C++ Template
全览

C++ Templates - The Complete Guide

说明

本书是在侯捷等翻译的繁体中文版基础上，用软件将PDF文件转换成word文件，然后将繁体转换成简体，再将一些台湾的术语改成了大陆的术语，并经过简单排版后的结果。对各位初学C++ Template的朋友应该有所帮助。

丁志强
zqding@hotmail.com

关于本书
About This Book

作为C++ 的一部分，尽管 templates（模板）已经存在二十多年（并且以其它多种面目存在了几 乎同样长的时间），但它还是会招引误解、误用和争议。在此同时，愈来愈多人觉察 templates 是产生更干净、更快速、更精明的软件的一个强而有力的手段。确实，templates 已经成为数种 新兴C++ 编程思维模型（programming paradigms）的基石。

但是我们发现，大多数现有书籍和文章对于C++ templates 的理论和应用方面的论述，显得过于 浅薄。有些书籍虽然很好地评述了各种 template-based 技法，却并没有精确阐述 C++ 语言本身 如何使这些技法得以施行。于是，无论新手或专家，都好像在和 templates 较劲，费尽心思去琢 磨为何他们的程序代码不能按想象的方式执行。

这样的观察结果，正是我们两人写这本书的主要动机之一。对于本书，我俩心中各有独立的要 旨和不同的写作方式：

David 的目标是为读者提供一份完整的参考手册，其中讲述 C++ template 语言机制的细节， 以及重要的 templates 高级编程技法。他比较关注内容的精确与完备。

Nico 希望为自己和其它日常生活中使用 templates 的程序员带来一本有帮助的书。这意味 本 书将以一种直观易懂、结合实践的方式呈现给读者。

从某种意义上，你可以把我们看做是科学家和工程师的组合：我们按照相同的原则写作，但侧 重点有些不同（当然，也有很多情况是重迭的）。

Addison-Wesley 把我俩的努力结合在一起，于是你拥有了一本我俩心目中两方面紧密结合的书籍：既是C++template 教本（tutorial），也是C++template 的详尽参考手册（reference）。作为 教本，本书不仅涵盖语言基本要素的介绍，也致力培养某种得以设计出切实可行之解决方案的 直觉。作为参考手册，本书既包括 C++ template 的语法和语意，也是各种广为人知和鲜为人知 的编程惯用手法（idioms）和编程技术的全面总览。

1.1 阅读本书之前你应该知道的事

要想具备学习本书的最佳状态，你应该先已了解 C++。本书中我们只讲述某些语言特性的细节， 并不涉及语言基础知识。你应该熟知类别（classes）、继承（inheritance）等概念，你应该能够 利用 C++ 标准库所提供的组件（例如 iostreams 和各种容器, containers）编写 C++ 程序。 如有必要，我们会回顾某些微妙问题 — 即使这些问题并不直接与 templates 相关。这可确保本 书对于专家和中级水平的程序员皆适用。

本书大部份以 1998 年的 C++*Standard*（[Standard98]）为依据，同时兼顾 C++ 标准委员会释出 的第一份技术勘误（[Standard02]）中的说明。如果你觉得你的C++ 基础还不够好，我们推荐你 阅读 [StroustrupC++PL]、[JosuttisOOP] 和 [JosuttisLib] 等书籍来充实知识。这些书非常好地讲 述了最新的C++ 语言及其标准库。你可以在附录 B.3.5 找到更多参考数据。

1.2 本书组织结构

我们的目标是为两个族群提供必要信息：(1) 准备开始使用 templates 并从中获益者，(2) 富有经验并希望紧追最新技术者。基于此种想法，我们决定将本书组织为以下四篇：

第一篇介绍 templates 涉及的基本概念。这部份采用循序渐进的教本（tutorial）方式。

第二篇展现语言细节，对 template 相关构件（constructs）来说是一份便利的参考手册。

第三篇讲述 C++ templates 的基础设计技术，从近乎微不足道的构想到精巧复杂的编程技法都有。某些内容在其它出版物中甚至从未被提到。

第四篇在前两篇的基础上讨论 templates 的各种普遍应用。 每一篇都包含数章。此外我们还提供多份附录，涵盖内容不限于 templates，例如附录 B 的「C++重载解析机制（overload resolution）」综览。

第一篇各章应该循序阅读，例如第 3 章内容就是是建立在第 2 章内容之上。其它三篇各章之间

的联系较松。例如你可以阅读 functors 那一章（第 22 章），不必先阅读 smart pointers 那一章（第 20 章）。

最后，我们提供了一份相当完备的索引，便利读者以自己的方式，跳脱任何顺序来阅读本书。

1.3 如何阅读本书

如果你是一位 C++ 程序员，打算学习或巩固templates 概念，请仔细阅读第一篇（基础篇）。即使你已经对templates 相当熟悉，快速翻阅第一篇也可以帮助你熟悉本书的写作风格和我们惯用的术语。该篇内容也可以帮助你有条理地组织你的一些templates 程序代码。

视个人学习方式的不同，你可以先消化第二篇的众多templates细节，也可以跳到第三篇领会实际程序中的templates 编程技术，再回头阅读第二篇以了解更多的语言细节。后一种方式对于那些每天和templates打交道的人可能更有好处。第四篇和第三篇有点类似，但更侧重于如何运用 templates 协助完成某个特定问题，而不是以templates 来辅助设计。钻研第四篇之前，最好先熟悉第三篇的内容。

本书附录包括了正文之中反复提及的一些内容。我们尽量使这些内容更有趣。经验显示，学习新知识的最好方法是假以实例。所以全书贯穿了诸多实例。某些实例以寥寥数行程序代码阐明一个抽象概念，某些实例则是与正文内容对应的一个完整范例。后一类实例会在开头以一行注释标明其文件名。你可以在本书支持网站 http://www.josuttis.com/tmplbook/ 中找到这些文件。

1.4 本书编程风格（Programming Style）

每一位 C++ 程序员都有自己的一套编程风格，我俩也不例外。这就引来了各种问题：哪儿应该插入空白符号、怎么摆放分隔符（大括号、小括号）…等等。我们尽量保持全书风格一致，当然有时候我们也对特殊问题作出让步。例如在教本（初阶）部份我们鼓励以空白符号和较具体的命名方式提高程序可读性，而在高阶主题中，较紧凑的风格可能更加适宜。

我们有一个他人不太常用的习惯 用以声明类型（types）、参数（parameters）和变量（variables），希望你能多加注意。下面数种方式无疑都是合理的：

```
void foo (const int &x);
void foo (const int& x);
void foo (int const &x);
void foo (int const& x);
```

尽管较为罕见，我们还是决定在表达「固定不变的整数」（constant integer）时使用 int const 而不写成 const int。这么做有两个原因，第一，这很容易显现出「什么是不能变动的（*what is constant*）」。不能变动的量总是 const饰词之前的那个东西。尽管以下两式等价：

```
const int N = 100;    //一般人可能的写法
int const N = 100;    //本书习惯写法
```

但对以下述句来说就不存在所谓的等价形式了：

```
int* const bookmark; // 指针 bookmark不能变动，但指针所指内容（int）可以变动
```

如果你把 const饰词放在运算符 * 之前，那就改变了原意。本例之中不能变动的是指针本身，不是指针所指的内容。

第二个原因和语法替换原则（syntactical substitution principle）有关，那是处理 template 程序代码时常会遭遇的问题。考虑下面两个类型定义：

```
typedef char* CHARS;
typedef CHARS const CPTR; // 一个用以「指向 chars」的 const指针
```

如果我们做文字上的替换，把 CHARS替换为其代表物，上述第二个声明的原意就得以保留：

```
typedef char* const CPTR; // 一个用以「指向 chars」的 const指针
```

然而如果我们把 const写在被修饰物之前，上述规则便不适用。考虑上述声明的另一种变化：

```
typedef char* CHARS;
typedef const CHARS CPTR; // 一个用以「指向 chars」的 const指针
```

现在，对 CHARS进行文字替换，会导出不同的含义：

```
typedef const char* CPTR; // 一个用以「指向 const chars」的指针
```

面对volatile饰词，也有同样考虑。关于空白符号，我们决定把他放在"&"符号和参数名称中间：

```
void foo (int const& x);
```

这样可以更加突出参数的类型和名称。无可否认，以下声明方式可能较易引起疑惑：

```
char *a, b;
```

根据从 C 语言继承下来的规则，a是个指针而b是个一般的 char。为了避免这种混淆，我们可以一次声明一个变量，不要集中于同一行声明语句。

本书并不是一本讨论C++标准库的书，但我们确实在一些例子中用到了标准库。一般 来说，我们使用C++特有的头文件（例如<iostream> 而非<stdio.h>）。惟一的例外是<stddef.h>，我们使用它而不使用<cstddef>，以避免类型 size_t和 ptrdiff_t被冠以 std::前缀词。这样做更具可移植性，而且 std::size_t并不比 size_t多出什么好处。

1.5 标准 vs.现实（Standard versus Reality）

*C++ Standard*自1998年末就已制定完备。但是直到2002年，仍然没有一个编译器公开宣称自己「完全符合标准」。各家编译器对于*C++ Standard* 的支持程度仍然表现各异。有些编译器可以 顺利编译本书大部份程序代码，但也有一些很流行的编译器无法编译本书中相当数量的实例。我们尽量提供另一种实现技术，使那些「只支持*C++ Standard* 子集」的编译器得以顺利工作。但是某些范例并不存在第二种实现技术。我们希望*C++ Standard* 支持问题得以解决 — 全世界程序员都迫切需要编译器厂商提供对*C++Standard* 的完整支持。

即便如此，C++ 语言仍然随着时间的推移而不断演进。已经有为数众多的 C++ 社群专家（无 论他们是否为 C++ 标准委员会成员）讨论着 C++ 语言的各种改进方案，其中有一些直接影响到 templates。本书第 13 章讨论了一些语言演化趋势。

第一篇 基本认识

The Basic

本篇介绍C++templates 的基本概念和语言特性。首先展示 function templates 和 class templates 的范例 开启对大体目标和基本概念的讨论 随后讲述其它 template 基础技术 例如 nontype template parameters、关键词 typename、以及 member templates。最后给出一些 templates 实际应用心得。

《*Object-Oriented Programming in C++*》（by Nicolai M. Josuttis, John Wiley & Sons Ltd., ISBN 0-470-84399-3）和本书共享了一部份 templates 入门相关内容。那本书循序渐进地教你C++ 语言和C++ 标准库的所有特性及实际用法。

为什么使用 Templates?

C++ 要求我们使用各种特定类型（specific types）来声明变量、函数和其它各种实体（entities）；然而，很多用以处理「不同类型之数据」的程序代码看起来都差不多。特别是当你实作算法（像是quicksort），或实作如 linked-list 或 binary tree 之类的数据结构时，除了所处理的类型不同，程序代码实际上是一样的。

如果你使用的编程语言并没有针对这个问题支持某种特殊的语言特性，那么你有数种退而次之的选择：

1. 针对每一种类型写一份程序代码。
2. 使用 common base type（通用基础类型，例如 `Object`或 `void*`）来编写代码。
3. 使用特殊的 preprocessors（预处理器。译注：意指编译之前预先处理的宏, macros）。

如果你是从 C、Java 或其它类似语言转到C++ 阵营，可能这三种方法你都用过。但是每一种方 法都有其缺点：

1. 如果为每一种类型写一份程序代码，你就是「不断做重复的事情」。你会在每一份程序代码中犯下相同的错误（如果有的话），而且不敢使用更复杂但更好的算法，因为这会带来更多错误。
2. 如果你使用一个 common base class（通用基础类别），就无法获益于「类型检验」。而且某些 classes 可能必须从其它特殊的 base classes 继承而来，这就给程序维护工作带来更多困难。
3. 如果你使用特殊的 preprocessors（预处理器），例如 C/C++ preprocessors，你将失去「格式化源码」（formatted source code）的好处。预处理机制对于作用域（scope）和类型（types）一无所知，只是进行简单的文字替换。

Templates 可以解决你的问题，而又不带上述提到的缺点。所谓 templates，是为「尚未确定之型别」所写的 functions 或 classes。使用 templates 时，你可以显式（明确）或隐式（隐喻）地将型 别当做自变量（argument）来传递。由于 templates 是一种语言特性，类型检查（type checking）和 作用域（scope）的好处不会丧失。

Templates 如今已被大量运用。就拿 C++标准库来说 几乎所有程序代码都以 templates 写成。 标准库提供各式各样的功能：对 objects 和 values 排序的各种算法、管理各种元素的数据 结构（容器类别，container classes）、支持各种字符集的 string（字符串）。Templates 使我们得以将程序的行为参数化、对程序代码优化（最佳化），并将各种信息参数化。这些都会在后续章节中讲述。现在，让我们从最简单的 templates 开始。

2
Function Templates
函数模板

本章将介绍 function templates。所谓 function templates 是指藉由参数化手段表现一整个族群的 functions（函数）。

2.1 Function Templates 初窥

Function templates 可为不同类型的数据提供作用行为。一个 function template 可以表示一族（一整群）functions，其表现和一般的 function 并无二致，只是其中某些元素在编写时尚未确定。换言之，那些「尚未确定的元素」被「参数化」了。让我们看一个实例。

2.1.1 定义 Template

下面的 function template 传回两个数值中的较大者：

```
// basics/max.hpp
template <typename T>
inline T const& max (T const& a, T const& b)
{
    // 如果 a<b就传回 b，否则传回 a
    return a < b ? b : a;
}
```

这一份 template 定义式代表了一整族 functions，它们的作用都是传回a 和 b 两参数中的较大者。两个参数的类型尚未确定，我们说它是 "template parameter T"。如你所见，template parameters 必须以如此形式加以声明：

```
template < 以逗号分隔的参数列 >
```

上述例子中，参数列就是 typename T。请注意，例中的「小于符号」和「大于符号」在这里被当作角括号（尖括号）使用。关键词 typename引入了一个所谓的 type parameter（类型参数）— 这是目前为止C++程序中最常使用的一种template parameter，另还存在其它种类的template parameter（译注：如 nontype parameter，「非类型参数」），我们将在第 4 章讨论。

此处的类型参数是T，你也可以使用其它任何标识符（identifier）来表示类型参数，但习惯写成 T（译注：代表 Type）。Type parameters 可表示任意类型，在 function template 被调用时，经由传递具体类型而使 T得以被具体指定你可以使用任何类型（包括基本类型和 class 类型等等），只要它支持T所要完成的操作。本例中类型 T必须支持operator< 以比较两值大小。

由于历史因素，你也可以使用关键词 **class**代替关键词 **typename**来定义一个 type parameter。关键字 **typename** 是 C++ 发展晚期才引进的，在此之前只能经由关键字 **class** 引入 type parameter。关键词 **class**目前依然可用。因此 template max()也可以被写成如下等价形式：

```
template <class T>
inline T const& max (T const& a, T const& b)
{
    // 如果 a<b就传回 b，否则传回 a
    return a < b ? b : a;
}
```

就语意而言，前后两者毫无区别，即便使用关键词 **class**，你还是可以把任意类型（包括 non-class 类型）当作实际的 template arguments。但是这么写可能带来一些误导（让人误以为 T必须是 class 类型），所以最好还是使用关键词 **typename**。请注意，这和 class 的类型声明并不是同一回事： 声明 type parameters 时我们不能把关键词 **typename**换成关键词 **struct**。

2.1.2 使用 Template

以下程序示范如何使用 max() function template：

```
// basics/max.cpp

#include <iostream>
#include <string>
#include "max.hpp"

int main()
{
    int i = 42;
    std::cout << "max(7,i):   " << ::max(7,i) << std::endl;

    double f1 = 3.4;
    double f2 = -6.7;
    std::cout << "max(f1,f2): " << ::max(f1,f2) << std::endl;
    std::string s1 = "mathematics";
    std::string s2 = "math";
    std::cout << "max(s1,s2): " << ::max(s1,s2) << std::endl;
}
```

程序调用了 max()三次。第一次所给自变量是两个 int，第二次所给自变量是两个 double，最后一 次给的是两个 std::string。每一次 max()均比较两值取其大者。程序运行结果为：

```
max(7,i):   42
max(f1,f2): 3.4
max(s1,s2): mathematics
```

注意程序对max()的三次调用都加了前缀字 "::"，以便确保被调用的是我们在全局命名空间（global namespace）中定义的 max()。标准库内也有一个 std::max() template，可能会在某些情况下被调用，或在调用时引发模棱两可（ambiguity，歧义性）。

一般而言，templates 不会被编译为「能够处理任意类型」的单一实体（entity），而是被编译为多个个别实体，每一个处理某一特定类型。因此，针对三个类型，max()被编译成三个实体。 例如第一次调用 max()：

```
int i = 42;
... max(7,i) ...
```

使用的是「以int为 template parameter T」的 function template，语意上等同于调用以下函数：

```
inline int const& max (int const& a, int const& b)
{
    // 如果 a<b就传回 b，否则传回 a
    return a < b ? b : a;
}
```

以具体类型替换 template parameters 的过程称为「实例化」（*instantiation*，或称「实体化」）。过程中会产生 template 的一份实体（*instance*）。不巧的是，*instantiation*（实例化、实例化产品）和*instance*（实体）这两个术语在OO（面向对象）编程领域中有其它含义，通常用来表示一个 class 的具体对象（concrete object）。本书专门讨论 templates，因此当我们运用这个术语时，除非另有明确指示，表达的是 templates 方面的含义。

注意，只要 function template 被使用，就会自动引发实例化过程。程序员没有必要个别申请实例化过程。

类似情况，另两次对 max()的调用被实例化为：

```
const double& max (double const&, double const&);
const std::string& max (std::string const&, std::string const&);
```

如果试图以某个类型来实例化 function template，而该类型并未支持 function template 中用到的操作，就会导致编译错误。例如：

```
std::complex<float> c1, c2;     // 此类型并不提供 operator<
...
max(c1,c2);                     // 编译期出错
```

实际上，templates 会被编译两次：

1. 不实例化，只是对 template 程序代码进行语法检查以发现诸如「缺少分号」等等的语法错误。
2. 实例化时，编译器检查 template 程序代码中的所有调用是否合法，诸如「未获支持之函数调用」

