5

实现

Implementations

大多数情况下，适当提出你的 classes（和 class templates）定义以及 functions（和 function templates）声明，是花费最多心力的两件事。一旦正确完成它们，相应的实现大多直截了当。尽管如此，还是有些东西需要小心。太快定义变量可能造成效率上的拖延；过度使用转型（casts）可能导致代码变慢又难维护，又招来微妙难解的错误；返回对象“内部数据之号码牌（handles）”可能会破坏封装并留给客户虚吊号码牌（dangling handles）；未考虑异常带来的冲击则可能导致资源泄漏和数据败坏；过度热心地 inlining 可能引起代码膨胀；过度耦合（coupling）则可能导致让人不满意的冗长建置时间（build times）。

所有这些问题都可避免。本章逐一解释各种做法。

条款 26：尽可能延后变量定义式的出现时间

Postpone variable definitions as long as possible.

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。即使这个变量最终并未被使用，仍需耗费这些成本，所以你应该尽可能避免这种情形。

或许你会认为，你不可能定义一个不使用的变量，但话不要说得太早！考虑下面这个函数，它计算通行密码的加密版本而后返回，前提是密码够长。如果密码太短，函数会丢出一个异常，类型为 `logic_error`（定义于 C++ 标准程序库，见条款 54）：

```
//这个函数过早定义变量 "encrypted"
std::string encryptPassword(const std::string& password)
{
    using namespace std;
    string encrypted;
    if (password.length() < MinimumPasswordLength) {
        throw logic_error("Password is too short");
    }
    ...                    //必要动作，俾能将一个加密后的密码
                           //置入变量 encrypted 内
    return encrypted;
}
```

对象 encrypted 在此函数中并非完全未被使用，但如果有个异常被丢出，它就真的没被使用。也就是说如果函数 encryptPassword 丢出异常，你仍得付出 encrypted 的构造成本和析构成本。所以最好延后 encrypted 的定义式，直到确实需要它：

```
//这个函数延后 "encrypted" 的定义，直到真正需要它
std::string encryptPassword(const std::string& password)
{
    using namespace std;
    if (password.length() < MinimumPasswordLength) {
        throw logic_error("Password is too short");
    }
    string encrypted;
    ...                    //必要动作，俾能将一个加密后的密码
                           //置入变量 encrypted 内
    return encrypted;
}
```

但是这段代码仍然不够秾纤合度，因为 encrypted 虽获定义却无任何实参作为初值。这意味调用的是其 ***default*** 构造函数。许多时候你该对对象做的第一次事就是给它个值，通常是通过一个赋值动作达成。条款 4 曾解释为什么“通过 ***default*** 构造函数构造出一个对象然后对它赋值”比“直接在构造时指定初值”效率差。那个分析当然也适用于此。举个例子，假设 encryptPassword 的艰难部分在以下函数中进行：

```
void encrypt(std::string& s); //在其中的适当地点对 s 加密
```

于是 encryptPassword 可实现如下，虽然还不算是最好的做法：

```
//这个函数延后 "encrypted" 的定义，直到需要它为止。
//但此函数仍然有着不该有的效率低落。
std::string encryptPassword(const std::string& password)
{
    ...                          //检查 length，如前。
    std::string encrypted;       //default-construct encrypted
    encrypted = password;        //赋值给 encrypted
    encrypt(encrypted);
    return encrypted;
}
```

更受欢迎的做法是以 password 作为 encrypted 的初值，跳过毫无意义的 ***default*** 构造过程：

```
//终于，这是定义并初始化 encrypted 的最佳做法
std::string encryptPassword(const std::string& password)
{
    ...                                    //检查长度。
    std::string encrypted(password);       //通过 copy 构造函数
                                           //定义并初始化。
    encrypt(encrypted);
    return encrypted;
}
```

这让我们联想起本条款所谓“尽可能延后”的真正意义。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。更深一层说，以“具明显意义之初值”将变量初始化，还可以附带说明变量的目的。

“但循环怎么办？”你可能会感到疑惑。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？也就是说下面左右两个一般性结构，哪一个比较好？

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

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

这里我把对象的类型从 string 改为 Widget，以免造成读者对于“对象执行构造、析构、或赋值动作所需的成本”有任何特殊偏见。

在 Widget 函数内部，以上两种写法的成本如下：

- 做法 A：1 个构造函数 +1 个析构函数 +n 个赋值操作
- 做法 B：n 个构造函数 +n 个析构函数

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。尤其当 n 值很大的时候。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，有时那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。

条款 27：尽量少做转型动作

Minimize casting.

C++ 规则的设计目标之一是，保证“类型错误”绝不可能发生。理论上如果你的程序很“干净地”通过编译，就表示它并不企图在任何对象身上执行任何不安全、无意义、愚蠢荒谬的操作。这是一个极具价值的保证，可别草率地放弃它。

不幸的是，转型（casts）破坏了类型系统（type system）。那可能导致任何种类的麻烦，有些容易辨识，有些非常隐晦。如果你来自 C, Java 或 C# 阵营，请特别注意，因为那些语言中的转型（casting）比较必要而无法避免，也比较不危险（与 C++ 相较）。但 C++ 不是 C，也不是 Java 或 C#。在 C++ 中转型是一个你会想带着极大尊重去亲近的一个特性。

让我们首先回顾转型语法，因为通常有三种不同的形式，可写出相同的转型动作。C 风格的转型动作看起来像这样：

```
(T) expression          //将 expression 转型为 T
```

函数风格的转型动作看起来像这样：

```
T(expression)           //将 expression 转型为 T
```

两种形式并无差别，纯粹只是小括号的摆放位置不同而已。我称此二种形式为“旧式转型”（*old-style casts*）。

C++ 还提供四种新式转型（常常被称为 *new-style* 或 *C++-style casts*）：

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

各有不同的目的：

- const_cast 通常被用来将对象的常量性转除（cast away the constness）。它也是唯一有此能力的 C++-style 转型操作符。
- dynamic_cast 主要用来执行“安全向下转型”（safe downcasting），也就是用来决定某对象是否归属继承体系中的某个类型。它是唯一无法由旧式语法执行的动作，也是唯一可能耗费重大运行成本的转型动作（稍后细谈）。
- reinterpret_cast 意图执行低级转型，实际动作（及结果）可能取决于编译器，这也就表示它不可移植。例如将一个 *pointer to* int 转型为一个 int。这一类转型在低级代码以外很少见。本书只使用一次，那是在讨论如何针对原始内存（raw memory）写出一个调试用的分配器（debugging allocator）时，见条款 50。
- static_cast 用来强迫隐式转换（implicit conversions），例如将 non-const 对象转为 const 对象（就像条款 3 所为），或将 int 转为 double 等等。它也可以用来执行上述多种转换的反向转换，例如将 void* 指针转为 typed 指针，将 *pointer-to*-base 转为 *pointer-to*-derived。但它无法将 const 转为 non-const——这个只有 const_cast 才办得到。

旧式转型仍然合法，但新式转型较受欢迎。原因是：第一，它们很容易在代码中被辨识出来（不论是人工辨识或使用工具如 grep），因而得以简化“找出类型系统在哪个地点被破坏”的过程。第二，各转型动作的目标愈窄化，编译器愈可能诊断出错误的运用。举个例子，如果你打算将常量性（constness）去掉，除非使用新式转型中的 const_cast 否则无法通过编译。

我唯一使用旧式转型的时机是，当我要调用一个 explicit 构造函数将一个对象传递给一个函数时。例如：

```
class Widget {
public:
   explicit Widget(int size);
   ...
};
void doSomeWork(const Widget& w);
doSomeWork(Widget(15));                  //以一个 int 加上“函数风格”的
                                         //转型动作创建一个 Widget。
doSomeWork(static_cast<Widget>(15));     //以一个 int 加上“C++ 风格”的
                                         //转型动作创建一个 Widget。
```

从某个角度来说，蓄意的“对象生成”动作感觉不怎么像“转型”，所以我很可能使用函数风格的转型动作而不使用 static_cast。但我要再说一次，当我们写下一段日后出错导致“核心倾印”（core dump）的代码时，撰写之时我们往往“觉得”通情达理，所以或许最好是忽略你的感觉，始终理智地使用新式转型。

