgImage.reset(std::make_shared<Image>(imgSrc));
```

```
    ++pNew->imageChanges;
    std::swap(pImpl, pNew);
}
```

当一个函数调用其它函数时，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。

强烈保证并非永远都是可实现的，特别是当函数在操控非局部对象时，这时就只能退而求其次选择不那么美好的基本承诺，并将该决定写入文档，让其他人维护时不至于毫无心理准备。

条款 30：透彻了解 inlining 的里里外外

将函数声明为内联一共有两种方法，一种是为其显式指定 inline 关键字，另一种是直接将成员函数的定义式写在类中，如下所示：

```
class Person {
public:
    int Age() const { return theAge; }  // 隐式声明为 inline
private:
    int theAge;
};
```

在 inline 诞生之初，它被当作是一种对编译器的优化建议，即将“对此函数的每一个调用”都以函数本体替换之。但在编译器的具体实现中，该行为完全被优化等级所控制，与函数是否内联无关。

在现在的 C++ 标准中，inline 作为优化建议的含义已经被完全抛弃，取而代之的是“允许函数在不同编译单元中多重定义”，使得可以在头文件中直接给出函数的实现。

在 C++17 中，引入了一个新的 inline 用法，使静态成员变量可以在类中直接定义：

```
class Person {
private:
    static inline int theAge = 0;  // since C++17
};
```

条款 31：将文件间的编译依存关系降至最低

C++ 坚持将类的实现细节放置于类的定义式中，这就意味着，即使你只改变类的实现而不改变类的接口，在构建程序时依然需要重新编译。这个问题的根源

出在编译器必须在编译期间知道对象的大小，如果看不到类的定义式，就没有办法为对象分配内存。也就是说，C++ 并没有把“将接口从实现中分离”这件事做得很好。

用“声明的依存性”替换“定义的依存性”：

我们可以玩一个“将对象实现细目隐藏于一个指针背后”的游戏，称作 **pimpl idiom（pimpl 是 pointer to implemention 的缩写）**：将原来的一个类分割为两个类，一个只提供接口，另一个负责实现该接口，称作**句柄类（handle class）**：

```
// person.hpp 负责声明类

class PersonImpl;

class Person {
public:
    Person();
    void Print();
    ...
private:
    std::shared_ptr<PersonImpl>;
};

// person.cpp 负责实现类

class PersonImpl {
public:
    int data{ 0 };
};

Person::Person() {
    pImpl = std::make_shared<PersonImpl>();
}

void Person::Print() {
    std::cout << pImpl->data;
}
```

这样，假如我们要修改 Person 的 private 成员，就只需要修改 PersonImpl 中的内容，而 PersonImpl 的具体实现是被隐藏起来的，对它的任何修改都不会使得 Person 客户端重新编译，真正实现了“类的接口和实现分离”。

如果使用对象引用或对象指针可以完成任务，就不要使用对象本身：

你可以只靠一个类型声明式就定义出指向该类型的引用和指针；但如果定义某类型的对象，就需要用到该类型的定义式。

如果能够，尽量以类声明式替换类定义式：

当你在声明一个函数而它用到某个类时，你不需要该类的定义；但当你触及到该函数的定义式后，就必须也知道类的定义：

```
class Date;                          // 类的声明式
Date Today();
void ClearAppointments(Data d); // 此处并不需要得知类的定义
```

为声明式和定义式提供不同的头文件：

为了避免频繁地添加声明，我们应该为所有要用的类声明提供一个头文件，这种做法对 template 也适用：

```
#include "datefwd.h"              // 这个头文件内声明 class Date
Date Today();
void ClearAppointments(Data d);
```

此处的头文件命名方式"datefwd.h"取自标准库中的<iosfwd>。

上面我们讲述了接口与实现分离的其中一个方法——提供句柄类，另一个方法就是将句柄类定义为抽象基类，称作**接口类（interface class）**：

```
class Person {
public:
    virtual ~Person() {}
    virtual void Print();
    ...
};
```

为了将 Person 对象实际创建出来，我们一般采用工厂模式。可以尝试在类中塞入一个静态成员函数 Create 用于创建对象：

```
class Person {
public:
    static std::shared_ptr<Person> Create();
};
```

但此时 Create 函数还无法使用，需要在派生类中给出 Person 类中的函数的具体实现：

```
class RealPerson : public Person {
```

```
public:
    RealPerson(...) { ... }
    virtual ~RealPerson() {}
    void Print() override { ... }
private:
    int data{ 0 };
};
```

完成 Create 函数的定义：

```
static std::shared_ptr<Person> Person::Create() {
    return std::make_shared<RealPerson>();
}
```

毫无疑问的是，句柄类和接口类都需要额外的开销：句柄类需要通过 pimpl 取得对象数据，增加一层间接访问、指针大小和动态分配内存带来的开销；而接口类会增加存储虚表指针和实现虚函数跳转带来的开销。

而当这些开销过于重大以至于类之间的耦合度在相形之下不成为关键时，就以具象类（concrete class）替换句柄类和接口类。

第六章：继承与面向对象设计

条款 32：确定你的 public 继承塑模出 is-a 关系

“public 继承”意味着 is-a，所谓 is-a，就是指适用于基类身上的每一件事情一定也适用于继承类身上，因为我们可以认为每一个派生类对象也都是一个基类对象。

考虑 Bird 类和 Penguin 类的继承关系：

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

class Penguin : public Bird {
    ...
};
```