便会在这个阶段被检查出来。

这会导致一个严重问题：当 function template 被运用而引发实例化过程时，某些时候编译器需要用到template 的原始定义。一般情况下，对普通的（non-template）functions而言，编译和链接两步骤是各自独立的，编译器只检查各个functions的声明语句是否和调用语句相符，然而template 的编译破坏了这个规则。解决办法在第6章讨论。眼下我们可以用最简单的解法：把template 程序代码以 inline 形式写在头文件（header）中。

2.2 自变量推导（Argument Deduction）

当我们使用某一类型的自变量调用 max()时，template parameters 将以该自变量类型确定下来。如果 我们针对参数类型 T const& 传递两个 ints，编译器必然能够推导出 T 是int。注意这里并不允许「自动类型转换」。是的，每个T都必须完全匹配其自变量。例如：

```
template <typename T>
inline T const& max(T const& a, T const& b);
...
max(4, 7);       // OK，两个 T都被推导为 int
max(4, 4.2);     // 错误：第一个 T被推导为 int，第二个 T被推导为 double
```

有三种方法可以解决上述问题：

1. 把两个自变量转型为相同类型：

```
max(static_cast<double>(4), 4.2);       // OK
```

2. 明确指定 T的类型：

```
max<double>(4, 4.2);                 // OK
```

3. 对各个 template parameters 使用不同的类型（译注：意思是不要像上面那样都叫做 T）。

下一节详细讨论这些问题。

2.3 Template Parameters（模板参数）

Function templates 有两种参数：

1. Template parameters（模板参数），在function template 名称前的一对角（尖）括号中声明：

```
template <typename T>      // T是个 template parameter
```

2. Call parameters（调用参数），在 function template 名称后的小（圆）括号中声明：

```
... max (T const& a, T const& b);    // a和 b是调用参数
```

template parameters 的数量可以任意，但你不能在 function templates 中为它们指定预设自变量值（这一点与 class templates 不同）。例如你可以在 max()template 中定义两个不同类型的调用参数：

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
```

```
    return a < b ? b : a;
}
...
max(4, 4.2);      // OK。返回类型和第一自变量类型相同
```

这似乎是一个可以为 max()template 的参数指定不同类型的好办法，但它也有不足。问题在于你 必须声明返回值的类型。如果你使用了其中一个类型，另一个类型可能被转型为该类型。C++ 没有提供一个机制用以选择「效力更大的类型, *the more powerful type*」（然而你可以藉由某些巧妙的template 编程手段来提供这种机制，参见 15.2.4 节）。因此，对于42和66.66两个调用自变量，max()的返回值要么是double 66.66，要么是int 66。另一个缺点是，把第二参数转型为第一参数的类型，会产生一个局部临时对象（local temporary object），因而无法以*by reference* 方式传回结果。因此在本例之中，返回类型必须是 T1，不能是 T1 const&。

由于call parameters 的类型由template parameters 建立，所以两者往往互相关联。我们把这种概念称为 function template argument deduction（函数模板自变量推导）。它使你可以像调用一个常规（意即 non-template）函数一样来调用 function template。

然而正如先前提到的那样，你也可以「明确指定类型」来实例化一个 template：

```
template <typename T>
inline T const& max (T const& a, T const& b);
...
max<double>(4,4.2); // 以 double类型实例化 T
```

当template parameters和call parameters之间没有明显联系，而且编译器无法推导出template arameters 时，你必须明确地在调用时指定template arguments。例如你可以为max()引入第三个template argument type 作为返回类型：

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
```

然而「自变量推导机制」并不对返回类型进行匹配，而且上述的RT 也并非函数调用参数（call parameters)中的一个; 因此编译器无法推导出 RT。你不得不像这样明确指出 template arguments：

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
...
max<int,double,double>(4, 4.2);
    // OK，但是相当冗长（译注：因为其实只需明写第三自变量类型，却连前两个自变量类型都得写出来）
```

以上我们所看到的是，要么所有function template arguments 都可被推导出来，要么一个也推导不出来。另有一种作法是只明确写出第一自变量，剩下的留给编译器去推导，你要做的只是把所 有「无法被自动推导出来的自变量类型」写出来。因此，如果把上述例子中的参数顺序改变一下， 调用时就可以只写明返回类型：

```
template <typename RT, typename T1, typename T2>
inline RT max (T1 const& a, T2 const& b);
...
max<double>(4,4.2);                    // OK，返回类型为 double
```

此例之中，我们调用 max()时，只明确指出返回类型RT为double，至于T1和T2两个参数类型会被编译器根据调用时的自变量推导为int和 double。

注意，这些max()修改版本并没带来什么明显好处。在「单一参数」版本中，如果两个自变量的 类型不同，你可以指定参数类型和回返值类型。总之，为尽量保持程序代码简单，使用「单一参 数」的 max()是不错的主意。讨论其它 template 相关问题时，我们也会遵守这个原则。

自变量推导过程的细节将在第 11 章讨论。

2.4 重载（Overloading）Function Templates

就像常规（意即non-template）functions一样，function templates 也可以被重载（译注：C++标准库中的许多STL算法都是如此）。这就是说，你可以写出多个不同的函数定义，并使用相同的函数名称；当客户调用其中某个函数时，C++编译器必须判断应该调用哪一个函数。即使不牵扯templates，这个推断过程也非常复杂。本节讨论的是，一旦涉及templates，重载将是一个怎样的过程。如果你对non-templates 情况下的重载机制还不太清楚，可以先参考附录B，那里我们对重载机制做了相当深入的讲解。

下面这个小程序展示如何重载一个 function template：

```
// basics/max2.cpp
// 传回两个 ints中的较大者
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}

// 传回两任意类型的数值中的较大者
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

```
// 传回三个任意类型值中的最大者
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);
}

int main()
{
    ::max(7, 42, 68);           // 调用「接受三个自变量」的函数
    ::max(7.0, 42.0);           // 调用 max<double>（经由自变量推导）
    ::max('a', 'b');            // 调用 max<char>（经由自变量推导）
    ::max(7, 42);               // 调用「接受两个 int自变量」的 non-template 函数
    ::max<>(7, 42);             // 调用 max<int>（经由自变量推导）
    ::max<double>(7, 42);       // 调用 max<double>（无需自变量推导）
    ::max('a', 42.7);           // 调用「接受两个 int自变量」的 non-template 函数
}
/* 译注：ICL7.1/g++ 3.2顺利通过本例。VC6无法把最后一个调用匹配到常规的（non-template）函数
max(),造成编译失败VC7.1可顺利编译但对倒数第二个调用给出警告虽然它调用的是 function template
max()，但它发现常规函数 max()与这个调用更匹配。*/
```

这个例子说明：non-template function 可以和同名的 function template 共存，也可以和其相同类型的具现体共存。当其它要素都相等时，重载解析机制会优先选择 non-template function，而不选择由 function template 实例化后的函数实体。上述第四个调用便是遵守这条规则：

```
::max(7, 42);       // 两个自变量都是 int，吻合对应的 non-template function
```

但是如果可由 template 产生更佳匹配，则 template 具现体会被编译器选中。前述的第二和第三 个调用说明了这一点：

```
::max(7.0, 42.0);           // 调用 max<double>（经由自变量推导）
::max('a', 'b');            // 调用 max<char>（经由自变量推导）
```

调用端也可以使用空的 template argument list，这种形式告诉编译器「只从 template 具现体中挑选适当的调用对象」，所有template parameters 都自call parameters 推导而得：

```
::max<>(7, 42);             // 调用 max<int>（经由自变量推导）
```

另外，「自动类型转换」只适用于常规函数，在templates 中不予考虑，因此前述最后一个调用调用的是 non-template 函数。在该处，'a' 和 42.7都被转型为 int：

```
::max('a', 42.7);       // 本例中只有 non-template 函数才可以接受两个不同类型的自变量
```

下面是一个更有用的例子，为指针类型和 C-style 字符串类型重载了 max()template：

```
// basics/max3.cpp
#include <iostream>
#include <cstring>
#include <string>

// 传回两个任意类型值的较大者
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// 传回两个指针的较大者（所谓较大是指「指针所指之物」较大）
template <typename T>
inline T* const& max (T* const& a, T* const& b)
{
    return *a < *b ? b : a;
}

// 传回两个 C-style 字符串的较大者（译注：C-style 字符串必须自行定义何谓「较大」）。
inline char const* const& max (char const* const& a, char const* const&
                               b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}

int main()
{
    int a=7;
    int b=42;
    ::max(a,b);     // 调用「接受两个 int」的 max()
    std::string s = "hey";
    std::string t = "you";
    ::max(s,t);     // 调用「接受两个 std::string」的 max()

    int *p1 = &b;
    int *p2 = &a;
    ::max(p1,p2);   // 调用「接受两个指针」的 max()

    char const* s1 = "David";
```

```
    char const* s2 = "Nico";
    ::max(s1, s2); // 调用「接受两个 C-style 字符串」的 max()
}
```

注意，所有重载函数都使用*by reference*方式来传递自变量。一般说来，不同的重载形式之间最好只存在「绝对必要的差异」。各重载形式之间应该只存在「参数个数的不同」或「参数类型 的明确不同」，否则可能引发各种副作用。举个例子，如果你以一个「*by value* 形式的 max()」重载一个「*by reference* 形式的 max()」（译注：两者之间的差异不够明显），就无法使用「三自变量」版本的 max()来取得「三个 C-style 字符串中的最大者」：

```
// basics/max3a.cpp
#include <iostream>
#include <cstring>
#include <string>

// 传回两个任意类型值的较大者（call-by-reference）
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
// 传回两个 C-style 字符串的较大者（call-by-value）
inline char const* max (char const* a, char const* b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}

// 传回三个任意类型值的最大者（call-by-reference）
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);  // 当 max(a,b)采用 by value 形式时，此行错误
}

int main()
{
    ::max(7, 42, 68);      // OK

    const char* s1 = "frederic";
    const char* s2 = "anica";
    const char* s3 = "lucas";
    ::max(s1, s2, s3);     // ERROR
```

```
}
```

本例中针对三个 C-style 字符串调用 max()，会出现问题。以下这行述句是错误的：

```
return ::max (::max(a,b), c);
```

因为C-style字符串的max(a,b)重载函数创建了一个新而暂时的区域值（a new, temporary local value），而该值却以 *by reference* 方式被传回（那当然会造成错误）。

这只是细微的重载规则所引发的非预期行为例子之一。当函数调用动作发生时，如果不是所有重载形式都在当前范围内可见，那么上述错误可能发生，也可能不发生。事实上，如果把「三自变量」版本的 max()写在接受两个ints的max()前面（于是后者对前者而言不可见），那么在调用「三自变量」max()时，会间接调用「双自变量」max() function template：

```
// basics/max4.cpp
// 传回两个任意类型值的较大者
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// 传回三个任意类型值的最大者
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return ::max (::max(a,b), c);
      // 即使自变量类型都是 int，这里也会调用 max()template。因为下面的函数定义来得太迟。
}

// 传回两个 ints的较大者
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}
```

9.2 节,详细讨论这个问题。就目前而言，你应该遵循一条准则：总是把所有形式的重载函数写在它们被调用之前。

2.5 摘要

Function templates 可以针对不同的 template arguments 定义一整族（a family of）函数。

Function templates 将依照传递而来的自变量（arguments）的类型而被实例化（instantiated）。

你可以明确指出 template parameters。

Function templates 可以被重载（overloaded）。

重载 function templates 时，不同的重载形式之间最好只存在「绝对必要的差异」。

请确保所有形式的重载函数都被写在它们的被调用点之前。

3
Class Templates
类别模板

就像上一章所说的 functions 那样，classes 也可以针对一或多个类型被参数化。用来管理「各种 不同类型的元素」的 container classes（容器类别）就是典型例子。运用 class templates 你可以实 作出可包容各种类型的 container class。本章将以一个 stack class 作为 class templates 实例。

3.1 实作 Class Template Stack

和 function templates 一样，我们在单一头文件（header）中声明和定义 class Stack<> 如下（6.3 节将讨论声明和定义分离的模型）：

```
// basics/stack1.hpp
#include <vector>
#include <stdexcept>

template <typename T>
class Stack {
private:
    std::vector<T> elems;       // 元素
public:
    void push(T const&);        // push 元素
    void pop();                 // pop 元素
    T top() const;              // 传回最顶端元素
    bool empty() const {        // stack是否为空
        return elems.empty();
    }
};


template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem); // 追加（附于尾）
}
```

```
template <typename T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop: empty stack");
    }
    elems.pop_back();      // 移除最后一个元素
}

template <typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top: empty stack");
    }
    return elems.back();    // 传回最后一个元素的拷贝
}
```

正如你所见，这个 class template 是以标准库的 class template `vector<>`为基础实作出来的，这样我们就可以不考虑内存管理、*copy*构造函数、*assignment* 运算符等等，从而得以把注意力放在这个 class template 的界面上。

3.1.1 Class Templates 的声明

声明class templates 的动作和声明function templates 类似：在声明语句之前加一条述句，以任意标识符声明类型参数（type parameters）。我们还是以`T`作为标识符：

```
template <typename T>
class Stack {
    ...
};
```

相同的规则再次适用：关键词 **`class`**可以代替关键词 **`typename`**：

```
template <class T>
class Stack {
    ...
};
```

在 class template 内部，`T`就像其它任意类型一样，可用来声明成员变量（member variables）和成员函数（member functions）。本例之中`T`用来声明`vector<>` 所容纳的元素类型、声明一个「接受 `T const&`」的 `push()`函数、以及声明一个「返回类型为 `T`」的 `top()`函数：

```
template <typename T>
class Stack {
private:
    std::vector<T> elems;       // 元素
public:
    Stack();                    // 构造函数
    void push(T const&);        // push元素
    void pop();                 // pop元素
    T top() const;              // 传回最顶端的元素
};
```

这个 class 的类型为 `Stack<T>`，`T`是一个 template parameter。现在，无论何时你以这个class声明变量或函数时，都应该写成 `Stack<T>`。例如，假设你要声明自己的 *copy* 构造函数和 *assignment* 运算符，可以写为：

```
template <typename T>
class Stack {
    ...
    Stack (Stack<T> const&);                    // copy构造函数
    Stack<T>& operator= (Stack<T> const&);      // assignment运算符
    ...
};
```

然而如果只是需要class名称而不是class类型时，只需写`Stack`即可。构造函数和析构函数的声明就属于这种情况。

3.1.2 成员函数（Member Functions）的实作

为了定义 class template 的成员函数，你必须指出它是个 function template，而且你必须使用 class template 的全称。因此 `Stack<T>` 的成员函数 `push()`看起来便像这样：

```
template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem); // 将传入的元素 elem 附加于尾
}
```

这里调用了`vector`的成员函数`push_back()`，把元素`elem`追加到`elems`尾端。

注意，对一个vector进行 `pop_back()`，只是把最后一个元素移除，并不传回该元素。这种行为乃是基于异常安全性（exception safety）考虑。实现一个「移除最后元素并传回，而且完全顾及异常安全性」的 `pop()` 是不可能的（这个问题最早由 Tom Cargill 在[CargillExceptionSafety]中讨论过），[SutterExceptional] 条款 10 也有讨论）。然而，如果抛开可能的危险，我们可以实作出一个「移除最后元素并传回」的`pop()`，但必须声明一个类型为 `T`的区域变量：

```
template <typename
```

```
T>
T Stack<T>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop: empty stack");
    }
    T elem = elems.back();       // 保存最后元素的拷贝
    elems.pop_back();            // 移除最后一个元素
    return elem;                 // 传回先前保存的最后元素
}
```

当vector为空时，对它进行back()或pop_back()操作会导致未定义行为。所以我们必须在操作前先检查stack是否为空。如果stack为空，就抛出一个std::out_of_range异常。top()之中也需进行相同检查；该函数传回 stack 的最后一个元素，但不移除之：

```
template <typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top: empty stack");
    }
    return elems.back();    // 传回最后元素的拷贝
}
```

当然，你也可以把任何 class templates 的成员函数实作码写在 class 声明语句中，形成一个 inline 函
式。例如：

```
template <typename T>
class Stack {
    ...
    void push(T const& elem) {
        elems.push_back(elem); // 将传入的元素 elem 附加于尾
    }
    ...
};
```

3.2 使用 Class Template Stack

为了使用 class template object，你必须明确指出其 template arguments。下面例子说明如何使用
Stack<> class template：

```
// basics/stack1test.cpp
#include <iostream>
```

```
#include <string>
#include <cstdlib>
#include "stack1.hpp"

int main()
{
    try {
        Stack<int>         intStack;        // stack of ints
        Stack<std::string> stringStack;     // stack of strings

        // 操控 int stack
        intStack.push(7
        );
        std::cout << intStack.top() << std::endl;

        // 操控 string stack
        stringStack.push("hello");
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;    // 传回错误状态码
    }
}
```

上例声明了 `Stack<int>`这样一个类型，表示在该 class template 中 `T`被替换为 `int`。于是 `intStack` 成为这样一个object：内部使用「可容纳`int`数据做为元素」的vector，并将被调用之任何成员函数都「以`int` 类型进行实例化」。同样道理，`Stack<std::string>` 表示：`stringStack` 被创建为这样一个object：内部使用「可容纳`string`数据做为元素」的vector，并将被调用之任何成员函数都「以`std::string`类型进行实例化」。

注意，惟有被调用到的成员函数，才会被实例化（instantiated）。对class templates 而言，只有当某个成员函数被使用时，才会进行实例化。无疑地这么做可以节省时间和空间。另一个好处是，你甚至可以实例化一个class template，而具现类型并不需要完整支持「class template 内与该 类型有关的所有操作」—前提是那些操作并未真正被叫用。举个例子，考虑某个class，其某些成员函数使用`operator<` 对内部元素排序；只要避免调用这些函数，就可以以一个「并不支持 `operator<`」的类型来实例化这个 class template。

本例中的 *default* 构造函数、`push()`函数和 `top()`函数都同时被 `int` 和 `string` 类型加以实现（实例化）。然而 `pop()`只被 `string`实例化（译注：因为 `pop()`只被 `stringStack`调用）。如果 class template 拥有 static 成员，这些 static 成员会针对每一种被使用的类型完成实例化。

