Effective C++ (3rd) 学习笔记

吴金龙

北京大学数学科学学院

2008年12月

目录

第一章 让自己习惯C++ **1**

1.1 条款01: 视C++为一个语言联邦 1

1.2 条款02: 尽量以const,enum,inline替换#define 1

1.3 条款03: 尽可能使用const 2

1.4 条款04: 确定对象被使用前已先被初始化 3

第二章 构造/析构/赋值运算 **5**

2.1 条款05: 了解C++ 默默编写并调用哪些函数 5

2.2 条款06: 若不想使用编译器自动生成的函数,
就该明确拒绝 5

2.3 条款07: 为多态基类声明virtual析构函数 6

2.4 条款08: 别让异常逃离析构函数 7

2.5 条款09: 绝不在构造和析构过程中调用virtual函数 7

2.6 条款10: 令operator=返回一个reference to *this 7

2.7 条款11: 在operator=中处理“自我赋值” 8

2.8 条款12: 复制对象时勿忘其每一个成分 10

第三章 资源管理 **11**

3.1 条款16: 成对使用new和delete时要采取相同形式 11

第四章 设计与声明 **13**

4.1 条款19: 设计class犹如设计type 13

4.2 条款20: 宁以pass-by-reference-to-const替换pass-by-value 13

第五章 实现 **15**

5.1 条款26: 尽可能延后变量定义式的出现时间 15

5.2 条款28: 避免返回handles指向对象内部成分 16

5.3 条款30: 透彻了解inlining的里里外外 16

第六章 继承与面向对象设计 **19**

6.1 条款33: 避免遮掩继承而来的名称 19

6.2 条款36: 绝不重新定义继承而来的non-virtual 函数 21

6.3 条款37: 绝不重新定义继承而来的缺省参数值 22

6.4 条款38: 通过复合塑模出has-a 或“根据某物实现出” 23

6.5 条款40: 明智而审慎地使用多重继承 23

第七章 模版和泛型编程 **25**

7.1 条款42: 了解typename的双重意义 25

7.2 条款44: 将与参数无关的代码抽离templates 26

第一章　让自己习惯C++

1.1　条款01：视C++为一个语言联邦

今天的C++已经是个多重范型编程语言（multiparadigm programming language），一个同时支持过程形式（procedural）、面向对象形式（object-oriented）、函数形式（functional）、泛型形式（generic）、元编程形式（metaprogramming）的语言。

最简单的方法是将C++视为一个由相关次语言（sublanguage）组成的联邦而非单一语言。幸运的是，次语言总共有四个：

- C。
- Object-Oriented C++。包括：class（包括构造函数和析构函数）、封装（encapsulation）、继承（inheritance）、多态（polymorphism）、virtual 函数（动态绑定）、... ... 等等。
- Template C++。这是C++ 的泛型编程（generic programming）部分。
- STL。STL 是个template 程序库，但它是非常特殊的一个。它对容器（containers）、迭代器（iterators）、算法（algorithms）以及函数对象（function objects）的规约有极佳的紧密配合与协调，然而templates 及程序库也可以其他想法建置出来。

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

这个条款或许可以改为“宁可用编译器替换预处理器”。

通常C++要求你对所使用的任何东西提供一个定义式，但如果它是个class专属常量又是static且为整数类型（integral type,例如int,char,bool），则可特殊处理。只要不取它们的地址，你可以声明并使用它们而无需提供定义式。

例：

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

上面的是NumTurns的声明式而非定义式。如果你要取这个class专属常量的地址，你就必须另外提供定义式如下:

```
const int GamePlayer::NumTurns;    //NumTurns的定义;
```

上面的这个式子应该**放入一个实现文件而非头文件!** 由于它已经在声明时获得了初值，定义时不可以再设初值。

旧式编译器也许不支持上述语法，他们不允许static成员在其声明式上获得初值。此外，所谓的“in-class初值设定”也只允许对整数常量进行。如果你的编译器不支持上述语法，你可以将初值放在定义式:

例:

```
class CostEstimate{
private:
    static const double FudgeFactor;    //static class 常量声明，位于头文件内
...
};
const double CostEstimate::FudgeFactor = 1.35; //static class 常量定义，位于实现文件中
```