Penguin 类会获得来自 Bird 类的飞行方法，这就造成了误解，因为企鹅恰恰是不会飞的鸟类。一种解决方法是当调用 Penguin 类中的 Fly 函数时，抛出一个运行期错误，但这种做法通常不够直观；另一个解决方法是使用双继承，区分会飞和不会飞的鸟类：

但若要处理鸟类的多钟不同属性时，双继承模式就不太管用了，因此我们总是说程序设计没有银弹。

另一个常见的例子是用 Square 类继承自 Rectangle 类，从几何学的角度来讲这很自然，然而正方形的长宽是相等的，矩形却不是如此，因此 Square 类和 Rectangle 类也无法满足严格的 is-a 关系。

条款 33：避免遮掩继承而来的名称

之前我们了解过 C++ 名称查找法则，这在继承体系中也是类似的，当我们在派生类中使用到一个名字时，编译器会优先查找派生类覆盖的作用域，如果没找到，再去查找基类的作用域，最后再查找全局作用域。

考虑以下情形：

```
class Base {
public:
    void mf();
    void mf(double);
};

class Derived : public Base {
public:
    void mf();
};
```

这样会导致派生类无法使用来自基类的重载函数，因为派生类中的名称 mf 掩盖了来自基类的名称 mf。

对于名称掩盖问题的一种方法是使用 using 关键字：

```
class Derived : public Base {
public:
    using Base::mf;
};
```

using 关键字会将基类中所有使用到名称 mf 的函数全部包含在派生类中，包括其重载版本。

若有时我们不想要一个函数的全部版本，只想要单一版本（特别是在 private 继承时），可以考虑使用**转发函数（forwarding function）**：

```
class Base {
public:
    virtual void mf();
    virtual void mf(double);
};

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

条款 34：区分接口继承和实现继承

1. 接口继承和实现继承不一样。在 public 继承下，派生类总是继承基类的接口。
2. 声明一个纯虚函数的目的，是为了让派生类只继承函数接口。
3. 声明简朴的非纯虚函数的目的，是让派生类继承该函数的接口和缺省实现。
4. 声明非虚函数的目的，是为了令派生类继承函数的接口及一份强制性实现。

用非纯虚函数提供缺省的默认实现：

```
class Airplane {
public:
    virtual void Fly() {
        // 缺省实现
    }
};

class Model : public Airplane { ... };
```

这是最简朴的做法，但是这样做会带来的问题是，由于不强制对虚函数的覆写，在定义新的派生类时可能会忘记进行覆写，导致错误地使用了缺省实现。

使用纯虚函数并提供默认实现：

```
class Airplane {
public:
```

```
    virtual void Fly() = 0;
protected:
    void DefaultFly() {
        // 缺省实现
    }
};

class Model : public Airplane {
public:
    virtual void Fly() override {
        DefaultFly();
    }
};
```

上述写法可以替代为：

```
class Airplane {
public:
    virtual void Fly() = 0;
};

void Airplane::Fly() {
        // 缺省实现
}

class Model : public Airplane {
public:
    virtual void Fly() override {
        Airplane::Fly();
    }
};
```

条款 35：考虑虚函数以外的其它选择

由非虚接口手法实现 template method：

非虚接口（non-virtual interface，NVI） 设计手法的核心就是用一个非虚函数作为 wrapper，将虚函数隐藏在封装之下：

```
class GameCharacter {
public:
    int HealthValue() const {
        ...    // 做一些前置工作
        int retVal = DoHealthValue();
```

```
        ...    // 做一些后置工作
        return retVal;
    }
private:
    virtual int DoHealthValue() const {
        ...    // 缺省算法
    }
};
```

NVI 手法的一个优点就是在 wrapper 中做一些前置和后置工作，确保得以在一个虚函数被调用之前设定好适当场景，并在调用结束之后清理场景。如果你让客户直接调用虚函数，就没有任何好办法可以做这些事。

NVI 手法允许派生类重新定义虚函数，从而赋予它们“如何实现机能”的控制能力，但基类保留诉说“函数何时被调用”的权利。

由函数指针实现 Strategy 模式

```
using HealthCalcFunc = int(*)(const GameCharacter&);
// 定义函数指针类型
    explicit GameCharacter(HealthCalcFunc hcf = DefaultHealthCalc)
        : healthFunc(hcf) {}
```

由 std::function 完成 Strategy 模式

