7

模板与泛型编程

Templates and Generic Programming

C++ templates 的最初发展动机很直接：让我们得以建立“类型安全”（type-safe）的容器如 vector, list 和 map。然而当愈多人用上 templates，他们发现 templates 有能力完成愈多可能的变化。容器当然很好，但泛型编程（generic programming）——写出的代码和其所处理的对象类型彼此独立——更好。STL 算法如 for_each, find 和 merge 就是这一类编程的成果。最终人们发现，C++ template 机制自身是一部完整的图灵机（Turing-complete）：它可以被用来计算任何可计算的值。于是导出了模板元编程（template metaprogramming），创造出“在 C++ 编译器内执行并于编译完成时停止执行”的程序。近来这些日子，容器反倒只成为 C++ template 馅饼上的一小部分。然而尽管 template 的应用如此宽广，有一组核心观念一直支撑着所有基于 template 的编程。那些观念便是本章焦点。

本章无法使你变成一个专家级的 template 程序员，但可以使你成为一个比较好的 template 程序员。本章也会给你必要信息，使你能够扩展你的 template 编程，到达你所渴望的境界。

条款 41：了解隐式接口和编译期多态

Understand implicit interfaces and compile-time polymorphism.

面向对象编程世界总是以显式接口（*explicit* interfaces）和运行期多态（*runtime* polymorphism）解决问题。举个例子，给定这样（无甚意义）的 class：

```
class Widget {
public:
    Widget();
    virtual ~Widget();
```

```
    virtual std::size_t size( ) const;
    virtual void normalize();
    void swap(Widget& other);                    //见条款25
    ...
};
```

和这样（也是无甚意义）的函数：

```
void doProcessing(Widget& w)
{
    if (w.size() > 10 && w != someNastyWidget) {
        Widget temp(w);
        temp.normalize();
        temp.swap(w);
    }
}
```

我们可以这样说 doProcessing 内的 w：

- 由于 w 的类型被声明为 Widget，所以 w 必须支持 Widget 接口。我们可以在源码中找出这个接口（例如在 Widget 的.h 文件中），看看它是什么样子，所以我称此为一个显式接口（*explicit interface*），也就是它在源码中明确可见。
- 由于 Widget 的某些成员函数是 virtual，w 对那些函数的调用将表现出运行期多态（*runtime polymorphism*），也就是说将于运行期根据 w 的动态类型（见条款37）决定究竟调用哪一个函数。

Templates 及泛型编程的世界，与面向对象有根本上的不同。在此世界中显式接口和运行期多态仍然存在，但重要性降低。反倒是隐式接口（*implicit interfaces*）和编译期多态（*compile-time polymorphism*）移到前头了。若想知道那是什么，看看当我们将 doProcessing 从函数转变成函数模板（function template）时发生什么事：

```
template<typename T>
void doProcessing(T& w)
{
    if (w.size() > 10 && w != someNastyWidget) {
        T temp(w);
        temp.normalize();
        temp.swap(w);
    }
}
```

现在我们怎么说 doProcessing 内的 w 呢？

- w 必须支持哪一种接口，系由 template 中执行于 w 身上的操作来决定。本例看来 w 的类型 T 好像必须支持 size, normalize 和 swap 成员函数、*copy* 构造函数（用

以建立 temp)、不等比较(*inequality* comparison，用来比较 someNasty-Widget)。我们很快会看到这并非完全正确，但对目前而言足够真实。重要的是，这一组表达式（对此 template 而言必须有效编译）便是 T 必须支持的一组隐式接口（*implicit interface*）。

- 凡涉及 w 的任何函数调用，例如 operator>和 operator!=，有可能造成 template 具现化（instantiated），使这些调用得以成功。这样的具现行为发生在编译期。“以不同的 template 参数具现化 function templates”会导致调用不同的函数，这便是所谓的编译期多态（*compile-time polymorphism*）。

纵使你从未用过 templates，应该不陌生“运行期多态”和“编译期多态”之间的差异，因为它类似于“哪一个重载函数该被调用”（发生在编译期）和“哪一个 virtual 函数该被绑定”（发生在运行期）之间的差异。显式接口和隐式接口的差异就比较新颖，需要更多更贴近的说明和解释。

通常显式接口由函数的签名式（也就是函数名称、参数类型、返回类型）构成。例如 Widget class：

```
class Widget {
public:
   Widget();
   virtual ~Widget();
   virtual std::size_t size( ) const;
   virtual void normalize();
   void swap(Widget& other);
};
```

其 public 接口由一个构造函数、一个析构函数、函数 size, normalize, swap 及其参数类型、返回类型、常量性（constnesses）构成。当然也包括编译器产生的 *copy* 构造函数和 *copy assignment* 操作符（见条款 5）。另外也可以包括 typedefs，以及如果你大胆违反条款 22（令成员变量为 private）而出现的 public 成员变量。

隐式接口就完全不同了。它并不基于函数签名式，而是由有效表达式（valid *expressions*）组成。再次看看 doProcessing template 一开始的条件：

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

T（w的类型）的隐式接口看来好像有这些约束：

- 它必须提供一个名为 size 的成员函数，该函数返回一个整数值。
- 它必须支持一个 operator!= 函数，用来比较两个 T 对象。这里我们假设 someNastyWidget 的类型为 T。

真要感谢操作符重载（operator overloading）带来的可能性，这两个约束都不需要满足。是的，T 必须支持 size 成员函数，然而这个函数也可能从 base class 继承而得。这个成员函数不需返回一个整数值，甚至不需返回一个数值类型。就此而言，它甚至不需要返回一个定义有 operator>的类型！它唯一需要做的是返回一个类型为 X 的对象，而 X对象加上一个 int(10 的类型)必须能够调用一个 operator>。这个 operator> 不需要非得取得一个类型为 X 的参数不可，因为它也可以取得类型 Y 的参数，只要存在一个隐式转换能够将类型 X 的对象转换为类型 Y 的对象！

同样道理，T 并不需要支持 operator!=，因为以下这样也是可以的：operator!= 接受一个类型为 X 的对象和一个类型为 Y 的对象，T 可被转换为 X 而 someNastyWidget 的类型可被转换为 Y，这样就可以有效调用 operator!=。

（偷偷告诉你，以上分析并未考虑这样的可能性：operator&& 被重载，从一个连接词改变为或许完全不同的某种东西，从而改变上述表达式的意义。）

当人们第一次以此种方式思考隐式接口，大多数的他们会感到头疼。但真的不需要阿司匹林来镇痛。隐式接口仅仅是由一组有效表达式构成，表达式自身可能看起来很复杂，但它们要求的约束条件一般而言相当直接又明确。例如以下条件式：

```
if (w.size() > 10 && w != someNastyWidget) ...
```

关于函数 size, oper-ator>, operator&&或 operator!= 身上的约束条件，我们很难就此说得太多，但整体确认表达式约束条件却很容易。if 语句的条件式必须

是个布尔表达式，所以无论涉及什么实际类型，无论"w.size() > 10 && w != someNastyWidget"导致什么，它都必须与 bool 兼容。这是 template doProcessing 加诸于其类型参数（type parameter）T 的隐式接口的一部分。doProcessing 要求的其他隐式接口：*copy* 构造函数、normalize 和 swap 也都必须对 T 型对象有效。

加诸于 template 参数身上的隐式接口，就像加诸于 class 对象身上的显式接口一样真实，而且两者都在编译期完成检查。就像你无法以一种“与 class 提供之显式接口矛盾”的方式来使用对象（代码将通不过编译），你也无法在 template 中使用“不支持 template 所要求之隐式接口”的对象（代码一样通不过编译）。

请记住

- classes 和 templates 都支持接口（interfaces）和多态（polymorphism）。
- 对 classes 而言接口是显式的(explicit)，以函数签名为中心。多态则是通过 virtual 函数发生于运行期。
- 对 template 参数而言，接口是隐式的（implicit），奠基于有效表达式。多态则是通过 template 具现化和函数重载解析（function overloading resolution）发生于编译期。

条款 42：了解 typename 的双重意义

Understand the two meanings of typename.

提一个问题：以下 template 声明式中，class 和 typename 有什么不同？

```
template<class T> class Widget;              //使用 "class"
template<typename T> class Widget;           //使用 "typename"
```

答案：没有不同。当我们声明 template 类型参数，class 和 typename 的意义完全相同。某些程序员始终比较喜欢 class，因为可以少打几个字。其他人（包括我）比较喜欢 typename，因为它暗示参数并非一定得是个 class 类型。少数开发人员在接受任何类型时使用 typename，而在只接受用户自定义类型时保留旧式的 class。然而从 C++ 的角度来看，声明 template 参数时，不论使用关键字 class 或 typename，意义完全相同。

然而 C++ 并不总是把 class 和 typename 视为等价。有时候你一定得使用 typename。为了解其时机，我们必须先谈谈你可以在 template 内指涉（*refer to*）的两种名称。

假设我们有个 template function，接受一个 STL 兼容容器为参数，容器内持有的对象可被赋值为 ints。进一步假设这个函数仅仅只是打印其第二元素值。这是一个无聊的函数，以无聊的方式实现，而且如稍后所言，它甚至不能通过编译。但请暂时漠视那些事，下面是实践这个愚蠢想法的一种方式：

```
template<typename C>
void print2nd(const C& container)          //打印容器内的第二元素
{                                          //注意这不是有效的 C++代码
  if (container.size() >= 2) {
     C::const_iterator iter(container.begin());//取得第一元素的迭代器
     ++iter;                                //将 iter 移往第二元素
     int value = *iter;                     //将该元素复制到某个 int.
     std::cout << value;                    //打印那个 int.
  }
}
```