一个例外是当你在class编译期间需要一个class常量值，例如上面的GamePlayer::scores的数组声明式中（编译器坚持必须在编译期间知道数组的大小）。这时候万一你的编译器不允许“static整数型class常量”完成“in class 初值设定”，可改为所谓的“the enum hack”补偿做法。其理论基础是:“一个属于枚举类型（enumerated type）的数值可权充int被使用”，于是GamePlayer可定义如下:

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

1.3 条款03：尽可能使用const

STL 迭代器是以指针为根据塑模出来的，所以迭代器的作用就像个T* 指针。声明迭代器为const就像声明指针为const一样（即声明一个T* const 指针）。

例:

```
std::vector<int> vec ;
...
const std::vector<int>::iterator iter=vec.begin() ; // iter 的作用像个T* const
++ iter ; //错误! iter 是const
```

如果你希望迭代器所指的东西不可被改动（即希望STL 模拟一个const T* 指针），你需要的是const_iterator：

```
std::vector<int>::const_iterator cIter=vec.begin() ;
*cIter = 10 ; // 错误! *cIter 是const
++cIter ; // 正确。
```

令函数返回一个常量值,往往可以降低因客户错误而造成的意外,而又不至于放弃安全性和高效性。

例:

```
class Rational{ ... } ;
const Rational operator* ( const Rational& lhs, const Rational& rhs ) ;
Rational a, b, c ;
...
(a * b) = c ; //错误! operator* 返回常量值
```

1.4 条款04：确定对象被使用前已先被初始化

读取未初始化的值会导致不明确的行为。

通常如果你使用C part of C++（见条款01）而且初始化可能招致运行期成本，那么就不保证发生初始化。一旦进入non-C parts of C++，规则有些变化。这就是为什么array（来自C part of C++）不保证其内容被初始化，而vector（来自STL part of C++）却有此保证。

C++规定，对象的成员变量的初始化动作发生在进入构造函数本体之前。所以，相对于在构造函数本体里为成员变量赋值，一个更好的办法是在member initialization list（成员初值列）里初始化成员变量。如果成员变量是const或references，它们就一定需要初值（所以只能放在成员初值列中），不能被赋值。

C++有着十分固定的“成员初始化次序”：base classes更早于derived classes被初始化，而class的成员变量总是以其声明次序（类定义中的次序）被初始化。即使它们在成员初值列中以不同的次序出现（很不幸那是合法的），也不会有任何影响。

一旦你已经很小心地将“内置型成员变量”明确地加以初始化，而且也确保你的构造函数运用“成员初值列”初始化base classes和成员变量，那就只剩唯一一件事需要操心—“不同编译单元内定义之non-local static对象”的初始化次序（P30-P33）。

请记住

- 为内置型对象进行手工初始化，因为C++不保证初始化它们。
- 构造函数最好使用成员初值列（member initialization list），而不要在构造函数本体内使用赋值操作（assignment）。初值列列出的成员变量，其排列次序应该（不是必须）和它们在class中的声明次序相同。
- 为免除“跨编译单元之初始化次序”问题，请以local static对象替换non-local static对象。

第二章　构造/析构/赋值运算

2.1　条款05：了解C++ 默默编写并调用哪些函数

如果你自己没声明，编译器就会为它声明（编译器版本）一个default 构造函数、一个copy 构造函数、一个copy assignment 操作符和一个析构函数。因此，如果你写下:

```
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
    Empty& operator=(const Empty& rhs) { ... } // copy assignment 操作符
};
```

惟有当这些函数被需要（被调用），它们才会被编译器创建出来。

所有这些函数都是public 且inline （见条款30）。注意，编译器产生的析构函数是个non-virtual（见条款07），除非这个class 的base class 自身声明有virtual 析构函数。

如果某个base classes 将copy assignment 操作符声明为private，编译器将拒绝为其derived classes 生成一个copy assignment 操作符。

2.2　条款06：若不想使用编译器自动生成的函数，就该明确拒绝

所有编译器产生的函数都是public。为阻止这些函数被创建出来，你得自行声明它们。为使产生的类不支持copying，你可以将copy构造函数或copy assignment操作符声明为private。