你可以像面对任何基本类型一样地使用一个实例化后的class template类型，前提是你的使用方式合法：

```
void foo (Stack<int> const& s) // 参数 s是一个 int stack
{
    Stack<int> istack[10];      // istack是一个「含有 10个 int stacks」的 array
    ...
}
```

运用 typedef，你可以更方便地使用 class templates：

```
typedef Stack<int> IntStack;

void foo (IntStack const& s) // 参数 s是一个 int stack
{
    IntStack istack[10];    // istack是一个「含有 10个 int stacks」的 array
    ...
}
```

注意，在C++中，typedef并不产生新类型，只是为既有类型产生一个别名（type alias）。因此在以下述句之后：

```
typedef Stack<int> IntStack;
```

IntStack和 Stack<int>拥有（代表）相同类型，可以互换使用，可以相互赋值（*assigned*）。

Template arguments 可以是任意类型，例如可以是「指向 float」的指针，或甚至是个 int stack：

```
Stack<float*>      floatPtrStack;    // stack of float pointers
Stack<Stack<int> > intStackStack;    // stack of stack of ints
```

惟一的条件是：该类型必须支持所有「实际被调用到的操作」。

注意，你必须在相邻两个右角括号之间插入一些空白符号（像上面那样），否则就等于使用了operator>>，那会导致语法错误：

```
Stack<Stack<int>> intStackStack;    // 错误：此处不允许使用 >>
```

3.3 Class Templates 的特化（Specializations）

你可以针对某些特殊的 template arguments，对一个 class template 进行「特化」。class templates 的特化与 function template 的重载类似，使你得以针对某些特定类型进行程序代码优化， 或修正某个特定类型在 class template 实例（instantiation）中的错误行为。然而如果你对一个 class template 进行特化，就必须特化其所有成员函数。虽然你可以特化某个单独的成员函数，但一旦这么做，也就不再是特化整个 class template。

欲特化某个class template，必须以template<> 开头声明此一class，后面跟着你希望的特化结果。特化类型（specialized type）将作为template arguments 并在class名称之后直接写明：

```
template<>
class Stack<std::string> {
    ...
};
```

对特化体（specializations）而言，每个成员函数都必须像常规的（一般的）成员函数那样定义，每一个 T出现处都必须更换为特化类型（specialized type）：

```
void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem);  // 将传入的 elem附加于尾
}
```

下面是一个针对 std::string类型而特化的 Stack<> 的完整范例：

```
// basics/stack2.hpp
#include <deque>
#include <string>
#include <stdexcept>
#include "stack1.hpp"

template<>
class Stack<std::string> {
private:
    std::deque<std::string> elems; // 元素
public:
    void push(std::string const&);   // push元素
    void pop();                      // pop元素
    std::string top() const;         // 传回 stack最顶端元素
    bool empty() const {             // stack是否为空
        return elems.empty();
    }
};

void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem);         // 追加元素
}

void Stack<std::string>::pop ()
{
    if (elems.empty()) {
```

```
        throw std::out_of_range("Stack<std::string>::pop(): empty stack");
    }
    elems.pop_back();              // 移除最后一个元素
}

std::string Stack<std::string>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<std::string>::top(): empty stack");
    }
    return elems.back();           // 传回最后一个元素的拷贝
}
```

此例之中，我们在stack内部改用deque代替vector来管理元素。这么做并没有特别的好处，但它示范「一个特化实作码可以和其primary template（主模板。译注：最原始的那一份定义）有 相当程度的差异」。

3.4 偏特化（Partial Specialization）

Class templates 可以被偏特化（partial specialized，或称部份特化、局部特化）。这使你得以在特定情形下使用特殊实作码，但仍然留给你（使用者）选择 template parameters 的能力。例如对于下面的 class template：

```
template <typename T1, typename T2>
class MyClass {
    ...
};
```

以下数种形式的偏特化都是合理的：

```
// 偏特化：两个 template parameter相同
template <typename T>
class MyClass<T,T> {
    ...
};

// 偏特化：第二个类型为
int template <typename
T> class
MyClass<T,int> {
    ...
};

// 偏特化：两个 template parameter均为指针类型
```

```
template <typename T1, typename T2>
class MyClass<T1*, T2*> {
    ...
};
```

以下例子示范，下列各种声明语句将使用上述哪一个 class template：

```
MyClass<int,float> mif;         // 使用 MyClass<T1,T2>
MyClass<float,float> mff;       // 使用 MyClass<T,T>
MyClass<float,int> mfi;         // 使用 MyClass<T,int>
MyClass<int*,float*> mp;        // 使用
MyClass<T1*,T2*>
```

如果某个声明语句与两个（或更多）偏特化版本产生同等的匹配程度，这个声明语句便被视为模棱两可（歧义）：

```
MyClass<int,int> m;         // 错误：同时匹配 MyClass<T,T> 和 MyClass<T,int>
MyClass<int*,int*> m;       // 错误：同时匹配 MyClass<T,T> 和 MyClass<T1*,T2*>
```

为解除上述第二声明的歧义性，你可以针对「指向相同类型」的指针，提供另一个偏特化版本：

```
template <typename T>
class MyClass<T*,T*> {
    ...
};
```

3.5 预设模板自变量（Default Template Arguments）

你可以针对 class templates 定义其 template parameters 的默认值这称为 default template arguments（预设模板自变量）。预设自变量值甚至可以引用前一步声明的 template parameters。例如在 class Stack<> 中，你可以把「用来容纳元素」的容器类型定义为第二个template parameter，并使用std::vector<> 作为其默认值：

```
// basics/stack3.hpp
#include <vector>
#include <stdexcept>

template <typename T, typename CONT = std::vector<T> >
class Stack {
private:
    CONT elems;             // 元素
public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 传回 stack 的顶端元素
```

```
    bool empty() const {    // stack 是否为空
        return elems.empty();
    }
};

template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem); // 追加元素
}

template <typename T, typename CONT>
void Stack<T,CONT>::pop()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();      // 移除最后一个元素
}

template <typename T, typename CONT>
T Stack<T,CONT>::top() const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back();    // 传回最后一个元素的拷贝
}
```

注意上述的 template 如今有两个参数，所以每一个成员函数的定义式中都必须包含这两个参数：

```
template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem); // 追加元素
}
```

你可以像面对「单一template parameter」版本那样地使用这个新版stack。这时如果你只传递一个自变量表示元素类型，stack会使用预设的vector来管理其内部元素：

```
template <typename T, typename CONT = std::vector<T> >
class Stack {
private:
    CONT elems; // 元素
```

```
...
};
```

当你在程序中声明一个Stack object时，也可以明确指定元素容器的类型：

```
// basics/stack3test.cpp
#include <iostream>
#include <deque>
#include <cstdlib>
#include "stack3.hpp"

int main()
{
    try {
        // stack of ints
        Stack<int> intStack;

        // stack of doubles，其内部使用 std::deque<> 来管理元素
        Stack<double,std::deque<double> > dblStack;
            //译注：千万不要声明为 Stack<double, std::deque<int> >，
            //      这是自己砸自己的脚，编译器无法为你做些什么。

        // 操控 int stack
        intStack.push(7
        );
        std::cout << intStack.top() << std::endl;
        intStack.pop();

        // 操控 double stack
        dblStack.push(42.42);
        std::cout << dblStack.top() << std::endl;
        dblStack.pop();
        dblStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;    // 以错误状态码离开程序
    }
}
```

那么，只要像下面这样做：

```
Stack<double,std::deque<double> >
```

你就声明了一个 double stack，其内部以 std::deque<> 来管理元素。

3.6 摘要

所谓 class template 是「包含一个或多个尚未确定之类型」的 class。

你必须将具体类型当作 template arguments 传入，才能使用 class template。于是该 class template 便以你所指定的那些类型，由编译器加以实例化并编译。

Class templates 之中，只有实际被调用的成员函数，才会被实例化。

你可以针对某些特定类型，对 class templates 进行特化（specialize）。

你可以针对某些特定类型，对 class templates 进行偏特化（partially specialize）。

你可以为 template parameters 定义默认值（称为 default template arguments），该预设自变量值 可以引用前一步定义的 template parameters。

4

Nontype Template Parameters

非类型模板参数

对 function templates 和 class templates 而言，template parameters 并不一定非要是类型（types）不可，它们也可以是常规的（一般的）数值。当你以类型（types）作为 template parameters 时，程序代码中尚未决定的是类型；当你以一般数值（non-types）作为 template parameter 时，程序代码中待定的内容便是某些数值。使用这种template时必须明确指定数值，程序代码才得以实例化。 本章将利用这个特性实作一个新版本的stack class template。此外我将举一个例子，展示如何在 function templates 中使用 nontype template paramaters，并讨论此技术的一些局限。

4.1 Nontype Class Template Parameters（非类型类别模板参数）

上一章实作了一个「元素个数可变」的stack class。与之对比，你也可以实作另一种stack，透过一个固定大小（fixed-size）的 array来容纳元素。这样做的好处是不必考虑诸如内存管理之类的问题。然而array大小的决定是一件比较困难的事：array愈小则stack愈容易满溢，array愈大则愈容易造成空间浪费一个可行的解决办法是让使用者指定array大小这个大小也就是stack 的最大元素个数。

为了完成以上想法，我们应该把大小值当作一个 template parameter：

```
// basics/stack4.hpp
#include <stdexcept>

template <typename T, int MAXSIZE>
class Stack {
private:
    T elems[MAXSIZE];       // 元素
    int numElems;           // 当前的元素个数
public:
    Stack();                // 构造函数
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;              // 传回 stack 顶端元素
    bool empty() const {        // stack 是否为空
        return numElems == 0;
    }
```

```
    bool full() const {          // stack 是否已满
        return numElems == MAXSIZE;
    }
};

// 构造函数
template <typename T, int MAXSIZE>
Stack<T,MAXSIZE>::Stack ()
    : numElems(0)          // 一开始并无任何元素
{
    // 不做任何事
}

template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push (T const& elem)
{
    if (numElems == MAXSIZE) {
        throw std::out_of_range("Stack<>::push(): stack is full.");
    }
    elems[numElems] = elem;     // 追加
    ++numElems;                 // 元素总数加 1
}

template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::pop ()
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::pop(): empty stack.");
    }
    --numElems;                 // 元素总数减 1
}

template <typename T, int MAXSIZE>
T Stack<T,MAXSIZE>::top () const
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::top(): empty stack.");
    }
    return elems[numElems - 1];      // 传回最后一个元素
}
```

新加入的第二个 template parameter MAXSIZE隶属 int类型，用来指定「容纳 stack 元素」的那

个底部 array 的大小：

```
template <typename T, int MAXSIZE>
class Stack {
private:
    T elems[MAXSIZE];      // 元素
    ...
};
```

push()便是使用 MAXSIZE来检查 stack 是否已满：

```
template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push (T const& elem)
{
    if (numElems == MAXSIZE) {
        throw std::out_of_range("Stack<>::push(): stack is full.");
    }
    elems[numElems] = elem;     // 追加
    ++numElems;                 // 元素总数加 1
}
```

使用上述 class template 时，必须同时指定 (1) 元素类型和 (2) stack 元素的最大数量：

```
// basics/stack4test.cpp
#include <iostream>
#include <string>
#include <cstdlib>
#include "stack4.hpp"

int main()
{
    try {
        Stack<int,20>         int20Stack;      // 最多容纳 20个 int元素
        Stack<int,40>         int40Stack;      // 最多容纳 40个 int元素
        Stack<std::string,40> stringStack;     // 最多容纳 40个 string元素

        // 操控「最多容纳 20个 int元素」的那个 stack
        int20Stack.push(7);
        std::cout << int20Stack.top() << std::endl;
        int20Stack.pop();

        // 操控「最多容纳 40个 string元素」的那个 stack
        stringStack.push("hello");
```

```
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;    // 传回一个错误状态码
    }
}
```

注意，每一个被实例化（instantiated）的 class template 都有各自的类型。（译注：常见的误会是：上述三个 stacks 隶属同一类型。这是错误观念。）因此 int20Stack和 int40Stack是两个不同类型，不能互相进行隐式或显式转换，两者不能换用（彼此取代），也不能互相赋值。

你可以指定 non-type template parameters 的默认值：

```
template <typename T = int, int MAXSIZE = 100>
class Stack {
    ...
};
```

然而从设计角度来看，这样做并不恰当。Template parameters 的默认值应该符合大多数情况下的 要求，然而把 int当做预设元素类型，或指定 stack 最多有 100 个元素，并不符合一个「通用型 stack」的需求。更好的作法是让使用者指定这两个参数的值，并在文件中说明它们的意义。

4.2 Nontype Function Template Parameters（非类型函数模板参数）

你也可以为 function template 定义 nontype parameters。例如下面的 function template 定义了一组 函数，可以将参数 x累加一个值（VAL）后传回：

```
// basics/addval.hpp
template <typename T, int VAL>
T addValue(T const& x)
{
    return x + VAL;
}
```

当我们需要把「函数」或「某种通用操作」作为参数传递时，这一类函数就很有用。例如使用 STL（Standard Template Library，标准模板库）时，你可以运用上述 function template 的实例（instantiation），将某值加到元素集内的每一个元素身上：

```
// (1)
std::transform (source.begin(), source.end(),      // 来源端起止位置
                dest.begin(),                      // 目的端起始位置
                addValue<int,5>);                  // 实际操作
```

最后一个自变量将 function template addValue()实例化了，使其操作成为「将5加进一个int数 值中」。算法transform()会对source 中的所有元素调用这个具现体（函数），然后把结果传入 dest中。

注意上述例子带来的一个问题：addValue<int,5> 是个 function template 实体（instance），而我们知道，所谓「function templates 实体」被认为是命名了一组重载函数集，即使该函数集内可能只有一个函数。根据目前标准，编译器无法借助「重载函数集」来进行 template parameter 的推导。因此你不得不把 function template argument 强制转型为精确类型：

```
// (2)
std::transform (source.begin(), source.end(),          // 来源端起止位置
                dest.begin(),                           // 目的端起始位置
                (int(*)(int const*)) addValue<int,5>); // 操作
```

C++ *Standard* 中已有一个提案要求修正这种行为，使你不必在这种场合强制转型（请参考[CoreIssue115]）。在尚未获得修正之前，为保证程序的可移植性，你还是得像上面那么做。

4.3 Nontype Template Parameters 的局限

注意，nontype template parameters 有某些局限：通常来说它们只能是常数整数（constant integral values），包括 enum，或是「指向外部链接（external linkage）之对象」的指针。

以浮点数或 class-type objects 作为 nontype template parameters 是不可以的：

```
template <double VAT>       // 错误：浮点值不能作为 template parameters
double process (double v)
{
    return v * VAT;
}

template <std::string name> // 错误：class objects 不能作为 template parameters
class MyClass {
    ...
};
```

不允许浮点字面常数（floating-point literals）或简单的常量浮点表达式（constant floating-point expressions）作为 template arguments，其实只是历史因素，并非技术原因。由于并没有什么实作上的困难，或许将来C++会支持它，请参考 13.4 节。

由于字符串字面常数（string literal）是一种采用内部链接（internal linkage）的对象，也就是说不同模块（modules）内的两个同值的字符串字面常数，其实是不同的对象，因此它们也不能被拿来作为 template arguments：

```
template <char const* name>
class MyClass {
```

```
    ...
};

MyClass<"hello"> x; // 错误：不能使用字符串常量"hello"
```

此外，全局指针也不能被拿来作为 template arguments：

```
template <char const* name>
Class MyClass {
    ...
};

char const* s = "hello";

MyClass<s> x;   // 错误：s是「指向内部链接（internal linkage）对象」的指针
```

但是你可以这么写：

```
template <char const* name>
Class MyClass {
    ...
};

extern char const s[] = "hello";

MyClass<s> x;   // OK
```

全局的 char array s被初始化为 "hello"，因此 s是一个外部链接（external linkage）对象。8.3.3 节, 13.4 节则讨论了这个问题未来的可能变化。

4.4 摘要

Templates parameters 不限只能是类型（types），也可以是数值（values）。

你不能把浮点数、class-type 对象、内部链接（internal linkage）对象（例如字符串字面常数） 当作 nontype template parameters 的自变量。

5

高阶基本技术

Tricky Basics

本章涵盖实际编程之中层次较高的一些 template 基本知识，包括关键词 typename 的另一种用途、将 member function（成员函数）和 nested class（嵌套类别）定义为 templates、奇特的 template template parameters、零值初始化（zero initialization）、以字符串字面常数（string literals）作为 function templates arguments 的细节问题…等等。有时候这些问题可能涉及较多技巧，但每一位程序员都应该至少对它们有大略的了解。

5.1 关键词 typename

关键词typename是C++标准化过程中被引入的，目的在于向编译器说明template 内的某个标识符是个类型（而不是其它什么东西）。考虑下面的例子：

```
template <typename T>
class MyClass {
    typename T::SubType * ptr;
    ...
};
```

在这里，第二个 typename关键词的意思是：SubType是 class T内部定义的一个类型，从而ptr是一个「指向 T::SubType类型」的指针。

如果上例没有使用关键词 typename，SubType会被认为是 class T的一个 static 成员，于是被编 译器理解为一个具体变量或一个对象，导致以下式子：

```
T::SubType * ptr
```

所表达的意义变成：class T的 static 成员 SubType与 ptr相乘。

通常如果某个与 template parameter 相关的名称是个类型（type）时，你就必须加上关键字
typename。更详细的讨论见 9.3.2 节。

typename的一个典型应用是在 template 程序代码中使用「STL 容器供应的迭代器（iterators）」：

```
// basics/printcoll.hpp
```

```
#include <iostream>
// 打印某个 STL容器内的所有元素
template <typename T>
void printcoll (T const& coll)
{
    typename T::const_iterator pos;                 // 一个迭代器，用于巡访 coll
    typename T::const_iterator end(coll.end());     // 末尾位置


    for (pos=coll.begin(); pos!=end; ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

在这个 function template 中，coll是个 STL 容器，其元素类型为 T。这里使用了 STL 容器的迭代器类型（iterator type）遍历 coll的所有元素。迭代器类型为const_iterator，每一个STL 容器都声明有这个类型：

```
class stlcontainer {
    ...
    typedef ... iterator;       // 可读可写的迭代器
    typedef ... const_iterator; // 惟读迭代器
    ...
};
```

使用 template type T的 const_iterator时你必须写出全名并在最前面加上关键词 typename：