我在代码中特别强调两个 local 变量 iter 和 value。iter 的类型是 C::const_iterator，实际是什么必须取决于 template 参数 C。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*）。C::const_iterator 就是这样一个名称。实际上它还是个嵌套从属类型名称（*nested dependent type name*），也就是个嵌套从属名称并且指涉某类型。

print2nd 内的另一个 local 变量 value，其类型是 int。int 是一个并不倚赖任何 template 参数的名称。这样的名称是谓非从属名称（*non-dependent names*）。我不知道为什么不叫做独立名称（independent names）。如果你和我一样认为术语 "non-dependent" 令人憎恶，你我之间起了共鸣。但毕竟 "non-dependent" 已被定为这一类名称的术语，所以请和我一样，眨眨眼睛然后顺从它吧。

嵌套从属名称有可能导致解析（parsing）困难。举个例子，假设我们令 print2nd 更愚蠢些，这样起头：

```
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。当然啦，这听起来有点疯狂，但却是可能的，而撰写 C++ 解析器的人必须操心所有可能的输入，甚至是这么疯狂的输入。

在我们知道 C 是什么之前，没有任何办法可以知道 C::const_iterator 是否为一个类型。而当编译器开始解析 template print2nd 时，尚未确知 C 是什么东西。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。

把这些记在心上，现在再次看看 print2nd 起始处：

```
template<typename C>
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
```

现在应该很清楚为什么这不是有效的 C++ 代码了吧。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。若要矫正这个形势，我们必须告诉 C++ 说 C::const_iterator 是个类型。只要紧临它之前放置关键字 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());
       ...
   }
}
```

一般性规则很简单：任何时候当你想要在 template 中指涉一个嵌套从属类型名称，就必须在紧临它的前一个位置放上关键字 typename。（再提醒一次，很快我会谈到一个例外。）

typename 只被用来验明嵌套从属类型名称；其他名称不该有它存在。例如下面这个 function template，接受一个容器和一个“指向该容器”的迭代器：

```
template<typename C>                    //允许使用 "typename"（或"class"）
void f(const C& container,              //不允许使用 "typename"
       typename C::iterator iter);      //一定要使用 "typename"
```

上述的 C 并不是嵌套从属类型名称（它并非嵌套于任何“取决于 template 参数”的东西内），所以声明 container 时并不需要以 typename 为前导，但 C::iterator 是个嵌套从属类型名称，所以必须以 typename 为前导。

“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<T>::Nested {     //base class list 中
public:                                      //不允许 "typename".
   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，
        ...                  //既不在 base class list 中也不在 mem. init. list 中，
   }                         //作为一个 base class 修饰符需加上 typename.
   ...
};
```

这样的不一致性真令人恼恨，但一旦你有了一些经验，勉勉强强还能接受它。

让我们看看最后一个 typename 例子，那是你将在真实程序中看到的代表性例子。假设我们正在撰写一个 function template，它接受一个迭代器，而我们打算为该迭代器指涉的对象做一份 local 复件（副本）temp。我们可以这么写：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
   typename std::iterator_traits<IterT>::value_type temp(*iter);
   ...
}
```

别让 std::iterator_traits<IterT>::value_type 惊吓了你，那只不过是标准 traits class（见条款 47）的一种运用，相当于说“类型为 IterT 之对象所指之物的类型”。这个语句声明一个 local 变量（temp），使用 IterT 对象所指物的相同类型，并将 temp 初始化为 iter 所指物。如果 IterT 是 vector<int>::iterator，temp 的类型就是 int。如果 IterT 是 list<string>::iterator，temp 的类型就是 string。由于 std::iterator_traits<IterT>::value_type 是个嵌套从属类型名称（value_type 被嵌套于 iterator_traits<IterT> 之内而 IterT 是个 template 参数），所以我们必须在它之前放置 typename。

如果你认为 std::iterator_traits<IterT>::value_type 读起来不畅快，想象一下打那么长的字又是什么光景。如果你像大多数程序员一样，认为多打几次这些字实在很恐怖，那么你应该会想建立一个 typedef。对于 traits 成员名称如 value_type（再次请看条款 47 提供的 traits 信息），普遍的习惯是设定 typedef 名称用以代表某个 traits 成员名称，于是常常可以看到类似这样的 local typedef：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
    typedef typename std::iterator_traits<IterT>::value_type value_type;
    value_type temp(*iter);
    ...
}
```

许多程序员最初认为把 "typedef typename" 并列颇不合谐，但它实在是指涉“嵌套从属类型名称”的一个合理附带结果。你很快会习惯它，毕竟你有强烈的动机——你希望多打几次 typename std::iterator_traits<IterT>::value_type 吗？

作为结语，我应该提到，typename 相关规则在不同的编译器上有不同的实践。某些编译器接受的代码原本该有 typename 却遗漏了；原本不该有 typename 却出现了；还有少数编译器（通常是较旧版本）根本就拒绝 typename。这意味 typename 和“嵌套从属类型名称”之间的互动，也许会在移植性方面带给你某种温和的头疼。

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。

条款 43：学习处理模板化基类内的名称

Know how to access names in templatized base classes.

假设我们需要撰写一个程序，它能够传送信息到若干不同的公司去。信息要不译成密码，要不就是未经加工的文字。如果编译期间我们有足够信息来决定哪一个信息传至哪一家公司，就可以采用基于 template 的解法：

```
class CompanyA {
public:
    ...
    void sendCleartext(const std::string& msg);
    void sendEncrypted(const std::string& msg);
    ...
};
class CompanyB {
public:
    ...
    void sendCleartext(const std::string& msg);
    void sendEncrypted(const std::string& msg);
    ...
};
...                              //针对其他公司设计的 classes.

class MsgInfo { ... };           //这个 class 用来保存信息，以备将来产生信息

template<typename Company>
class MsgSender {
public:
    ...                          //构造函数、析构函数等等。
    void sendClear(const MsgInfo& info)
    {
        std::string msg;

        在这儿，根据 info 产生信息;

        Company c;
        c.sendCleartext(msg);
    }
    void sendSecret(const MsgInfo& info)    //类似 sendClear，唯一不同是
    { ... }                                 //这里调用 c.sendEncrypted
};
```

这个做法行得通。但假设我们有时候想要在每次送出信息时志记（log）某些信息。derived class 可轻易加上这样的生产力，那似乎是个合情合理的解法：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
  ...                                       //构造函数、析构函数 等等.
  void sendClearMsg(const MsgInfo& info)
  {
     将“传送前”的信息写至 log;

     sendClear(info);              //调用 base class 函数；这段码无法通过编译。

     将“传送后”的信息写至 log;
  }
  ...
};
```

注意这个 derived class 的信息传送函数有一个不同的名称（sendClearMsg），与其 base class 内的名称（sendClear）不同。那是个好设计，因为它避免遮掩“继承而得的名称”（见条款 33），也避免重新定义一个继承而得的 non-virtual 函数（见条款 36）。但上述代码无法通过编译，至少对严守规律的编译器而言。这样的编译器会抱怨 sendClear 不存在。我们的眼睛可以看到 sendClear 的确在 base class 内，编译器却看不到它们。为什么？

问题在于，当编译器遭遇 class template LoggingMsgSender 定义式时，并不知道它继承什么样的 class。当然它继承的是 MsgSender<Company>，但其中的 Company 是个 template 参数，不到后来（当 LoggingMsgSender 被具现化）无法确切知道它是什么。而如果不知道 Company 是什么，就无法知道 class MsgSender<Company>看起来像什么——更明确地说是没办法知道它是否有个 sendClear 函数。

为了让问题更具体化，假设我们有个 class CompanyZ 坚持使用加密通讯：

```
class CompanyZ {                    //这个 class 不提供
public:                             //sendCleartext 函数
   ...
   void sendEncrypted(const std::string& msg);
   ...
};
```

一般性的 MsgSender template 对 CompanyZ 并不合适，因为那个 template 提供了一个 sendClear 函数（其中针对其类型参数 Company 调用了 sendCleartext 函数），而这对 CompanyZ 对象并不合理。欲矫正这个问题，我们可以针对 CompanyZ 产生一个 MsgSender 特化版：

```
template<>                          //一个全特化的
class MsgSender<CompanyZ> {         //MsgSender；它和一般 template 相同，
public:                             //差别只在于它删掉了 sendClear。
   ...
   void sendSecret(const MsgInfo& info)
   { ... }
};
```

注意 class 定义式最前头的 "template<>" 语法象征这既不是 template 也不是标准 class，而是个特化版的 MsgSender template，在 template 实参是 CompanyZ 时被使用。这是所谓的模板全特化（*total template specialization*）：template MsgSender 针对类型 CompanyZ 特化了，而且其特化是全面性的，也就是说一旦类型参数被定义为 CompanyZ，再没有其他 template 参数可供变化。

现在，MsgSender 针对 CompanyZ 进行了全特化，让我们再次考虑 derived class LoggingMsgSender:

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
    ...
    void sendClearMsg(const MsgInfo& info)
    {
      将“传送前”的信息写至 log;
      sendClear(info);        //如果 Company == CompanyZ，这个函数不存在。
      将“传送后”的信息写至 log;
    }
    ...
};
```

正如注释所言，当 base class 被指定为 MsgSender<CompanyZ> 时这段代码不合法，因为那个 class 并未提供 sendClear 函数！那就是为什么 C++ 拒绝这个调用的原因：它知道 base class templates 有可能被特化，而那个特化版本可能不提供和一般性 template 相同的接口。因此它往往拒绝在 templatized base classes（模板化基类，本例的 MsgSender<Company>）内寻找继承而来的名称（本例的 SendClear）。就某种意义而言，当我们从 Object Oriented C++ 跨进 Template C++（见条款 1），继承就不像以前那般畅行无阻了。

为了重头来过，我们必须有某种办法令 C++“不进入 templatized base classes 观察”的行为失效。有三个办法，第一是在 base class 函数调用动作之前加上 "this->":

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
    ...
    void sendClearMsg(const MsgInfo& info)
    {
      将“传送前”的信息写至 log;
      this->sendClear(info);        //成立，假设 sendClear 将被继承。
      将“传送后”的信息写至 log;
    }
    ...
};
```