许多程序员相信，转型其实什么都没做，只是告诉编译器把某种类型视为另一种类型。这是错误的观念。任何一个类型转换（不论是通过转型操作而进行的显式转换，或通过编译器完成的隐式转换）往往真的令编译器编译出运行期间执行的码。例如在这段程序中：

```
int x, y;
...
double d = static_cast<double>(x)/y;        //x 除以 y，使用浮点数除法
```

将 int x 转型为 double 几乎肯定会产生一些代码，因为在大部分计算器体系结构中，int 的底层表述不同于 double 的底层表述。这或许不会让你惊讶，但下面这个例子就有可能让你稍微睁大眼睛了：

```
class Base { ... };
class Derived: public Base { ... };
Derived d;
Base* pb = &d;                           //隐喻地将 Derived* 转换为 Base*
```

这里我们不过是建立一个 base class 指针指向一个 derived class 对象，但有时候上述的两个指针值并不相同。这种情况下会有个偏移量（offset）在运行期被施行于 Derived* 指针身上，用以取得正确的 Base* 指针值。

上个例子表明，单一对象（例如一个类型为 Derived 的对象）可能拥有一个以上的地址（例如“以 Base* 指向它”时的地址和“以 Derived* 指向它”时的地址。C 不可能发生这种事，Java 不可能发生这种事，C# 也不可能发生这种事。但 C++ 可能！实际上一旦使用多重继承，这事几乎一直发生着。即使在单一继承中也可能发

生。虽然这还有其他意涵，但至少意味你通常应该避免做出“对象在 C++ 中如何如何布局”的假设。当然更不该以此假设为基础执行任何转型动作。例如，将对象地址转型为 char*指针然后在它们身上进行指针算术，几乎总是会导致无定义（不明确）行为。

但请注意，我说的是有时候需要一个偏移量。对象的布局方式和它们的地址计算方式随编译器的不同而不同，那意味“由于知道对象如何布局”而设计的转型，在某一平台行得通，在其他平台并不一定行得通。这个世界有许多悲惨的程序员，他们历经千辛万苦才学到这堂课。

另一件关于转型的有趣事情是：我们很容易写出某些似是而非的代码（在其他语言中也许真是对的）。例如许多应用框架（application frameworks）都要求 derived classes 内的 virtual 函数代码的第一个动作就先调用 base class 的对应函数。假设我们有个 Window base class 和一个 SpecialWindow derived class，两者都定义了 virtual 函数 onResize。进一步假设 SpecialWindow 的 onResize 函数被要求首先调用 Window 的 onResize。下面是实现方式之一，它看起来对，但实际上错：

```
class Window {                                  //base class
public:
   virtual void onResize( ) { ... }             //base onResize 实现代码
   ...
};
class SpecialWindow: public Window {            //derived class
public:
  virtual void onResize( ) {                    //derived onResize 实现代码
    static_cast<Window>(*this).onResize();      //将*this 转型为 Window，
                                                //然后调用其 onResize;
                                                //这不可行!
        ...  //这里进行 SpecialWindow 专属行为。
    }
    ...
};
```

我在代码中强调了转型动作（那是个新式转型，但若使用旧式转型也不能改变以下事实）。一如你所预期，这段程序将*this 转型为 Window，对函数 onResize 的调用也因此调用了 Window::onResize。但恐怕你没想到，它调用的并不是当前对象上的函数，而是稍早转型动作所建立的一个“*this 对象之 base class 成分”的暂时副本身上的 onResize！（译注：函数就是函数，成员函数只有一份，“调用起哪个对象身上的函数”有什么关系呢？关键在于成员函数都有个隐藏的 this 指

针，会因此影响成员函数操作的数据。）再说一次，上述代码并非在当前对象身上调用 Window::onResize 之后又在该对象身上执行 SpecialWindow 专属动作。不，它是在“当前对象之 base class 成分”的副本上调用 Window::onResize，然后在当前对象身上执行 SpecialWindow 专属动作。如果 Window::onResize 修改了对象内容（不能说没有可能性，因为 onResize 是个 non-const 成员函数），当前对象其实没被改动，改动的是副本。然而 SpecialWindow::onResize 内如果也修改对象，当前对象真的会被改动。这使当前对象进入一种“伤残”状态：其 base class 成分的更改没有落实，而 derived class 成分的更改倒是落实了。

解决之道是拿掉转型动作，代之以你真正想说的话。你并不想哄骗编译器将 *this 视为一个 base class 对象，你只是想调用 base class 版本的 onResize 函数，令它作用于当前对象身上。所以请这么写：

```
class SpecialWindow: public Window {
public:
    virtual void onResize( ) {
        Window::onResize();     //调用 Window::onResize 作用于*this 身上
        ...
    }
    ...
};
```

这个例子也说明，如果你发现你自己打算转型，那活脱是个警告信号：你可能正将局面发展至错误的方向上。如果你用的是 dynamic_cast 更是如此。

在探究 dynamic_cast 设计意涵之前，值得注意的是，dynamic_cast 的许多实现版本执行速度相当慢。例如至少有一个很普遍的实现版本基于“class 名称之字符串比较”，如果你在四层深的单继承体系内的某个对象身上执行 dynamic_cast，刚才说的那个实现版本所提供的每一次 dynamic_cast 可能会耗用多达四次的 strcmp 调用，用以比较 class 名称。深度继承或多重继承的成本更高！某些实现版本这样做有其原因（它们必须支持动态连接）。然而我还是要强调，除了对一般转型保持机敏与猜疑，更应该在注重效率的代码中对 dynamic_casts 保持机敏与猜疑。

之所以需要 dynamic_cast，通常是因为你想在一个你认定为 derived class 对象身上执行 derived class 操作函数，但你的手上却只有一个“指向 base”的 pointer 或 reference，你只能靠它们来处理对象。有两个一般性做法可以避免这个问题。

第一，使用容器并在其中存储直接指向 derived class 对象的指针（通常是智能指针，见条款 13），如此便消除了“通过 base class 接口处理对象”的需要。假设先前的 Window/ SpecialWindow 继承体系中只有 SpecialWindows 才支持闪烁效果，试着不要这样做：

```
class Window { ... };
class SpecialWindow: public Window {
public:
   void blink();
   ...
};
typedef                                                    //关于 tr1::shared_ptr
std::vector<std::tr1::shared_ptr<Window> > VPW; //见条款 13.
VPW winPtrs;
...
for (VPW::iterator iter = winPtrs.begin( );      //不希望使用
     iter != winPtrs.end();++iter) {              //dynamic_cast.
 if (SpecialWindow * psw = dynamic_cast<SpecialWindow * >(iter->get()))
     psw->blink();
}
```

应该改而这样做：

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;
VPSW winPtrs;
...
for ( VPSW::iterator iter = winPtrs.begin();     //这样写比较好，
      iter != winPtrs.end();                      //不使用 dynamic_cast
      ++iter)
(*iter)->blink();
```

当然啦，这种做法使你无法在同一个容器内存储指针“指向所有可能之各种 Window 派生类”。如果真要处理多种窗口类型，你可能需要多个容器，它们都必须具备类型安全性（type-safe）。

另一种做法可让你通过 base class 接口处理“所有可能之各种 Window 派生类”，那就是在 base class 内提供 virtual 函数做你想对各个 Window 派生类做的事。举个例子，虽然只有 SpecialWindows 可以闪烁，但或许将闪烁函数声明于 base class 内并提供一份“什么也没做”的缺省实现码是有意义的：

```
class Window {
public:
   virtual void blink() { }            //缺省实现代码“什么也没做”;
      ...                              //条款34告诉你为什么
};                                     //  缺省实现代码可能是个馊主意。

class SpecialWindow: public Window {
public:
   virtual void blink() { ... };       //在此class内,
   ...                                 //blink做某些事。
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
VPW winPtrs;                           //容器,内含指针,指向
...                                    //  所有可能的Window类型。
for (VPW::iterator iter = winPtrs.begin( );
     iter != winPtrs.end();
     ++iter)                           //注意,这里没有
  (*iter)->blink();                    //  dynamic_cast。
```

不论哪一种写法——“使用类型安全容器”或“将virtual函数往继承体系上方移动”——都并非放之四海皆准，但在许多情况下它们都提供一个可行的dynamic_cast替代方案。当它们有此功效时，你应该欣然拥抱它们。

绝对必须避免的一件事是所谓的“连串（cascading）dynamic_casts”，也就是看起来像这样的东西：