```
typename T::const_iterator pos;
```

.template 构件（construct）

引入关键词 typename 之后，人们又发现了一个类似问题。考虑以下程序代码，其中使用标准的
bitset类型：

```
template<int N>
void printBitset (std::bitset<N> const& bs)
{
    std::cout << bs.template to_string<char,char_traits<char>,allocator<char> >();
}
```

此例中的 .template 看起来有些古怪，但是如果没有它，编译器无法得知紧跟其后的 "<" 代表的是个 template argument list 的起始，而非一个「小于」符号。注意，只有当位于 "." 之前的 构件（construct）取决于某个 template parameter 时，这个问题才会发生。以上例子中，参数 bs 便是取决（受控）于 template parameter N。

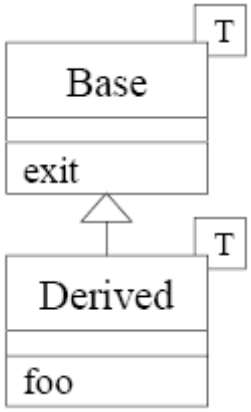
结论是 ".template" 或 "->template" 记号只在 templates 之内才能被使用，而且它们必须紧 跟着「与 template parameters 相关」的某物体。细节请见 9.3.3 节, p.132。

5.2 使用 this->

如果 class templates 拥有 base classes，那么其内出现的成员名称 x并非总是等价于 this->x，即使 x系继承而来。例如：template <typename T>

```
class Base {
public:
    void exit();
};

template <typename T>
class Derived : public Base<T> {
public:
    void foo() {
        exit();
    }
};
```



本例在 foo()内解析（*resolving*）"exit" 符号时，定义于 Base的exit()会被编译器忽略。因此，你要么获得一个编译错误，要么就是调用一个外部的 exit()。

我们将在 9.4.2 节详细讨论这个问题。目前可视为一个准则：使用与 template 相关的符号时，建议总是以this-> 或Base<T>:: 进行修饰。为避免任何不确定性，可考虑在templates 内对所有成员存取动作（member accesses）进行以上修饰。

5.3 Member Templates（成员模板）

Class的成员也可以是templates：既可以是nested class templates，也可以是member function templates。

让我再次使用 Stack<> class templates 来展示这项技术的优点，并示范如何运用这种技术。通常只有当两个 stacks 类型相同，也就是当两个stacks 拥有相同类型的元素时，你才能对它们相互赋值（*assign*），也就是将某个 stack 整体赋值给另一个。你不能把某种类型的 stack赋值给另一种类型的stack，即使这两种类型之间可以隐式转型：

```
Stack<int>     intStack1, intStack2;      // stacks for ints
Stack<float>   floatStack;                // stack for floats
...
intStack1 = intStack2;     // OK：两个 stacks 拥有相同类型
floatStack = intStack1;    // ERROR：两个 stacks 类型不同
```

default assignment 运算符要求左右两边拥有相同类型，而以上情况中，拥有不同类型元素的两个 stacks，其类型并不相同。

然而，只要把 *assignment* 运算符定义为一个 template，你就可以让两个「类型不同，但其元素 可隐式转型」的 stacks 互相赋值。为完成此事，Stack<> 需要这样的声明：

```
// basics/stack5decl.cpp
template <typename T>
class Stack {
private:
    std::deque<T> elems;    // 元素
public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 传回 stack 顶端元素
    bool empty() const {    // stack 是否为空
        return elems.empty();
    }

    // 以「元素类型为 T2」的 stack做为赋值运算的右手端。
    template <typename T2>
    Stack<T>& operator= (Stack<T2> const&);
};
```

对比原先的 Stack，这个版本有如下改动：

1. 增加一个 *assignment* 运算符，使 Stack可被赋予一个「拥有不同元素类型 T2」的 stack。
2. 这个 stack 如今使用 deque 作为内部容器。这个改动是上述改动的连带影响。

新增加的 *assignment* 运算符实作如下：

```
// basics/stack5assign.hpp
template <typename T>
 template <typename T2>
```

```
Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
{
    if ((void*)this == (void*)&op2) {     // 判断是否赋值给自己
        return *this;
    }

    Stack<T2> tmp(op2);                   // 建立 op2的一份拷贝
    elems.clear();                        // 移除所有现有元素
    while (!tmp.empty()) {                // 复制所有元素
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
```

我们先看看什么样的语法可以定义一个 member template。在拥有 template parameter T的 template 中定义一个内层的（inner）template parameter T2：

```
template <typename T>
 template <typename T2>
...
```

实作这个运算符（成员函数）时，你可能希望取得「赋值符号右侧之op2 stack」的所有必要资料，但是这个stack的类型和目前（此身）类型不同（是的，如果你以两个不同的类型实例化同一个class template，你会得到两个不同的类型），所以只能通过public接口来得到那些数据。 因此惟一能够取得 op2数据的办法就是调用其 top()函数。你必须经由 top()取得op2的所有数据，而这必须借助 op2 的一份拷贝来实现：每取得一笔数据，就运用 pop()把该数据从op2 的拷贝中移除。由于top()传回的是stack之中最后（最晚）被推入的元素，所以我们需要反方向把元素安插回去。基于这种需求，这里使用了deque，它提供push_front()操作，可以把一 个元素安插到容器最前面。

有了这个 member template，你就可以把一个 int stack 赋值（*assign*）给一个 float stack：

```
Stack<int>     intStack1, intStack2;      //stack for ints
Stack<float>    floatStack;               //stack for floats
...
floatStack = intStack1;    // OK：两个 stacks 类型不同，但 int可转型为 float。
```

当然，这个赋值动作并不会改动 stack和其元素的类型。赋值完成后，floatStack 的元素类型还是 float，而 top()仍然传回 float值。

也许你会认为，这么做会使得类型检查失效，因为你甚至可以把任意类型的元素赋值给另一个 stack。然而事实并非如此。必要的类型检查会在「来源端 stack」的元素（拷贝）被安插到「目 的端 stack」时进行：

```
elems.push_front(tmp.top());
```

如果你将一个 string stack 赋值给一个 float stack，以上述句编译时就会发生错误：「elems.push_front()无法接受 tmp.top()的返回类型」。具体的错误讯息因编译器而异，但含义类似。

```
Stack<std::string> stringStack;      // stack of strings
Stack<float>      floatStack;        // stack of floats
…
floatStack = stringStack;            // 错误：std::string无法转型为 float
```

注意，前述的 template *assignment* 运算符并不取代 *default assignment* 运算符。如果你在相同类型的 stack 之间赋值，编译器还是会采用 *default assignment* 运算符。

和先前一样，你可以把内部容器的类型也参数化：

```
// basics/stack6decl.hpp
template <typename T, typename CONT = std::deque<T> >
class Stack {
private:
    CONT elems;             // 元素
public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 传回 stack 的顶端元素
    bool empty() const {    // stack 是否为空
        return elems.empty();
    }

    // 以元素类型为 T2的 stack赋值
    template <typename T2, typename CONT2>
    Stack<T,CONT>& operator= (Stack<T2,CONT2> const&);
};
```

此时的 template *assignment* 运算符可实作如下：

```
// basics/stack6assign.hpp
template <typename T, typename CONT>
template <typename T2, typename CONT2>
Stack<T,CONT>&
Stack<T,CONT>::operator= (Stack<T2,CONT2> const& op2)
{
    if ((void*)this == (void*)&op2) {     // 判断是否赋值给自己
        return *this;                     // 译注：请见 p47之「英文版勘误」
```

```
    }

    Stack<T2,CONT2> tmp(op2);              // 创建 op2的一份拷贝
    elems.clear();                         // 移除所有现有元素
    while (!tmp.empty()) {                 // 复制所有元素
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
```

记住，对 class templates 而言，只有「实际被调用的成员函数」才会被实例化。因此如果你不至 于令不同（元素）类型的 stacks 彼此赋值，那么甚至可以拿 vector 当作内部元素的容器（译注： 而先前的程序代码完全不必改动）：

```
// stack for ints，使用 vector作为内部容器
Stack<int,std::vector<int> > vStack;
...
vStack.push(42
);
vStack.push(7)
;
std::cout << vStack.top() << std::endl;
```

由于 template *assignment* 运算符并未被用到，编译器不会产生任何错误讯息抱怨说「内部容器 无法支持 push_front()操作」。

以上例子的完整实作全部包含于以 stack6 开头的文件中，位于子目录 basics 之下。

5.4 Template Template Parameters（双重模板参数）

一个 template parameter 本身也可以是个 class template 这一点非常有用 我们将再次以 stack class template 说明这种用法。

为了使用其它类型的元素容器stack class 使用者必须两次指定元素类型:一次是元素类型本身，另一次是容器类型：

```
Stack<int,std::vector<int> > vStack;    // int stack，以 vector为容器
```

如果使用 template template parameter，就可以只指明元素类型，无须再指定容器类型：

```
Stack<int,std::vector> vStack;          // int stack，以 vector为容器
```

为了实现这种特性，你必须把第二个 template parameter 声明为 template template parameter。原 则上程序代码可以写为：

```
// basics/stack7decl.cpp
template <typename T,
          template <typename ELEM> class CONT = std::deque >
class Stack {
private:
    CONT<T> elems;          // 元素
public:
    void push(T const&);    // push 元素
    void pop();             // pop 元素
    T top() const;          // 传回 stack 顶端元素
    bool empty() const {    // stack 是否为空
        return elems.empty();
    }
};
```

与先前的 stack 差别在于，第二个 template parameter 被声明为一个 class template：

```
template <typename ELEM> class CONT
```

其默认值则由 std::deque<T> 变更为 std::deque。这个参数必须是个 class template，并以第一参数的类型完成实例化：

```
CONT<T> elems;
```

本例「以第一个 template parameter 对第二个 template parameter 进行实例化」只是基于例子本身的需要。实际运用时你可以使用 class template 内的任何类型来实例化一个 template template parameter。

和往常一样，你也可以改用关键词 class 而不使用关键字 typename 来声明一个 template parameter；但 CONT定义的是一个 class 类型，因此你必须使用关键词 class来声明它。所以，
下面的程序代码是正确的：

```
template <typename T,
          template <typename ELEM> class CONT = std::deque >    //OK
class Stack {
    ...
};
```

下面的程序代码则是错误的：

```
template <typename T,
          template <typename ELEM> typename CONT = std::deque > //ERROR
class Stack {
```

```
    ...
};
```

由于 template template parameter 中的 template parameter 实际并未用到，因此你可以省略其名称：

```
template <typename T,
          template <typename> class CONT = std::deque >
class Stack {
...
};
```

所有成员函数也必须按此原则修改：必须指定其第二个template parameter为template template parameter。同样的规则也适用于成员函数的实作部份。例如成员函数 push()应该实作如下：

```
template <typename T, template <typename> class CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem); // 追加元素
}
```

另请注意，function templates 不允许拥有 template template parameters。

Template Template Argument 的匹配（matching）

如果你试图使用上述新版 Stack，编译器会报告一个错误：默认值 std::deque不符合 template template parameter CONT的要求。问题出在 template template argument 不但必须是个 template，而且其参数必须严格匹配它所替换之 template template parameter 的参数。Template template argument 的默认值不被考虑，因此如果不给出拥有默认值的自变量值时，编译器会认为匹配失败。

本例的问题在于：标准库中的 std::deque template 要求不只一个参数。第二参数是个配置器（allocator），它虽有默认值，但当它被用来匹配CONT 的参数时，其默认值被编译器强行忽略了。

办法还是有的。我们可以重写 class 声明语句，使 CONT参数要求一个「带两个参数」的容器：

```
template <typename T,
          template <typename ELEM,
                    typename ALLOC = std::allocator<ELEM> >
                    class CONT = std::deque>
class Stack {
private:
    CONT<T> elems;         // 元素
    ...
};
```

由于 ALLOC并未在程序代码中用到，因此你也可以把它省略掉。

Stack template 的最终版本如下。此一版本支持对「不同元素类型」之 stacks 的彼此赋值动作：

```
// basics/stack8.hpp
```

```
#ifndef STACK_HPP
#define STACK_HPP

#include <deque>
#include <stdexcept>
#include <memory>

template <typename T,
          template <typename ELEM,
                    typename = std::allocator<ELEM> >
                    class CONT = std::deque>
class Stack {
private:
   CONT<T> elems;          // 元素
public:
   void push(T const&);    // push 元素

   void pop();             // pop 元素
   T top() const;          // 传回 stack 的顶端元素
   bool empty() const {    // stack 是否为空
      return elems.empty();
   }

   // 赋予一个「元素类型为 T2」的 stack
   template<typename T2,
            template<typename
            ELEM2,
                     typename = std::allocator<ELEM2>
                    > class CONT2>
   Stack<T,CONT>& operator= (Stack<T2,CONT2> const&);
};

template <typename T, template <typename,typename> class CONT>
void Stack<T,CONT>::push (T const& elem)
{
   elems.push_back(elem);    // 追加元素
}

template<typename T, template <typename,typename> class CONT>
void Stack<T,CONT>::pop ()
{
   if (elems.empty()) {
```

```
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();        // 移除最后一个元素
}

template <typename T, template <typename,typename> class CONT>
T Stack<T,CONT>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back();     // 传回最后一个元素的拷贝
}


template <typename T, template <typename,typename> class CONT>
template <typename T2, template <typename,typename> class CONT2>
Stack<T,CONT>&
Stack<T,CONT>::operator= (Stack<T2,CONT2> const& op2)
{
    if ((void*)this == (void*)&op2) {     // 是否赋值给自己
        return *this;
    }

    Stack<T2,CONT2> tmp(op2);             // 创建  assigned stack的一份拷贝

    elems.clear();                        // 移除所有元素
    while (!tmp.empty()) {                // 复制所有元素
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}

#endif // STACK_HPP
```

下面程序使用了上述最终版本的 stack：

```
// basics/stack8test.cpp

#include <iostream>
#include <string>
#include <cstdlib>
```

```
#include <vector>
#include "stack8.hpp"

int main()
{
    try {
        Stack<int>   intStack;         // stack of ints
        Stack<float> floatStack;       // stack of floats

        // 操控 int stack
        intStack.push(42
        );
        intStack.push(7)
        ;
        // 操控 float stack
        floatStack.push(7.7);

        // 赋予一个「不同类型」的 stack
        floatStack = intStack;

        // 打印 float stack
        std::cout << floatStack.top() << std::endl;
        floatStack.pop();
        std::cout << floatStack.top() << std::endl;
        floatStack.pop();
        std::cout << floatStack.top() << std::endl;
        floatStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
    }

    // int stack，以 vector为其内部容器
    Stack<int,std::vector> vStack;
    // ...
    vStack.push(42
    );
    vStack.push(7)
    ;
    std::cout << vStack.top() << std::endl;
    vStack.pop();
}
```

程序运行的输出结果为：

```
7
42
Exception: Stack<>::top(): empty stack
7
```

注意，template template parameter 是极晚近才加入的C++ 特性，因此上面这个程序可作为一个极佳工具，用来评估你的编译器对 template 特性的支持程度。

5.5 零值初始化（Zero Initialization）

对于基本类型如 int、double、pointer type（指针类型）来说，并没有一个 *default* 构造函数将它 们初始化为有意义的值。任何一个未初始化的区域变量（local variable），其值都是未定义的：

```
void foo()
{
    int x;        // x的值未有定义
    int* ptr;     // ptr指向某处(而不是哪儿都不指向)
}
```

你可能在 template 程序代码中声明某个变量，并且想令这个变量被初始化为其默认值；但是当变 数是个内建类型（built-in type）时，你无法确保它被正确初始化：

```
template <typename T>
void foo()
{
    T x;    // 如果 T是内建类型，则 x值未有定义
}
```

为解决这个问题，你可以在声明内建类型的变量时，明确调用其 *default* 构造函数，使其值为零（对bool类型而言则是 false）。也就是说 int()导致 0 值。这样一来你就可以确保内建类型的变 数有正确初值：

```
template <typename T>
void foo()
{
    T x = T(); // 如果 T是内建类型，则 x被初始化为 0或 false
}
```

Class template 的各个成员，其类型有可能被参数化。为确保初始化这样的成员，你必须定义一个构造函数，在其「成员初值列」（member initialization list）中对每个成员进行初始化：

```
template <typename
T> class MyClass
```

```
{ private:
    T x;
public:
    MyClass() : x() {   // 这么做可以确保：即使 T为内建类型，x也能被初始化。
    }
    ...
};
```

5.6 以字符串字面常数（String Literals）作为 Function Template Arguments

以 *by reference* 传递方式将「字符串字面常数」（string literals）传递给 function template parameters 时，有可能遇上意想不到的错误：

```
// basics/max5.cpp
#include <string>

// 注意：使用 reference parameters
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

int main()
{
    std::string s;

    ::max("apple","peach");     // OK：类型相同
    ::max("apple","tomato");    // ERROR：类型不同
    ::max("apple",s);           // ERROR：类型不同
}
```

问题出在这几个字符串字面常数（string literals）的长度不同，因而其底层的 array 类型也不同。

换句话说 "apple" 和 "peach" 的 array 类型都是 char const[6]，而 "tomato" 的 array 型 别是 char const[7]。上述调用只有第一个合法，因为两个参数具有相同类型；然而如果你使

用 *by value* 传递方式，就可以传递不同类型的字符串字面常数（string literals），其对应的 array 大小不同：

```
// basics/max6.hpp
#include <string>
```

```
// 注意：使用 non-reference parameters
template <typename T>
inline T max(T a, T b)
{
   return a < b ? b : a;
}

int main()
{
   std::string s;

   ::max("apple","peach");     // OK：类型相同
   ::max("apple","tomato");    // OK：退化为相同类型
   ::max("apple",s);           // 错误：类型不同
}
```

这种方式之所以可行，因为在自变量推导过程中，惟有当参数并不是一个 reference 类型时，「array 转为 pointer」的转型动作（常被称为退化, *decay*）才会发生。这个规则可藉以下例子加以说明：