请记住

- 为驳回编译器自动（暗自）提供的机能，可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（见P39）也是一种做法。

2.3 条款07：为多态基类声明virtual析构函数

C++明白指出，当derived class对象经由一个base class指针被删除，而该base class带着一个non-virtual析构函数，其结果未有定义—实际执行时通常发生的是对象的derived成分没被销毁。

任何class只要带有virtual函数都几乎确定应该也有一个virtual析构函数。如果class不含virtual函数，通常表示它并不意图被用做一个base class。当class不企图被当做base class，令其析构函数为virtual往往是个馊主意，因为这样会增加其对象的大小。

如果你希望让一个类成为抽象基类，你应该为它声明一个pure virtual析构函数。

例：

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

这个class有一个pure class函数，所以它是个抽象class，又由于它有个virtual析构函数，所以你不需要担心析构函数的问题。然而这里有个窍门：你必须为这个pure virtual析构函数提供一份定义：

```
AWOV::~AWOV()    { }          //pure virtual析构函数的定义
```

析构函数的运作方式是，最深层派生（most derived）的那个class其析构函数最先被调用，然后是其每一个base class 的析构函数被调用。编译器会在AWOV的derived classes的析构函数中创建一个对~AWOV的调用动作，所以你必须为这个函数提供一份定义。如果不这样做，连接器会发出抱怨。

“给base classes一个virtual析构函数”，这个规则只适用于polymorphic（带多态性质的）base classes身上。这种base classes的设计目的是为了用来“通过base class接口处理derived class对象”。

并非所有base classes的设计目的都是为了多态用途。例如标准string和STL容器都不被设计作为base classes使用，更别提多态了。某些classes的设计目的是作为base classes使用，但不是为了多态用途。

请记住

- polymorphic（带多态性质的）base classes应该声明一个virtual析构函数。如果class带有任何virtual函数，它就应该拥有一个virtual析构函数。

- 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（polymorphically），就不该声明virtual析构函数。

2.4 条款08：别让异常逃离析构函数

C++并不禁止析构函数吐出异常，但它不鼓励你这样做。如果某个操作可能在失败时抛出异常，而又存在某种需要必须处理该异常，那么这个异常必须来自析构函数以外的某个函数。

请记住

- 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，然后吞下它们（不传播）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。

2.5 条款09：绝不在构造和析构过程中调用virtual函数

Base class构造期间virtual函数绝不会下降到derived classes阶层。取而代之的是，对象的作为就像隶属base类型一样。也就是说，在base classes构造期间，virtual函数不是virtual函数。

根本的原因是，在derived class对象的base class构造期间，对象的类型是base class而不是derived class。

相同道理也适用于析构函数。

请记住

- 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（比起当前执行构造函数和析构函数的那层）。

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

赋值可以写成连锁形式：

```
x = y = z = 15 ;
```

赋值采用右结合律：

```
x = (y = (z = 15)) ;
```

为了实现“连锁赋值”，赋值操作符必须返回一个reference指向操作符的左侧实参。这是你为classes实现赋值操作符时应该遵循的协议：

例：

```
class Widget{
public:
    ...
    Widget & operator=(const Widget& rhs) //返回类型是个reference, 指向当前对象
    {
        ...
        return * this ; //返回左侧对象
    }
    ...
};
```

这个协议不仅适用于以上的标准赋值形式，也适用于所有赋值相关运算，如“+=、-=、*=”等。

请记住

- 令赋值（assignment）操作符返回一个reference to *this。

2.7 条款11：在operator=中处理“自我赋值”

“自我赋值”发生在对象被赋值给自己时。例如，假设你建立一个class用来保存一个指针指向一块动态分配的位图（bitmap）：

```
class Bitmap { ... };
class Widget {
    ...
private:
    Bitmap* pb; //指针，指向一个从heap分配而得的对象
};
```

下面是operator=实现代码，其自我赋值出现时并不安全（它也不具备异常安全性）。

```
Widget& Widget::operator=(const Widget& rhs) //一份不安全的operator=实现版本
{
    delete pb; //停止使用当前的bitmap，
    pb = new Bitmap(*rhs.pb); //使用rhs的bitmap副本（复件）。
    return *this; //见条款10。
}
```

