

Effective-CPP

第四：

初始化发生在构造函数被调用之前。构造函数内其实是赋值。应从此处初始化的方式，更为妥当。

ABEntry 构造函数的一个较佳写法是，使用所谓的 **member initialization list**（成员初值列）替换赋值动作：

```
ABEntry::ABEntry(const std::string& name, const std::string& address,  
                  const std::list<PhoneNumber>& phones)  
    :theName(name),  
     theAddress(address),           //现在，这些都是初始化 (initializations)  
     thePhones(phones),  
     numTimesConsulted(0)  
{ }                           //现在，构造函数本体不必有任何动作
```

第五章:

编译器暗自为 class 创建 default 构造函数,
copy 构造函数, copy assignment 以及析构函数。
(都为公有)

```
class Empty { };  
这就好像你写下这样的代码:  
  
class Empty {  
public:  
    Empty() { ... } //default 构造函数  
    Empty(const Empty& rhs) { ... } //copy 构造函数  
    ~Empty() { ... } //析构函数, 是否该是  
                      // virtual 见稍后说明.  
    Empty& operator=(const Empty& rhs) { ... } //copy assignment 操作符.  
};
```

这些函数只有在被调用时才会创建。

当你声明了一个 函数后, 编译器将不再创建 default 函数
当默认 copy assignment 试图修改 reference 的指向或试图修改
const 成员时, 则需自定义 copy assignment.

条款六：

不想使用编译器自动生成的函数，则“private”屏蔽。

```
class Uncopyable {  
protected:                                         //允许 derived 对象构造和析构  
    Uncopyable() {}  
    ~Uncopyable() {}  
private:  
    Uncopyable(const Uncopyable&);           //但阻止 copying  
    Uncopyable& operator=(const Uncopyable&);  
};  
  
为求阻止 HomeForSale 对象被拷贝，我们唯一需要做的就是继承 Uncopyable:  
  
class HomeForSale: private Uncopyable {          //class 不再声明  
    ...  
};  
//copy 构造函数或  
//copy assign. 操作符
```

也可使用 Boost 的 non_copyable。

(有必要看一下 Boost.)

条款七：

若 class 被用于做 base class, 则应声明一个 virtual 析构函数。

若 class 不被用于 base class, 则不应声明 virtual 析构函数。

当然也不要试图继承一个 non-virtual-class。

例如：STL 中的类。

条款八：

绝对不要在析构函数内抛出异常。

一定要在普通函数中抛出。

条款九：

在构造和析构期间不要调用 virtual 函数。

因为这会使调用从不下降。

构造和析构内的调用权当 base-class。在 base-class 构造析构期间 virtual 函数 不是 virtual 函数。

条款十：

赋值操作符返回一个 reference to `*this`。

这使得赋值连统形式变得可能。

这是一种协议，遵守便是了，不遵守也不会报错。

条款十一：

要确保 operator= 在操作多个同一对象时有良好的行为。

例如：

添加证明测试：

```
Widget& Widget::operator=(const Widget& rhs)
{
    if (this == &rhs) return *this; // 证明测试 (identity test) :
                                    // 如果是自我赋值，就不做任何事。
    delete pb;
    pb = new Bitmap(*rhs.pb);
    return *this;
}
```

添加一个副本在赋值给 pb 之前不会删除：

```
Widget& Widget::operator=(const Widget& rhs)
{
    Bitmap* pOrig = pb; // 记住原先的 pb
    pb = new Bitmap(*rhs.pb); // 令 pb 指向 *pb 的一个复印件 (副本)
    delete pOrig; // 删除原先的 pb
    return *this;
}
```

上面的简化了 copy and swap 技术

```
class Widget {
...
void swap(Widget& rhs); // 交换*this 和 rhs 的数据；详见条款 29
...
};

Widget& Widget::operator=(const Widget& rhs)
{
    Widget temp(rhs); // 为 rhs 数据制作一份复印件 (副本)
    swap(temp); // 将*this 数据和上述复印件的数据交换。
    return *this;
}
```

这是一种很不清晰的作法，但不可否认，有时很高效：

```
Widget& Widget::operator=(Widget rhs) //rhs 是被传对象的一份复印件（副本）
{
    swap(rhs); //注意这里是 pass by value.
    return *this; //将*this 的数据和复印件/副本的数据互换
}
```

条款十二：

Copying 函数应该确保复制“对象内的所有变量”及“所有 base class 成分”。

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
    : Customer(rhs),           //调用 base class 的 copy 构造函数
      priority(rhs.priority)
{
    logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
    logCall("PriorityCustomer copy assignment operator");
    Customer::operator=(rhs);   //对 base class 成分进行赋值动作
    priority = rhs.priority;
    return *this;
}
```

不要尝试以某个 Copying 函数实现另一个 Copying 函数。应该将共同机能放进第三个函数中，并由两个函数共同调用。

Copying 函数这指 Copy 构造函数以及 copy assignment。

条款十三：

为了防止资源泄漏，请使用 RAII 对象（资源取得时机便是初始化时的时机），他们在构造函数中获得资源并在析构函数中释放资源。

```
void f()
{
    Investment* pInv = createInvestment();           // 调用 factory 函数
    ...
    delete pInv;    若...处出现异常，则程序会跳过 // 释放 pInv 所指对象
}
```

```
void f()
{
    std::shared_ptr<Investment> pInv(createInvestment()); // 调用 factory 函数
    ...
} // 一如以往地使用 pInv // 经由 auto_ptr 的析构函数自动删除 pInv
```

两个被推荐使用的 RAII classes 分别是 `std::shared_ptr` 和 `auto_ptr`，但通常前者是较推荐，因为 copy 行为很直观，若选择 `auto_ptr` 复制动作会使被复制物指向 null。

第十四：

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。

```
class Lock {
public:
    explicit Lock(Mutex* pm)           // 以某个 Mutex 初始化 shared_ptr
        : mutexPtr(pm, unlock)         // 并以 unlock 函数为删除器
    {
        lock(mutexPtr.get());          // 条款 15 谈到 "get" 部署为 out 调用
    }
private:
    std::tr1::shared_ptr<Mutex> mutexPtr; // 使用 shared_ptr
};                                     // 替换 raw pointer
```

通过抑制的行是抑制 copying、施行引用计数法

private: (copying)

mutexPtr 是一个 shared_ptr<Mutex> 指针，而非一个 Mutex
类型的指针，若想获得其内部指针可调用 mutexPtr.get()，详见条款

十五。

第十六：

如果你在 new 表达式中使用 []，必须在相应的 delete 表达式中也使用 []。如果您在 new 表达式中不使用 []，一定不要在相应的 delete 表达式中使用 []

单一对象

Object

对象数组

n	Object	Object	Object	...
---	--------	--------	--------	-----

第六十七：

以独立语句将 newed 对象存储于 (互入) 智能指针内。如果不这样做，一旦异常抛出，有可能导致难以察觉的资源泄漏。

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

由调用顺序未知可能产生：

1. 执行 "new Widget"
2. 调用 priority
3. 调用 tr1::shared_ptr 构造函数

这样的顺序。

如果 priority 抛出异常，则会产生资源泄漏。

```
std::tr1::shared_ptr<Widget> pw(new Widget);           // 在单独语句内以  
                                                               // 智能指针存储  
                                                               // newed 所得对象。  
processWidget(pw, priority());                         // 这个调用动作绝不至于造成泄漏。
```