```
class Window { ... };
...                                    //derived classes定义在这里
typedef std::vector<std::tr1::shared_ptr<Window> > VPW;
VPW winPtrs;
...
for (VPW::iterator iter = winPtrs.begin( );
     iter != winPtrs.end(); ++iter)
{
   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())) { ... }
   ...
}
```

这样产生出来的代码又大又慢，而且基础不稳，因为每次 Window class 继承体系一有改变，所有这一类代码都必须再次检阅看看是否需要修改。例如一旦加入新的 derived class，或许上述连串判断中需要加入新的条件分支。这样的代码应该总是以某些“基于 virtual 函数调用”的东西取而代之。

优良的 C++ 代码很少使用转型，但若说要完全摆脱它们又太过不切实际。例如 p.118 从 int 转型为 double 就是转型的一个通情达理的使用，虽然它并非绝对必要（那段代码可以重新写过，声明一个类型为 double 的新变量并以 x 值初始化）。就像面对众多蹊跷可疑的构造函数一样，我们应该尽可能隔离转型动作，通常是把它隐藏在某个函数内，函数的接口会保护调用者不受函数内部任何肮脏龌龊的动作影响。

请记住

- 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设计。
- 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，而不需将转型放进他们自己的代码内。
- 宁可使用 C++-style（新式）转型，不要使用旧式转型。前者很容易辨识出来，而且也比较有着分门别类的职掌。

条款 28：避免返回 handles 指向对象内部成分

Avoid returning "handles" to object internals.

假设你的程序涉及矩形。每个矩形由其左上角和右下角表示。为了让一个 Rectangle 对象尽可能小，你可能会决定不把定义矩形的这些点存放在 Rectangle 对象内，而是放在一个辅助的 struct 内再让 Rectangle 去指它：

```
class Point {                    //这个 class 用来表述“点”
public:
   Point(int x, int y);
   ...
   void setX(int newVal);
   void setY(int newVal);
   ...
};
```

```
struct RectData {          //这些“点”数据用来表现一个矩形
   Point ulhc;             //ulhc = "upper left-hand corner"（左上角）
   Point lrhc;             //lrhc = "lower right-hand corner"（右下角）
};
class Rectangle {
   ...
private:
   std::tr1::shared_ptr<RectData> pData;   //关于 tr1::shared_ptr,
};                                         //见条款 13
```

Rectangle 的客户必须能够计算 Rectangle 的范围，所以这个 class 提供 upperLeft 函数和 lowerRight 函数。Point 是个用户自定义类型，所以根据条款 20 给我们的忠告（它说以 ***by reference*** 方式传递用户自定义类型往往比以 ***by value*** 方式传递更高效），这些函数于是返回 references，代表底层的 Point 对象：

```
class Rectangle {
public:
   ...
   Point& upperLeft( ) const { return pData->ulhc; }
   Point& lowerRight( ) const { return pData->lrhc; }
   ...
};
```

这样的设计可通过编译，但却是错误的。实际上它是自我矛盾的。一方面 upperLeft 和 lowerRight 被声明为 const 成员函数，因为它们的目的只是为了提供客户一个得知 Rectangle 相关坐标点的方法，而不是让客户修改 Rectangle（见条款 3）。另一方面两个函数却都返回 references 指向 private 内部数据，调用者于是可通过这些 references 更改内部数据！例如：

```
Point coord1(0, 0);
Point coord2(100, 100);
const Rectangle rec(coord1, coord2);        //rec 是个 const 矩形，
                                            //  从(0,0)到(100,100)
rec.upperLeft( ).setX(50);                  //现在 rec 却变成
                                            //  从(50,0)到(100,100)
```

这里请注意，upperLeft 的调用者能够使用被返回的 reference（指向 rec 内部的 Point 成员变量）来更改成员。但 rec 其实应该是不可变的（const）！

这立刻带给我们两个教训。第一，成员变量的封装性最多只等于“返回其 reference”的函数的访问级别。本例之中虽然 ulhc 和 lrhc 都被声明为 private，它们实际上却是 public，因为 public 函数 upperLeft 和 lowerRight 传出了它们的

references。第二，如果 const 成员函数传出一个 reference，后者所指数据与对象自身有关联，而它又被存储于对象之外，那么这个函数的调用者可以修改那笔数据。这正是 bitwise constness 的一个附带结果，见条款 3。

上面我们所说的每件事情都是由于“成员函数返回 references”。如果它们返回的是指针或迭代器，相同的情况还是发生，原因也相同。References、指针和迭代器统统都是所谓的 *handles*（号码牌，用来取得某个对象），而返回一个“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。同时，一如稍早所见，它也可能导致“虽然调用 const 成员函数却造成对象状态被更改”。

通常我们认为，对象的“内部”就是指它的成员变量，但其实不被公开使用的成员函数（也就是被声明为 protected 或 private 者）也是对象“内部”的一部分。因此也应该留心不要返回它们的 handles。这意味你绝对不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。如果你那么做，后者的实际访问级别就会提高如同前者（访问级别较高者），因为客户可以取得一个指针指向那个“访问级别较低”的函数，然后通过那个指针调用它。

然而“返回指针指向某个成员函数”的情况毕竟不多见，所以让我们把注意力收回，专注于 Rectangle class 和它的 upperLeft 以及 lowerRight 成员函数。我们在这些函数身上遭遇的两个问题可以轻松去除，只要对它们的返回类型加上 const 即可：

```
class Rectangle {
public:
    ...
    const Point& upperLeft( ) const { return pData->ulhc; }
    const Point& lowerRight( ) const { return pData->lrhc; }
    ...
};
```

有了这样的改变，客户可以读取矩形的 Points，但不能涂写它们。这意味当初声明 upperLeft 和 upperRight 为 const 不再是个谎言，因为它们不再允许客户更改对象状态。至于封装问题，我们总是愿意让客户看到 Rectangle 的外围 Points，所以这里是蓄意放松封装。更重要的是这是个有限度的放松：这些函数只让渡读取权。涂写权仍然是被禁止的。

但即使如此，upperLeft 和 lowerRight 还是返回了“代表对象内部”的 handles，有可能在其他场合带来问题。更明确地说，它可能导致 *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。例如某个函数返回 GUI 对象的外框(bounding box)，这个外框采用矩形形式：

```
class GUIObject { ... };
const Rectangle                          //以 by value 方式返回一个矩形
 boundingBox(const GUIObject& obj);      //条款 3 谈过为什么返回类型是 const
```

现在，客户有可能这么使用这个函数：

```
GUIObject* pgo;                          //让 pgo 指向某个 GUIObject
...
const Point* pUpperLeft =                //取得一个指针指向外框左上点
 &(boundingBox(*pgo).upperLeft());
```

对 boundingBox 的调用获得一个新的、暂时的 Rectangle 对象。这个对象没有名称，所以我们权且称它为 *temp*。随后 upperLeft 作用于 *temp* 身上，返回一个 reference 指向 *temp* 的一个内部成分，更具体地说是指向一个用以标示 *temp* 的 Points。于是 pUpperLeft 指向那个 Point 对象。目前为止一切还好，但故事尚未结束，因为在那个语句结束之后，boundingBox 的返回值，也就是我们所说的 *temp*，将被销毁，而那间接导致 *temp* 内的 Points 析构。最终导致 pUpperLeft 指向一个不再存在的对象；也就是说一旦产出 pUpperLeft 的那个语句结束，pUpperLeft 也就变成空悬、虚吊（*dangling*）！

这就是为什么函数如果“返回一个 handle 代表对象内部成分”总是危险的原因。不论这所谓的 handle 是个指针或迭代器或 reference，也不论这个 handle 是否为 const，也不论那个返回 handle 的成员函数是否为 const。这里的唯一关键是，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[]s 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、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。

条款 29：为“异常安全”而努力是值得的

Strive for exception-safe code.

异常安全性（Exception safety）有几分像是……呃……怀孕。但等等，在我们完成求偶之前，实在无法确实地谈论生育。

假设有个 class 用来表现夹带背景图案的 GUI 菜单。这个 class 希望用于多线程环境，所以它有个互斥器（mutex）作为并发控制（concurrency control）之用：