欲阻止这种错误，传统做法是借由operator=最前面的一个“证同测试（identity test）”达到“自我赋值”的检验目的：

```
Widget& Widget::operator=(const Widget& rhs) //一份不安全的operator=实现版本
{
    if( this == &rhs ) return *this; //证同测试 (identity test)。

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

这个版本仍然存在异常方面的麻烦。如果“new Bitmap”导致异常，Widget最终会持有一个指针指向一块被删除的Bitmap。令人高兴的是，让operator=具备“异常安全性”往往自动获得“自我赋值安全”的回报。本条款只要你注意“许多时候一群精心安排的语句就可以导出异常安全（以及自我赋值安全）的代码”，这就够了。例如以下代码，我们只需注意在复制pb所指东西之前别删除pb:

```
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的一个复件 (副本)
    delete pOrig; //删除原先的pb
    return *this;
}
```

在operator=函数内手工排列语句（确保代码不但“异常安全”而且“自我赋值安全”）的一个替代方案是，使用所谓的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数据制作一份复制 (副本)
    swap(temp); //将*this数据和上述复件的数据交换。
    return *this;
}
```

请记住

- 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”的地址、精心周到的语句顺序、以及copy-and-swap。
- 确保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。

2.8 条款12：复制对象时勿忘其每一个成分

设计良好之面向对象系统（OO-systems）会将对象的内部封装起来，只留两个函数负责对象拷贝（复制），那便是带着适切名称的copy构造函数和copy assignment操作符，我称它们为**copying函数**。

任何时候只要你承担起“为derived class撰写copying函数“的重责大任，必须很小心地也复制其base class成分。

例：

```
class Customer{
...
};

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

当你编写一个copying函数，请确保（1）复制所有local成员变量，（2）调用所有base classes内的适当的copying函数。

请记住

- Copying函数应该确保复制“对象内的所有成员变量”及“所有base class成分”。
- 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（通常叫init），并由两个copying函数共同调用。

第三章　资源管理

3.1　条款16：成对使用new和delete时要采取相同形式

当你使用new（也就是通过new动态生成一个对象），有两件事发生。第一，内存被分配出来（通过名为operator new的函数，见条款49和条款51）。第二，针对此内存会有一个（或更多）构造函数被调用。当你使用delete，也有两件事发生：针对此内存会有一个（或更多）析构函数被调用，然后内存才被释放。delete的最大问题在于：即将被删除的内存之内究竟存有多少对象？

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

如果你使用delete时加上**中括号（方括号）**，delete便认定指针指向一个数组，否则它便认定指针指向单一对象：

例：

```
std::string* stringPtr1 = new std::string;
std::string* stringPtr2 = new std::string[100];
...
delete stringPtr1; //删除一个对象
delete [] stringPtr2; //删除一个由对象组成的数组
```

游戏规则很简单：如果你调用new时使用[]，你必须在对应调用delete时也使用[]。如果你调用new时没有使用[]，那么也不该在对应调用delete时使用[]。

这个规则对于喜欢使用typedef的人也很重要。考虑下么这个typedef：

例：