第二是使用 using 声明式。如果你已读过条款 33，这个解法应该会令你感到熟悉。条款 33 描述 using 声明式如何将“被掩盖的 base class 名称”带入一个 derived class 作用域内。我们可以这样写下 sendClearMsg：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
    using MsgSender<Company>::sendClear;     //告诉编译器，请它假设
    ...                                      //sendClear 位于 base class 内。
    void sendClearMsg(const MsgInfo& info)
    {
        ...
        sendClear(info);          //OK，假设 sendClear 将被继承下来。
        ...
    }
    ...
};
```

（虽然 using 声明式在这里或在条款 33 都可有效运作，但两处解决的问题其实不相同。这里的情况并不是 base class 名称被 derived class 名称遮掩，而是编译器不进入 base class 作用域内查找，于是我们通过 using 告诉它，请它那么做。）

第三个做法是，明白指出被调用的函数位于 base class 内：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
    ...
    void sendClearMsg(const MsgInfo& info)
    {
        ...
        MsgSender<Company>::sendClear(info);    //OK，假设 sendClear
        ...                                     //将被继承下来。
    }
    ...
};
```

但这往往是最不让人满意的一个解法，因为如果被调用的是 virtual 函数，上述的明确资格修饰（explicit qualification）会关闭“virtual 绑定行为”。

从名称可视点（visibility point）的角度出发，上述每一个解法做的事情都相同：对编译器承诺“base class template 的任何特化版本都将支持其一般（泛化）版本所提供的接口”。这样一个承诺是编译器在解析（parse）像 LoggingMsgSender 这样的 derived class template 时需要的。但如果这个承诺最终未被实践出来，往后的编

译最终还是会还给事实一个公道。举个例子，如果稍后的源码内含这个：

```
LoggingMsgSender<CompanyZ> zMsgSender;
MsgInfo msgData;
...                                          //在 msgData 内放置信息。
zMsgSender.sendClearMsg(msgData);            //错误！无法通过编译。
```

其中对 sendClearMsg 的调用动作将无法通过编译，因为在那个点上，编译器知道 base class 是个 template 特化版本 MsgSender<CompanyZ>，而且它们知道那个 class 不提供 sendClear 函数，而后者却是 sendClearMsg 尝试调用的函数。

根本而言，本条款探讨的是，面对“指涉 base class members”之无效 references，编译器的诊断时间可能发生在早期（当解析 derived class template 的定义式时），也可能发生在晚期（当那些 templates 被特定之 template 实参具现化时）。C++ 的政策是宁愿较早诊断，这就是为什么“当 base classes 从 templates 中被具现化时”它假设它对那些 base classes 的内容毫无所悉的缘故。

请记住

- 可在 derived class templates 内通过 "this->" 指涉 base class templates 内的成员名称，或藉由一个明白写出的“base class 资格修饰符”完成。

条款 44：将与参数无关的代码抽离 templates

Factor parameter-independent code out of templates.

Templates 是节省时间和避免代码重复的一个奇方妙法。不再需要键入 20 个类似的 classes 而每一个带有 15 个成员函数，你只需键入一个 class template，留给编译器去具现化那 20 个你需要的相关 classes 和 300 个函数。（class templates 的成员函数只有在被使用时才被暗中具现化，所以只有在这 300 个函数的每一个都被使用，你才会获得这 300 个函数。）Function templates 有类似的诉求。替换写许多函数，你只需要写一个 function template，然后让编译器做剩余的事情。技术是不是很崇高伟大，呵呵。

是的，唔……有时候啦。如果你不小心，使用 templates 可能会导致代码膨胀（*code bloat*）：其二进制码带着重复（或几乎重复）的代码、数据，或两者。其结果有可能源码看起来合身而整齐，但目标码（object code）却不是那么回事。肥胖不结实很难被视为时尚，所以你需要知道如何避免这样的二进制浮夸。

你的主要工具有个气势恢宏的名称：共性与变性分析（*commonality and variability analysis*），但其概念十分平民化。纵使你从未写过一个 template，你始

终做着这样的分析。

当你编写某个函数，而你明白其中某些部分的实现码和另一个函数的实现码实质相同，你会很单纯地重复这些码吗？当然不。你会抽出两个函数的共同部分，把它们放进第三个函数中，然后令原先两个函数调用这个新函数。也就是说，你分析了两个函数，找出共同的部分和变化的部分，把共同部分搬到一个新函数去，保留变化的部分在原函数中不动。同样道理，如果你正在编写某个 class，而你明白其中某些部分和另一个 class 的某些部分相同，你也不会重复这共同的部分。取而代之的是你会把共同部分搬移到新 class 去，然后使用继承或复合（见条款 32,38,39），令原先的 classes 取用这共同特性。而原 classes 的互异部分（变异部分）仍然留在原位置不动。

编写 templates 时，也是做相同的分析，以相同的方式避免重复，但其中有个窍门。在 non-template 代码中，重复十分明确：你可以"看"到两个函数或两个 classes 之间有所重复。然而在 template 代码中，重复是隐晦的：毕竟只存在一份 template 源码，所以你必须训练自己去感受当 template 被具现化多次时可能发生的重复。

举个例子，假设你想为固定尺寸的正方矩阵编写一个 template。该矩阵的性质之一是支持逆矩阵运算（matrix inversion）。

```
template<typename T,          //template 支持 n x n 矩阵，元素是
         std::size_t n>       //类型为 T 的 objects；见以下
class SquareMatrix {          //关于 size_t 参数的信息
public:
  ...
  void invert( );             //求逆矩阵
};
```

这个 template 接受一个类型参数 T，除此之外还接受一个类型为 size_t 的参数，那是个非类型参数（*non-type parameter*）。这种参数和类型参数比起来较不常见，但它们完全合法，而且就像本例一样，相当自然。

现在，考虑这些代码：

```
SquareMatrix<double,5> sm1;
...
sm1.invert( );                //调用 SquareMatrix<double,5>::invert
SquareMatrix<double,10> sm2;
...
sm2.invert( );                //调用 SquareMatrix<double,10>::invert
```

这会具现化两份 invert。这些函数并非完完全全相同，因为其中一个操作的是 5*5 矩阵而另一个操作的是 10*10 矩阵，但除了常量 5 和 10，两个函数的其他部分完全相同。这是 template 引出代码膨胀的一个典型例子。

如果你看到两个函数完全相同，只除了一个使用 5 而另一个使用 10，你会怎么做？你的本能会为它们建立一个带数值参数的函数，然后以 5 和 10 来调用这个带参数的函数，而不重复代码。你的本能很好，下面是对 SquareMatrix 的第一次修改：

```
template<typename T>                          //与尺寸无关的 base class，
class SquareMatrixBase {                      //用于正方矩阵
protected:
   ...
   void invert(std::size_t matrixSize);       //以给定的尺寸求逆矩阵
   ...
};

template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
private:
   using SquareMatrixBase<T>::invert;         //避免遮掩 base 版的
                                              //invert；见条款 33
public:
   ...
   void invert( ) { this->invert(n); }        //制造一个 inline 调用，调用
};                                            //base class 版的 invert。稍后
                                              //说明为什么这儿出现 this->
```

就如你所看到，带参数的 invert 位于 base class SquareMatrixBase 中。和 SquareMatrix 一样，SquareMatrixBase 也是个 template，不同的是它只对“矩阵元素对象的类型”参数化，不对矩阵的尺寸参数化。因此对于某给定之元素对象类型，所有矩阵共享同一个（也是唯一一个）SquareMatrixBase class。它们也将因此共享这唯一一个 class 内的 invert。

SquareMatrixBase::invert 只是企图成为“避免 derived classes 代码重复”的一种方法，所以它以 protected 替换 public。调用它而造成的额外成本应该是 0，因为 derived classes 的 inverts 调用 base class 版本时用的是 inline 调用（这里的 inline 是隐晦的，见条款 30）。这些函数使用 "this->" 记号，因为若不这样做，便如条款 43 所说，模板化基类（templatized base classes，例如 SquareMatrixBase<T>）内的函数名称会被 derived classes 掩盖。也请注意 SquareMatrix 和 SquareMatrixBase 之间的继承关系是 private。这反应一个事实：

这里的 base class 只是为了帮助 derived classes 实现，不是为了表现 SquareMatrix 和 SquareMatrixBase 之间的 **is-a** 关系（关于 private 继承，见条款 39）。

目前为止一切都好，但还有一些棘手的题目没有解决。SquareMatrixBase::invert 如何知道该操作什么数据？虽然它从参数中知道矩阵尺寸，但它如何知道哪个特定矩阵的数据在哪儿？想必只有 derived class 知道。Derived class 如何联络其 base class 做逆运算动作？一个可能的做法是为 SquareMatrixBase::invert 添加另一个参数，也许是个指针，指向一块用来放置矩阵数据的内存起始点。那行得通，但十之八九 invert 不是唯一一个可写为“形式与尺寸无关并可移至 SquareMatrixBase 内”的 SquareMatrix 函数。如果有若干这样的函数，我们唯一要做的就是找出保存矩阵元素值的那块内存。我们可以对所有这样的函数添加一个额外参数，却得一次又一次地告诉 SquareMatrixBase 相同的信息，这样似乎不好。