```
using HealthCalcFunc = std::function<int(const GameCharacter&)>;
// 定义函数包装器类型
    explicit GameCharacter(HealthCalcFunc hcf = DefaultHealthCalc)
        : healthFunc(hcf) {}
```

条款 36：绝不重新定义继承而来的非虚函数

非虚函数和虚函数具有本质上的不同：非虚函数执行的是静态绑定，由对象类型本身（称之静态类型）决定要调用的函数；而虚函数执行的是动态绑定，决定因素不在对象本身，而在于“指向该对象之指针”当初的声明类型（称之动态类型）。

前面已经说过，public 继承意味着 is-a 关系，而在基类中声明一个非虚函数将会为该类建立起一种不变性，凌驾其特异性。而若在派生类中重新定义该非虚函数，则会使人开始质疑是否该使用 public 继承的形式；如果必须使用，则又打破了基类“不变性凌驾特异性”的性质，就此产生了设计上的矛盾。

综上所述，在任何情况下都不该重新定义一个继承而来的非虚函数。

条款 37：绝不重新定义继承而来的缺省参数值

在条款 36 中我们已经否定了重新定义非虚函数的可能性，因此此处我们只讨论带有缺省参数值的虚函数。

虚函数是动态绑定而来，意思是调用一个虚函数时，究竟调用哪一份函数实现代码，取决于发出调用的那个对象的动态类型。但与之不同的是，**缺省参数值却是静态绑定**，意思是你可能会在“调用一个定义于派生类的虚函数”的同时，却使用基类为它所指定的缺省参数值。考虑以下例子：

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

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

class Circle : public Shape {
public:
    virtual void Draw(ShapeColor color) const;
};
```

此时若对派生类对象调用 Draw 函数，则会发现：

```
Shape* pr = new Rectangle;
Shape* pc = new Circle;

pr->Draw(Shape::ShapeColor::Green);
// 调用 Rectangle::Draw(Shape::Green)
pr->Draw();
// 调用 Rectangle::Draw(Shape::Red)
pc->Draw();
// 调用 Rectangle::Draw(Shape::Red)
```

条款 38：通过复合塑模出 has-a 或“根据某物实现出”

所谓**复合**，指的是某种类型的对象内含它种类型的对象。复合通常意味着 **has-a** 或**根据某物实现出**的关系，当复合发生于应用域内的对象之间，表现出 has-a 的关系；当它发生于实现域内则是表现出“根据某物实现出”的关系。

下面是一个 has-a 关系的例子：

```
class Address { ... };
class PhoneNumber { ... };

class Person {
public:
    ...
private:
    std::string name;           // 合成成分物（composed object）
    Address address;            // 同上
    PhoneNumber voiceNumber;    // 同上
    PhoneNumber faxNumber;      // 同上
};
```

下面是一个“根据某物实现出”关系的例子：

```
// 将 list 应用于 Set
template<class T>
class Set {
public:
    bool member(const T& item) const;
    void insert(const T& item);
    void remove(const T& item);
    std::size_t size() const;

private:
    std::list<T> rep;           // 用来表述 Set 的数据
};
```

条款 39：明智而审慎地使用 private 继承

private 继承的特点：

1. 如果类之间是 private 继承关系，那么编译器不会自动将一个派生类对象转换为一个基类对象。
2. 由 private 继承来的所有成员，在派生类中都会变为 private 属性，换句话说，private 继承只继承实现，不继承接口。

private 继承的意义是“根据某物实现出”，如果你读过条款 38，就会发现 private 继承和复合具有相同的意义，事实上也确实如此，绝大部分 private 继承的使用场合都可以被“public 继承+复合”完美解决：

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

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

替代为：

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

使用后者比前者好的原因有以下几点：

1. private 继承无法阻止派生类重新定义虚函数，但若使用 public 继承定义 WidgetTimer 类并复合在 Widget 类中，就能防止在 Widget 类中重新定义虚函数。
2. 可以仅提供 WidgetTimer 类的声明，并将 WidgetTimer 类的具体定义移至实现文件中，从而降低 Widget 的编译依存性。

条款 40：明智而审慎地使用多重继承

多重继承是一个可能会造成很多歧义和误解的设计，因此反对它的声音此起彼伏，下面我们来接触几个使用多重继承的场景。

最先需要认清的一件事是，程序有可能从一个以上的基类继承相同名称，那会导致较多的歧义机会：

```
class BorrowableItem {
public:
```

```
    void CheckOut();
};

class ElectronicGadget {
public:
    void CheckOut() const;
};

class MP3Player : public BorrowableItem, public ElectronicGadget {
};

MP3Player mp;
mp.CheckOut();          // MP3Player::CheckOut 不明确!
```

如果真遇到这种情况，必须明确地指出要调用哪一个基类中的函数：

```
mp.BorrowableItem::CheckOut();      // 使用 BorrowableItem::CheckOut
```

在使用多重继承时，我们可能会遇到要命的“钻石型继承（菱形继承）”：

```
class File { ... };

class InputFile : public File { ... };
class OutputFile : public File { ... };

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

这时候必须面对这样一个问题：是否打算让基类内的成员变量经由每一条路径被复制？如果不想要这样，应当使用虚继承，指出其愿意共享基类：

```
class File { ... };

class InputFile : virtual public File { ... };
class OutputFile : virtual public File { ... };

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

然而由于虚继承会在派生类中额外存储信息来确认成员来自于哪个基类，虚继承通常会付出更多空间和速度的代价，并且由于虚基类的初始化责任是由继承体系中最底层的派生类负责，就导致了虚基类必须认知其虚基类并且承担虚基类的初始化责任。因此我们应当遵循以下两个建议：

1. 非必要不使用虚继承。
2. 如果必须使用虚继承，尽可能避免在虚基类中放置数据。

多重继承可用于结合 public 继承和 private 继承，public 继承用于提供接口，private 继承用于提供实现：

第七章：模板与泛型编程

条款 41：了解隐式接口和编译期多态

类与模板都支持接口和多态。对于类而言接口是显式的，以函数签名为中心，多态则是通过虚函数发生于运行期；而对模板参数而言，接口是隐式的，奠基于有效表达式，多态则是通过模板具现化和函数重载解析发生于编译期。