```
typedef std::string AddressLines[4];
std::string* pal = new AddressLines; //注意，“new AddressLines”返回一个string*，
                                     //就像“new string[4]”一样。
...
delete pal; //行为未有定义!
delete [] pal; //很好。
```

为避免此类错误，**最好尽量不要对数组形式做typedef动作**。

请记住

- 如果你调用new时使用[]，你必须在对应调用delete时也使用[]。如果你调用new时没有使用[]，那么也不该在对应调用delete时使用[]。

第四章　设计与声明

4.1　条款19：设计class犹如设计type

C++就像在其他OOP（面向对象编程）语言一样，当你定义一个新class，也就定义了一个新type。你并不只是class设计者，还是type设计者。

那么，如何设计高效的classes呢？首先你必须了解你面对的问题。几乎每一个class都要求你面对一下提问，而你的回答往往导致你的设计规范:

- 新type的对象应该如何被创建和销毁？
- 对象的初始化和对象的赋值该有什么样的差别？
- 新type的对象如果被passed by value，意味着什么？
- 什么是新type的“合法值”？
- 你的新type需要配合某个继承图系（inheritance graph）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？
- 该谁取用新type的成员？
- 什么是新type的“未声明接口”（undeclared interface）?
- 你的新type有多么一般化？
- 你真的需要一个新type吗？

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

Pass-by-reference-to-const的效率高得多，没有任何构造函数或析构函数被调用，因为没有任何新对象被创建。

以by reference方式传递参数也可以避免slicing(对象切割)问题。当一个derived class对象以by value方式传递并被视为一个base class对象，base class的copy构造函数会被调用，而“造成此对象的行为像个derived class对象”的那些特化性质全被切割掉了，仅仅留下一个base class对象。

references往往以指针实现出来，因此pass by reference通常意味真正传递的是指针。因此如果你有个对象属于内置类型(例如int)，pass by value往往比pass by reference的效率高些。这个忠告也适用于STL的迭代器和函数对象。

并不是小型types都是pass-by-value的合格候选人，对象小并不就意味其copy构造函数不昂贵。即使小型对象拥有并不昂贵的copy构造函数，还是可能有效率上的争议。某些编译器对待“内置类型”和“用户自定义类型”的态度截然不同，纵使两者拥有相同的底层表述(underlying representation)。

“小型的用户自定义类型不必然成为pass-by-value优良候选人”的另一个理由是，作为一个用户自定义类型，其大小容易有所变化。

一般而言，你可以合理假设“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**。

请记住

- 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(slicing problems)。
- 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。对它们而言，pass-by-value往往比较适当。

第五章　实现

5.1　条款26：尽可能延后变量定义式的出现时间

只要你定义了一个变量而其类型带有一个构造函数或析构函数，那么当程序的控制流（control flow）到达这个变量定义式时，你便得承受构造成本；当这个变量离开其作用域时，你便得承受析构成本。

你不只应该延后变量的定义，直到非得使用该变量的前一刻为止，甚至应该尝试延后这份定义直到能够给它初值实参为止。如果这样，不仅能够避免构造（和析构）非必要对象，还可以避免无意义的default构造行为。

“但循环怎么办？”如果变量只在循环内使用，那么把它定义于循环外并在每次循环迭代时赋值给它比较好，还是该把它定义于循环内？

例：

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

和

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

如果classes的一个赋值成本低于一组构造+析构成本，做法A大体而言比较高效。否则做法B或许较好。此外，做法A 造成名称w的作用域（覆盖整个循环）比做法B更大，有时会对程序的可理解性和易维护性造成冲突。因此除非（1）你知道赋值成本比“构造+析构”成本低，（2）你正在处理代码中效率高度敏感（performance-sensitive）的部分，否则你应该使用做法B。

请记住

- 尽可能延后变量定义式的出现。这样做可增加程序的清晰度并改善程序效率。

5.2 条款28：避免返回handles指向对象内部成分

References、指针和迭代器都是所谓的handles（号码牌，用来取得某个对象）。返回一个“代表对象内部数据”的handle，随之而来的便是“降低对象封装性”的风险。

对象的“内部”是指它的成员变量和不被公开使用的成员函数（也就是被声明为protected或private者），因此也要留心不要返回它们的handles。

返回handles时可能导致dangling handles（空悬的号码牌）：这种handles所指东西（的所属对象）不复存在。这种“不复存在的对象”最常见的来源就是函数返回值。这里唯一的关键是，有个handle被传出去了，一旦如此你就是暴露在“handle比其所指对象更长寿”的风险下。

这并不意味着你绝对不可以让成员函数返回handle。有时候你必须那么做。例如operator[]就允许你“摘采”strings和vectors的个别元素，而这些operator[]就是返回references指向“容器内的数据”（见条款3），那些数据会随着容器被销毁而销毁。尽管如此，这样的函数毕竟是例外，不是常态。

请记住

- 避免返回handles（包括references、指针和迭代器）指向对象内部。遵守这个条款可增加封装性，帮助const成员函数的行为像个const，并将发生“虚吊号码牌”（dangling handles）的可能性降至最低。

5.3 条款30：透彻了解inlining的里里外外

Inline只是对编译器的一个申请，不是强制命令。

Inline函数通常一定被置于头文件内，因为大多数建置环境（build environments）在编译过程中进行inlining，而为了将一个“函数调用”替换为“被调用函数的本体”，编译器必须知道那个函数长什么样子。

Templates通常也被置于头文件内，因为它一旦被使用，编译器为了将它具现化，需要知道它长什么样子。（某些建置环境可以再连接期才执行template具现化。）

大部分编译器拒绝将太过复杂（例如带有循环或递归）的函数inlining，而所有对virtual函数的调用（除非是最平淡无奇的）也都会使inlining落空。因为virtual意味“等待，直到运行期才确定调用哪个函数”，而inline意味“执行前，先将调用动作替换为被调用函数的本体”。

如果程序要取某个inline函数的地址，编译器通常必须为此函数生成一个outlined函数本体。毕竟编译器不可能提出一个指针指向并不存在的函数。与此并提的是，编译器通常不对“通过函数指针而进行的调用”实施inlining，这意味对inline 函数的调用有可能被inlined，也可能不被inlined，取决于该调用的实施方式。

例:

```
inline void f() { ... } //假设编译器有意愿inline“对f的调用”
void (*pf) () = f; //pf指向f
...
f(); //这个调用将被inlined，因为它是一个正常调用。
pf(); //这个调用或许不被inlined，因为它通过函数指针达成。
```

实际上构造函数和析构函数往往是inlining的糟糕候选人，因为编译器在编译期间会安插一些代码到程序中。

程序库设计者必须评估“将函数声明为inline”的冲击：inline函数无法随着程序库的升级而升级。一旦一个inline函数被修改，所有用到此函数的程序都必须被重新编译。但对于non-inline函数，一旦它有任何修改，只需重新连接就好。

若从纯粹实用观点出发，有一个事实比其他因素更重要：**大部分调试器面对inline函数都束手无策。**毕竟你如何在一个并不存在的函数内设立断点呢？

请记住

- 将大多数inlining限制在小型、被频繁调用的函数身上。这可使日后的调试过程和二进制升级（binary upgradability）更容易，也可使潜在的代码膨胀问题最小化，使程序的速度提升机会最大化。
- 不要只因为function templates出现在头文件，就将它们声明为inline。

第六章　继承与面向对象设计

6.1　条款33：避免遮掩继承而来的名称

C++的名称遮掩规则（name-hiding rules）所做的唯一事情就是：**遮掩名称**。

对于继承来说，derive class作用域被嵌套在base class作用域内。

例：

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

以作用域为基础的“名称遮掩规则”并没有改变，因此base class内所有名为mf1和mf3的函数都被derived class内的mf1和mf3函数遮掩掉了。

```
Derived d;
int x;
...
d.mf1(); //没问题，调用Derived::mf1
d.mf1(x); //错误！因为Derived:mf1遮掩了Base::mf1
d.mf2(); //没问题，调用Base::mf2
d.mf3(); //没问题，调用Derived::mf3
d.mf3(x); //错误！因为Derived::mf3遮掩了Base::mf3
```

如你所见，上述规则都适用，即使base classes和derived classes内的函数有不同的参数类型也适用，而且不论函数是virtual或non-virtual一体适用。

不幸的是你通常会想继承重载函数。实际上如果你正在使用public继承而又不继承那些重载函数，就是违反base和derived classes之间的**is-a**关系，而条款32说过**is-a**是public继承的基石。

你可以使用using声明式达成目标:

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

class Derived: public Base{
public:
    using Base::mf1; //让Base class内名为mf1和mf3的所有东西
    using Base::mf3; //在Derived作用域内都可见（并且public）
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};
```

现在，继承机制将一如往昔地运作:

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

这意味如果你继承base classes并加上重载函数，而你又希望重新定义或覆写（推翻）其中一部分，那么你必须为那些原本会被遮掩的每个名称引人一个using声明式，否则某些你希望继承的名称会被遮掩。

请记住

- 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。
- 为了让被遮掩的名称再见天日，可使用using声明式或转交函数（forwarding functions）。

6.2 条款36：绝不重新定义继承而来的non-virtual 函数

举例说明之。

例:

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

D x ;
B* pB = &x ; //获得一个指针指向x
D* pD = &x ; //获得另一个指针指向x
pB->mf() ; // 调用B::mf
pD->mf() ; // 调用D::mf
```

你可能会相当惊讶。毕竟两者都通过对象x 调用成员函数mf。造成这种两面行为的原因是，non-virtual 函数如B::mf 和D::mf 都是静态绑定（statically bound）。这意味着通过pB 调用的non-virtual 函数永远是B 所定义的版本，即使pB 指向一个类型为“B 派生之class”的对象。

但另一方面，virtual 函数却是动态绑定（dynamically bound）。如果mf 是个virtual 函数，不论是通过pB 或pD 调用mf，都会导致调用D::mf，因为pB 和pD 真正指的都是一个类型为D 的对象。

6.3 条款37：绝不重新定义继承而来的缺省参数值

既然重新定义一个继承而来的non-virtual 函数永远是错误的（见条款36），所以我们将本条款局限于“继承一个带有缺省参数值的virtual 函数”。

对象的所谓静态类型（static type），就是它在程序中被声明时所采用的类型。

例：

```
class Shape{
public:
    enum ShapeColor { Red, Green, Blue };
    virtual void draw( ShapeColor color = Red ) const = 0 ;
...
};
class Rectangle : public Shape {
public:
    //注意，赋予不同的缺省参数值。这真糟糕!
    virtual void draw( ShapeColor color = Green ) const ;
...
};
```

现在考虑这些指针：

```
Shape* ps ; //静态类型为Shape*
Shape* pr = new Rectangle ; //静态类型为Shape*
```

本例中ps 和pr 都被声明为**pointer-to-**Shape 类型，所以它们都以它为静态类型。不论它们真正指向什么，他们的静态类型都是Shape*。

对象的所谓动态类型（dynamic type）则是指“目前所指对象的类型”。也就是说，动态类型可以表现出一个对象将会有什么行为。以上例而言，pr 的动态类型是Rectangle*，ps 没有动态类型，因为它尚未指向任何对象。

virtual 函数系动态绑定而来，意思是调用一个virtual 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：

```
pr->draw(Shape::Red) ; //调用Rectangle::draw(Shape::Red)
```

virtual 函数是动态绑定，而缺省参数值却是静态绑定的。意思是你可能会在“调用一个定义于derived class 内的virtual 函数”的同时，却使用base class 为它所指定的缺省参数值：

```
pr->draw( ) ; //调用Rectangle::draw(Shape::Red)!
```

请记住

- 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——你唯一应该覆写的东西——却是动态绑定。

6.4 条款38：通过复合塑模出has-a 或“根据某物实现出”

复合（composition）是类型之间的一种关系，当某种类型的对象内含它种类型的对象，便是这种关系。

例：

```
class Address { ... } ;
class PhoneNumber { ... } ;
class Person {
public:
    ...
private:
    Address address ;
    PhoneNumber voiceNumber ;
} ;
```

本例之中Person 对象由Address, PhoneNumber 构成。在程序员之间复合（composition）这个术语有许多同义词，包括layering（分层），containment（内含），aggregation（聚合）和embedding（内嵌）。

条款32曾说，**“public继承”带有is-a（是一种）的意义。复合意味has-a（有一个）或is-implemented-in-terms-of（根据某物实现出）。**

程序中的对象其实相当于你所塑造的世界中的某些事物，例如人、汽车、一张张视频画面等等。这样的对象属于应用域（application domain）部分。其他对象则纯粹是实现细节上的人工制品，像是缓冲区（buffer）、互斥器（mutexes）、查找树（search trees）等等。这些对象相当于你的软件的实现域（implementation domain）。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is-implemented-in-terms-of 的关系。

6.5 条款40：明智而审慎地使用多重继承

使用virtual 继承的那些classes 所产生的对象往往比使用non-virtual 继承的兄弟们体积大，访问virtual base classes 的成员变量时，也比访问non-virtual base classes 的成员变量速度慢。总之，你得为virtual 继承付出代价。

支持“virtual base classes 初始化”的规则比起non-virtual bases 的情况远为复杂且不直观。virtual base 的初始化责任是由继承体系中的最底层（most derived）class 负责，中间层的classes 对其virtual bases 的初始化都将被屏蔽。也就是说当一个新的derived class 加入到继承体系的底层时，它必须承担其virtual bases（不论直接或间接）的初始化责任。

在产生一个新的derived class 对象时，所有virtual bases 的构造函数总是先于所有non-virtual bases 的构造函数被调用。

我对virtual base classes（亦相当于对virtual 继承）的忠告是:

1. 非必要不使用virtual bases 。
2. 如果你必须使用virtual base classes，尽可能避免在其中放置数据。

如果你有个单一继承的设计方案，而它大约等价于一个多重继承设计方案，那么单一继承设计方案几乎一定比较受欢迎。

请记住

- 多重继承比单一继承复杂。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（及赋值）复杂度等等成本。如果virtual base classes 不带任何数据，将是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。

第七章　模版和泛型编程

7.1　条款42：了解typename的双重意义

声明template参数时，不论使用关键字class或typename，意义完全相同。下面两个template声明式完全相同：