另一个办法是令 SquareMatrixBase 贮存一个指针，指向矩阵数值所在的内存。而只要它存储了那些东西，也就可能存储矩阵尺寸。成果看起来像这样：

```
template<typename T>
class SquareMatrixBase {
protected:
   SquareMatrixBase(std::size_t n, T* pMem)      //存储矩阵大小和一个
   : size(n), pData(pMem) { }                    //指针，指向矩阵数值。
   void setDataPtr(T* ptr) { pData = ptr; }      //重新赋值给 pData。
   ...
   private:
   std::size_t size;                             //矩阵的大小。
   T* pData;                                     //指针，指向矩阵内容。
};
```

这允许 derived classes 决定内存分配方式。某些实现版本也许会决定将矩阵数据存储在 SquareMatrix 对象内部：

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
   SquareMatrix( )                               //送出矩阵大小和
     : SquareMatrixBase<T>(n, data) { }          //数据指针给 base class。
   ...
private:
   T data[n*n];
};
```

这种类型的对象不需要动态分配内存，但对象自身可能非常大。另一种做法是把每一个矩阵的数据放进 heap（也就是通过 new 来分配内存）：

```
template<typename T, std::size_t n>
class SquareMatrix: private SquareMatrixBase<T> {
public:
    SquareMatrix( )                          //将 base class 的数据指针设为 null，
      : SquareMatrixBase<T>(n, 0),           //为矩阵内容分配内存，
    pData(new T[n*n])                        //将指向该内存的指针存储起来，
    { this->setDataPtr(pData.get()); } //然后将它的一个副本交给 base class
    ...
private:
    boost::scoped_array<T> pData;            //关于boost::scoped_array，
};                                           //见条款 13。
```

不论数据存储于何处，从膨胀的角度检讨之，关键是现在许多——说不定是所有——SquareMatrix 成员函数可以单纯地以 inline 方式调用 base class 版本，后者由“持有同型元素”（不论矩阵大小）之所有矩阵共享。在此同时，不同大小的 SquareMatrix 对象有着不同的类型，所以即使（例如 SquareMatrix<double,5>和 SquareMatrix<double,10>）对象使用相同的 SquareMatrixBase<double> 成员函数，我们也没机会传递一个 SquareMatrix<double,5> 对象到一个期望获得 SquareMatrix<double,10> 的函数去。很棒，对吗？

是的，很棒，但必须付出代价。硬是绑着矩阵尺寸的那个 invert 版本，有可能生成比共享版本（其中尺寸乃以函数参数传递或存储在对象内）更佳的代码。例如在尺寸专属版中，尺寸是个编译期常量，因此可以藉由常量的广传达到最优化，包括把它们折进被生成指令中成为直接操作数。这在“与尺寸无关”的版本中是无法办到的。

从另一个角度看，不同大小的矩阵只拥有单一版本的 invert，可减少执行文件大小，也就因此降低程序的 working set（译注：见下说明）大小，并强化指令高速缓存区内的引用集中化（locality of reference）。这些都可能使程序执行得更快速，超越“尺寸专属版”invert 的最优化效果。哪一个影响占主要地位？欲知答案，唯一的办法是两者都尝试并观察你的平台的行为以及面对代表性数据组时的行为。

> 译注：所谓 working set 是指对一个在“虚内存环境”下执行的进程（process）而言，其所使用的那一组内存页（pages）。

另一个效能评比所关心的主题是对象大小。如果你不介意，可将前述“与矩阵大小无关的函数版本”搬至 base class 内，这会增加每一个对象的大小。例如在我

刚才展示的例子中，每一个 SquareMatrix 对象都有一个指针指向 SquareMatrixBase class 内的数据。虽然每个 derived class 已经有一种取得数据的办法，这会对每一个 SquareMatrix 对象增加至少一个指针那么大。当然也可以修改设计，拿掉这些指针，但是再一次，这其中需要若干取舍。例如令 base class 贮存一个 protected 指针指向矩阵数据，会导致丧失封装性，如条款 22 所言。也可能导致资源管理上的混乱和复杂；是的，如果 base class 存储一个指针指向矩阵数据，那些数据空间也许是动态分配获得，也许存储于 derived class 对象内（如稍早所见），如何判断这个指针该不该被删除呢？这样的问题有其答案，但你愈是尝试精密的做法，事情变得愈是复杂。从某个角度看，一点点代码重复反倒看起来有点幸运了。

这个条款只讨论由 non-type template parameters（非类型模板参数）带来的膨胀，其实 type parameters（类型参数）也会导致膨胀。例如在许多平台上 int 和 long 有相同的二进制表述，所以像 vector<int> 和 vector<long> 的成员函数有可能完全相同——这正是膨胀的最佳定义。某些连接器（linkers）会合并完全相同的函数实现码，但有些不会，后者意味某些 templates 被具现化为 int 和 long 两个版本，并因此造成代码膨胀（在某些环境下）。类似情况，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*>, list<SquareMatrix<long,3>*> 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。某些 C++ 标准程序库实现版本的确为 vector, deque 和 list 等 templates 做了这件事。如果你关心你的 templates 可能出现代码膨胀，也许你会想让你的 templates 也做相同的事情。

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。
- 因非类型模板参数（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。

条款 45：运用成员函数模板接受所有兼容类型

Use member function templates to accept "all compatible types."

所谓智能指针（*Smart pointers*）是“行为像指针”的对象，并提供指针没有的机能。例如条款 13 曾经提及 `std::auto_ptr` 和 `tr1::shared_ptr` 如何能够被用来在正确时机自动删除 heap-based 资源。STL 容器的迭代器几乎总是智能指针；无疑地你不会奢望使用 "++" 将一个内置指针从 linked list 的某个节点移到另一个节点，但这在 `list::iterators` 身上办得到。

真实指针做得很好的一件事是，支持隐式转换（implicit conversions）。Derived class 指针可以隐式转换为 base class 指针，“指向 non-const 对象”的指针可以转换为“指向 const 对象”……等等。下面是可能发生于三层继承体系的一些转换：

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top* pt1 = new Middle;                  //将 Middle* 转换为 Top*
Top* pt2 = new Bottom;                  //将 Bottom* 转换为 Top*
const Top* pct2 = pt1;                  //将 Top* 转换为 const Top*
```

但如果想在用户自定的智能指针中模拟上述转换，稍稍有点麻烦。我们希望以下代码通过编译：

```
template<typename T>
class SmartPtr {
   public:                              //智能指针通常
   explicit SmartPtr(T* realPtr);       //以内置（原始）指针完成初始化
   ...
};

SmartPtr<Top> pt1 =                     //将 SmartPtr<Middle> 转换为
   SmartPtr<Middle>(new Middle);        // SmartPtr<Top>
SmartPtr<Top> pt2 =                     //将 SmartPtr<Bottom> 转换为
   SmartPtr<Bottom>(new Bottom);        // SmartPtr<Top>
SmartPtr<const Top> pct2 = pt1;         //将 SmartPtr<Top> 转换为
                                        // SmartPtr<const Top>
```

但是，同一个 template 的不同具现体（instantiations）之间并不存在什么与生俱来的固有关系（译注：这里意指如果以带有 base-derived 关系的 B, D 两类型分别具现化某个 template，产生出来的两个具现体并不带有 base-derived 关系），所以编译器视 `SmartPtr<Middle>` 和 `SmartPtr<Top>` 为完全不同的 classes，它们之间的关系并不比……唔……并不比 `vector<float>` 和 `Widget` 更密切，呵呵。为了获得我们希望获得的 `SmartPtr` classes 之间的转换能力，我们必须将它们明确地编写出来。

Templates 和泛型编程（Generic Programming）

在上述智能指针实例中，每一个语句创建了一个新式智能指针对象，所以现在我们应该关注如何编写智能指针的构造函数，使其行为能够满足我们的转型需要。一个很具关键的观察结果是：我们永远无法写出我们需要的所有构造函数。在上述继承体系中，我们根据一个 SmartPtr<Middle> 或一个 SmartPtr<Bottom> 构造出一个 SmartPtr<Top>，但如果这个继承体系未来有所扩充，SmartPtr<Top> 对象又必须能够根据其他智能指针构造自己。假设日后添加了：

```
class BelowBottom: public Bottom { ... };
```

我们因此必须令 SmartPtr<BelowBottom> 对象得以生成 SmartPtr<Top> 对象，但我们当然不希望一再修改 SmartPtr template 以满足此类需求。

就原理而言，此例中我们需要的构造函数数量没有止尽，因为一个 template 可被无限量具现化，以致生成无限量函数。因此，似乎我们需要的不是为 SmartPtr 写一个构造函数，而是为它写一个构造模板。这样的模板（templates）是所谓 *member function templates*（常简称为 *member templates*），其作用是为 class 生成函数：

```
template<typename T>
class SmartPtr {
public:
   template<typename U>                        //member template,
   SmartPtr(const SmartPtr<U>& other);         //为了生成 copy 构造函数
   ...
};
```

以上代码的意思是，对任何类型 T 和任何类型 U，这里可以根据 SmartPtr<U> 生成一个 SmartPtr<T> ——因为 SmartPtr<T> 有个构造函数接受一个 SmartPtr<U> 参数。这一类构造函数根据对象 u 创建对象 t（例如根据 SmartPtr<U> 创建一个 SmartPtr<T>），而 u 和 v 的类型是同一个 template 的不同具现体，有时我们称之为泛化（*generalized*）*copy* 构造函数。

上面的泛化 *copy* 构造函数并未被声明为 explicit。那是蓄意的，因为原始指针类型之间的转换（例如从 derived class 指针转为 base class 指针）是隐式转换，无需明白写出转型动作（cast），所以让智能指针仿效这种行径也属合理。在模板化构造函数（templatized constructor）中略去 explicit 就是为了这个目的。