考虑以下例子：

```
template<typename T>
void DoProcessing(T& w) {
    if (w.size() > 10 && w != someNastyWidget) {
    ...
```

以上代码中，T 类型的隐式接口要求：

1. 提供一个名为 size 的成员函数，该函数的返回值可与 int（10 的类型）执行 operator>，或经过隐式转换后可执行 operator>。
2. 必须支持一个 operator!=函数，接受 T 类型和 someNastyWidget 的类型，或其隐式转换后得到的类型。

此处没有考虑 operator&&被重载的可能性。

加诸于模板参数身上的隐式接口，就像加诸于类对象身上的显式接口“一样真实”，两者都在编译期完成检查，你无法在模板中使用“不支持模板所要求之隐式接口”的对象（代码无法通过编译）。

条款 42：了解 typename 的双重含义

在模板声明式中，使用 class 和 typename 关键字并没有什么不同，但在模板内部，typename 拥有更多的一重含义。

为了方便解释，我们首先需要引入一个模板相关的概念：模板内出现的名称如果相依于某个模板参数，我们称之为**从属名称**；如果从属名称在类内呈嵌套状，我们称之为**嵌套从属名称**；如果一个名称并不倚赖任何模板参数的名称，我们称之为**非从属名称**。

这段代码看起来没有任何问题，但实际编译时却会报错，这一切的罪魁祸首便是 C::const_iterator。此处的 C::const_iterator 是一个指向某类型的**嵌套从属类型名称**，而嵌套从属名称可能会导致解析困难，因为在编译器知道 C 是什么之前，没有任何办法知道 C::const_iterator 是否为一个类型，这就导致出现了歧义状态，而 C++ 默认假设嵌套从属名称不是类型名称。

显式指明嵌套从属类型名称的方法就是将 typename 关键字作为其前缀词：

```
typename C::const_iterator iter(container.begin());
```

同样地，若嵌套从属名称出现在模板函数声明部分，也需要显式地指明是否为类型名称：

```
template<typename C>
void Print2nd(const C& container, const typename C::iterator iter);
```

这一规则的例外是，typename 不可以出现在基类列表内的嵌套从属类型名称之前，也不可以在成员初始化列表中作为基类的修饰符：

```
template<typename T>
class Derived : public Base<T>::Nested {
// 基类列表中不允许使用 typename
public:
    explicit Derived(int x)
        : Base<T>::Nested(x) {
// 成员初始化列表中不允许使用 typename
        typename Base<T>::Nested temp;
    }
};
```

在类型名称过于复杂时，可以使用 using 或 typedef 来进行简化：

```
using value_type = typename std::iterator_traits<IterT>::value_type;
```

条款 43：学习处理模板化基类内的名称

在模板编程中，模板类的继承并不像普通类那么自然，考虑以下情形：

```
class MsgInfo { ... };

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

```
};

template<typename Company>
class LoggingMsgSender : public MsgSender<Company> {
public:
    void SendClearMsg(const MsgInfo& info) {
        SendClear(info);        // 调用基类函数，这段代码无法通过编译
    }
};
```

很明显，由于直到模板类被真正实例化之前，编译器并不知道 MsgSender<Company>具体长什么样，有可能它是一个全特化的版本，而在这个版本中不存在 SendClear 函数。由于 C++ 的设计策略是宁愿较早进行诊断，所以编译器会拒绝承认在基类中存在一个 SendClear 函数。

为了解决这个问题，我们需要令 C++“进入模板基类观察”的行为生效，有三种办法达成这个目标：

为了解决这个问题，我们需要令 C++“进入模板基类观察”的行为生效，有三种办法达成这个目标：

第一种：**在基类函数调用动作之前加上 this->**：

```
this->SendClear(info);
```

第二种：**使用 using 声明式**：

```
using MsgSender<Company>::SendClear;
SendClear(info);
```

第三种：**指出被调用的函数位于基类内**：

```
MsgSender<Company>::SendClear(info);
```

第三种做法是最不令人满意的，如果被调用的是虚函数，上述的明确资格修饰（explicit qualification）会使“虚函数绑定行为”失效。

条款 44：将与参数无关的代码抽离模板

模板可以节省时间和避免代码重复，编译器会为填入的每个不同模板参数具现化出一份对应的代码，但长此以外，可能会造成代码膨胀（code bloat），生成浮夸的二进制目标码。

基于**共性和变性分析（commonality and variability analysis）** 的方法，我们需要分析模板中重复使用的部分，将其抽离出模板，以减轻模板具现化带来的代码量。

- 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类成员变量替换模板参数。
- 因类型模板参数而造成的代码膨胀，往往可以降低，做法是让带有完全相同二进制表述的具现类型共享实现代码。

条款 45：运用成员函数模板接受所有兼容类型

C++ 视模板类的不同具现体为完全不同的的类型，但在泛型编程中，我们可能需要一个模板类的不同具现体能够相互类型转换。

考虑设计一个智能指针类，而智能指针需要支持不同类型指针之间的隐式转换（如果可以的话），以及普通指针到智能指针的显式转换。很显然，我们需要的是模板拷贝构造函数：

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

    template<typename U>
    explicit SmartPtr(U* p)
        : heldPtr(p) { ... }

    T* get() const { return heldPtr; }
private:
    T* heldPtr;
};
```