```
// basics/refnonref.cpp
#include <typeinfo>
#include <iostream>

template <typename T>
void ref (T const& x)
{
   std::cout << "x in ref(T const&): "
             << typeid(x).name() << std::endl;
}

template <typename T>
void nonref (T x)
{
   std::cout << "x in nonref(T):    "
             << typeid(x).name() << std::endl;
}

int main()
{
   ref("hello");
   nonref("hello");
}
```

在这个例子中，同一个字符串字面常数（string literal）分别被传递给两个 function templates，其一声明参数为 reference，其二声明参数为 non-reference。两个函数都使用 typeid运算符打印其具现化后的参数类型。

typeid 运算符会传回一个左值（lvalue），其类型为 std::type_info，其内封装了「typeid运算符所接收之算式（expression）」的类型表述（representation）。std::type_info的成员函式 name()把这份「类型表述」以易读的字符串形式传回给调用者。*C++ Standard* 并不要求 name() 传回有意义的内容，但是在良好的 C++ 编译器中，它会传回一个字符串内含「typeid运算符的 参数类型」的完好描述。在某些编译器实作品中，这个字符串可能以重排（*mangled*）形式出现，但也有些反重排工具（*demangler*）可以把它调整回人类可读的形式。例如上面程序的输出可能如下：

```
x in ref(T const&): char [6]
x in nonref(T):     const char *
```

如果你曾经在程序中把「char array」和「char pointer」混用，你可能被这个令人惊奇的问题搞得头昏脑胀。不幸的是，没有一个普遍适用的办法可以解决这个问题。根据所处情况的不同，你可以：

以*by value*传递方式代替*by reference*传递方式。然而这会带来不必要的拷贝。

分别对*by value*传递方式和*by reference*传递方式进行重载。然而这会带来模棱两可问题（ambiguities, 歧义性），请参考 B.2.2 节。

以具体类型（例如 std::string）重载之 以 array 类型重载之。例如：

```
template <typename T, int N, int M>
T const* max (T const (&a)[N], T const (&b)[M])
{
   return a < b ? b : a;
}
```

强迫使用者进行显式转型（explicit conversions）

本例之中，最好的方式是对 string重载（请参考 2.4 节）。这么做有其必要。如果不这么 做，对两个字符串字面常数（string literals）调用 max()是合法的，但 max()会以 operator< 比较两个指针的大小，而所比较的其实是指针的地址，不是两个字符串的字面值。这也是为什么使用 std::string比使用 C-style 字符串更好的原因之一。

5.7 摘要

当你要操作一个取决于（受控于）template parameter 的类型名称时，应该在其前面冠以关键字 typename。嵌套类别（nested classes）和成员函数（member functions）也可以是 templates。应用之一是， 你可以对「不同类型但彼此可隐式转型」的两个 template classes 互相操作，而类型检验（type checking）仍然起作用。

assignment（赋值）运算符的 template 版本并不会取代 *default assignment* 运算符。

你可以把 class templates 作为 template parameters 使用，称为 template template parameters。

Template template arguments 必须完全匹配其对应参数。预设的 template arguments 会被编译 器忽略，要特别小心。

当你实例化（*instantiated*）一个隶属内建类型（built-in type）的变量时，如果打算为它设定 初值，可明确调用其 *default* 构造函数。

只有当你以 *by value* 方式使用字符串字面常数（string literals）时，字符串底部的 array 才会被转 型（退化）为一个字符指针（也就是发生 "array-to-pointer" 转换）。

6

实际运用 Templates

Using Templates in Practice

和一般程序代码相比，template 程序代码有些不同。某种程度而言，templates 处于macros（宏）和一般（non-template）声明之间。虽然这么说恐怕是过于简化了，但这种说法颇适用于我们撰写算法和数据结构时所使用的templates，也适用于我们日常后勤处理时用以表达和分析程序所使用的 templates 身上。

本章将探讨实际编程可能碰到的部份问题，然而并不试图探寻这些问题背后的技术细节。技术 细节将在第10章讨论。为了让讨论更简单些，这里假设C++编译系统由传统的「编译器＋链接器」组成（事实上此种结构以外的 C++编译系统也相当少见）。

6.1 置入式模型（Inclusion Model）

有很多种方法可以组织你的 template 程序代码。本节介绍截至本书完稿为止最常见的一种方法： 置入式模型。

6.1.1 链接错误（Linker Errors）

大多数 C/C++ 程序员大致上都按照以下方式来组织他们的 non-template 程序代码：

Classes 和其它类型被全体放置于头文件（header files）。通常头文件的后缀名称（扩展名）为.hpp（或 .H, .h, .hh, .hxx等等）。

全局变量和 non-inline 函数只在头文件中置入声明语句，定义式则置于 .C 文件。这里的 .C 文件是个统称，通常其后缀名称（扩展名）为.cpp（或.C, .c, .cc, .cxx等等）。

这种方式运作良好，程序能够轻易找到各个类型的定义，并避免一个变量或函数被多次重复定义，于是链接器（linker）可以正常工作。

如果牢记上述规则，很多 template 初学者就会犯一个常见错误。让我以下面这个例子加以说明。就像组织一般 non-template 程序代码一样，我们把 template 声明于某个头文件：

```
// basics/myfirst.hpp
#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// template声明语句
template <typename T>
void print_typeof (T const&);
```

```
#endif // MYFIRST_HPP
```

print_typeof()是一个简单的辅助函数，它打印参数的类型信息（type information）。函数的实际代码被置于一个.C 文件中：

```
// basics/myfirst.cpp
#include <iostream>
#include <typeinfo>
#include "myfirst.hpp"

// template 的实作码/定义式
template <typename T>
void print_typeof (T const& x)
{
    std::cout << typeid(x).name() << std::endl;
}
```

这个例子使用 typeid运算符，把参数的类型以字符串形式打印出来（见 5.6 节）。

此后，我们又在另外一个.C文件使用这个 template，并将 template 声明语句（的所在文件）以
#include包含进来：

```
// basics/myfirstmain.cpp
#include "myfirst.hpp"

// 使用含入之 template
int main()
{
    double ice = 3.0;
    print_typeof(ice); // 以 double值调用 function template
}
```

大多数 C++ 编译器都可以正确编译上述程序代码，然而大多数链接器会报告一个错误，表示无法 找到 print_typeof()函数的定义。

错误的原因在于，function template print_typeof()的定义并没有被实例化。为了实例化一个 template，编译器必须知道「以哪一份定义式」以及「以哪些 template arguments」对它实例化。 不幸的是先前这个例子中，这两项信息被分置于两个分开编译的文件。因此当编译器看到对 print_typeof()的调用时，看不到其定义，无法以 double 类型来实例化 print_typeof()。 于是编译器假设这个 template 的定义位于其它某处，因而只生成一个对该定义的 reference，并将这个reference所指的定义式留给链接器去决议（*resolve*）。另一方面，当编译器处理 myfirst.cpp时，它又察觉不到该以哪个自变量类型来实例化其定义式。

6.1.2 把 Templates 放进头文件（Header Files）

就像处理 macro（宏）和 inline 函数一样，解决上面问题的常见办法是：把 template 定义式放到其声明语句所在的头文件中。面对前例，我们可以在 myfirst.hpp最后一行加入：

```
#include "myfirst.cpp"
```

也可以在用到该template 的每一个 .C文件中#include "myfirst.cpp"。第三种作法是完全丢开 myfirst.cpp，把声明和定义全部放进 myfirst.hpp：

```
// basics/myfirst2.hpp
#ifndef MYFIRST_HPP
#define MYFIRST_HPP

#include <iostream>
#include <typeinfo>

// template的声明语句
template <typename T>
void print_typeof (T const&);

// template的实作码/定义式
template <typename T>
void print_typeof (T const& x)
{
    std::cout << typeid(x).name() << std::endl;
}

#endif // MYFIRST_HPP
```

这种 templates 组织法称为「置入式模型」（inclusion model）。现在你会发现，上述程序可被正确编译、链接、执行。

有一些需要注意的地方。最该引起注意的是：「含入 myfirst.hpp」的代价相当大。此处所谓 代价，不仅是指 template 定义式的体积增加，也包括含入 myfirst.hpp时一并含入的其它（由 myfirst.hpp 含入的）文件，本例是<iostream> 和 <typeinfo>。你会发现，这可能导致成千上万行程序代码被含进来，因为诸如<iostream> 这样的文件也是以此种方式来定义 templates。

这是一个很实际的问题。这显然会在编译大型程序时显著增加编译时间。稍后数节将试图寻找 解决此问题的一些办法。很多现实世界中的程序会吃掉你数小时的编译+链接时间（关于这个我 们有多次经验，编译和链接一个程序，竟用掉悠悠数天工夫）。

尽管存在这个问题，我们仍然强烈推荐：只要可能，你应该使用置入式模型（inclusion model）来组织你的 template 程序代码。虽然稍后还将审查另两种办法，但那些办法所带来工程缺陷看起 来远比时间占用问题更严重得多（不过它们也可能带来软件工程之外的其它好处）。

另一个关于置入式模型的考虑比较微妙：与 inline 函数及 macros（宏）明显不同的是，non-inline function templates 并不在调用端被展开。每当它们被实例化一次，编译器便从头创建一份函数拷 贝。由于这个过程完全自动化，编译器可能最终在两个不同的文件中创建出同一个 templates 具 现体的两份拷贝，而某些链接器会因为看到同一函数的两份定义而报错。理论上这不该我们操 心，这应该是编译系统的事。实际工作中大多数时候都能够运作良好，无需我们操心。然而对 于「可能自行建立库」的大型项目而言，问题偶尔也可能冒出来。第 10 章会讨论实例化的 细节，你的C++ 编译器文件往往也会对此有所说明，这些都有助于解决问题。

最后要指出的是，本节例中对于 function templates 所谈的内容，也适用于 class templates 的成员函数和 static 成员变数。当然，也同样适用于 member function templates。

6.2 显式实例化（Explicit Instantiation）

置入式模型（inclusion model）保证所有用到的 templates 都能被实例化。之所以有这层保障， 是因为C++ 编译器会在 templates 被使用时自动具现它。*C++ Standard* 另提供一种方式，使你得 以手动将 templates 实例化。这种方式称为「显式（明确）实例化指令」（explicit instantiation directive）。

6.2.1 显式实例化（Explicit Instantiation）示例

为说明「显式实例化」概念，让我们回头看看那个导致链接错误的例子。为避免链接错误，我们可以为程序添加如下的一个文件：

```
// basics/myfirstinst.cpp
#include "myfirst.cpp"

// 明确以 double类型将 print_typeof()实例化
template void print_typeof<double>(double const&);
```

这个「显式实例化指令」由两部份组成：先是关键词 template，然后是一个「template parameters 已被完全替换」后的函数声明。本例是对函数做显式（明确）实例化动作，我们也可以运用相 同手法对一个成员函数或一个 static 成员变数做显式（明确）实例化动作。例如：

```
// 明确地以 int类型对 MyClass<> 的构造函数进行实例化动作
template MyClass<int>::MyClass();

// 明确地以 int类型对 function template max()进行实例化动作
template int const& max (int const&, int const&);
```

你也可以显式（明确）实例化一个 class template。这是一种简便做法，相当于要求编译器具现化该 class template 的所有「可被实例化」成员，但不包括先前已被特化（specialized）或已被实例化的成员：

```
// 明确地以 int类型对 Stack<> class进行实例化动作
template class Stack<int>;

// 明确地以 string类型对 Stack<> 的一部份成员进行实例化动作
template Stack<std::string>::Stack();
template void Stack<std::string>::push(std::string const&);
template std::string Stack<std::string>::top() const;

// 错误：不能对「已被显式（明确）实例化的 class」的某个成员再次实例化
// 译注：Stack<int> 已在本例第一行被显式（明确）实例化。
template Stack<int>::Stack();
```

在程序之中每一个不同物体(distinct entity)最多只能有一份显式具现体(explicit instantiation)。换句话说你可以明确具现出 print_typeof<int> 和 print_typeof<double>，但程序中每一条显式实例化指令（explicit instantiation directive）只能出现一次。如果不遵守这个规则，往往引发链接错误，链接器会告诉你：具现体（instantiated entities）被重复定义了。

手动进行实例化，有一个明显缺点：我们必须非常仔细地搞清楚应该实例化哪些物体。在大型项目中，这对程序员来说是一个极重的负担；基于这个原因，我们并不推荐这种用法。我们曾经在一些实际项目开始时低估了这个负担的严重性。项目日趋复杂后，我们都为当初的错误决定懊悔不已。

然而「显式（明确、手动）实例化」也有一些好处，毕竟实例化动作因而得以根据程序的实际需求进行最佳调整。很明显，这么做得以避免含入巨大的头文件。Template 定义式的源码得以隐藏，不过也因此客户端无法产生更多具现体。最后要注意的一点是，控制 template 被具现于某个精确位置（我是指目标文件, object file）有时很有用，而「自动实例化」不可能做到这一点（细节请见第 10 章）。

stack.hpp:

```
#ifndef STACK_HPP
#define STACK_HPP

#include <vector>

template <typename T>
class Stack {
  private:
    std::vector<T> elems;
  public:
    Stack();
    void push(T const&);
    void pop();
    T top() const;
};

#endif
```

stackdef.hpp:

```
#ifndef STACKDEF_HPP
#define STACKDEF_HPP

#include <stack.hpp>

template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem);
}
...

#endif
```

图 6.1 template 的声明语句与定义式分离

6.2.2 结合置入式模型（Includsion model）和显式实例化（Explicit Instantiation）

为了进一步讨论究竟该使用「置入式模型」或使用「显式实例化」，我们将 templates 的声明语句 和定义式分别置于不同的两个文件。通常我们会令这两个文件的扩展名看起来像头文件（意欲 被含入），这种作法是聪明的。于是我们把 `myfirst.cpp` 易名为 `myfirstdef.hpp`，并在文件 首尾加入「防止重复含入」的防卫式预处理指令（所谓 guard preprocessor）。图 6.1 展示这样一个 `Stack<>` class template。

现在，如果我们想要使用置入式模型，可以简单地将定义式所在的头文件`stackdef.hpp` 包含进来。如果我们想要使用显式（明确）实例化，可以将声明语句所在的头文件`stack.hpp` 包含进 来，并提供一个.C文件，其中有必要的显式实例化指令（见图 6.2）。

stacktest1.cpp:

```
#include "stack.hpp"
#include <iostream>
#include <string>

int main()
{
  Stack<int> intStack;
  intStack.push(42);
  std::cout << intStack.top() << std::endl;
  intStack.pop();

  Stack<std::string>  stringStack;
  stringStack.push("hello");
  std::cout << stringStack.top() << std::endl;
}
```

stack_inst.cpp:

```
#include "stackdef.hpp"
#include <string>

// instantiate class Stack<> for int
template class Stack<int>;

// instantiate some member functions of Stack<> for strings
template Stack<std::string>::Stack();
template void Stack<std::string>::push(std::string const&);
template std::string Stack<std::string>::top() const;
```

图 6.2 搭配两个 template 头文件，完成显式（明确）实例化

6.3 分离式模型（Separation Model）

前面讲述的两种作法都可以有效解决问题，而且完全符合C++*Standard*。但是C++*Standard* 还提 供了另一种机制，可以汇出（*export*）一个 template。这种机制称为C++ template 的分离式模型（separation model）。

6.3.1 关键词 export

原则上，export的使用相当容易：将 template 定义于某文件中，并将其定义式及其所有「非定 义声明」（nondefinition declarations）加上关键词 export。以先前出现的例子而言，这会导致 如下的 function template 声明：

```
// basics/myfirst3.hpp
```

```
#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// template 声明
export
template <typename T>
void print_typeof (T const&);

#endif // MYFIRST_HPP
```

exported templates 可被直接拿来使用，不需现场看到 templates 定义文件。换言之，template 的使 用和定义可分隔于两个不同的编译单元。上面的例子中，文件 myfirst.hpp 如今只含入 class template 的成员函数声明，这样就足以使用这些成员函数。原先的程序代码会引发链接错误，新版 本可以顺利链接。对比两个版本，我们只是添加了关键词 export。

在一个预处理档（preprocessed file）内（亦即在一个编译单元内），只需将 template 的第一个宣 告加上关键词 export；此后再次声明或定义，都会自动加入相同属性，也就是说均会被汇出（*exported*）。这就是 myfirst.cpp 不需修改的原因 — 因为 template 在头文件中被声明为 export，因此 myfirst.cpp 内的 template 定义都会被汇出（*exported*）。不过，从另一方面来 说，如果你愿意在 template 定义式中加入（其实多余的）关键词 export 也非常好，可提高程序 码可读性。

关键词 export适用于 function templates、class templates 成员函数、member function templates、class templates 的 static 成员变数。关键词 export也可以被用在 class template 声明语句中，其意义 是「汇出所有可被汇出的成员」，但 class template 本身并不会被汇出（其定义仍然只在头文件
中有效）。你也可以一如往常地隐式或显式定义一些 inline 成员函数，但它们不能被汇出：

```
export template <typename T>
class MyClass {
public:
    void memfun1();     // 会被汇出
    void memfun2() {    // 不会被汇出，因为它暗自成为 inline
        ...
    }
    void memfun3();     // 不会被汇出，因为它明确为 inline（见下方定义式）
    ...
};

template <typename T>
inline void MyClass<T>::memfun3 ()
{
    ...
}
```

请注意，关键词 export不能和关键词 inline合用，而且关键词 export应该总是写在关键词 template之前。以下程序代码是不合法的：