```
template<class T> class Widget; // 使用“class”
template<typename T> class Widget; // 使用“typename”
```

template内出现的名称如果相依于某个template参数，称之为**从属名称（dependent names）**。如果从属名称在class 内呈嵌套状，我们称它为**嵌套从属名称（nested dependent name）**。

嵌套从属名称有可能导致解析（parsing）困难。

例：

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

看起来好像我们声明x为一个local变量，它是个指针，指向一个C::const_iterator。但它之所以被那么认为，只因为我们“已经知道”C::const_iterator是个类型。如果C::const_iterator不是个类型呢？如果C有个static成员变量而碰巧被命名为const_iterator，或如果x碰巧是个global变量名称呢？那样的话上述代码就不再是声明一个local变量，而是一个相乘动作：C::const_iterator乘以x。

C++有个规则可以解析（resolve）此一歧义状态：**如果解析器在template中遭遇一个嵌套从属名称，它便假设这名称不是个类型，除非你告诉它是。**所以缺省情况下嵌套从属名称不是类型。此规则有个例外，稍后我会提到。所以下面的C++代码不是有效的：

```
template<typename C> //这是无效的C++代码
void print2nd(const C& container)
{
    if( container.size() >= 2 ) {
        C::const_iterator iter(container.begin()); //这个名称被假设为非类型
        ...
```

iter声明式只有在C::const_iterator是个类型时才合理，但我们并没有告诉C++说它是，于是C++假设它不是。所以我们必须告诉C++说C::const_iterator是个类型。只要紧邻它之前放置关键字typename即可：

```
template<typename C> //这是合法的C++代码
void print2nd(const C& container)
{
    if( container.size() >= 2 ) {
        typename C::const_iterator iter(container.begin());
        ...
    }
}
```

typename只被用来验明嵌套从属类型名称；其他名称不该用它。

“typename必须作为嵌套从属类型名称的前缀词”这一规则的例外是，typename不可以出现在base classes list内的嵌套从属类型名称之前，也不可在member initialization list（成员初值列）中作为base class修饰符。例如：

```
template<typename T>
class Derived: public Base<Y>::Nested { //base class list中不允许“typename”。
public:
    explicit Derived(int x)
    : Base<T>::Nested(x) //mem.init.list中不允许“typename”。
    {
        typename Base<T>::Nested temp; //嵌套从属类型名称，
                                       //作为一个base class修饰符需加上typename。
        ...
    }
    ...
};
```

请记住

- 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（基类列）或member initialization list（成员初值列）内以它作为base class修饰符。

7.2 条款44：将与参数无关的代码抽离templates

在大多数平台上，所有指针类型都有相同的二进制表述，因此凡templates持有指针者（例如list<int*>，list<const int*>等等）往往应该对每一个成员函数使用唯一一份底层实现。这很具代表性地意味，如果你实现某些成员函数而它们操作强型指针（strongly typed pointers，即T*），你应该令它们调用另一个操作无类型指针（untyped pointers，

即void*）的函数，由后者完成实际工作。

请记住

- Templates生成多个classes和多个函数，所以任何template代码都不该与某个造成膨胀的template参数产生相依关系。
- 因非类型模版参数（non-type template parameters）而造成的代码膨胀，往往可消除，做法是以函数参数或class成员变量替换template参数。
- 因类型参数（type parameters）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（binary representations）的具现类型（instantiation types）共享实现码。