```
class PrettyMenu {
public:
   ...
   void changeBackground(std::istream& imgSrc);      //改变背景图像
   ...
private:
   Mutex mutex;                                      //互斥器
   Image* bgImage;                                   //目前的背景图像
   int imageChanges;                                 //背景图像被改变的次数
};
```

下面是 PrettyMenu 的 changeBackground 函数的一个可能实现：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
   lock(&mutex);                                     //取得互斥器（见条款 14）
   delete bgImage;                                   //摆脱旧的背景图像
   ++imageChanges;                                   //修改图像变更次数
   bgImage = new Image(imgSrc);                      //安装新的背景图像
   unlock(&mutex);                                   //释放互斥器
}
```

从“异常安全性”的观点来看，这个函数很糟。“异常安全”有两个条件，而这个函数没有满足其中任何一个条件。

当异常被抛出时，带有异常安全性的函数会：

- **不泄漏任何资源**。上述代码没有做到这一点，因为一旦 "new Image(imgSrc)" 导致异常，对 unlock 的调用就绝不会执行，于是互斥器就永远被把持住了。
- **不允许数据败坏**。如果 "new Image(imgSrc)" 抛出异常，bgImage 就是指向一个已被删除的对象，imageChanges 也已被累加，而其实并没有新的图像被成功安装起来。（但从另一个角度说，旧图像已被消除，所以你可能会争辩说图像

还是“改变了”）。

解决资源泄漏的问题很容易，因为条款 13 讨论过如何以对象管理资源，而条款 14 也导入了 Lock class 作为一种“确保互斥器被及时释放”的方法：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
   Lock ml(&mutex);          //来自条款 14：获得互斥器并确保它稍后被释放
   delete bgImage;
   ++imageChanges;
   bgImage = new Image(imgSrc);
}
```

关于“资源管理类”（resource management classes）如 Lock 者，一个最棒的事情是，它们通常使函数更短。你看，不再需要调用 unlock 了不是吗？有个一般性规则是这么说的：较少的码就是较好的码，因为出错机会比较少，而且一旦有所改变，被误解的机会也比较少。

把资源泄漏抛诸脑后，现在我们可以专注解决数据的败坏了。此刻我们需要做个抉择，但是在我们能够抉择之前，必须先面对一些用来定义选项的术语。

异常安全函数（Exception-safe functions）提供以下三个保证之一：

- **基本承诺**：如果异常被抛出，程序内的任何事物仍然保持在有效状态下。没有任何对象或数据结构会因此而败坏，所有对象都处于一种内部前后一致的状态（例如所有的 class 约束条件都继续获得满足）。然而程序的现实状态（exact state）恐怕不可预料。举个例子，我们可以撰写 changeBackground 使得一旦有异常被抛出时，PrettyMenu 对象可以继续拥有原背景图像，或是令它拥有某个缺省背景图像，但客户无法预期哪一种情况。如果想知道，他们恐怕必须调用某个成员函数以得知当时的背景图像是什么。
- **强烈保证**：如果异常被抛出，程序状态不改变。调用这样的函数需有这样的认知：如果函数成功，就是完全成功，如果函数失败，程序会回复到“调用函数之前”的状态。

和这种提供强烈保证的函数共事，比和刚才说的那种只提供基本承诺的函数共事，容易多了，因为在调用一个提供强烈保证的函数后，程序状态只有两种可能：如预期般地到达函数成功执行后的状态，或回到函数被调用前的状态。与此成对比的是，如果调用一个只提供基本承诺的函数，而真的出现异常，程序有可能处于任何状态——只要那是个合法状态。

- **不抛掷（nothrow）保证**，承诺绝不抛出异常，因为它们总是能够完成它们原先承诺的功能。作用于内置类型（例如 ints，指针等等）身上的所有操作都提供 nothrow 保证。这是异常安全码中一个必不可少的关键基础材料。

如果我们假设，函数带着“空白的异常明细”（empty exception specification）者必为 nothrow 函数，似乎合情合理，其实不尽然。举个例子，考虑以下函数：

```
int doSomething() throw();     //注意“空白的异常明细”
                               //（empty exception spec）
```

这并不是说 doSomething 绝不会抛出异常，而是说如果 doSomething 抛出异常，将是严重错误，会有你意想不到的函数被调用[1]。实际上 doSomething 也许完全没有提供任何异常保证。函数的声明式（包括其异常明细——如果有的话）并不能够告诉你是否它是正确的、可移植的或高效的，也不能够告诉你它是否提供任何异常安全性保证。所有那些性质都由函数的实现决定，无关乎声明。

异常安全码（Exception-safe code）必须提供上述三种保证之一。如果它不这样做，它就不具备异常安全性。因此，我们的抉择是，该为我们所写的每一个函数提供哪一种保证？除非面对不具异常安全性的传统代码（我将在本条款末尾讨论那种情况），否则你应该只在一种情况下才不提供任何异常安全保证：你那“天才班”需求分析团队确认你的应用程序有“泄漏资源”并“在执行过程中带着败坏数据”的需要。

一般而言你应该会想提供可实施之最强烈保证。从异常安全性的观点视之，nothrow 函数很棒，但我们很难在 C part of C++ 领域中完全没有调用任何一个可能抛出异常的函数。任何使用动态内存的东西（例如所有 STL 容器）如果无法找到足

[1] 关于所谓“意想不到的函数”，请咨询你最常用的搜索引擎或广泛的 C++ 文件。搜寻 set_unexpected 或许会得到较好的结果；此函数用来指定那个“意想不到的函数”。

够内存以满足需求，通常便会抛出一个 bad_alloc 异常（见条款 49）。是的，可能的话请提供 nothrow 保证，但对大部分函数而言，抉择往往落在基本保证和强烈保证之间。

对 changeBackground 而言，提供强烈保证几乎不困难。首先改变 PrettyMenu 的 bgImage 成员变量的类型，从一个类型为 Image* 的内置指针改为一个“用于资源管理”的智能指针（见条款 13）。坦白说，这个好构想纯粹只是帮助我们防止资源泄漏。它对“强烈之异常安全保证”的帮助仅仅只是强化了条款 13 的论点：以对象（例如智能指针）管理资源是良好设计的根本。以下代码中我使用 tr1::shared_ptr，因为它比 auto_ptr 更直观的行为使它更受欢迎。

第二，我们重新排列 changeBackground 内的语句次序，使得在更换图像之后才累加 imageChanges。一般而言这是个好策略：不要为了表示某件事情发生而改变对象状态，除非那件事情真的发生了。

下面是结果：

```
class PrettyMenu {
    ...
    std::tr1::shared_ptr<Image> bgImage;
    ...
};
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    Lock ml(&mutex);
    bgImage.reset(new Image(imgSrc));  //以"new Image" 的执行结果
                                       //设定bgImage 内部指针
    ++imageChanges;
}
```

注意，这里不再需要手动 delete 旧图像，因为这个动作已经由智能指针内部处理掉了。此外，删除动作只发生在新图像被成功创建之后。更正确地说，tr1::shared_ptr::reset 函数只有在其参数（也就是 "new Image(imgSrc)"的执行结果）被成功生成之后才会被调用。delete 只在 reset 函数内被使用，所以如果从未进入那个函数也就绝对不会使用 delete。也请注意，以对象（tr1::shared_ptr）管理资源（这里是动态分配而得的 Image）再次缩减了 changeBackground 的长度。

如我稍早所言，这两个改变几乎足够让 changeBackground 提供强烈的异常安全保证。美中不足的是参数 imgSrc。如果 Image 构造函数抛出异常，有可能输入

流（input stream）的读取记号（read marker）已被移走，而这样的搬移对程序其余部分是一种可见的状态改变。所以 changeBackground 在解决这个问题之前只提供基本的异常安全保证。

然而，让我们把它放在一旁，佯装 changeBackground 的确提供了强烈保证（我有信心你可以想出个什么办法顺利过渡，或许你可以改变它的参数类型，从 istream 改为一个内含图像数据的文件名称）。有个一般化的设计策略很典型地会导致强烈保证，很值得熟悉它。这个策略被称为 **copy and swap**。原则很简单：为你打算修改的对象（原件）做出一份副本，然后在那副本身上做一切必要修改。若有任何修改动作抛出异常，原对象仍保持未改变状态。待所有改变都成功后，再将修改过的那个副本和原对象在一个不抛出异常的操作中置换（swap）。

实现上通常是将所有“隶属对象的数据”从原对象放进另一个对象内，然后赋予原对象一个指针，指向那个所谓的实现对象（implementation object，即副本）。这种手法常被称为 **pimpl idiom**，条款 31 详细描述了它。对 PrettyMenu 而言，典型写法如下：