```
template <typename
T> class Invalid
{ public:
    export void wrong(T);            // 错误：export应该在 template之前
};

export template <typename T>         // 错误：export与 inline合用
inline void Invalid<T>::wrong<T>
{
}

export template <typename T>
inline T const& max (T const& a, T const& b)
{                                    // 错误：export与 inline合用
    return a < b ? b : a;
}
```

6.3.2 分离式模型（Separation Model）的局限

读到这里，你可能会奇怪：既然 export提供了这么多好处，为什么我们仍然提倡使用「置入式 模型」呢？事实是，使用 export会带来一些问题。

首先，即使在C++ *Standard* 已经制定四年的今天（译注：原书写于 2002 年），仍然只有一家公 司实作出 export。因此 export 并没有像其它C++ 特性那样地被广为运用，这也意味在目前 的时间点上，所有关于 export的知识或经验都颇为有限；而我们关于此特性的讨论也都十分谨 慎和保守。也许将来某一天，我们的疑惑会被解开（稍后将告诉你如何未雨绸缪）。

其次尽管 export看起来近乎魔术然而它确确实实存在实例化过程最终必须处理两个问题：何时具现 templates，以及 templates 被定义于何处。因此尽管这两个问题从源码角度来看并不互 相影响(neatly decoupled)，但在幕后编译系统却为它们建立了一种不可见的耦合关系(invisible coupled)。这个关系或许可以这样解释：当包含 template 定义式的文件发生变化时，这个文件，以及所有「实例化该 template」的文件，都不得不重新编译。这和「置入式模型」看起来并无本 质上的区别，但是从源码角度，这种关系却没有那么明显。这样的后果是，许多以「源码级
(source-base)技术」来管理依存关系(dependency)的工具，例如十分普及的 make 和 nmake 工具程序，如果像对待传统 non-template 程序代码一样地对待 exported template 程序代码，将无法正 确运作。这也意味编译器不得不花很多功夫去逐一牢记（bookkeeping）这些依存关系。最终结 果是：使用 exported template 并不比使用「置入式模型」节省多少编译时间。

最后一点：exported templates 有可能导致令人大吃一惊的语意问题。第 10 章会谈到此一问题的

细节。

很多人对 export 有一个误解：他们觉得这么一来就有可能在不提供源码的情况下发售 template 库（就像发售 non-template 库那样）。这是一个错误观念，因为「隐藏程序代码」并非语言层面上的议题：提供一个机制用以隐藏 exported template 定义式，差不多相当于提供一个机制用以隐藏 included template 定义式。虽然这么做或许可行（目前编译器并不支持这种作法），但不幸的是，这又带来新的困难：当编译器使用这种 exported template 并发生错误时，如何在错误信息中引用被隐藏的程序源码？

6.3.3 为分离式模型（Separation Model）预做准备

为了可以随时在「置入式模型」和「分离式模型」之间灵活切换，一个实际可行的办法是：使用预处理指令（preprocessor directives）。具体实作如下：

```
// basics/myfirst4.cpp
#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// 如果定义了 USE_EXPORT，就使用关键词 export
#if defined(USE_EXPORT)
#define EXPORT export
#else
#define EXPORT
#endif

// template的声明语句
EXPORT
template <typename T>
void print_typeof (T const&);

// 如果未定义 USE_EXPORT，就将 template定义式含入
#if !defined(USE_EXPORT)
#include "myfirst.cpp"
#endif

#endif // MYFIRST_HPP
```

藉由定义或取消 USE_EXPORT 预处理符号，我们便可以在两种模型之间切换。如果程序在含入 myfirst.hpp之前定义有 USE_EXPORT，就使用「分离式模型」：

```
// 使用分离式模型：
#define USE_EXPORT
#include "myfirst.hpp"
...
```

如果程序未定义 USE_EXPORT，则使用置入式模型，因为 myfirst.hpp会自动含入 myfirst.cpp

内的定义式：

```
// 使用置入式模型：
#include "myfirst.hpp"
...
```

尽管有这样的灵活性，我们还是重申，除了明显的逻辑差异外，「置入式模型」和「分离式模型」之间还有一些微妙的语意差异。
请注意，我们也可以显式实例化 exported templates。如此一来 templates 便可以在不同的文件中定义。为了能够在「置入式模型」、「分离式模型」和「显式实例化」之间切换，我们可以使用 USE_EXPORT来控制程序结构，并辅以 6.2.2 节所说的办法。

6.4 Templates 与关键词 inline

将短小的函数声明为 inline，是提高程序执行速度的一个惯用手法。关键词 inline 用来告诉编 译器，最好将「函数调用」替换为「被调用函数之实作码」。然而编译器并不一定进行此一替 换工作，换句话说它握有决定权。

藉由「将定义式置于头文件，并将该头文件含入多个 .C文件内」的手法，function templates 和 inline functions 都可以被定义于多个编译单元内。

这会产生一种假象：function templates 预设情况下就是 inline；其实它们并不是。如果你希望编 写一个 inline function template，应该明确使用关键词 inline（除非它本来就是 inline，例如它被 写于一个 class 定义式内）。

所以，对于没有被写在 class 定义式内的小型 template functions，你应该将它们声明为 inline。

6.5 预编译头文件（Precompiled Headers）

即使没有templates，C++头文件也可能很大，编译它们需要很长时间。Templates 可能使得编译时间更长。许多耐不住煎熬的程序员在很多场合中对此大声疾呼，促使编译器厂商实作出一个名为「预编译头文件」（precompiled headers）的机制。这个方案超越C++ *Standard* 涵盖范围，其细节因编译器厂商而异。虽然我们把「如何产生和使用预编译头文件」的细节留给支持这一 特性的编译系统所带文件，但在这里简短地说明其工作方式，还是非常有益的。

编译器编译某个文件时，会从文件起始处扫描至文件尾端。当编译器处理文件中的每一个语汇 单元（token；可能来自被 #include 的文件），它便对其内部数据与状态进行调整，包括将一 些条目（entries）添加到符号表（symbols table）中，以便这些符号稍后可被搜寻到。处理过程
中，编译器可能会在目标文件（object file）中产生一些码。

「预编译头文件」便是基于这样一个事实：程序中往往有很多文件一开头都含入了相同的程序 码。为了更好地讨论这个问题，假设每个文件的前 N 行程序代码相同。我们可以把这 N

行程序代码 编译出来，并将编译器此时的内部数据和状态完整保存于一个所谓的「预编译头文件」中。之 后编译程序的其它每一个文件时，编译器便可将此「预编译头文件」加载，然后从第 N+1 行开始继续编译。值得注意的是，编译器加载「预编译头文件」的速度，比实际编译最前面N行程序码，要快上几个数量级，但第一次把编译器内部数据与状态储存到「预编译头文件」时，通 常比编译 N 行程序代码的代价大得多，往往慢上 20% 到 200%。

有效利用「预编译头文件」的办法是，保证各个文件最前面的「相同程序代码」尽可能多。实际 情况中，这意味各文件最前面的「相同的 #include 指令」尽可能多，因为正是这些 #include 指令占用了极长的编译时间。此外最好能注意头文件被含入的次序。例如下面两段程序代码：

```
#include <iostream>
#include <vector>
#include <list>
```

和

```
#include <list>
#include <vector>
```

「预编译头文件机制」在这里不起作用，因为这两段程序代码的次序并不相同。

有些程序员决定多含入一些或许用不到的头文件，他们认为这至少比无法使用「预编译头文件」要好。这不失为一个可以大幅简化头文件含入问题的良策。例如我们可以写一个std.hpp头文件，把所有标准库的头文件都包含进来：

```
#include <iostream>
#include <string>
#include <vector>
#include <deque>
#include <list>
...
```

这个头文件可以被预编译。每一个需要用到标准库的程序，只需在开头简单加上这么一行：

```
#include "std.hpp"
```

注意，这会花费相当长一段编译时间，但如果你的系统有足够的内存，「预编译头文件」机制会显著减少编译时间 — 尤其是和缺乏「预编译头文件机制」相比。以这种方式来组织标准库的头文件非常方便，因为这些文件很少发生变化。这么一来 std.hpp的「预编译头文件」只被生成一次。否则「预编译头文件」应该作为依存关系（dependency）的一部份被加入项目 组态（project confiuration）中。例如当所含入的文件发生变化时，make 之类的工具就可以发觉 并令编译器重新编译它。

一个非常不错的「预编译头文件」管理办法是，将「预编译头文件」分层：从「使用最广泛且 最不易发生变化者（例如上述的std.hpp）」到「较不易发生变化且预编译仍有价值者」。然而如果头文件正处于密集开发期而频繁有着变化，为它们生成「预编译头文件」所用的时间， 会比预编译头文件所节省的时间还长.这个方案的关键在于;重复使用较稳定层a more stable layer）的预编译头文件，可以改善较不稳定层（a less stable layer）的预编译时间。举个例子， 除提供一个std.hpp 头文件（可被预编译）之外，我们还定义一个core.hpp 头文件，它

又含入项目中的数个额外文件，这么做是为了导入一个稳定层：

```
#include "std.hpp"
#include "core_data.hpp"
#include "core_algos.hpp"
...
```

由于这个文件以 `#include "std.hpp"` 开始，编译器就可以将相关的「预编译头文件」加载，然后从下一行继续编译，从而避免重新编译所有的标准库头文件。当整个文件被处理完毕，编译器会生成一个新的「预编译头文件」。使用者可经由 `core.hpp` 快速存取其所含入的大量功能函数 — 因为编译器会加载那个新的「预编译头文件」。

6.6 Templates 的除错（Debugging）

对 templates 除错，可能遭遇两类难题。第一类对 templates 撰写者颇为麻烦：如何保证我们所撰 写的 templates 对于「符合文件要求」的任何 template arguments 都能够正常运作？另一类问题正 好相对：作为使用者，当 templates 的行为未与文件一致时，如何得知我们的程序代码违反了哪一个 template parameter 的条件？

深入探讨这个问题之前，我们应该先静下心来细想：template parameters 可能带来哪些种类的约 束条件（constraints）。本节大部份内容都是讨论那些「如果违反就会引发编译错误」的约束条 件，我们把这些条件称为语法约束（syntactic constraints），包括：必须存在特定某种构造函数、 对某些特殊函数的调用会造成模棱两可（歧义性）等等。其它种类的约束条件称为语意约束
（semantic constraints）。

这些约束很难以机械化方式查验出来，通常那是不实际的。例如我们可以要求template type parameter 必须定义一个 `operator<`（这是语法约束），但通常我们也会要求这个 `operator<`能够在它所接受的范围内比较两值大小（这是语意约束）。

concepts（概念）这一术语经常被用到，表示一个「约束集」（constraints set），在 template 程式库中会被反复提起C++ 标准库倚赖这样一些 concepts「随机存取迭代器」（*random access iterator*）和「可被 *default* 构造函数加以建构」（*default constructible*）。各个 concept 之间可形成 继承体系，也就是说一个 concept 可以是另一个 concept 的强化（精炼, refinement）。所谓「更 强概念」（more refined concepts）通常不仅包括原概念的约束条件，还可能加上更多约束。例 如「随机存取迭代器」就是「双向迭代器」（*bi-directional iterator*）的强化。在这些术语的基 础上，我们可以这么说：对 templates 除错（debugging），就是找出在 templates 的实作和运用
中违反了哪些 concepts（概念）。

6.6.1 解读长篇错误讯息（Decoding the Error Novel）

一般而言，编译错误讯息通常简洁并直指问题所在。例如，当编译器给出这样的错误讯息：class X has no member `'fun'`，对我们而言找到程序代码中的问题并非难事（也许我们错把 `run`

写成了 fun）。但面对 templates 情况便不相同。考虑下面这个简单的程序片段，它使用C++标准程序 库并犯了一个错误。对 list<string> 进行搜寻时，我们使用 greater<int> 仿函数（function object），而正确的做法应该使用 greater<string>：

```
std::list<std::string> coll;
...
// 搜寻第一个大于 "A" 的元素
std::list<std::string>::iterator pos;
pos = std::find_if(coll.begin(),coll.end(),                    // 范围
                   std::bind2nd(std::greater<int>(),"A")); // 搜寻准则
```

这类错误通常是由于剪贴/复制程序代码而却了修改。

GNUC++ 是一个流传普遍的编译器，它的某一版本对以上程序代码报出如下错误讯息：

```
/local/include/stl/_algo.h: In function `struct _STL::_List_iterator<_STL::basic
_string<char,_STL::char_traits<char>,_STL::allocator<char>
>,_STL::_Nonconst_tra
its<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > > >
_STL::find_if<_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<
cha                                                    r>,_STL::allocator<char>
>,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::
char_traits<char>,_STL::allocator<char> > > >, _STL::binder2nd<_STL::greater<int
> > >(_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL:
:allocator<char>
>,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_tra
its<char>,_STL::allocator<char> > > >, _STL::_List_iterator<_STL::basic_string<c
har,_STL::char_traits<char>,_STL::allocator<char>
>,_STL::_Nonconst_traits<_STL:
:basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > > >, _STL::bi
nder2nd<_STL::greater<int> >, _STL::input_iterator_tag)':
/local/include/stl/_algo.h:115:  instantiated from `_STL::find_if<_STL::_List_
iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> >
,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::
allocator<char> > > >, _STL::binder2nd<_STL::greater<int> > >(_STL::_List_itera
tor<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> >,_STL
::_Nonconst_traits<_STL::basic_string<char,_STL::char_traits<char>,_STL::allo
cat               or<char>               >               >               >,
_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<c
har>,_STL::allocator<char>
>,_STL::_Nonconst_traits<_STL::basic_string<char,_STL
::char_traits<char>,_STL::allocator<char> > > >, _STL::binder2nd<_STL::greater<i
nt> >)'
testprog.cpp:18:  instantiated from here
```

```
/local/include/stl/_algo.h:78: no match for call to `(_STL::binder2nd<_STL::grea
ter<int>                                                                    >)
(_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<cha r> > &)'
/local/include/stl/_function.h:261: candidates are: bool _STL::binder2nd<_STL::g
reater<int> >::operator ()(const int &) const
```

这样的错误讯息不像是诊断信息，倒像天书一般。这会吓倒 template 初级用户。然而，如果你有一些实践经验的话，这类错误讯息倒也不是无法读懂，找出错误所在也没有那么困难。

这条错误讯息的第一部份意思是，在某个 function template 具现体（其名字特长）中存在一个错 误，对应的头文件深藏于 /local/include/stl/_algo.h中。接下来是这个特定具现体得以具 现的原因。在这个例子中，testprog.cpp（上例文件名）第 18 行引发了 find_if template 的具 现动作，该 template 位于 _algo.h 头文件第 115 行。编译器把这一切都报告给我们。我们并不 关心所有这些实例化动作，但这使我们得以确定整个实例化过程的来龙去脉。

上述例子中，我们相信所有这些 templates（译注）的确需要被实例化，我们想知道为什么具现 化过程没有成功。答案在错误讯息的最后一部份。"no match for call" 这句话告诉我们，由于参 数和自变量的类型不匹配，所以无法解析（*resolved*）函数调用语句。不仅如此，错误讯息还告诉我 们，类型 int（参数型别为 const int&）可能是引发错误的原由。回头看看程序第 18 行（std::bind2nd(std::greater<int>(),"A")），我们确实在那儿使用了一个 int，它和我们 要搜寻的 list 的类型并不相容。把 <int> 换成 <std::string> 问题便解决了。

译注：读者可能对「所有这些 templates」的含义有所疑惑，这里做个简单解释：上面的错误讯息本质上等同于「在 template1 中有一个错误，template1 的实例化是由于 template2 被实例化，而 template2 的实例化是由于 template3 被实例化」..等等等这里的 template1, template2, template3 就是作者所说的「所有 templates」。

毫无疑问，错误讯息的结构性还可以更好。有些问题可能出现在实例化开始之前。此外，比起使用「完全展开」的 template 具现体名称（例如 MyTemplate<YourTemplate<int> >），将它加以分解可缩短过长的名称（例如可以写成 MyTemplate<T> with T=YourTemplate<int>）。然而某些情况下得到完整的错误讯息很有用，因此其它编译器也给出类似的（冗长）错误讯息就不令人惊奇了（尽管有些编译器使用了上面所说的讯息结构）。

补充：Leor Zolman 撰写了一个实用的程序 STLFilt，可解读多种编译器输出的 STL 错误讯息。请参考 http://www.bdsoft.com/tools/stlfilt.html。

6.6.2 浅实例化（Shallow Instantiation）

如果实例化过程链（chain of instantiation）很长，编译器也可能给出前面那种长篇诊断讯息。让我以下面这个人为捏造的例子来说明问题：

```
template <typename T>
void clear (T const& p)
```

```
{
    *p = 0; // 假设 T是一个 pointer-like 类型
}

template <typename T>
void core (T const& p)
{
    clear(p);
}

template <typename T>
void middle (typename T::Index p)
{
    core(p);
}

template <typename T>
void shell (T const& env)
{
    typename T::Index i;
    middle<T>(i);
}

class Client {
public:
    typedef int Index;
};

Client main_client;

int main()
{
    shell(main_client);
}
```

这个例子展现软件开发中的一种典型分层方式：高层的 function templates（例如 `shell()`）倚赖其它部份（例如 `middle()`）完成，它们都用到了提供基本设施的函数（例如 `core()`）。当我们实例化 `shell()`时，它的所有下层函数也都会被实例化。本例之中，最深层的 `core()` 以 `int` 类型实例化（这是由于 `middle()`以 `Client::Index` 类型实例化的缘故），却试图从该类型中提领（*dereference*）数值，这么做是错误的。良好的一般性诊断讯息会包含「引发错误」的每一层调用轨迹，但我们的观察是，这么多讯息反而没有多大用处。

你可以在 [StroustrupDnE] 找到一段非常好的论述，围绕着此问题的核心思想。Bjarne Stroustrup 在讨论中给出两种办法，使能尽早判定 template arguments 是否满足某一组约束条件：(1) 经由 某种语言扩展性质；(2) 尽早使用 template parameters。我们将在 13.11 节简单讨论前一种 办法。后一种办法是藉由浅实例化（*shallow instantiation*）来曝露问题：安插一些用不到的程序 码，它们不干别的，只为了在「template arguments 无法满足更深层 templates 的需求」时得以引 爆错误。

我们可以在 `shell()`中添加一段程序代码，试图提领一个隶属于 `T::Index`类型的数值。例如：

```
template <typename T>
inline void ignore(T const&)
{
}

template <typename T>
void shell (T const& env)
{
    class ShallowChecks {
        void deref(T::Index ptr) {
            ignore(*ptr);
        }
    };
    typename T::Index i;
    middle(i);
}
```

如果`T`类型使得`T::Index`无法被提领（*dereferenced*），那么错误讯息就首先会针对`shell()`内的`ShallowChecks`local class。注意，由于这个 local class 实际上并未被使用，因此它不会对 `shell()`的执行速度造成影响。不幸的是许多编译器会给出一个警告，说 `ShallowChecks`class 和其成员没有被用到。这里我们使用了一个小技巧，藉由一个什么都不做的`ignore()`来抑制这个警告，但是这样做会增加程序代码的复杂度。

很明显，写出本例中的这种哑码（dummy code）可能和写出实作码一样复杂。为了控制复杂度不让它蔓生，把这一类小程序代码集合在某种库中是很自然的作法。这一类库可能包含一些宏（macros），它们可以被展开为程序代码，当 template parameter 的替换物违反了 parameter的 concept（概念）时，这些宏就能引发相应的错误讯息。这一类库中最流行的称为 Concept Check Library，是 Boost 库的一部份（请参考 [BCCL]）。

不幸的是这种技术的可移植性不高：不同编译器之间的错误分析方式差异极大，有时在较高层面上很难捕捉到问题。

6.6.3 长符号（Long Symbols）

6.6.1 节所分析的错误讯息，展现了 templates 的另一个问题：被实例化的 templates 程序代码 有

可能导致很长的符号。例如前面用到的 std::string可能被展开为：