使用 get 获取原始指针，并将在原始指针之间进行类型转换本身提供了一种保障，如果原始指针之间不能隐式转换，那么其对应的智能指针之间的隐式转换会造成编译错误。

模板构造函数并不会阻止编译器暗自生成默认的构造函数，所以如果你想要控制拷贝构造的方方面面，你必须同时声明泛化拷贝构造函数和普通拷贝构造函数，相同规则也适用于赋值运算符：

```
template<typename T>
class shared_ptr {
```

```
public:
    shared_ptr(shared_ptr const& r);                // 拷贝构造函数

    template<typename Y>
    shared_ptr(shared_ptr<Y> const& r);             // 泛化拷贝构造函
数

    shared_ptr& operator=(shared_ptr const& r);     // 拷贝赋值运算符

    template<typename Y>
    shared_ptr& operator=(shared_ptr<Y> const& r);  // 泛化拷贝赋值运
算符
};
```

条款 46：需要类型转换时请为模板定义非成员函数

模板实参在推导过程中，从不将隐式类型转换纳入考虑。当我们编写一个 class template，而它所提供的“与此 template 相关的”函数支持“所有参数的隐式类型转换”时，请将那些函数定义为“class template 内部的 friend 函数”。

条款 47：请使用 traits classes 表现类型信息

traits classes 可以使我们在编译期就能获取某些类型信息，它被广泛运用于 C++ 标准库中。traits 并不是 C++ 关键字或一个预先定义好的构件：它们是一种技术，也是 C++ 程序员所共同遵守的协议，并要求对用户自定义类型和内置类型表现得一样好。

设计并实现一个 trait class 的步骤如下：

1. 确认若干你希望将来可取得的类型相关信息。
2. 为该类型选择一个名称。
3. 提供一个模板和一组特化版本，内含你希望支持的类型相关信息。

以迭代器为例，标准库中拥有多种不同的迭代器种类，它们各自拥有不同的功用和限制：

1. input_iterator_tag：单向输入迭代器，只能向前移动，一次一步，客户只可读取它所指的东西。
2. output_iterator_tag：单向输出迭代器，只能向前移动，一次一步，客户只可写入它所指的东西。
3. forward_iterator_tag：单向访问迭代器，只能向前移动，一次一步，读写均允许。

4. bidirectional_iterator_tag：双向访问迭代器，去除了只能向前移动的限制。
5. random_access_iterator_tag：随机访问迭代器，没有一次一步的限制，允许随意移动，可以执行“迭代器算术”。

trait 并不是 C++关键字或一个预先定义好的构件，它们是一种技术，也是一个 C++程序员共同遵守的协议。这个技术的要求之一是，它对内置类型和用户自定义类型的表现必须一样好。

在 C++17 之前，利用函数重载（也是原书中介绍的做法）：

```
template<typename IterT, typename DisT>
void doAdvance(IterT& iter, DisT d, std::random_access_iterator_tag)
{}

template<typename IterT, typename DisT>
void doAdvance(IterT& iter, DisT d, std::bidirectional_iterator_tag)
{}

template<typename IterT, typename DisT>
void doAdvance(IterT& iter, DisT d, std::input_iterator_tag) {
    if (d < 0) {
        throw std::out_of_range("Negative distance");
// 单向迭代器不允许负距离
    }
}

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

在 C++17 之后，我们有了更简单有效的做法——使用 if constexpr：

```
template<typename IterT, typename DisT>
void Advance(IterT& iter, DisT d) {
    if constexpr
(typeid(std::iterator_traits<IterT>::iterator_category)
        == typeid(std::random_access_iterator_tag)) {
        ...
    }
```

```
}
```

可以把 if constexpr 理解为编译时 if，只有被选中的 if constexpr 分支才会被实例化。

总结如何使用一个 traits class

建立一组重载函数，彼此间差异只在于各自的 traits 参数，另实现与接受的 traits 相符合。建立一个控制函数，调用以上的重载函数并传递 traits class 所提供的信息。这样可以在编译器对类型执行 if else 测试。

条款 48：认识模板元编程

模板元编程（Template metaprogramming，TMP）是编写基于模板的 C++ 程序并执行于编译期的过程，它并不是刻意被设计出来的，而是当初 C++ 引入模板带来的副产品，事实证明模板元编程具有强大的作用，并且现在已经成为 C++ 标准的一部分。实际上，在条款 47 中编写 traits classes 时，我们就已经在进行模板元编程了。

由于模板元程序执行于 C++ 编译期，因此可以将一些工作从运行期转移至编译期，这可以帮助我们在编译期时发现一些原本要在运行期时才能察觉的错误，以及得到较小的可执行文件、较短的运行期、较少的内存需求。当然，副作用就是会使编译时间变长。

模板元编程已被证明是“图灵完备”的，并且以“函数式语言”的形式发挥作用，因此在模板元编程中没有真正意义上的循环，所有循环效果只能藉由递归实现，而递归在模板元编程中是由 **“递归模板具现化”** 实现的。

常用于引入模板元编程的例子是在编译期计算阶乘：

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

template<>
struct Factorial<0> {           // 处理特殊情况：Factorial<0> = 1
    enum { value = 1 };
};

std::cout << Factorial<5>::value;
```