```
struct PMImpl {                                  //PMImpl = "PrettyMenu Impl";
   std::tr1::shared_ptr<Image> bgImage;          //稍后说明为什么它是个 struct
   int imageChanges;
};

class PrettyMenu {
   ...
private:
   Mutex mutex;
   std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
   using std::swap;                              //见条款 25
   Lock ml(&mutex);                              //获得 mutex 的副本数据
   std::tr1::shared_ptr<PMImpl>
      pNew(new PMImpl(*pImpl));

   pNew->bgImage.reset(new Image(imgSrc)); //修改副本
   ++pNew->imageChanges;

   swap(pImpl, pNew);                            //置换（swap）数据，释放 mutex
}
```

此例之中我选择让 PMImpl 成为一个 struct 而不是一个 class，这是因为 PrettyMenu 的数据封装性已经由于"pImpl 是 private"而获得了保证。如果令 PMImpl 为一个 class，虽然一样好，有时候却不太方便（但也保持了面向对象纯度）。如果你要，也可以将 PMImpl 嵌套于 PrettyMenu 内，但打包问题（packaging，例如"独立撰写异常安全码"）是我们这里所挂虑的事。

"copy-and-swap" 策略是对对象状态做出"全有或全无"改变的一个很好办法，但一般而言它并不保证整个函数有强烈的异常安全性。为了解原因，让我们考虑 changeBackground 的一个抽象概念：someFunc。它使用 copy-and-swap 策略，但函数内还包括对另外两个函数 f1 和 f2 的调用：

```
void someFunc()
{
    ...                    //对 local 状态做一份副本
    f1();
    f2();
    ...                    //将修改后的状态置换过来
}
```

很显然，如果 f1 或 f2 的异常安全性比"强烈保证"低，就很难让 someFunc 成为"强烈异常安全"。举个例子，假设 f1 只提供基本保证，那么为了让 someFunc 提供强烈保证，我们必须写出代码获得调用 f1 之前的整个程序状态、捕捉 f1 的所有可能异常、然后恢复原状态。

如果 f1 和 f2 都是"强烈异常安全"，情况并不就此好转。毕竟如果 f1 圆满结束，程序状态在任何方面都可能有所改变，因此如果 f2 随后抛出异常，程序状态和 someFunc 被调用前并不相同，甚至当 f2 没有改变任何东西时也是如此。

问题出在"连带影响"（side effects）。如果函数只操作局部性状态（local state，例如 someFunc 只影响其"调用者对象"的状态），便相对容易地提供强烈保证。但是当函数对"非局部性数据"（non-local data）有连带影响时，提供强烈保证就困难得多。举个例子，如果调用 f1 带来的影响是某个数据库被改动了，那就很难让 someFunc 具备强烈安全性。一般而言在"数据库修改动作"送出之后，没有什么做法可以取消并恢复数据库旧观，因为数据库的其他客户可能已经看到了这一笔新数据。

这些议题想必会阻止你为函数提供强烈保证——即使你想那么做。另一个主题

是效率。copy-and-swap 的关键在于“修改对象数据的副本，然后在一个不抛异常的函数中将修改后的数据和原件置换”，因此必须为每一个即将被改动的对象做出一个副本，那得耗用你可能无法（或无意愿）供应的时间和空间。是的，大家都希望提供“强烈保证”；当它可被实现时你的确应该提供它，但“强烈保证”并非在任何时刻都显得实际。

当“强烈保证”不切实际时，你就必须提供“基本保证”。现实中你或许会发现，你可以为某些函数提供强烈保证，但效率和复杂度带来的成本会使它对许多人而言摇摇欲坠。只要你曾经付出适当的心力试图提供强烈保证，万一实际不可行，使你退而求其次地只提供基本保证，任何人都不该因此责难你。对许多函数而言，“异常安全性之基本保证”是一个绝对通情达理的选择。

如果你写的函数完全不提供异常安全保证，情况又有点不同。因为他人可以合理假设你在这方面有缺失，直到你证明自己的清白。是的，你应当写出异常安全码。不过你也可能有令人信服的理由。再次考虑先前出现的那份调用函数 f1 和 f2 的 someFunc 实现代码。假设 f2 完全没有提供异常安全保证，甚至连基本保证都没有，那便意味一旦 f2 抛出异常，程序有可能在 f2 内泄漏资源。这意味 f2 可能败坏数据结构，例如带序数组（sorted arrays）可能不再处于排序状态下、从某数据结构搬移至另一数据结构的对象有可能遗失……等等。someFunc 没办法补偿那些问题。如果 someFunc 调用的函数没有提供任何异常安全保证，someFunc 自身也不可能提供任何保证。

这令我想到怀孕。一位女性若非怀孕，就是没怀孕。不可能说她“部分怀孕”。同样道理，一个软件系统要不就具备异常安全性，要不就全然否定，没有所谓的“局部异常安全系统”。如果系统内有一个（惟有一个）函数不具备异常安全性，整个系统就不具备异常安全性，因为调用那个（不具备异常安全性的）函数有可能导致资源泄漏或数据结构败坏。不幸的是许多老旧 C++ 代码并不具备异常安全性，所以今天许多系统仍然不能够说是“异常安全”的，因为它们并入了一些并非“异常安全”的代码。

没有理由让这种情况永垂不朽。当你撰写新码或修改旧码时，请仔细想想如何让它具备异常安全性。首先是“以对象管理资源”（条款 13），那可阻止资源泄漏。然后是挑选三个“异常安全保证”中的某一个实施于你所写的每一个函数身上。你应该挑选“现实可施作”条件下的最强烈等级，只有当你的函数调用了传统代码，

才别无选择地将它设为“无任何保证”。将你的决定写成文档，这一来是为你的函数用户着想，二来是为将来的维护者着想。函数的“异常安全性保证”是其可见接口的一部分，所以你应该慎重选择，就像选择函数接口的其他任何部分一样。

四十年前，满载 goto 的代码被视为一种美好实践，而今我们却致力写出结构化控制流（structured control flows）。二十年前，全局数据（globally accessible data）被视为一种美好实践，而今我们却致力于数据的封装。十年前，撰写“未将异常考虑在内”的函数被视为一种美好实践，而今我们致力于写出“异常安全码”。

时间不断前进。我们与时俱进！

请记住

- 异常安全函数（Exception-safe functions）即使发生异常也不会泄漏资源或允许任何数据结构败坏。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。

条款 30：透彻了解 inlining 的里里外外

Understand the ins and outs of inlining.

Inline 函数，多棒的点子！它们看起来像函数，动作像函数，比宏好得多（见条款 2），可以调用它们又不需蒙受函数调用所招致的额外开销。你还能要求更多吗？

你实际获得的比想到的还多，因为“免除函数调用成本”只是故事的一部分而已。编译器最优化机制通常被设计用来浓缩那些“不含函数调用”的代码，所以当你 inline 某个函数，或许编译器就因此有能力对它（函数本体）执行语境相关最优化。大部分编译器绝不会对着一个“outlined 函数调用”动作执行如此之最优化。

然而编写程序就像现实生活一样，没有白吃的午餐。inline 函数也不例外。inline 函数背后的整体观念是，将“对此函数的每一个调用”都以函数本体替换之。我想不需要统计学博士来告诉你，这样做可能增加你的目标码（object code）大小。在

一台内存有限的机器上，过度热衷 inlining 会造成程序体积太大（对可用空间而言）。即使拥有虚内存，inline 造成的代码膨胀亦会导致额外的换页行为（paging），降低指令高速缓存装置的击中率（instruction cache hit rate），以及伴随这些而来的效率损失。

换个角度说，如果 inline 函数的本体很小，编译器针对“函数本体”所产出的码可能比针对“函数调用”所产出的码更小。果真如此，将函数 inlining 确实可能导致较小的目标码（object code）和较高的指令高速缓存装置击中率！

记住，inline 只是对编译器的一个申请，不是强制命令。这项申请可以隐喻提出，也可以明确提出。隐喻方式是将函数定义于 class 定义式内：