完成声明之后，这个为 SmartPtr 而写的“泛化 *copy* 构造函数”提供的东西比我们需要的更多。是的，我们希望根据一个 SmartPtr<Bottom> 创建一个 SmartPtr<Top>，却不希望根据一个 SmartPtr<Top> 创建一个 SmartPtr<Bottom>，因为那对 public 继承而言（见条款 32）是矛盾的。我们也不希望根据一个

SmartPtr<double>创建一个 SmartPtr<int>，因为现实中并没有“将 int*转换为 double*”的对应隐式转换行为。是的，我们必须从某方面对这一 member template 所创建的成员函数群进行拣选或筛除。

假设 SmartPtr 遵循 auto_ptr 和 tr1::shared_ptr 所提供的榜样，也提供一个 get 成员函数，返回智能指针对象（见条款 15）所持有的那个原始指针的副本，那么我们可以在“构造模板”实现代码中约束转换行为，使它符合我们的期望：

```
template<typename T>
class SmartPtr {
public:
  template<typename U>
  SmartPtr(const SmartPtr<U>& other)   //以 other 的 heldPtr
    : heldPtr(other.get()) { ... }     //  初始化 this 的 heldPtr
  T* get() const { return heldPtr; }
    ...
private:
  T* heldPtr;                  //这个 SmartPtr 持有的内置（原始）指针
};
```

我使用成员初值列（member initialization list）来初始化 SmartPtr<T> 之内类型为 T* 的成员变量，并以类型为 U* 的指针（由 SmartPtr<U> 持有）作为初值。这个行为只有当“存在某个隐式转换可将一个 U*指针转为一个 T*指针”时才能通过编译，而那正是我们想要的。最终效益是 SmartPtr<T> 现在有了一个泛化 *copy* 构造函数，这个构造函数只在其所获得的实参隶属适当（兼容）类型时才通过编译。

member function templates（成员函数模板）的效用不限于构造函数，它们常扮演的另一个角色是支持赋值操作。例如 TR1 的 shared_ptr（见条款 13）支持所有“来自兼容之内置指针、tr1::shared_ptrs、auto_ptrs 和 tr1::weak_ptrs（见条款 54）”的构造行为，以及所有来自上述各物（tr1::weak_ptrs 除外）的赋值操作。下面是 TR1 规范中关于 tr1::shared_ptr 的一份摘录，其中强烈倾向声明 template 参数时采用关键字 class 而不采用 typename（条款 42 曾说过，两者的意义在此语境下完全相同）。

```
template<class T>
class shared_ptr {
public:
  template<class Y>                                  //构造，来自任何兼容的
    explicit shared_ptr(Y* p);                       //内置指针、
  template<class Y>
    shared_ptr(shared_ptr<Y> const& r);              //或 shared_ptr、
  template<class Y>
    explicit shared_ptr(weak_ptr<Y> const& r);       //或 weak_ptr、
  template<class Y>
    explicit shared_ptr(auto_ptr<Y>& r);             //或 auto_ptr.
```

```
  template<class Y>                                    //赋值，来自任何兼容的
    shared_ptr& operator=(shared_ptr<Y> const& r);     //shared_ptr、
  template<class Y>                                    //或auto_ptr.
    shared_ptr& operator=(auto_ptr<Y>& r);
  ...
};
```

上述所有构造函数都是 explicit，惟有“泛化 *copy* 构造函数”除外。那意味从某个 shared_ptr 类型隐式转换至另一个 shared_ptr 类型是被允许的，但从某个内置指针或从其他智能指针类型进行隐式转换则不被认可（如果是显式转换如 *cast* 强制转型动作倒是可以）。另一个趣味点是传递给 tr1::shared_ptr 构造函数和 *assignment* 操作符的 auto_ptrs 并未被声明为 const，与之形成对比的则是 tr1::shared_ptrs 和 tr1::weak_ptrs 都以 const 传递。这是因为条款 13 说过，当你复制一个 auto_ptrs，它们其实被改动了。

member function templates（成员函数模板）是个奇妙的东西，但它们并不改变语言基本规则。条款 5 说过，编译器可能为我们产生四个成员函数，其中两个是 *copy* 构造函数和 *copy assignment* 操作符。现在，tr1::shared_ptr 声明了一个泛化 *copy* 构造函数，而显然一旦类型 T 和 Y 相同，泛化 *copy* 构造函数会被具现化为“正常的” *copy* 构造函数。那么究竟编译器会暗自为 tr1::shared_ptr 生成一个 *copy* 构造函数呢？或当某个 tr1::shared_ptr 对象根据另一个同型的 tr1::shared_ptr 对象展开构造行为时，编译器会将“泛化 *copy* 构造函数模板”具现化呢？

一如我所说，member templates 并不改变语言规则，而语言规则说，如果程序需要一个 *copy* 构造函数，你却没有声明它，编译器会为你暗自生成一个。在 class 内声明泛化 *copy* 构造函数（是个 member template）并不会阻止编译器生成它们自己的 *copy* 构造函数（一个 non-template），所以如果你想要控制 *copy* 构造的方方面面，你必须同时声明泛化 *copy* 构造函数和“正常的” *copy* 构造函数。相同规则也适用于赋值（*assignment*）操作。下面是 tr1::shared_ptr 的一份定义摘要，例证上述所言：

```
template<class T>
class shared_ptr {
public:
   shared_ptr(shared_ptr const& r);                  //copy构造函数.
   template<class Y>                                 //泛化 copy构造函数.
   shared_ptr(shared_ptr<Y> const& r);
   shared_ptr& operator=(shared_ptr const& r);       //copy assignment.
   template<class Y>                                 //泛化 copy assignment.
      shared_ptr& operator=(shared_ptr<Y> const& r);
   ...
};
```

请记住

- 请使用 member function templates（成员函数模板）生成“可接受所有兼容类型”的函数。
- 如果你声明 member templates 用于“泛化 *copy* 构造”或“泛化 *assignment* 操作”，你还是需要声明正常的 *copy* 构造函数和 *copy assignment* 操作符。

条款 46：需要类型转换时请为模板定义非成员函数

Define non-member functions inside templates when type conversions are desired.

条款 24 讨论过为什么惟有 non-member 函数才有能力“在所有实参身上实施隐式类型转换”，该条款并以 Rational class 的 operator* 函数为例。我强烈建议你继续看下去之前先让自己熟稔那个例子，因为本条款首先以一个看似无害的改动扩充条款 24 的讨论；本条款将 Rational 和 operator* 模板化了：

```
template<typename T>
  class Rational {
  public:
     Rational(const T& numerator = 0,          //条款 20 告诉你为什么参数以
              const T& denominator = 1);       // passed by reference 方式传递。
     const T numerator() const;                //条款 28 告诉你为什么返回值
     const T denominator() const;              // 以 passed by value 方式传递。
     ...                                       //条款 3 告诉你为什么它们是 const.
  };

  template<typename T>
  const Rational<T> operator* (const Rational<T>& lhs,
                               const Rational<T>& rhs)
  { ... }
```

就像条款 24 一样，我们希望支持混合式（mixed-mode）算术运算，所以我们希望以下代码顺利通过编译。我们也预期它会，因为它正是条款 24 所列的同一份代码，唯一不同的是 Rational 和 operator* 如今都成了 templates：

```
Rational<int> oneHalf(1, 2);          //这个例子来自条款 24，
                                      //唯一不同是 Rational 改为 template。
Rational<int> result = oneHalf * 2;   //错误！无法通过编译。
```

上述失败给我们的启示是，模板化的 Rational 内的某些东西似乎和其 non-template 版本不同。事实的确如此。在条款 24 内，编译器知道我们尝试调用什么函数（就是接受两个 Rationals 参数的那个 operator* 啦），但这里编译器不知道我们想要调用哪个函数。取而代之的是，它们试图想出什么函数被名为 operator*

的 template 具现化(产生)出来。它们知道它们应该可以具现化某个“名为 operator* 并接受两个 Rational<T>参数”的函数，但为完成这一具现化行动，必须先算出 T 是什么。问题是它们没有这个能耐。

为了推导 T，它们看了看 operator* 调用动作中的实参类型。本例中那些类型分别是 Rational<int>（oneHalf 的类型）和 int（2 的类型）。每个参数分开考虑。

以 oneHalf 进行推导，过程并不困难。operator* 的第一参数被声明为 Rational <T>，而传递给 operator* 的第一实参（oneHalf）的类型是 Rational<int>，所以 T 一定是 int。其他参数的推导则没有这么顺利。operator* 的第二参数被声明为 Rational<T>，但传递给 operator* 的第二实参（2）类型是 int。编译器如何根据这个推算出 T？你或许会期盼编译器使用 Rational<int> 的 non-explicit 构造函数将 2 转换为 Rational<int>，进而将 T 推导为 int，但它们不那么做，因为在 template 实参推导过程中从不将隐式类型转换函数纳入考虑。绝不！这样的转换在函数调用过程中的确被使用，但在能够调用一个函数之前，首先必须知道那个函数存在。而为了知道它，必须先为相关的 function template 推导出参数类型（然后才可将适当的函数具现化出来）。然而 template 实参推导过程中并不考虑采纳“通过构造函数而发生的”隐式类型转换。条款 24 不涉及 templates，所以 template 实参推导不成为讨论议题。现在我们却是处在 template part of C++(见条款 1）领域内，template 实参推导是我们的重大议题。