```
STL::basic_string<char,_STL::char_traits<char>,
                  _STL::allocator<char> >
```

某些运用C++标准库的程序，可能产生长度超过10,000个字符的符号。这么长的符号可能会使编译器、链接器或除错器发出错误或警告。现代编译器运用压缩技术来降低这个问题，但在错误讯息中的效果并不明显。

6.6.4 追踪器（Tracers）

目前为止我们已经讨论了编译或链接template程序代码时可能遭遇的各种问题。然而，即使编译 链接成功，接踵而来的难题是：如何保证程序运行正常。templates 有时候会加深这个问题，因 为每一个以templates 表述的泛型程序代码，其行为都取决于templates 的客户端（译注：也就是取 决于「实例化类型」）。这种情况比起在一般（non-templates）classes和functions中要严重得多。追踪器（tracer）是一种软件装置，有助于在开发初期检查template 定义式的问题，以减轻日后除错工作的负担。

追踪器是一个用户自定的 class，可作为「待测试之template」的template argument。通常它恰恰 符合待测template 的要求，此外不具更多功能。然而更重要的是，追踪器可以报告出「在它身上进行的操作的具体轨迹」（a trace of the operations that are invoked on it）。这使我们得以更身 临其境地检测某个算法的效能efficiency），并给出演算过程中所有操作的完整序列sequenceof operations）。

下面是一个追踪器示例，用来测试一个排序算法（sorting algorithm）：

```
// basics/tracer.hpp
#include <iostream>
class SortTracer {
private:
    int value;                  // 用来排序的整数
    int generation;             // 此追踪器的生成个数
    static long n_created;      // 构造函数被调用的次
    数 static long n_destroyed; // 析构函数被调用的次
    数 static long n_assigned;  // 赋值次数
    static long n_compared;     // 比较次数
    static long n_max_live;     // 同一时间最多存在几个 objects

    // 重新计算「同一时间最多存在几个 objects」
    static void update_max_live() {
        if (n_created-n_destroyed > n_max_live) {
            n_max_live = n_created-n_destroyed;
        }
    }

public:
```

```
    static long creations() {
        return n_created;
    }
    static long destructions() {
        return n_destroyed;
    }
    static long assignments() {
        return n_assigned;
    }
    static long comparisons() {
        return n_compared;
    }
    static long max_live() {
        return n_max_live;
    }

public:
    // 构造函数（constructor）
    SortTracer (int v = 0) : value(v), generation(1) {
        ++n_created;
        update_max_live();
        std::cerr << "SortTracer #" << n_created
                  << ", created generation " << generation
                  << " (total: " << n_created - n_destroyed
                  << ')' << std::endl;
    }

    // copy 构造函数（copy constructor）
    SortTracer (SortTracer const& b)
     : value(b.value), generation(b.generation+1) {
        ++n_created;
        update_max_live();
        std::cerr << "SortTracer #" << n_created
                  << ", copied as generation " << generation
                  << " (total: " << n_created - n_destroyed
                  << ')' << std::endl;
    }

    // 析构函数（destructor）
    ~SortTracer() {
        ++n_destroyed;
```

```
        update_max_live();
        std::cerr << "SortTracer generation " << generation
                  << " destroyed (total: "
                  << n_created - n_destroyed << ')' << std::endl;
    }

    // 赋值（assignment）
    SortTracer& operator= (SortTracer const& b) {
        ++n_assigned;
        std::cerr << "SortTracer assignment #" << n_assigned
                  << " (generation " << generation
                  << " = " << b.generation
                  << ')' << std::endl;
        value = b.value;
        return *this;
    }

    // 比较（comparison）
    friend bool operator < (SortTracer const& a,
                            SortTracer const& b) {
        ++n_compared;
        std::cerr << "SortTracer comparison #" << n_compared
                  << " (generation " << a.generation
                  << " < " << b.generation
                  << ')' << std::endl;
        return a.value < b.value;
    }

    int val() const {
        return value;
    }
};
```

除了追踪待排序值value外，这个追踪器还提供若干成员，用来追踪实际排序的过程：generation追踪每个对象被拷贝的次数其它 static 成员则追踪构造函数和析构函数的被调用次数、 赋值和比较次数，以及「同一时间最多存在几个对象」。

static 成员被定义在一个分离的.C 文件中：

```
// basics/tracer.cpp
#include "tracer.hpp"

long SortTracer::n_created = 0;
```

```
long SortTracer::n_destroyed = 0;
long SortTracer::n_max_live = 0;
long SortTracer::n_assigned = 0;
long SortTracer::n_compared = 0;
```

这个特定的追踪器负责追踪「在一个template 内，建构、解构、赋值、比较等操作究竟以怎样的方式进行」。下面这个测试程序使用上述追踪器来追踪C++标准库的std::sort 演算法：

```
// basics/tracertest.cpp
#include <iostream>
#include <algorithm>
#include "tracer.hpp"

int main()
{
    // 准备样本数据源
    SortTracer input[] = { 7, 3, 5, 6, 4, 2, 0, 1, 9, 8 };

    // 打印起始值
    for (int i=0; i<10; ++i) {
        std::cerr << input[i].val() << ' ';
    }
    std::cerr << std::endl;

    // 记录起始值
    long created_at_start = SortTracer::creations();
    long max_live_at_start = SortTracer::max_live();
    long assigned_at_start = SortTracer::assignments();
    long compared_at_start = SortTracer::comparisons();

    // 执行算法
    std::cerr << "---[ Start std::sort() ]--------------------" << std::endl;
    std::sort<>(&input[0], &input[9]+1);
    std::cerr << "---[ End std::sort() ]----------------------" << std::endl;

    // 检验结果
    for (int i=0; i<10; ++i) {
        std::cerr << input[i].val() << ' ';
    }
    std::cerr << std::endl << std::endl;

    // 最终报告
    std::cerr << "std::sort() of 10 SortTracer's"
```

```
                << " was performed by: " << std::endl
                << SortTracer::creations() - created_at_start
                << " temporary tracers" << std::endl << ' '
                << "up to "
                << SortTracer::max_live()
                << " tracers at the same time ("
                << max_live_at_start << " before)" << std::endl << ' '
                << SortTracer::assignments() - assigned_at_start
                << " assignments" << std::endl << ' '
                << SortTracer::comparisons() - compared_at_start
                << " comparisons" << std::endl << std::endl;
}
```

这个程序一执行，会产生相当数量的输出，但我们可以由最终结果获得很多有用信息。对于某个 std::sort()函数实作版本，我们获得以下结果：

```
std::sort() of 10 SortTracer's was performed by:
 15 temporary tracers
 up to 12 tracers at the same time (10 before)
 33 assignments
 27 comparisons
```

我们看到了，程序执行期间产生 15 个临时追踪器，任一时间点最多有两个附加追踪器。

译注：此一执行结果视 STL 实作版本而异。上述是原作者在 g++ with SGI STL 中得到的结果。改用 VC7.1 with P.J.Plauger STL，执行结果为：

```
std::sort() of 10 SortTracer's was performed by:
 8 temporary tracers
 up to 11 tracers at the same time (10 before)
 32 assignments
 27 comparisons
```

这个追踪器成功扮演了两个角色：它证明我们的追踪器所提供的功能已经完全满足标准sort()算法的需要（例如 sort()并不需要operator== 和 operator>），此外它使我们对 sort演 算法的执行代价有了一些感受。然而它对于检测 std::sort template 的正确性并没有太大帮助。

6.6.5 Oracles（铭码）

追踪器既简单又有效，但它只能让我们针对特定数据或特定行为，追踪templates 的执行。我们也许还想知道，sorting算法中的比较运算符（comparison operator）有些什么要求，然而从上 面的例子中我们只能测试出，这个比较运算符的行为相当于一个「整数小于」功能。

有一个被某些人称为"oracles"（或 run-time analysis oracles）的「追踪器增强版」。它也是追踪 器，但与一个所谓的「推论引擎」（inference engine）程序相连，该程序可从各种因果关系中得出 某 些

结论。这种系统的某一实作版本已被 C++ 标准库采用，称为 MELAS，在 [MusserWangDynaVeri] 中有一些讨论。

Oracles使我们能够在某些情况下动态检验 template 算法，而无需指定 template arguments 的型别（它实际上被替换为 oracles 本身）或指定算法输入数据（当 oracles 的推理引擎受阻时，可能会要求某种假设性输入）。然而即便是oracles也只能处理复杂度不高的算法（此乃受限于推理引擎）。基于这些原因，我们并不深入研究 oracles，有兴趣的读者可以参考 [MusserWangDynaVeri]，或其中所引用的其它相关书籍。

6.6.6 原型/模本（Archetypes）

前面提到过，追踪器通常提供一个接口，使能满足「受追踪之 template 的最小要求」。当这样一个追踪器不在执行期产生任何输出时，我们把它叫做一个「原型/模本」（archetype）。这种东西使我们能够检验「一个 template 实作品除了计划内的约束外，没有其它更多语法约束」。很多 template 实作者都会针对 template library 中的每一个 concepts（概念）开发出一个原型/模本（archetype）。

6.7 后记

根据「单一定义规则」（one-definition rule, ODR），人们在实践中采用「将源码分置于头文件和 .C 文件」的组织形式。附录 A 对这条规则有深入的讨论。

置入式模型（Inclusion Model）和分离式模型（Separation Model）的争论由来已久。C++ 编译器实作品显示，置入式模型相当实用。然而第一个C++编译器并非如此，它对template 定义式的含入是隐寓的（implicitly），这往往使人误以为它使用了分离式模型。原始模型请参考第 10 章。

[StroustrupDnE] 极好地描述了 Stroustrup 对于 template 程序代码组织方式的一个设想，同时指出它所带来的实作困难。很明显，这种组织方式并非「置入式模型」。C++ 标准化过程中的某一段时间，人们认为置入式模型是惟一可行的组织方式。然而经过激烈论战后，人们对于可降低「程序代码耦合」的新模型渐渐有了充份的认识，这个模型便是「分离式模型」。和置入式模型不同的是，它并非基于任何既有实作品，而是一个理论上的模型，从被提出到最终实作出来（2002年 5 月），共耗用了 5 年以上的时间。

如果能够将「预编译头文件」的概念加以扩充，使得在一次编译时能够含入多个头文件，将十分具有诱惑力。原则上它会对预编译带来更好的粒度（a finer grained approach）。这里主要的障碍是预处理器（preprocessor）：某个文件内的宏（macros）到了另一个文件中可能有完全不同的作用。然而一旦文件被预编译，宏的处理过程便告完成。实际经验中很难在处理另一个头文件时，对先前预编译好的头文件修修补补。

Jeremy Siek 的 Concept Check Library（请参考 [BCCL]）相当系统化地透过「在高层 templates 添加部份哑码（dummy code）」来改善C++ 编译器的诊断讯息。它是 Boost 库的一部份（请参考 [Boost]）。

6.8 摘要

Templates 对于古典的「编译器＋链接器」模型提出新挑战。人们提出了 template 程序代码的多 种不同组织法：置入式模型（Inclusion Model）、显式实例化（Explicit Instantiation），以及分离式模型（Separation Model）。

通常情况下应该使用置入式模型（也就是把所有 template 程序代码写在头文件中）。

如果把 template 程序代码的声明语句和定义式放置于不同的头文件中，可以更加容易地在「置入 式模型」和「显式实例化模型」之间切换。

C++ *Standard* 为 templates 定义了一个「分离式模型」（使用关键词 `export`）。然而很多编译器并不支持这个特性。

对 template 程序代码侦错，可能相当困难。

Templates 具现体（instantiation）的名称可能很长。

为了从「预编译头文件」机制中获取最大好处，你应该确保各文件中的`#include` 指令的次序相同。

7

Template 基本术语

Basic Template Terminology

目前为止我们已经介绍了C++ templates 的基本概念。进一步讲述细节之前，让我们先关注一下 本书各概念所使用的术语。这非常有必要，因为在C++社群（甚至C++标准委员会）中，对概念和术语的运用不见得精准。

7.1 是 Class Template 还是 Template Class?

在C++ 中structs, classes 和 unions 统称为 class types。如果没有附加额外饰词一般字体的 "class" 表示的是关键词 `class`和 `struct`所代表的类型。特别请注意，"class type" 包括 unions，而 "class" 不包括 unions。

面对一个「本身是 template」的 class，人们的称谓相当混乱：

术语 class template 表示此 class 是个 template。这就是说它是「一整族 classes」的参数化描 述（parameterized description）。

术语 template class 有以下用法（其中第二和第三种用法的差异甚微，在本书其它部份并无重 要用途）：

作为 class template 的同义语。

表示由 templates 产生的 classes。

表示一个「以 template-id 为名称」的 classes。（译注：template-id 在 p.90 介绍）

鉴于 template class 的含义不够精确，本书避免使用这个术语，统一使用 class template。类似道理，我们使用术语 function template 和 member function template，避免使用 template function 和 template member function。

7.2 实例化（Instantiation）与特化（Specialization）

在 template 中，「以实际值（actual values）做为 template arguments，从而产生常规的（*regular*）class、function 或 member function」，这个过程称为「template 实例化」（template instantiation）。这些随之产生的物体（*entity*；包括 class, function 或 member function）通称为特化体（*specialization*）。（译注：我所阅读的众多泛型编程书籍中，很少将这些物体称为特化体，较 常称呼的是具现体, *instantiation*）

在 C++ 中，实例化并非产生特化体的惟一方法。另一种机制可使程序员明白地以某些特定的 template parameters 声明某些物体。我们在 3.3 节, p.27 中已经介绍过这种方法：透过 `template<>` 进行特化：

```
template <typename T1, typename T2>       // primary class template
class MyClass {
    ...
};

template<>                                // 显式（明确）特化
class MyClass<std::string, float> {
    ...
};
```

严格地说，这应该称作「显式特化」（explicit specialization），以别于因实例化或其它方式而产生的特化。

3.4 节我们提到，「仍带有 template parameters」的特化称为偏特化（partial specializations）：

```
template <typename T>                // 偏特化
class MyClass<T,T> {
    ...
};

template <typename T>                // 偏特化
class MyClass<bool,T> {
    ...
};
```

当我们讨论显式特化或偏特化时，常把「最泛化（*general*）的那个 template」称为 primary template（主模板/原始模板）。

7.3 声明（Declarations）vs. 定义（Definitions）

截至目前，本书只在为数不多的场合用上了「声明（式）」和「定义（式）」两个词。在C++ 中 这两个词都有各自的精确含义，本书的用法与之完全符合。

所谓「声明」是这样一种 C++ 构件（construct）：将一个名称引入（或重新引入）至某个 C++ 作用域（scope）中。这个「被引入体」总是包括该名称的某部份信息，不过一个合法声明并不 需要过多细节。例如：

```
class C;          // 声明：C是一个 class
void f(int p);    // 声明：f()是一个以 p为具名参数的函数
extern int v;     // 声明：v是一个变数
```

注意，虽然宏定义（macro definitions）和 goto 标签（goto labels）也有名称，但C++ 并不把它们当作一种声明。

两种情况下「声明」会变成「定义」：(1) 当它们的结构细节被写明白，(2) 当编译器必须

为变 数配置储存空间。对 class type 和 function 而言，这意味你必须为它们的定义提供「以大括号封 起来的程序代码」。对 variable（变数）而言，这意味一个初始化动作或一个不以关键词 extern 为前导的声明。以下补足上例中的每一个声明语句所欠缺的定义：

```
class C {};              // class C的定义（及声明）

void f(int p) {          // 函数 f()的定义（及声明）
    std::cout << p << std::endl;
}

extern int v = 1;        // 一个初始化动作（这使它成为变量 v的一个定义）

int w;                   // 全局变量而且不带 extern饰词（所以既是声明也是定义）
```

另外，如果 class templates 或 function templates 的声明（式）带有实作码，我们也称它们为定义（式）：

```
template <typename T>
void func (T);
```

以上是声明而不是定义。然而以下是一份定义：

```
template <typename T>
class S {};
```

7.4 单一定义规则（One-Definition Rule, ODR）

C++ 语言对于各种实体（entities）的再声明（redeclaration）做出了一些限制。这些限制总称为「单一定义规则」（One-Definition Rule, ODR）。这条规则的细节颇为复杂，并且适用于相当 大的语言跨度上。本书后续章节会阐述这条规则在各种适用情形下的各方面细节。附录 A 对此 规则亦有完整描述。目前只需记住 ODR 的数条基本守则就够了：

Non-inline 函数和成员函数，以及全局变量和 static 成员变量，在整个程序中只能定义一次。

Class 类型（包括 structs 和 unions）及 inline 函数，在每个编译单元中最多只能定义一次。如 果跨越不同的编译单元，则其定义必须完全相同。

所谓「编译单元」是指一个源码文件所涉及的所有文件；也就是说包括 #include 指令所含入 的所有文件。

在本书剩余章节中，所谓「可链接物」（linkable entity）可以是下列任何一个东西：non-inline 函数或 non-inline 成员函数、全局变量或 static 成员变量，以及所有由 template 程序代码产生的上述四种物体。

7.5 Template Arguments（模板自变量）vs. Template Parameters（模板参数）

比较下面两个 classes，第一个是 class template，第二个是与之类似的常规（non-template）class：

```
template <typename T, int N>
class ArrayInClass {
public:
    T array[N];
};

class DoubleArrayInClass {
public:
    double array[10];
};
```

如果我们分别把 T换成 double，把 N换成 10，那么两个 class 本质上等价。C++ 的这种替换系 透过下面这种写法完成：