```
class Person {
public:
   ...
   int age() const { return theAge; }      //一个隐喻的 inline 申请：
   ...                                     //age 被定义于 class 定义式内。
private:
   int theAge;
};
```

这样的函数通常是成员函数，但条款 46 说 friend 函数也可被定义于 class 内，如果真是那样，它们也是被隐喻声明为 inline。

明确声明 inline 函数的做法则是在其定义式前加上关键字 inline。例如标准的 max template（来自<algorithm>）往往这样实现出来：

```
template<typename T>                                  //明确申请 inline：
inline const T& std::max(const T& a, const T& b)      //std::max 之前有
{ return a < b ? b : a; }                             //关键字"inline"
```

“max 是个 template”带出了一项观察结果：我们发现 inline 函数和 templates 两者通常都被定义于头文件内。这使得某些程序员以为 function templates 一定必须是 inline。这个结论不但无效而且可能有害，值得深入看一看。

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。某些建置环境可以在连接期完成 inlining，少量建置环境如基于 .NET CLI（Common Language Infrastructure：公共语言基础设施）的托管环境（managed environments）竟可在运行期完成 inlining。然而这样的环境毕竟是例外，不是通例。Inlining 在大多数 C++ 程序中是编译期行为。

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。（这其实也不是世界一统的准则。某些建置环境可以在连接期才执行 template 具现化。只不过编译期完成具现化动作比较常见。）

Template 的具现化与 inlining 无关。如果你正在写一个 template 而你认为所有根据此 template 具现出来的函数都应该 inlined，请将此 template 声明为 `inline`；这就是上述 `std::max` 代码的作为。但如果你写的 template 没有理由要求它所具现的每一个函数都是 inlined，就应该避免将这个 template 声明为 inline（不论显式或隐式）。Inlining 需要成本，你不会想在没有事先考虑的情况下就招来那些成本吧。我已经提过 inlining 如何引发代码膨胀（这对 template 作者特别重要，见条款 44），但还存在其他成本，稍后再讨论。

现在让我们先结束“`inline` 是个申请，编译器可加以忽略”的观察。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。这不该令你惊讶，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”。如果编译器不知道该调用哪个函数，你就很难责备它们拒绝将函数本体 inlining。

这些叙述整合起来的意思就是：一个表面上看似 inline 的函数是否真是 inline，取决于你的建置环境，主要取决于编译器。幸运的是大多数编译器提供了一个诊断级别：如果它们无法将你要求的函数 inline 化，会给你一个警告信息（见条款 53）。

有时候虽然编译器有意愿 inlining 某个函数，还是可能为该函数生成一个函数本体。举个例子，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。毕竟编译器哪有能力提出一个指针指向并不存在的函数呢？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：

```
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。
```

即使你从未使用函数指针，“未被成功 inlined”的 inline 函数还是有可能缠住你，因为程序员并非唯一要求函数指针的人。有时候编译器会生成构造函数和析构函数的 outline 副本，如此一来它们就可以获得指针指向那些函数，在 array 内部元素的构造和析构过程中使用。

实际上构造函数和析构函数往往是 inlining 的糟糕候选人——虽然漫不经心的情况下你不会这么认为。考虑以下 Derived class 构造函数：

```
class Base {
public:
   ...
private:
   std::string bm1, bm2;          //base 成员 1 和 2
};
class Derived: public Base {
public:
   Derived() { }                  //Derived 构造函数是空的，哦，是吗?
   ...
private:
   std::string dm1, dm2, dm3;     //derived 成员 1-3
};
```

这个构造函数看起来是 inlining 的绝佳候选人，因为它根本不含任何代码。但是你的眼睛可能会欺骗你。

C++ 对于“对象被创建和被销毁时发生什么事”做了各式各样的保证。当你使用 new，动态创建的对象被其构造函数自动初始化；当你使用 delete，对应的析构函数会被调用。当你创建一个对象，其每一个 base class 及每一个成员变量都会被自动构造；当你销毁一个对象，反向程序的析构行为亦会自动发生。如果有个异常在对象构造期间被抛出，该对象已构造好的那一部分会被自动销毁。在这些情况中 C++ 描述了什么一定会发生，但没有说如何发生。“事情如何发生”是编译器实现者的权责，不过至少有一点很清楚，那就是它们不可能凭空发生。你的程序内一定有某些代码让那些事情发生，而那些代码——由编译器于编译期间代为产生并安插到你的程序中的代码——肯定存在于某个地方。有时候就放在你的构造函数和

析构函数内，所以我们可以想象，编译器为稍早说的那个表面上看起来为空的Derived构造函数所产生的代码，相当于以下所列：

```
Derived::Derived()                    //“空白 Derived构造函数”的观念性实现
{
    Base::Base();                                   //初始化“Base成分”
    try { dm1.std::string::string(); }              //试图构造 dm1。
    catch (...) {                                   //如果抛出异常就
        Base::~Base();                              //销毁 base class 成分，并
        throw;                                      //传播该异常。
    }

    try { dm2.std::string::string(); }              //试图构造 dm2。
    catch(...) {                                    //如果抛出异常就
        dm1.std::string::~string( );                //销毁 dm1，
        Base::~Base();                              //销毁 base class 成分，并
        throw;                                      //传播该异常。
    }

    try { dm3.std::string::string(); }              //试图构造 dm3。
    catch(...) {                                    //如果抛出异常就
        dm2.std::string::~string( );                //销毁 dm2，
        dm1.std::string::~string( );                //销毁 dm1，
        Base::~Base();                              //销毁 base class 成分，并
        throw;                                      //传播该异常。
    }
}
```

这段代码并不能代表编译器真正制造出来的代码，因为真正的编译器会以更精致复杂的做法来处理异常。尽管如此，这已能准确反映 Derived 的空白构造函数必须提供的行为。不论编译器在其内所做的异常处理多么精致复杂，Derived 构造函数至少一定会陆续调用其成员变量和 base class 两者的构造函数，而那些调用（它们自身也可能被 inlined）会影响编译器是否对此空白函数 inlining。

相同理由也适用于 Base 构造函数，所以如果它被 inlined，所有替换“Base 构造函数调用”而插入的代码也都会被插入到“Derived 构造函数调用”内（因为 Derived 构造函数调用了 Base 构造函数）。如果 string 构造函数恰巧也被 inlined，Derived 构造函数将获得五份“string 构造函数代码”副本，每一份副本对应于 Derived 对象内的五个字符串（两个来自继承，三个来自自己的声明）之一。现在或许很清楚了，“是否将 Derived 构造函数 inline 化”并非是个轻松的决定。类似思考也适用于 Derived 析构函数，在那儿我们必须看到“被 Derived 构造函数初始化的所有对象”被一一销毁，无论以哪种方式进行。

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。换句话说如果 f 是程序库内的一个 inline 函数，客户将“f 函数本体”编进其程序中，一旦程序库设计者决定改变 f，所有用到 f 的客户端程序都必须重新编译。这往往是大家不愿意见到的。然而如果 f 是 non-inline 函数，一旦它有任何修改，客户端只需重新连接就好，远比重新编译的负担少很多。如果程序库采取动态连接，升级版函数甚至可以不知不觉地被应用程序吸纳。

对程序开发而言，将上述所有考虑牢记在心很是重要，但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。这对你应该不是太大的意外，毕竟你如何在一个并不存在的函数内设立断点（break point）呢？虽然某些建置环境勉力支持对 inlined 函数的调试，其他许多建置环境仅仅只能“在调试版程序中禁止发生 inlining”。

这使我们在决定哪些函数该被声明为 inline 而哪些函数不该时，掌握一个合乎逻辑的策略。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”（见条款 46）或“十分平淡无奇”（例如 p.135 Person::age）的函数身上。慎重使用 inline 便是对日后使用调试器带来帮助，不过这么一来也等于把自己推向手工最优化之路。不要忘记 80-20 经验法则：平均而言一个程序往往将 80% 的执行时间花费在 20% 的代码上头。这是一个重要的法则，因为它提醒你，作为一个软件开发者，你的目标是找出这可以有效增进程序整体效率的 20% 代码，然后将它 inline 或竭尽所能地将它瘦身。但除非你选对目标，否则一切都是虚功。

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。

条款 31：将文件间的编译依存关系降至最低

Minimize compilation dependencies between files.

假设你对 C++ 程序的某个 class 实现文件做了些轻微修改。注意，修改的不是 class 接口，而是实现，而且只改 private 成分。然后重新建置这个程序，并预计只花数秒就好。毕竟只有一个 class 被修改。你按下 "Build" 按钮或键入 make（或其他类似命令），然后大吃一惊，然后感到窘困，因为你意识到整个世界都被重新编译和连接了！当这种事情发生，难道你不气恼吗？

问题出在 C++ 并没有把“将接口从实现中分离”这事做得很好。Class 的定义式不只详细叙述了 class 接口，还包括十足的实现细目。例如：