模板元编程很酷，但对其进行调试可能是灾难性的，因此在实际应用中并不常见。我们可能会在下面几种情形中见到它的出场：

1. 确保量度单位正确。
2. 优化矩阵计算。
3. 可以生成客户定制之设计模式（custom design pattern）实现品。

第八章：定制 new 和 delete

条款 49：了解 new-handler 的行为

当 operator new 无法满足某一内存分配需求时，会不断调用一个客户指定的错误处理函数，即所谓的 **new-handler**，直到找到足够内存为止，调用声明于<new>中的 set_new_handler 可以指定这个函数。new_handler 和 set_new_handler 的定义如下：

```
namespace std {
    using new_handler = void(*)();
    new_handler set_new_handler(new_handler) noexcept;    // 返回值为原来持有的 new-handler
}
```

一个设计良好的 new-handler 函数必须做以下事情之一：

让更多的内存可被使用： 可以让程序一开始执行就分配一大块内存，而后当 new-handler 第一次被调用，将它们释还给程序使用，造成 operator new 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 无法取得更多内存，可以调换为另一个可以完成目标的 new-handler（令 new-handler 修改“会影响 new-handler 行为”的静态或全局数据）。

卸除 new-handler： 将 nullptr 传给 set_new_handler，这样会使 operator new 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 捕捉，因此会被传播到内存分配处。

不返回： 通常调用 std::abort 或 std::exit。

有的时候我们或许会希望在为不同的类分配对象时，使用不同的方式处理内存分配失败情况。这时候使用静态成员是不错的选择：

```
public:
    static std::new_handler set_new_handler(std::new_handler p) noexcept;
```

```
    static void* operator new(std::size_t size);
private:
    static std::new_handler currentHandler;
};

// 做和 std::set_new_handler 相同的事情
std::new_handler Widget::set_new_handler(std::new_handler p) noexcept
{//noexcept 不抛出异常
    std::new_handler oldHandler = currentHandler;
    currentHandler = p;
    return oldHandler;
}

void* Widget::operator new(std::size_t size) {
    auto globalHandler = std::set_new_handler(currentHandler);
    // 切换至 Widget 的专属 new-handler
    void* ptr = ::operator new(size);
    // 分配内存或抛出异常
    std::set_new_handler(globalHandler);
    // 切换回全局的 new-handler
    return globalHandler;
}

std::new_handler Widget::currentHandler = nullptr;
```

Widget 的客户应该类似这样使用其 new-handling：

```
void OutOfMem();

Widget::set_new_handler(OutOfMem);
auto pw1 = new Widget;              // 若分配失败，则调用 OutOfMem

Widget::set_new_handler(nullptr);
auto pw2 = new Widget;              // 若分配失败，则抛出异常
```

实现这一方案的代码并不因类的不同而不同，因此对这些代码加以复用是合理的构想。一个简单的做法是建立起一个“mixin”风格的基类，让其派生类继承它们所需的 set_new_handler 和 operator new，并且使用模板确保每一个派生类获得一个实体互异的 currentHandler 成员变量：

```
template<typename T>
class NewHandlerSupport {       // “mixin”风格的基类
public:
    static std::new_handler set_new_handler(std::new_handler p)
noexcept;
```

```
    static void* operator new(std::size_t size);
    // 其它的 operator new 版本，见条款 52
private:
    static std::new_handler currentHandler;
};

template<typename T>
std::new_handler
NewHandlerSupport<T>::set_new_handler(std::new_handler p) noexcept {
    std::new_handler oldHandler = currentHandler;
    currentHandler = p;
    return oldHandler;
}

template<typename T>
void* NewHandlerSupport<T>::operator new(std::size_t size) {
    auto globalHandler = std::set_new_handler(currentHandler);
    void* ptr = ::operator new(size);
    std::set_new_handler(globalHandler);
    return globalHandler;
}

template<typename T>
std::new_handler NewHandlerSupport<T>::currentHandler = nullptr;

class Widget : public NewHandlerSupport<Widget> {
public:
    // 不必再声明 set_new_handler 和 operator new
};
```

注意此处的模板参数 T 并没有真正被当成类型使用，而仅仅是用来区分不同的派生类，使得模板机制为每个派生类具现化出一份对应的 currentHandler。

这个做法用到了所谓的 **CRTP（curious recurring template pattern，奇异递归模板模式）** ，除了在上述设计模式中用到之外，它也被用于实现**静态多态**：

```
template <class Derived>
struct Base {
    void Interface() {
        static_cast<Derived*>(this)->Implementation();
 // 在基类中暴露接口
    }
};
```

```
struct Derived : Base<Derived> {
    void Implementation();
     // 在派生类中提供实现
};
```

