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。