```
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;          //实现细目
    Date theBirthDate;            //实现细目
    Address theAddress;           //实现细目
};
```

这里的 class Person 无法通过编译——如果编译器没有取得其实现代码所用到的 classes string, Date 和 Address 的定义式。这样的定义式通常由 #include 指示符提供，所以 Person 定义文件的最上方很可能存在这样的东西：

```
#include <string>
#include "date.h"
#include "address.h"
```

不幸的是，这么一来便是在 Person 定义文件和其含入文件之间形成了一种编译依存关系（compilation dependency）。如果这些头文件中有任何一个被改变，或这些头文件所倚赖的其他头文件有任何改变，那么每一个含入 Person class 的文件就得重新编译，任何使用 Person class 的文件也必须重新编译。这样的连串编译依存关系（cascading compilation dependencies）会对许多项目造成难以形容的灾难。

你或许会奇怪，为什么 C++ 坚持将 class 的实现细目置于 class 定义式中？为什么不这样定义 Person，将实现细目分开叙述？

```
namespace std {
   class string;          //前置声明（不正确，详下）
}                         //
class Date;               //前置声明
class Address;            //前置声明
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;
   ...
};
```

如果可以那么做，Person 的客户就只需要在 Person 接口被修改过时才重新编译。

这个想法存在两个问题。第一，string 不是个 class，它是个 typedef（定义为 basic_string<char>）。因此上述针对 string 而做的前置声明并不正确；正确的前置声明比较复杂，因为涉及额外的 templates。然而那并不要紧，因为你本来就不该尝试手工声明一部分标准程序库。你应该仅仅使用适当的 #includes 完成目的。标准头文件不太可能成为编译瓶颈，特别是如果你的建置环境允许你使用预编译头文件（precompiled headers）。如果解析（parsing）标准头文件真的是个问题，你可能需要改变你的接口设计，避免使用标准程序库中“引发不受欢迎之 #includes”那一部分。

关于“前置声明每一件东西”的第二个（同时也是比较重要的）困难是，编译器必须在编译期间知道对象的大小。考虑这个：

```
int main()
{
   int x;                 //定义一个 int
   Person p( params );    //定义一个 Person
   ...
}
```

当编译器看到 x 的定义式，它知道必须分配多少内存（通常位于 stack 内）才够持有一个 int。没问题，每个编译器都知道一个 int 有多大。当编译器看到 p 的定义

式，它也知道必须分配足够空间以放置一个 Person，但它如何知道一个 Person 对象有多大呢？编译器获得这项信息的唯一办法就是询问 class 定义式。然而如果 class 定义式可以合法地不列出实现细目，编译器如何知道该分配多少空间？

此问题在 Smalltalk, Jave 等语言上并不存在，因为当我们以那种语言定义对象时，编译器只分配足够空间给一个指针（用以指向该对象）使用。也就是说它们将上述代码视同这样子：

```
int main()
{
   int x;                  //定义一个 int
   Person* p;              //定义一个指针指向 Person 对象
   ...
}
```

这当然也是合法的 C++ 代码，所以你也可以自己玩玩“将对象实现细目隐藏于一个指针背后”的游戏。针对 Person 我们可以这样做：把 Person 分割为两个 classes，一个只提供接口，另一个负责实现该接口。如果负责实现的那个所谓 implementation class 取名为 PersonImpl，Person 将定义如下：

```
#include <string>       //标准程序库组件不该被前置声明。
#include <memory>       //此乃为了 tr1::shared_ptr 而含入；详后。

class PersonImpl;       //Person 实现类的前置声明。
class Date;             //Person 接口用到的 classes 的前置声明。
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::tr1::shared_ptr<PersonImpl> pImpl; //指针，指向实现物；
};                                         //std::tr1::shared_ptr 见条款 13.
```

在这里，main class(Person)只内含一个指针成员(这里使用 tr1::shared_ptr，见条款 13)，指向其实现类（PersonImpl）。这般设计常被称为 pimpl idiom（pimpl

是 "pointer to implementation" 的缩写）。这种 classes 内的指针名称往往就是 pImpl，就像上面代码那样。

这样的设计之下，Person 的客户就完全与 Dates, Addresses 以及 Persons 的实现细目分离了。那些 classes 的任何实现修改都不需要 Person 客户端重新编译。此外由于客户无法看到 Person 的实现细目，也就不可能写出什么“取决于那些细目”的代码。这真正是“接口与实现分离”！

这个分离的关键在于以“声明的依存性”替换“定义的依存性”，那正是编译依存性最小化的本质：现实中让头文件尽可能自我满足，万一做不到，则让它与其他文件内的声明式（而非定义式）相依。其他每一件事都源自于这个简单的设计策略：

- **如果使用 object references 或 object 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，尽量以 class 声明式替换 class 定义式**。注意，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；纵使函数以 *by value* 方式传递该类型的参数（或返回值）亦然：

```
class Date;                              //class 声明式。
Date today();                            //没问题 — 这里并不需要
void clearAppointments(Date d);          //  Date 的定义式。
```

当然，*pass-by-value* 一般而言是个糟糕的主意（见条款 20），但如果你发现因为某种因素被迫使用它，并不能够就此为“非必要之编译依存关系”导入正当性。

声明 today 函数和 clearAppointments 函数而无需定义 Date，这种能力可能会令你惊讶，但它并不是真的那么神奇。一旦任何人调用那些函数，调用之前 Date 定义式一定得先曝光才行。那么或许你会纳闷，何必费心声明一个没人调用的函数呢？嗯，并非没人调用，而是并非每个人都调用。假设你有一个函数库内含数百个函数声明，不太可能每个客户叫遍每一个函数。如果能够将“提供 class 定义式”（通过 #include 完成）的义务从“函数声明所在”之头文件移转到“内含函数调用”之客户文件，便可将“并非真正必要之类型定义”与客户端之间的编译依存性去除掉。

■ **为声明式和定义式提供不同的头文件**。为了促进严守上述准则，需要两个头文件，一个用于声明式，一个用于定义式。当然，这些文件必须保持一致性，如果有个声明式被改变了，两个文件都得改变。因此程序库客户应该总是#include一个声明文件而非前置声明若干函数，程序库作者也应该提供这两个头文件。举个例子，Date的客户如果希望声明today和clearAppointments，他们不该像先前那样以手工方式前置声明Date，而是应该#include适当的、内含声明式的头文件：