条款 50：了解 new 和 delete 的合理替换时机

以下是常见的替换默认 operator new 和 operator delete 的理由：

用来检测运用上的错误： 如果将“new 所得内存”delete 掉却不幸失败，会导致内存泄漏；如果在“new 所得内存”身上多次 delete 则会导致未定义行为。如果令 operator new 持有一串动态分配所得地址，而 operator delete 将地址从中移除，就很容易检测出上述错误用法。此外各式各样的编程错误可能导致 **“overruns”（写入点在分配区块尾端之后）** 和 **“underruns”（写入点在分配区块起点之前）**，以额外空间放置特定的 byte pattern 签名，检查签名是否原封不动就可以检测此类错误，下面给出了一个这样的范例：

```
static const int signature = 0xDEADBEEF;
// 调试“魔数”
using Byte = unsigned char;

void* operator new(std::size_t size) {
    using namespace std;
    size_t realSize = size + 2 * sizeof(int);
    // 分配额外空间以塞入两个签名

    void* pMem = malloc(realSize);
    // 调用 malloc 取得内存
    if (!pMem) throw bad_alloc();

    // 将签名写入内存的起点和尾端
    *(static_cast<int*>(pMem)) = signature;
    *(reinterpret_cast<int*>(static_cast<Byte*>(pMem) + realSize -
sizeof(int))) = signature;

    return static_cast<Byte*>(pMem) + sizeof(int);
    // 返回指针指向第一个签名后的内存位置
}
```

实际上这段代码不能保证内存对齐，并且有许多地方不遵守 C++ 规范，我们将在条款 51 中进行详细讨论。

为了收集使用上的统计数据： 定制 new 和 delete 动态内存的相关信息：分配区块的大小分布，寿命分布，FIFO（先进先出）、LIFO（后进先出）或随机次序的倾向性，不同的分配/归还形态，使用的最大动态分配量等等。

为了增加分配和归还的速度： 泛用型分配器往往（虽然并非总是）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之对象设计时。类专属的分配器可以做到“区块尺寸固定”，例如 Boost 提供的 Pool 程序库。又例如，编译器所带的内存管理器是线程安全的，但如果你的程序是单线程的，你也可以考虑写一个不线程安全的分配器来提高速度。当然，这需要你对程序进行分析，并确认程序瓶颈的确发生在那些内存函数身上。

为了降低缺省内存管理器带来的空间额外开销： 泛用型分配器往往（虽然并非总是）还比定制型分配器使用更多内存，那是因为它们常常在每一个分配区块身上招引某些额外开销。针对小型对象而开发的分配器（例如 Boost 的 Pool 程序库）本质上消除了这样的额外开销。

为了弥补缺省分配器中的非最佳内存对齐（suboptimal alignment）： 许多计算机体系架构要求特定的类型必须放在特定的内存地址上，如果没有奉行这个约束条件，可能导致运行期硬件异常，或者访问速度变低。
std::max_align_t 用来返回当前平台的最大默认内存对齐类型，对于 malloc 分配的内存，其对齐和 max_align_t 类型的对齐大小应当是一致的，但若对 malloc 返回的指针进行偏移，就没有办法保证内存对齐。

在 C++11 中，提供了以下内存对齐相关方法：

```
// alignas 用于指定栈上数据的内存对齐要求
struct alignas(8) testStruct { double data; };

// alignof 和 std::alignment_of 用于得到给定类型的内存对齐要求
std::cout << alignof(std::max_align_t) << std::endl;
std::cout << std::alignment_of<std::max_align_t>::value << std::endl;

// std::align 用于在一大块内存中获取一个符合指定内存要求的地址
char buffer[] = "memory alignment";
void* ptr = buffer;
std::size_t space = sizeof(buffer) - 1;
std::align(alignof(int), sizeof(char), ptr, space);
```

在 C++17 后，可以使用 std::align_val_t 来重载需求额外内存对齐的 operator new：

```
void* operator new(std::size_t count, std::align_val_t al);
```

为了将相关对象成簇集中： 如果你知道特定的某个数据结构往往被一起使用，而你又希望在处理这些数据时将“内存页错误（page faults）”的频率降至最

低，那么可以考虑为此数据结构创建一个堆，将它们成簇集中在尽可能少的内存页上。一般可以使用 placement new 达成这个目标（见条款 52）。

为了获得非传统的行为： 有时候你会希望 operator new 和 operator delete 做编译器版不会做的事情，例如分配和归还共享内存（shared memory），而这些事情只能被 C API 完成，则可以将 C API 封在 C++ 的外壳里，写在定制的 new 和 delete 中。

条款 51：编写 new 和 delete 时需固守常规

我们在条款 49 中已经提到过一些 operator new 的规矩，比如内存不足时必须不断调用 new-handler，如果无法供应客户申请的内存，就抛出 std::bad_alloc 异常。C++ 还有一个奇怪的规定，即使客户需求为 0 字节，operator new 也得返回一个合法的指针，这种看似诡异的行为其实是为了简化语言其他部分。

根据这些规约，我们可以写出非成员函数版本的 operator new 代码：

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

    if (size == 0)       // 处理 0 字节申请
        size = 1;        // 将其视为 1 字节申请

    while (true) {
        if (...)         // 如果分配成功
            return ...; // 返回指针指向分配得到的内存

        // 如果分配失败，调用目前的 new-handler
        auto globalHandler = get_new_handler(); // since C++11

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

operator new 的成员函数版本一般只会分配大小刚好为类的大小的内存空间，但是情况并不总是如此，假设我们没有为派生类声明其自己的 operator new，那么派生类会从基类继承 operator new，这就导致派生类可以使用其基类的 new 分配方式，但派生类和基类的大小很多时候是不同的。