只要利用一个事实，我们就可以缓和编译器在 template 实参推导方面受到的挑战：template class 内的 friend 声明式可以指涉某个特定函数。那意味 class Rational<T>可以声明 operator* 是它的一个 friend 函数。Class templates 并不倚赖 template 实参推导（后者只施行于 function templates 身上），所以编译器总是能够在 class Rational<T> 具现化时得知 T。因此，令 Rational<T> class 声明适当的 operator*为其 friend 函数，可简化整个问题：

```
template<typename T>
class Rational {
public:
   ...
```

```
    friend                                                    //声明
    const Rational operator*(const Rational& lhs,             //operator* 函数,
                             const Rational& rhs);            //细节详下。
};
template<typename T>                                          //定义
const Rational<T> operator*(const Rational<T>& lhs,          //operator* 函数。
                            const Rational<T>& rhs)
{ ... }
```

现在对 operator* 的混合式调用可以通过编译了，因为当对象 oneHalf 被声明为一个 Rational<int>，class Rational<int>于是被具现化出来，而作为过程的一部分，friend 函数 operator*（接受 Rational<int>参数）也就被自动声明出来。后者身为一个函数而非函数模板（function template），因此编译器可在调用它时使用隐式转换函数（例如 Rational 的 non-explicit 构造函数），而这便是混合式调用之所以成功的原因。

但是，此情境下的“成功”是个有趣的字眼，因为虽然这段代码通过编译，却无法连接。稍后我马上回来处理这个问题，首先我要谈谈在 Rational 内声明 operator* 的语法。

在一个 class template 内，template 名称可被用来作为“template 和其参数”的简略表达方式，所以在 Rational<T> 内我们可以只写 Rational 而不必写 Rational<T>。本例中这只节省我们少打几个字，但若出现许多参数，或参数名称很长，这可以节省我们的时间，也可以让代码比较干净。我谈这个是因为，本例中的 operator* 被声明为接受并返回 Rationals（而非 Rational<T>s）。如果它被声明如下，一样有效：

```
template<typename T>
class Rational {
public:
   ...
friend
   const Rational<T> operator*(const Rational<T>& lhs,
                               const Rational<T>& rhs);
   ...
};
```

然而使用简略表达方式（速记式）比较轻松也比较普遍。

现在回头想想我们的问题。混合式代码通过了编译，因为编译器知道我们要调用哪个函数（就是接受一个 Rational<int> 以及又一个 Rational<int> 的那个

operator*），但那个函数只被声明于 Rational 内，并没有被定义出来。我们意图令此 class 外部的 operator* template 提供定义式，但是行不通——如果我们自己声明了一个函数（那正是 Rational template 内的作为），就有责任定义那个函数。既然我们没有提供定义式，连接器当然找不到它！

或许最简单的可行办法就是将 operator* 函数本体合并至其声明式内：

```
template<typename T>
class Rational {
public:
...
friend const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
   return Rational(lhs.numerator() * rhs.numerator( ),      //实现码与
                   lhs.denominator() * rhs.denominator()); //条款 24 同
}
};
```

这便如同我们所期望地正常运作了起来：对 operator*的混合式调用现在可编译连接并执行。万岁！

这项技术的一个趣味点是，我们虽然使用 friend，却与 friend 的传统用途“访问 class 的 non-public 成分”毫不相干。为了让类型转换可能发生于所有实参身上，我们需要一个 non-member 函数（条款 24）；为了令这个函数被自动具现化，我们需要将它声明在 class 内部；而在 class 内部声明 non-member 函数的唯一办法就是：令它成为一个 friend。因此我们就这样做了。不习惯？是的。有效吗？不必怀疑。

一如条款 30 所说，定义于 class 内的函数都暗自成为 inline，包括像 operator* 这样的 friend 函数。你可以将这样的 inline 声明所带来的冲击最小化，做法是令 operator*不做任何事情，只调用一个定义于 class 外部的辅助函数。在本条款的例子中，这样做并没有太大意义，因为 operator*已经是个单行函数，但对更复杂的函数而言，那么做也许就有价值。“令 friend 函数调用辅助函数”的做法的确值得细究一番。

“Rational 是个 template”这一事实意味上述的辅助函数通常也是个 template，所以定义了 Rational 的头文件代码，很典型地长这个样子：

```
template<typename T> class Rational;        //声明 Rational template
```

```
template<typename T>                          //声明 helper template
const Rational<T> doMultiply(const Rational<T>& lhs,
                             const Rational<T>& rhs);

template<typename T>
class Rational {
public:
   ...
friend
   const Rational<T> operator*(const Rational<T>& lhs,
                               const Rational<T>& rhs)
   { return doMultiply(lhs, rhs); }           //令 friend 调用 helper
   ...
};
```

许多编译器实质上会强迫你把所有 template 定义式放进头文件内，所以你或许需要在头文件内定义 doMultiply（一如条款 30 所言，这样的 templates 不需非得是 inline 不可），看起来像这样：

```
template<typename T>                                    //若有必要，
const Rational<T> doMultiply(const Rational<T>& lhs,   //在头文件内定义
                             const Rational<T>& rhs)    //helper template
{
   return Rational<T>(lhs.numerator() * rhs.numerator(),
                      lhs.denominator() * rhs.denominator());
}
```

作为一个 template，doMultiply 当然不支持混合式乘法，但它其实也不需要。它只被 operator* 调用，而 operator* 支持了混合式操作！本质上 operator* 支持了类型转换所需的任何东西，确保两个 Rational 对象能够相乘，然后它将这两个对象传给一个适当的 doMultiply template 具现体，完成实际的乘法操作。协作为成功之本，不是吗？

请记住

- 当我们编写一个 class template，而它所提供之“与此 template 相关的”函数支持“所有参数之隐式类型转换”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

Use traits classes for information about types.

STL 主要由“用以表现容器、迭代器和算法”的 templates 构成，但也覆盖若干工具性 templates，其中一个名为 advance，用来将某个迭代器移动某个给定距离：

```
template<typename IterT, typename DistT>   //将迭代器向前移动 d 单位。
void advance(IterT& iter, DistT d);        //如果 d < 0 则向后移动。
```

观念上 advance 只是做 iter += d 动作，但其实不可以全然那么实践，因为只有 *random access*（随机访问）迭代器才支持 +=操作。面对其他威力不那么强大的迭代器种类，advance 必须反复施行 ++ 或 --，共 d 次。

嗯，你不记得你的 STL 迭代器分类（categories）了吗？没关系，让我们来一次迷你回顾。STL 共有 5 种迭代器分类，对应于它们支持的操作。*Input* 迭代器只能向前移动，一次一步，客户只可读取（不能涂写）它们所指的东西，而且只能读取一次。它们模仿指向输入文件的阅读指针（read pointer）；C++ 程序库中的 istream_iterators 是这一分类的代表。*Output* 迭代器情况类似，但一切只为输出：它们只向前移动，一次一步，客户只可涂写它们所指的东西，而且只能涂写一次。它们模仿指向输出文件的涂写指针（write pointer）；ostream_iterators 是这一分类的代表。这是威力最小的两个迭代器分类。由于这两类都只能向前移动，而且只能读或写其所指物最多一次，所以它们只适合“一次性操作算法”（one-pass algorithms）。

另一个威力比较强大的分类是 *forward* 迭代器。这种迭代器可以做前述两种分类所能做的每一件事，而且可以读或写其所指物一次以上。这使得它们可施行于多次性操作算法（multi-pass algorithms）。STL 并未提供单向 linked list，但某些程序库有（通常名为 slist），而指入这种容器的迭代器就是属于 *forward* 迭代器。指入 TR1 hashed 容器（见条款 54）的也可能是这一分类。（译注：这里说“可能”是因为 hashed 容器的迭代器可为单向也可为双向，取决于实现版本。）

Bidirectional 迭代器比上一个分类威力更大：它除了可以向前移动，还可以向后移动。STL 的 list 迭代器就属于这一分类，set, multiset, map 和 multimap 的迭代器也都是这一分类。

最有威力的迭代器当属 *random access* 迭代器。这种迭代器比上一个分类威力更大的地方在于它可以执行“迭代器算术”，也就是它可以在常量时间内向前或向后跳跃任意距离。这样的算术很类似指针算术，那并不令人惊讶，因为 *random access* 迭代器正是以内置（原始）指针为榜样，而内置指针也可被当做 *random access* 迭代器使用。vector, deque 和 string 提供的迭代器都是这一分类。

对于这 5 种分类，C++标准程序库分别提供专属的卷标结构（tag struct）加以确认：

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

这些 structs 之间的继承关系是有效的 *is-a* 关系(见条款 32):是的,所有 *forward* 迭代器都是 *input* 迭代器,依此类推。很快我们会看到这个继承关系的效力。

现在回到 advance 函数。我们已经知道 STL 迭代器有着不同的能力,实现 advance 的策略之一是采用“最低但最普及”的迭代器能力,以循环反复递增或递减迭代器。然而这种做法耗费线性时间。我们知道 *random access* 迭代器支持迭代器算术运算,只耗费常量时间,因此如果面对这种迭代器,我们希望运用其优势。

我们真正希望的是以这种方式实现 advance:

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    if (iter is a random access iterator) {
        iter += d;      //针对 random access 迭代器使用迭代器算术运算
    }
    else {
        if (d >= 0) { while (d--) ++iter; }     //针对其他迭代器分类
        else { while (d++) - -iter; }           //反复调用 ++ 或 --
    }
}
```

这种做法首先必须判断 iter 是否为 *random access* 迭代器,也就是说需要知道类型 IterT 是否为 *random access* 迭代器分类。换句话说我们需要取得类型的某些信息。那就是 *traits* 让你得以进行的事:它们允许你在编译期间取得某些类型信息。

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(built-in)类型和用户自定义(user-defined)类型的表现必须一样好。举个例子,如果上述 advance 收到的实参是一个指针(例如 const char*)和一个 int,上述 advance 仍然必须有效运作,那意味 traits 技术必须也能够施行于内置类型如指针身上。

“traits 必须能够施行于内置类型”意味“类型内的嵌套信息(nesting information)”这种东西出局了,因为我们无法将信息嵌套于原始指针内。因此类型的 traits 信息必须位于类型自身之外。标准技术是把它放进一个 template 及其一或多个特化版本中。这样的 templates 在标准程序库中有若干个,其中针对迭代器者被命名为 iterator_traits:

```
template<typename IterT>          //template，用来处理
struct iterator_traits;           //迭代器分类的相关信息
```

如你所见，iterator_traits 是个 struct。是的，习惯上 traits 总是被实现为 structs，但它们却又往往被称为 traits *classes*。

iterator_traits 的运作方式是，针对每一个类型 IterT，在 struct iterator_traits<IterT> 内一定声明某个 typedef 名为 iterator_category。这个 typedef 用来确认 IterT 的迭代器分类。

iterator_traits 以两个部分实现上述所言。首先它要求每一个“用户自定义的迭代器类型”必须嵌套一个 typedef，名为 iterator_category，用来确认适当的卷标结构（tag struct）。例如 deque 的迭代器可随机访问，所以一个针对 deque 迭代器而设计的 class 看起来会是这样子：

```
template < ... >                  //略而未写 template 参数
class deque {
public:
   class iterator {
   public:
        typedef random_access_iterator_tag iterator_category;
        ...
   };
   ...
};
```

list 的迭代器可双向行进，所以它们应该是这样：

```
template < ... >
class list {
public:
   class iterator {
   public:
        typedef bidirectional_iterator_tag iterator_category;
        ...
   };
   ...
};
```

至于 iterator_traits，只是鹦鹉学舌般地响应 iterator class 的嵌套式 typedef：

```
//类型 IterT 的 iterator_category 其实就是用来表现“IterT 说它自己是什么”。
//关于 "typedef typename" 的运用，见条款 42。
template<typename IterT>
struct iterator_traits {
   typedef typename IterT::iterator_category iterator_category;
   ...
};
```

这对用户自定义类型行得通，但对指针（也是一种迭代器）行不通，因为指针不可能嵌套 typedef。iterator_traits 的第二部分如下，专门用来对付指针。

为了支持指针迭代器，iterator_traits 特别针对指针类型提供一个偏特化版本（*partial template specialization*）。由于指针的行径与 ***random access*** 迭代器类似，所以 iterator_traits 为指针指定的迭代器类型是：

```
template<typename IterT>                    //template 偏特化
struct iterator_traits<IterT*>              //针对内置指针
{
   typedef random_access_iterator_tag iterator_category;
   ...
};
```

现在，你应该知道如何设计并实现一个 traits class 了：

- 确认若干你希望将来可取得的类型相关信息。例如对迭代器而言，我们希望将来可取得其分类（category）。
- 为该信息选择一个名称（例如 iterator_category）。
- 提供一个 template 和一组特化版本（例如稍早说的 iterator_traits），内含你希望支持的类型相关信息。

好，现在有了 iterator_traits（实际上是 std::iterator_traits，因为它是 C++标准程序库的一部分），我们可以对 advance 实践先前的伪码（pseudocode）：

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

虽然这看起来前景光明，其实并非我们想要。首先它会导致编译问题，但我将在条款 48 才探讨这一点，此刻有更根本的问题要考虑。IterT 类型在编译期间获知，所以 iterator_traits<IterT>::iterator_category 也可在编译期间确定。但 if 语句却是在运行期才会核定。为什么将可在编译期完成的事延到运行期才做呢？这不仅浪费时间，也造成可执行文件膨胀。

我们真正想要的是一个条件式（也就是一个 if...else 语句）判断“编译期核定成功”之类型。恰巧 C++ 有一个取得这种行为的办法，那就是重载（overloading）。

当你重载某个函数 f，你必须详细叙述各个重载件的参数类型。当你调用 f，编译器便根据传来的实参选择最适当的重载件。编译器的态度是“如果这个重载件最匹配传递过来的实参，就调用这个 f；如果那个重载件最匹配，就调用那个 f；如果第三个 f 最匹配，就调用第三个 f！”依此类推。看到了吗，这正是一个针对类型而发生的“编译期条件句”。为了让 advance 的行为如我们所期望，我们需要做的是产生两版重载函数，内含 advance 的本质内容，但各自接受不同类型的 iterator_category 对象。我将这两个函数取名为 doAdvance：

```
template<typename IterT, typename DistT>          //这份实现用于
void doAdvance(IterT& iter, DistT d,              //random access
               std::random_access_iterator_tag)   //迭代器
{
   iter += d;
}

template<typename IterT, typename DistT>          //这份实现用于
void doAdvance(IterT& iter, DistT d,              //bidirectional
               std::bidirectional_iterator_tag)   //迭代器
{
   if (d >= 0) { while (d--) ++iter; }
   else { while (d++) --iter; }
}

template<typename IterT, typename DistT>          //这份实现用于
void doAdvance(IterT& iter, DistT d,              //input迭代器
               std::input_iterator_tag)
{
   if (d < 0 ) {
       throw std::out_of_range("Negative distance");      //详下
   }
   while (d--) ++iter;
}
```

由于 forward_iterator_tag 继承自 input_iterator_tag，所以上述 doAdvance 的 input_iterator_tag 版本也能够处理 *forward* 迭代器。这是 iterator_tag structs 继承关系带来的一项红利。实际上这也是 public 继承带来的部分好处：针对 base class 编写的代码用于 derived class 身上也行得通。

advance 函数规范说，如果面对的是 *random access* 和 *bidirectional* 迭代器，则接受正距离和负距离；但如果面对的是 *forward* 或 *input* 迭代器，则移动负距离会导致不明确（未定义）行为。我所检验过的实现码都假设 d 不为负，于是直接进入一个冗长的循环迭代，等待计数器降为 0。上述代码中我以抛出异常取而代之。两种做法都有根据，但“无法预言发生何事”是“不明确行为”之祸源所在。

有了这些 doAdvance 重载版本，advance 需要做的只是调用它们并额外传递一个对象，后者必须带有适当的迭代器分类。于是编译器运用重载解析机制（overloading resolution）调用适当的实现代码：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    doAdvance(                    //调用的 doAdvance 版本
      iter, d,                    //对 iter 之迭代器分类而言
      typename                    //必须是适当的。
        std::iterator_traits<IterT>::iterator_category()
    );
}
```

现在我们可以总结如何使用一个 traits class 了：

- 建立一组重载函数（身份像劳工）或函数模板（例如 doAdvance），彼此间的差异只在于各自的 traits 参数。令每个函数实现码与其接受之 traits 信息相应和。
- 建立一个控制函数（身份像工头）或函数模板（例如 advance），它调用上述那些“劳工函数”并传递 traits class 所提供的信息。

Traits 广泛用于标准程序库。其中当然有上述讨论的 iterator_traits，除了供应 iterator_category 还供应另四份迭代器相关信息（其中最有用的是 value_type，见条款 42）。此外还有 char_traits 用来保存字符类型的相关信息，以及 numeric_limits 用来保存数值类型的相关信息，例如某数值类型可表现之最小值和最大值等等；命名为 numeric_limits 有点让人惊讶，因为 traits classes 的名称常以 "traits" 结束，但 numeric_limits 却没有遵守这种风格。

TR1（条款 54）导入许多新的 traits classes 用以提供类型信息，包括 is_fundamental <T>（判断 T 是否为内置类型），is_array<T>（判断 T 是否为数组类型），以及 is_base_of<T1, T2>（T1 和 T2 相同，抑或 T1 是 T2 的 base class）。总计 TR1 一共为标准 C++ 添加了 50 个以上的 traits classes。

请记住

- Traits classes 使得“类型相关信息”在编译期可用。它们以 templates 和“templates 特化”完成实现。
- 整合重载技术（overloading）后，traits classes 有可能在编译期对类型执行 if...else 测试。

条款 48：认识 template 元编程

Be aware of template metaprogramming.

Template metaprogramming（TMP，模板元编程）是编写 template-based C++ 程序并执行于编译期的过程。花一分钟想想这个：所谓 template metaprogram（模板元程序）是以 C++ 写成、执行于 C++ 编译器内的程序。一旦 TMP 程序结束执行，其输出，也就是从 templates 具现出来的若干 C++ 源码，便会一如往常地被编译。

如果这没有带给你异乎寻常的印象，你一定是没有足够认真地思考它。

C++ 并非是为 template metaprogramming 而设计，但自从 TMP 于 1990s 初期被发现以后，由于日渐被证明十分有用，其延伸部分很可能加入语言和标准程序库内，使 TMP 更容易进行。是的，TMP 是被发现而不是被发明出来的。当 templates 加入 C++ 时 TMP 底层特性也就被引进了。对某些人而言唯一需要注意的是如何以熟练巧妙而意想不到的方式使用 TMP。

TMP 有两个伟大的效力。第一，它让某些事情更容易。如果没有它，那些事情将是困难的，甚至不可能的。第二，由于 template metaprograms 执行于 C++ 编译期，因此可将工作从运行期转移到编译期。这导致的一个结果是，某些错误原本通常在运行期才能侦测到，现在可在编译期找出来。另一个结果是，使用 TMP 的 C++ 程序可能在每一方面都更高效：较小的可执行文件、较短的运行期、较少的内存需求。然而将工作从运行期移转至编译期的另一个结果是，编译时间变长了。是的，程序如果使用 TMP，其编译时间可能远长于不使用 TMP 的对应版本。

考虑 p.228 导入的 STL advance 伪码（位于条款 47。或许你会想现在就阅读它，因为本条款中我假设你已经熟悉条款 47 的内容）。就像 p.228 所示，我特别强调那段代码的伪码部分（pseudo part）：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
    if (iter is a random access iterator) {
        iter += d;          //针对 random access 迭代器使用迭代器算术运算
    }
    else {
        if (d >= 0) { while (d--) ++iter; }     //针对其他迭代器类型
        else { while (d++) - -iter; }           //反复调用 ++ 或 --
    }
}
```

我们可以使用 typeid 让其中的伪码成真，取得 C++ 对此问题的一个“正常”解决方案——所有工作都在运行期进行：