```
#include "datefwd.h"              //这个头文件内声明（但未定义）class Date。
Date today( );                    //同前。
void clearAppointments(Date d);
```

只含声明式的那个头文件名为"datefwd.h"，命名方式取法C++标准程序库头文件（见条款54）的<iosfwd>。<iosfwd>内含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<sstream>, <streambuf>, <fstream>和<iostream>。

<iosfwd>深具启发意义的另一个原因是，它分外彰显“本条款适用于templates也适用于non-templates”。虽然条款30说过，在许多建置环境（build environments）中template定义式通常被置于头文件内，但也有某些建置环境允许template定义式放在“非头文件”内，这么一来就可以将“只含声明式”的头文件提供给templates。<iosfwd>就是这样一份头文件。

C++也提供关键字export，允许将template声明式和template定义式分割于不同的文件内。不幸的是支持这个关键字的编译器目前非常少，因此现实中使用这个关键字的经验也非常少。目前若要评论export在高效C++编程中扮演什么角色，恐怕言之过早。

像Person这样使用pimpl idiom的classes，往往被称为Handle classes。也许你会纳闷，这样的classes如何真正做点事情。办法之一是将它们的所有函数转交给相应的实现类（implementation classes）并由后者完成实际工作。例如下面是Person两个成员函数的实现：

```
#include "Person.h"           //我们正在实现Person class，
                              //所以必须#include其class定义式。
```

```
#include "PersonImpl.h"      //我们也必须#include PersonImpl 的
                             // class 定义式，否则无法调用其成员函数；
                             //注意，PersonImpl 有着和 Person
                             //完全相同的成员函数，两者接口完全相同。
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( );
}
```

请注意，Person 构造函数以 new（见条款 16）调用 PersonImpl 构造函数，以及 Person::name 函数内调用 PersonImpl::name。这是重要的，让 Person 变成一个 Handle class 并不会改变它做的事，只会改变它做事的方法。

另一个制作 Handle class 的办法是，令 Person 成为一种特殊的 abstract base class（抽象基类），称为 Interface class。这种 class 的目的是详细一一描述 derived classes 的接口（见条款 34），因此它通常不带成员变量，也没有构造函数，只有一个 virtual 析构函数（见条款 7）以及一组 pure virtual 函数，用来叙述整个接口。

Interface classes 类似 Java 和 .NET 的 Interfaces，但 C++ 的 Interface classes 并不需要负担 Java 和 .NET 的 Interface 所需负担的责任。举个例子，Java 和 .NET 都不允许在 Interfaces 内实现成员变量或成员函数，但 C++ 不禁止这两样东西。C++ 这种更为巨大的弹性有其用途，因为一如条款 36 所言，“non-virtual 函数的实现”对继承体系内所有 classes 都应该相同，所以将此等函数实现为 Interface class（其中写有相应声明）的一部分也是合理的。

一个针对 Person 而写的 Interface class 或许看起来像这样：

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

这个 class 的客户必须以 Person 的 pointers 和 references 来撰写应用程序，因为它不可能针对“内含 pure virtual 函数”的 Person classes 具现出实体。（然而却有可能对派生自 Person 的 classes 具现出实体，详下。）就像 Handle classes 的客户一样，除非 Interface class 的接口被修改否则其客户不需重新编译。

Interface class 的客户必须有办法为这种 class 创建新对象。他们通常调用一个特殊函数，此函数扮演“真正将被具现化”的那个 derived classes 的构造函数角色。这样的函数通常称为 factory（工厂）函数（见条款 13）或 virtual 构造函数。它们返回指针（或更为可取的智能指针，见条款 18），指向动态分配所得对象，而该对象支持 Interface class 的接口。这样的函数又往往在 Interface class 内被声明为 static：

```
class Person {
public:
  ...
  static std::tr1::shared_ptr<Person> //返回一个 tr1::shared_ptr，指向
    create(const std::string& name,    //一个新的 Person，并以给定之参数
           const Date& birthday,       //初始化。条款 18 告诉你
           const Address& addr);       //为什么返回的是 tr1::shared_ptr
    ...
};
```

客户会这样使用它们：

```
std::string name;
Date dateOfBirth;
Address address;
...
//创建一个对象，支持 Person 接口
std::tr1::shared_ptr<Person> pp(Person::create(name, dateOfBirth,
                                               address));
...
std::cout << pp->name()                  //通过 Person 的接口使用这个对象
          << " was born on "
          << pp->birthDate()
          << " and now lives at "
          << pp->address( );
...                                      //当 pp 离开作用域，
                                         //对象会被自动删除，
                                         //见条款 13。
```

当然，支持 Interface class 接口的那个具象类（concrete classes）必须被定义出来，而且真正的构造函数必须被调用。一切都在 virtual 构造函数实现码所在的文件

内秘密发生。假设 Interface class Person 有个具象的 derived class RealPerson，后者提供继承而来的 virtual 函数的实现：

```
class RealPerson: public Person {
public:
   RealPerson(const std::string& name, const Date& birthday,
              const Address& addr)
     : theName(name), theBirthDate(birthday), theAddress(addr)
     {}

   virtual ~RealPerson() { }
   std::string name() const;          //这些函数的实现码并不显示于此，
   std::string birthDate() const;     //但它们很容易想象。
   std::string address() const;

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

有了 RealPerson 之后，写出 Person::create 就真的一点也不稀奇了：

```
std::tr1::shared_ptr<Person> Person::create(const std::string& name,
                                            const Date& birthday,
                                            const Address& addr)
{
   return
     std::tr1::shared_ptr<Person>(new RealPerson(name, birthday,
                                                 addr));
}
```

一个更现实的 Person::create 实现代码会创建不同类型的 derived class 对象，取决于诸如额外参数值、读自文件或数据库的数据、环境变量等等。

RealPerson 示范实现 Interface class 的两个最常见机制之一：从 Interface class（Person）继承接口规格，然后实现出接口所覆盖的函数。Interface class 的第二个实现法涉及多重继承，那是条款 40 探索的主题。

Handle classes 和 Interface classes 解除了接口和实现之间的耦合关系，从而降低文件间的编译依存性（compilation dependencies）。如果你是犬儒学派（译注：犬儒学派希望过一种符合自然的简朴生活，摈弃一切社会习俗和人为引导的种种欲望），我知道你正等着我有义务给出的旁注。“所有这些戏法得付出多少代价？”你咕哝着。答案是计算器科学中通常需要付出的那些：它使你在运行期丧失若干速度，又让你为每个对象超额付出若干内存。

在 Handle classes 身上，成员函数必须通过 implementation pointer 取得对象数据。那会为每一次访问增加一层间接性。而每一个对象消耗的内存数量必须增加 implementation pointer 的大小。最后，implementation pointer 必须初始化（在 Handle class 构造函数内），指向一个动态分配得来的 implementation object，所以你将蒙受因动态内存分配（及其后的释放动作）而来的额外开销，以及遭遇 `bad_alloc` 异常（内存不足）的可能性。

至于 Interface classes，由于每个函数都是 virtual，所以你必须为每次函数调用付出一个间接跳跃（indirect jump）成本（见条款 7）。此外 Interface class 派生的对象必须内含一个 vptr（virtual table pointer，再次见条款 7），这个指针可能会增加存放对象所需的内存数量——实际取决于这个对象除了 Interface class 之外是否还有其他 virtual 函数来源。

最后，不论 Handle classes 或 Interface classes，一旦脱离 inline 函数都无法有太大作为。条款 30 解释过为什么函数本体为了被 inlined 必须（很典型地）置于头文件内，但 Handle classes 和 Interface classes 正是特别被设计用来隐藏实现细节如函数本体。

然而，如果只因为若干额外成本便不考虑 Handle classes 和 Interface classes，将是严重的错误。Virtual 函数不也带来成本吗？你并不会想要弃绝它们对不对？（如果是的话，那你读错书了。）你应该考虑以渐进方式使用这些技术。在程序发展过程中使用 Handle classes 和 Interface classes 以求实现码有所变化时对其客户带来最小冲击。而当它们导致速度和/或大小差异过于重大以至于 classes 之间的耦合相形之下不成为关键时，就以具象类（concrete classes）替换 Handle classes 和 Interface classes。

请记住

- 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。
- 程序库头文件应该以“完全且仅有声明式”（full and declaration-only forms）的形式存在。这种做法不论是否涉及 templates 都适用。