处理此情况的最佳做法是将“内存申请量错误”的调用行为改为采用标准的 operator new：

```
void* Base::operator new(std::size_t size) {
    if (size != sizeof(Base))
        return ::operator new(size);
// 转交给标准的 operator new 进行处理
}
```

注意在 operator new 的成员函数版本中我们也不需要检测分配的大小是否为 0 了，因为在条款 39 中我们提到过，非附属对象必须有非零大小，所以 sizeof(Base)无论如何也不能为 0。

如果你打算实现 operator new[]，即所谓的 array new，那么你唯一要做的一件事就是分配一块未加工的原始内存，因为你无法对 array 之内迄今尚未存在的元素对象做任何事情，实际上你甚至无法计算这个 array 将含有多少元素对象。

operator delete 的规约更加简单，你需要记住的唯一一件事情就是 C++ 保证**“删除空指针永远安全”**：

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

    // 归还 rawMemory 所指的内存
}
```

operator delete 的成员函数版本要多做的唯一一件事就是将大小有误的删除行为转交给标准的 operator delete：

```
void Base::operator delete(void* rawMemory, std::size_t size)
noexcept {
    if (rawMemory == 0) return;
    if (size != sizeof(Base)) {
        ::operator delete(rawMemory);
// 转交给标准的 operator delete 进行处理
        return;
    }

    // 归还 rawMemory 所指的内存
}
```

如果即将被删除的对象派生自某个基类而后者缺少虚析构函数，那么 C++ 传给 operator delete 的 size 大小可能不正确，这或许是“为多态基类声明虚析构函数”的一个足够的理由，能作为对条款 7 的补充。

条款 52：写了 placement new 也要写 placement delete

placement new 最初的含义指的是“接受一个指针指向对象该被构造之处”的 operator new 版本，它在标准库中的用途广泛，其中之一是负责在 vector 的未使用空间上创建对象，它的声明如下：

```
void* operator new(std::size_t, void* pMemory) noexcept;
```

我们此处要讨论的是广义上的 placement new，即带有附加参数的 operator new，例如下面这种：

```
void* operator new(std::size_t, std::ostream& logStream);

auto pw = new (std::cerr) Widget;
```

当我们在使用 new 表达式创建对象时，共有两个函数被调用：一个是用以分配内存的 operator new，一个是对象的构造函数。假设第一个函数调用成功，而第二个函数却抛出异常，那么会由 C++ runtime 调用 operator delete，归还已经分配好的内存。

这一切的前提是 C++ runtime 能够找到 operator new 对应的 operator delete，如果我们使用的是自定义的 placement new，而没有为其准备对应的 placement delete 的话，就无法避免发生内存泄漏。因此，合格的代码应该是这样的：

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

    static void operator delete(void* pMemory);
// delete 时调用的正常 operator delete
    static void operator delete(void* pMemory, std::ostream&
logStream);    // placement delete
};
```

另一个要注意的问题是，由于成员函数的名称会掩盖其外部作用域中的相同名称（见条款 33），所以提供 placement new 会导致无法使用正常版本的 operator new：

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

```
    ...
};

auto pb = new Base;             // 无法通过编译!
auto pb = new (std::cerr) Base; // 正确
```

同样道理，派生类中的 operator new 会掩盖全局版本和继承而得的 operator new 版本：

```
class Derived : public Base {
public:
    static void* operator new(std::size_t size);
};

auto pd = new (std::clog) Derived;  // 无法通过编译!
auto pd = new Derived;              // 正确
```

为了避免名称遮掩问题，需要确保以下形式的 operator new 对于定制类型仍然可用，除非你的意图就是阻止客户使用它们：

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

一个最简单的实现方式是，准备一个基类，内含所有正常形式的 new 和 delete：

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

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

```
        ::operator delete(pMemory, ptr);
    }

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

凡是想以自定义形式扩充标准形式的客户，可以利用继承和 using 声明式（见条款 33）取得标准形式：

```
class Widget: public StandardNewDeleteForms{
public:
    using StandardNewDeleteForms::operator new;
    using StandardNewDeleteForms::operator delete;

    static void* operator new(std::size_t size, std::ostream&
logStream);
    static void operator detele(std::size_t size, std::ostream&
logStream) noexcept;
};
```

第九章：杂项讨论

条款 53：不要轻忽编译器的警告

1. 严肃对待编译器发出的警告信息。努力在你的编译器的最高（最严苛）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警告能力，因为不同的编译器对待事情的态度不同。一旦移植到另一个编译器上，你原本依赖的警告信息可能会消失。

条款 54：让自己熟悉包括 TR1 在内的标准程序库

如今 TR1 草案已完全融入 C++ 标准当中，没有再过多了解 TR1 标准库的必要。

条款 55：让自己熟悉 Boost

Boost 是若干个程序库的集合，并且当中的许多库已经被 C++ 吸纳为标准库的一部分。不过在现在的 Modern C++ 时代，是否该在项目中使用 Boost 仍然有一定的争议，一些 Boost 组件并无法做到像 C++ 标准库那样高性能，零开销抽象，但毫无疑问的是，Boost 的参考价值是无法忽视的，你可以在 Boost 中找到许多非常值得学习和借鉴的实现。