```
template<typename IterT, typename DistT>
void advance(IterT& iter, DistT d)
{
 if (typeid(typename std::iterator_traits<IterT>::iterator_category)
      == typeid(std::random_access_iterator_tag)) {
    iter += d;          //针对 random access 迭代器，使用迭代器算术运算。
 }
 else {
    if (d >= 0) { while (d--) ++iter; }    //针对其他迭代器分类
    else { while (d++) - -iter; }          //反复调用 ++ 或 --
 }
}
```

条款 47 指出，这个 typeid-based 解法的效率比 traits 解法低，因为在此方案中，(1) 类型测试发生于运行期而非编译期，(2)“运行期类型测试”代码会出现在（或说被连接于）可执行文件中。实际上这个例子正可彰显 TMP 如何能够比“正常的”C++ 程序更高效，因为 traits 解法就是 TMP。别忘了，traits 引发“编译期发生于类型身上的 if...else 计算”。

稍早我曾谈到，某些东西在 TMP 比在“正常的”C++ 容易，对此 advance 也提供了一个好例子。条款 47 曾经提过 advance 的 typeid-based 实现方式可能导致编译期问题，下面就是个例子：

```
std::list<int>::iterator iter;
...
advance(iter, 10);      //移动 iter 向前走 10 个元素;
                        //上述实现无法通过编译。
```

下面这一版 advance 便是针对上述调用而产生的。将 template 参数 IterT 和 DistT 分别替换为 iter 和 10 的类型之后，我们得到这些：

```
void advance(std::list<int>::iterator& iter, int d)
{
 if (typeid(std::iterator_traits<std::list<int>::iterator>::iterator_category)
      == typeid(std::random_access_iterator_tag)) {
    iter += d;          //错误!
 }
 else {
      if (d >= 0) { while (d--) ++iter; }
      else        { while (d++) --iter; }
 }
}
```

问题出在我所强调的那一行代码使用了 += 操作符，那便是尝试在一个 list<int>::iterator 身上使用 +=，但 list<int>::iterator 是 *bidirectional* 迭代器（见条款 47），并不支持 +=。只有 *random access* 迭代器才支持 +=。此刻我们知道绝不会执行起 += 那一行，因为测试 typeid 的那一行总是会因为 list<int>::iterators 而失败，但编译器必须确保所有源码都有效，纵使是不会执行起来的代码！而当 iter 不是 *random access* 迭代器时 "iter += d" 无效。与此对比的是 traits-based TMP 解法，其针对不同类型而进行的代码，被拆分为不同的函数，每个函数所使用的操作（操作符）都可施行于该函数所对付的类型。

TMP 已被证明是个“图灵完全”（Turing-complete）机器，意思是它的威力大到足以计算任何事物。使用 TMP 你可以声明变量、执行循环、编写及调用函数……但这般构件相对于“正常的”C++对应物看起来很是不同，例如条款 47 展示的 TMP if...else 条件句是藉由 templates 和其特化体表现出来。不过那毕竟是汇编语言层级的 TMP。针对 TMP 而设计的程序库（例如 Boost's MPL，见条款 55）提供更高层级的语法——尽管目前还不足以让你误以为那是“正常的”C++。

为了再次浮光掠影地认识一下“事物在 TMP 中如何运作”，让我们看看循环。TMP 并没有真正的循环构件，所以循环效果系藉由递归（recursion）完成。如果你对递归不太适应，恐怕必须在大胆投入 TMP 之前先解决它。TMP 主要是个“函数式语言”（functional language），而递归之于这类语言就像电视之于美国通俗文化一样地无法分割。TMP 的递归甚至不是正常种类，因为 TMP 循环并不涉及递归函数调用，而是涉及“递归模板具现化”（recursive template instantiation）。

TMP 的起手程序是在编译期计算阶乘（factorial）。这不是个令人特别兴奋的程序，但 "hello world" 程序也不是，而两者对于语言的导入都很有帮助。TMP 的阶乘运算示范如何通过“递归模板具现化”（recursive template instantiation）实现循环，以及如何在 TMP 中创建和使用变量：

```
template<unsigned n>                        //一般情况：Factorial<n> 的值是
struct Factorial {                          //  n 乘以 Factorial<n-1> 的值。
   enum { value = n * Factorial<n-1>::value };
};

template<>                                  //特殊情况：
struct Factorial<0> {                       //Factorial<0> 的值是 1
   enum { value = 1 };
};
```

有了这个 template metaprogram（其实只是个单一的 template metafunction Factorial），只要你指涉 Factorial<n>::value 就可以得到 n 阶乘值。

循环发生在 template 具现体 Factorial<n>内部指涉另一个 template 具现体 Factorial<n-1> 之时。和所有良好递归一样，我们需要一个特殊情况造成递归结束。这里的特殊情况是 template 特化体 Factorial<0>。

每个 Factorial template 具现体都是一个 struct，每个 struct 都使用 enum hack（见条款 2）声明一个名为 value 的 TMP 变量，value 用来保存当前计算所得的阶乘值。如果 TMP 拥有真正的循环构件，value 应该在每次循环内获得更新。但由于 TMP 系以“递归模板具现化”（recursive template instantiation）取代循环，每个具现体有自己的一份 value，而每个 value 有其循环内的适当值。

你可以这样使用 Factorial：

```
int main()
{
    std::cout << Factorial<5>::value;        //印出 120
    std::cout << Factorial<10>::value;       //印出 3628800
}
```

如果你认为这比冬天吃冰淇淋还酷，你就是取得了成为一个 template metaprogrammer 的必要条件。如果 templates 和其特化版本，以及递归具现化和 enum hacks，以及键入 Factorial<n-1>::value 这样的东西会使你汗毛直竖，唔……你是个十分“正常化”的 C++ 程序员。

当然，Factorial 示范 TMP 的用途就只是像 "hello world" 示范任何传统语言的用途一样。为求领悟 TMP 之所以值得学习，很重要的一点是先对它能够达成什么目标有一个比较好的理解。下面举出三个例子：

- 确保量度单位正确。在科学和工程应用程序中，确保量度单位（例如质量、距离、时间……等等）正确结合是绝对必要的。举个例子，将一个质量变量赋值给一个速度变量是错误的，但是将一个距离变量除以一个时间变量并将结果赋值给一个速度变量则成立。如果使用 TMP，就可以确保（在编译期）程序中所有量度单位的组合都正确，不论其计算多么复杂。这也就是为什么 TMP 可被用来进行早期错误侦测。这种 TMP 用途的一个有趣情况是，就连因次为分数的指数（fractional dimensional exponents）也可支持，但分数必须先在编译期被约简，

例如 time$^{1/2}$ 的单位和 time$^{4/8}$ 的单位相同。

- 优化矩阵运算。条款 21 曾经提过某些函数包括 operator* 必须返回新对象，而条款 44 又导入了一个 SquareMatrix class。考虑以下代码：

```
typedef SquareMatrix<double, 10000> BigMatrix;
BigMatrix m1, m2, m3, m4, m5;                    //创建矩阵并
...                                              //赋予它们数值。
BigMatrix result = m1 * m2 * m3 * m4 * m5;       //计算它们的乘积。
```

以“正常的”函数调用动作来计算 result，会创建 4 个暂时性矩阵，每一个用来存储对 operator* 的调用结果。犹有进者，各自独立的乘法产生了 4 个作用于矩阵元素身上的循环。如果使用高级、与 TMP 相关的 template 技术，即所谓 *expression templates*，就有可能消除那些临时对象并合并循环，这一切都无需改变客户端的用法（像上面那样）。于是 TMP 软件使用较少的内存，执行速度又有戏剧性的提升。

- 可以生成客户定制之设计模式（custom design pattern）实现品。设计模式如 ***Strategy***（见条款 35）,***Observer***, ***Visitor*** 等等都可以多种方式实现出来。运用所谓 *policy-based design* 之 TMP-based 技术，有可能产生一些 templates 用来表述独立的设计选项（所谓 "policies"），然后可以任意结合它们，导致模式实现品带着客户定制的行为。这项技术已被用来让若干 templates 实现出智能指针的行为政策（behavioral policies），用以在编译期间生成数以百计不同的智能指针类型。这项技术已经超越编程工艺领域如设计模式和智能指针，更广义地成为 *generative programming*（殖生式编程）的一个基础。

不是每个人都喜欢 TMP。其语法不直观，其支持工具目前还不充分（template metaprograms 的调试器？哈，还早咧！）由于 TMP 是一个在相对短时间之前才意外发现的语言，其编程方式还多少需要倚赖经验。尽管如此，将工作从运行期移往编译期所带来的效率改善还是令人印象深刻，而 TMP 对“难以或甚至不可能于运行期实现出来的行为”的表现能力也很吸引人。

TMP 仿佛旭日东升。有可能下一版 C++ 会对它提供明确的支持，甚至 TR1 已经这样做了（见条款 54）。TMP 书籍已逐渐出现，网络上的 TMP 信息愈来愈丰富。

TMP 或许永远不会成为主流，但对某些程序员——特别是程序库开发人员——几乎确定会成为他们的主要粮食。

请记住

- Template metaprogramming（TMP，模板元编程）可将工作由运行期移往编译期，因而得以实现早期错误侦测和更高的执行效率。
- TMP 可被用来生成“基于政策选择组合”（based on combinations of policy choices）的客户定制代码，也可用来避免生成对某些特殊类型并不适合的代码。