```
ArrayInClass<double,10>
```

注意上一行的 template 名称之后紧跟着以角括号括起来的 template arguments（模板自变量）。

无论这些 template arguments 本身是否倚赖(受控于、取决于; *dependent on*)template parameters， 我们把「template name + 大括号括起的 template arguments」组合体称为 template-id。
这样一个 template-id 可以像其对应的 non-template 实体一样地被使用，例如：

```
int main()
{
    ArrayInClass<double,10> ad;        //这是一个 template-id
    ad.array[0] = 1.0;
}
```

区分 template parameters 和 template arguments 是非常重要的(你可以说把自变量传为参数)(*pass arguments to become parameters*）[19]，或者更准确地说：

Template parameters 是在 template 声明语句或定义式中位于关键词 template之后的那些名称
（前例中的 T和 N）。

Template arguments 是用以替换 template parameters 的东西（前例中的 double 和 10）。和 template parameters 不同的是，template arguments 并不限于「名称」（译注：意思是也可以 为数据）。

当我们使用 template-id 时，用以替换 template parameters 的那些 template arguments 是被显式(明确）指定的，但也存在隐式替换的情况，例如 template parameters 可被其默认值替换。

一个基本原则是：任何 template arguments 都必须是编译期可确定的数量（quantity）或实值
（value）。这个要求大大提高了 template 物体的执行期效率，本书后继章节还会详细说明这一 点。由于 template parameters 最终会被替换为编译期实值，因此你也可以用它们构成编

译期运算式（compile-time expressions）。`ArrayInClass` template 就是利用这一点来设定其成员 `array` 的大小 — 那必须是个常数表达式，而 template parameter `N` 符合这一要求。

我们可以把这个论据更推进一步：由于 template parameters 是编译期物体（compile-time entities），因此它们也可以被用以产生合法的 template arguments。下面是个例子：

```
template <typename T>
class Dozen {
public:
   ArrayInClass<T,12> contents;
};
```

注意，本例之中 `T` 既是 template parameter 也是 template argument。因此我们可以藉由这样的机制，以简单的 templates 搭建复杂的 templates。当然，这和「组合类型（types）和函数，以拼装出更复杂的物体」并没有本质上的区别。

第二篇 深入模板

Templates in epth

本书第一篇介绍了C++templates 的大部份语言概念足够解答日常编程可能遭遇的大部份问题。本书第二篇带有参考价值：当你进行复杂软件开发时，会碰到一些不寻常的问题，你可以在本 篇找到答案。你也可以跳过这一篇先阅读后续章节，那些章节（或书后索引）有可能涉及本篇 内容，那个时候你可以再回过头来阅读本篇。

讲解清晰、内容完备，并保持简明扼要，是我们的写作目标。基于此，书中例子往往短小并只针对特定问题。这样做可以保证我们不会偏离主题太远。

我们还讨论了C++语言的template 特性的未来可能变化。本篇主题包括：

Template 声明语句的根本问题

Templates 中的名称含义

C++template 的实例化机制（instantiation mechanisms）

Template argument deduction（模板自变量推导）规则

特化（specialization）和重载（overloading）

未来可能的发展

8

基础技术更深入

Fundamentals in Depth

本章将回顾并深入讲述第一篇介绍的一些根本知识：templates 的声明、template parameters 的局限（restrictions）、template arguments 的约束条件（constraints）…等等。

8.1 参数化声明（Parameterized Declarations）

C++ 目前支持两大templates 基本类型：class templates 和 function templates（这方面的未来可能变化请参考 13.6 节）。这个分类还包括 member templates。Templates 的声明与一般 class和 functions 的声明颇为类似，差别在于templates 声明语句有一个参数化子句（parameterization clause）：

```
template<...parameters here...>
```

也可能长像如下：

```
export template<...parameters here...>
```

（关键词 export的详细讨论出现于 6.3 节和 10.3.3 节）。

稍后我们会回头讨论template parameter 的声明。以下实例展示两种 templates，一种是在 class 之内(译注: 亦即 member templates)，一种是在 class 之外且 namespace scope 之内(译注: global scope 也被视为一种 namespace scope）：

```
template <typename T>
class List {                          // 一个 namespace scope class template
public:
    template <typename T2>            // 一个 member function template
    List (List<T2> const&);           // (这是个构造函数)
    ...
};

template <typename T>
 template <typename T2>
List<T>::List (List<T2> const& b)  // 一个定义于 class 外的 member function template
{
    ...
}

template <typename T>
```

```
int length (List<T> const&);    // 一个 namespace scope function template

class Collection {
    template <typename T>       // 一个定义于 class 内的 member class template
    class Node {
        ...
    };

    template <typename T>       // 又一个 member class template，无定义
    class Handle;

    template <typename T>       // 一个定义于 class 内的 member function template
    T* alloc() {                // 隐寓为 inline
       ...
    }
    ...
};

template <typename T>           // 一个定义于 class 外的 member class template
class Collection::Handle {
    ...
};
```

注意，定义于 class 外的 member templates 可有多重 `template<...>` 参数化子句，其中一个代
表 template 本身，其余各个子句代表外围的每一层 class template。这些子句必须从最外层 class templates 开始写起。

你也可以撰写 union templates（被视为 class template 的一种）：

```
template <typename T>
union AllocChunk
    { T object;
    unsigned char bytes[sizeof(T)];
};
```

Function template 可以有预设的调用自变量（default call arguments），和一般 function 一样：

```
template <typename T>
void report_top (Stack<T> const&, int number = 10);

template <typename T>
void fill (Array<T>*, T const& = T()); // 若 T为内建类型，T()为 0 或 false)
```

后一个声明语句示范如何让一个 default call arguments 取决于某个 template parameter。当 `fill()`被 调用时，如果调用者提供了第二自变量值，预设自变量便不会被实例化。这可确

保如果预设自变量无 法被某个特定类型 T实例化，不会引发编译错误。举个例子：

```
class Value {
public:
    Value(int);         // 无 default 构造函数
};

void init (Array<Value>* array)
{
    Value zero(0);

    fill(array, zero); // OK：前述声明中的 T()未用上。
    fill(array);        // ERROR：前述声明中的 T()被用上，然而  T=Value 却不合法。
      // 译注：让我进一步解释：前述声明中的 T被推导为 Value; fill()第二自变量预设为  T(),
      //      也就成为 Value()。然而根据  class Value的定义，并不存在 Value()。所以失败。
}
```

除了两种 template 基本类型，另有三种声明也可以被参数化，它们都使用类似写法。三者均相当于 class template 的成员定义：

1. class templates 的成员函数定义
2. class templates 的 nested class members（嵌套类别成员）定义
3. class templates 的 static 成员变量定义

虽然它们也可以被参数化，但它们并不是第一级（first-class）templates。它们的参数完全由它 们所隶属的 template 决定。下面是个例子：

```
template <int I>
class CupBoard {
    void open();                //译注：隶属于  CupBoard class template
    class Shelf;                //译注：隶属于  CupBoard class template
    static double total_weight; //译注：隶属于  CupBoard class template
    ...
};

template <int I>
void CupBoard<I>::open()
{
    ...
}

template <int I>
class CupBoard<I>::Shelf {
    ...
};
```

```
template <int I>
double CupBoard<I>::total_weight = 0.0;
```

虽说这种参数化定义通常也被称为 templates，但很多场合中这种称谓并不合适。

8.1.1 虚拟成员函数（Virtual Member Functions）

Member function templates 不能声明成 virtual。这个限制有其原因。通常虚拟函数调用机制使 用一个大小固定不变动的表格，其中每一笔条目（entry）记录一个虚拟函数入口（entry point）。然而直到整个程序编译完成编译器才能知道有多少个 member function templates 需要被实例化。因此，如果要支持 virtual member function templates，C++ 编译器和链接器需要一个全新机制。

然而一般的 class template members 可以是虚拟函数，因为当 class 被实例化时，那些 members 的 数量是固定的：

```
template <typename
T> class Dynamic
{ public:
    // 译注：下面是 class template 的 member function，可为 virtual。
    virtual ~Dynamic();     // OK：每个 Dynamic<T> 具现体都有一个析构函数


    // 译注：下面是 member (function) template，不可为 virtual。
    template <typename T2>
    virtual void copy (T2 const&);
        // ERROR：编译器此时并不知道在一个 Dynamic<T> 具现体中要产生多少个 copy()具现体。
        // 译注：因此编译器无法备妥虚拟表格（virtual table, vtbl）。
};
```

8.1.2 Template 的链接（Linkage）

每个 template 在其作用域（scope）内必须有一个独一无二的名称，除非是被重载的（overloaded）function templates（见 12 章）。特别请注意，class template 不能和其它不同种类的物体（entities）共享同一个名称，这点与一般（non-template）class 不同。

```
int C;

class C;    // OK：class 名称和 nonclass 名称处于不同的空间(space)内


int X;

template <typename T>
class X;    // ERROR：名称与上述变量 X冲突
```

```
struct S;

template <typename T>
class S;    // ERROR：名称与上述 struct S冲突
```

Template 名称需要链接（linkage），但不能够使用 C 链接方式（C linkage）。非标准的链接方 式可能会有「与实作相依」（implementation-dependent）的意义（但我们无法知道是否某个编译 系统支持非标准的 templates 链接方式）：

```
extern "C++" template <typename T>
void normal();
    // 这是预设方式。其链接规格（linkage specification，亦即 extern "C++"）可省略不写。


extern "C" template <typename T>
void invalid();
    // 这是无效（不合法）的：templates 不能使用 C链接方式。


extern "Xroma" template <typename T>
void xroma_link();
    // 这是非标准的，但可能某些编译器会在某一天支持兼容于 Xroma语言的链接方式。
```

Templates 通常使用外部链接（external linkage）。惟一例外是 static namespace scope function templates：

```
template <typename T>
void external();
          // 指涉（refer to）另一文件中同名且同作用域（scope）的相同物体（entity）。

template <typename T>
static void internal();  // 与另一文件中的同名 template 无关
```

注意，函数内不能再声明 templates。

8.1.3 Primary Templates（主模板/原始模板）

正规、标准的 template 声明语句声明的是 primary templates。这一类声明语句在 template 名称之后并 不添加由角括号括起的 template argument list：

```
template<typename T> class Box;                // OK：primary template
```

```
template<typename T> class Box<T>;            // ERROR

template<typename T> void translate(T*);      // OK：primary template

template<typename T> void translate<T>(T*);   // ERROR
```

一旦我们声明一个偏特化 templates，就是产生一个 non-primary class templates，第 12 章将讨论 这种情况。Function templates 必须是 primary templates。（C++语言在这一方面可能将有所变动， 请参考 13.7 节）

8.2 Template Parameter（模板参数）

Template parameters 有三种类型：

1. Type parameters（类型参数）；这种参数最常用。
2. Nontype parameters（非类型参数）
3. Template template parameters（双重模板参数）

所谓 template parameters 是在 template 声明语句的参数化子句（parameterization clause）中声明的 变量。Template parameters 不一定得具名：

```
template <typename, int>   // 译注：两个 template parameters 都没有名称
class X;
```

但是当 template 程序代码中需要用到某个 template parameter 时，后者就必须具名。请注意，后宣 告的 template parameters 可以用到先声明的 template parameters 的名称，反之不然：

```
template <typename T,              // 第一个参数 T，被用于第二和第三参数的声明之中。
          T* Root,
          template<T*> class Buf>    // 译注：Buf就是个 template template parameter
class Structure;
```

8.2.1 Type Parameters（型别参数）

Type parameters 可以采用关键词 typename 或关键词 class 导入，两者完全等价[21]。其声明形 式是：关键词 typename 或 class 后面跟一个简单标识符（做为参数名称），该符号后面可 跟一个逗号以便区隔下一参数，也可以使用一个闭锁角括号（>）结束参数子句，或跟随一个等 号（=）标示出 default template argument（预设模板自变量）。

在 template 声明语句中，type parameter 的作用非常类似 typedef 的名称，例如，你不能使用如 class T这样的完整名称，即使 T确实表示一个 class type：

```
template <typename Allocator>
class List {
    class Allocator* allocator;      // ERROR
    friend class Allocator;          // ERROR
```

```
    ...
};
```

C++ 将来有可能接受这种型式的 friend 声明。

8.2.2 Nontype Parameters（非类型参数）

Nontype template parameter 是指那些可在编译期或链接期确定其值的常数。此种参数的类型必须是以下三者之一：

整数（integral）或列举（enumeration）类型

pointer 类型；包括指向常规 objects、指向 functions 和指向 members。

reference 类型；包括指向（指涉、代表）objects 和指向 functions。 目前不允许使用其它类型（将来有可能允许使用浮点数类型，见 13.4 节, p.210）。

也许你会感到惊讶，nontype parameter 的声明在某种情况下也以关键词 typename为前缀词：

```
template<typename T,                              // 一个 type parameter
         typename T::Allocator* Allocator>        // 一个 nontype parameter
class List;
```

这两种情况很容易区分：头一种情况中的 typename后面总是紧跟一个简单标识符，第二种情况中的 typename后面跟一个带饰词的名称（qualified name），亦即一个包含双冒号（::）的名称。本书 5.1 节和 9.3.2 节告诉你何时需要在 nontype parameter 之前方使用关键词 typename。

Nontype parameters 也可以是 function 类型或 array 类型，但它们都会退化*decay*为对应的 pointer 类型：

```
template<int buf[5]> class Lexer;       // buf实际被当作 int*
template<int* buf> class Lexer;         // OK：这是一个再声明（redeclaration）
```

Nontype template parameters 的声明方式非常类似变量声明，但你不能加上诸如 static、mutable 之类的修饰。你可以加上 const或 volatile，但如果这些饰词出现在参数类型的最外层，编译器会忽略它们：

```
template<int const length> class Buffer;      // const被忽略
template<int length> class Buffer;            // 与上一行声明语句等价
```

最后一点，nontype parameters 总是右值（rvalues）：它们不能被取址，也不能被赋值。

8.2.3 Template Template Parameters（双重模板参数）

Template template parameters 是一种「class templates 占位符号（placeholder）」，其声明方式和 class templates 类似，只是不能使用关键词 struct和 union：

```
template <template<typename X> class C>   // OK
void f(C<int>* p);

template <template<typename X> struct C> // ERROR：不能使用关键词 struct
void f(C<int>* p);

template <template<typename X> union C>   // ERROR：不能使用关键词 union
void f(C<int>* p);
```

在它们的作用域内，你可以像使用 class templates 那样地使用 template template parameters。

Template template parameters 的参数也可以有 default template arguments（预设模板自变量）。如果 客户端没有为相应的参数指定自变量，编译器就会使用这些预设自变量：

```
template <template<typename T,
                   typename A = MyAllocator> class Container>
class Adaptation {
    Container<int> storage; // 暗自（隐寓）等价于 Container<int, MyAllocator>
    ...
};
```

在template template parameters 中，template parameter 的名称只能被用于 template template parameter 的其它参数声明中。这一点可以下面例子来解释：

```
template <template<typename T, T*> class Buf>
class Lexer {
    static char storage[5];
    Buf<char, &Lexer<Buf>::storage[0]> buf;
    ...
};

template <template<typename T> class List>
class Node {
    static T* storage; // ERROR：这里不能使用 template template parameters 的参数 T
    ...
};
```

通常 template template parameter 中 template parameters 名称并不会在其它地方被用到，因此，未被用到的 template parameters 可以不具名。例如上页的 Adaptation template 可被声明为：

```
// 译注：原先（上页最下）的两个参数名称 T和 A都被省略了。
template <template <typename,
                    typename = MyAllocator> class Container>
class Adaptation
```

```
{
    Container<int> storage; // 暗自（隐隐）等价于 Container<int, MyAllocator>
    ...
};
```

8.2.4 Default Template Arguments（预设的模板引数）

目前，只有 class template 的声明语句可以存在（拥有）default template arguments（将来C++ 语言 可能会对此做出修改，见 13.3 节）。无论何种 template parameters 都可以有其预设自变量，当然它必须匹配（吻合）对应参数。很明显，预设自变量不能相依于其自身参数，但可以相依于 在它之前声明的参数：

```
template <typename T, typename Allocator = allocator<T> >
class List;
```

和函数的预设自变量一样，某个参数（译注：不论是 call parameters 或 template parameters）带有 预设自变量的条件是：其后续所有参数也都有预设自变量。

后续参数的预设自变量通常写在同一个 template 声明语句中，但也可以写在该 template 更早的某个
声明语句中。例如：

```
template <typename T1, typename T2, typename T3,
          typename T4 = char, typename T5 = char>
class Quintuple; // OK

template <typename T1, typename T2, typename T3 = char,
          typename T4, typename T5>
class Quintuple; // OK：T4和 T5先前已有默认值


template <typename T1 = char, typename T2, typename T3,
         typename T4, typename T5>
class Quintuple; // 错误：T1不能拥有默认值，因为 T2没有默认值
```

此外我们也不能重复指定 default template arguments：

```
template<typename T = void>
class Value;

template<typename T = void> // ERROR：预设自变量被重复定义了
class Value;
```

8.3 Template Arguments（模板引数）

所谓 templates arguments 是当编译器实例化一个 template 时用来替换 template parameters 的值。编译器以数种不同的机制来决定以何值替换 template parameters：

explicit template arguments（明白指定之模板自变量）：可在 template 名称之后跟着一个或多个 明确的 template arguments，并以角括号括起。这个完整名称被称为 template-id。

injected class name（内植式类别名称）：在带有参数 P1,P2…的 class template X作用域中，

template X的名称与 template-id X<P1, P2,...> 等价。细节请见 9.2.3 节。

default template arguments：如果存在可用的 default template arguments，我们便可在 class templates 具现体中省略不写 explicit template arguments。然而即使每一个参数都有默认值， 你也必须把开闭两个角括号写上（即使括号内什么都没有）。

argument deduction（自变量推导）：编译器可根据 function call arguments 推导出 function template arguments。具体推导细节将在第 11 章讲述。其它某些情况下，编译器也可能动用推导机制。 如果所有 template arguments 都可以推导得出，你就无需在 function template 名称后面加写角 括号。

8.3.1 Function Template Arguments（函式模板自变量）

你既可以明白指定 function template 的 template arguments，也可以让它们被编译器推导出来。例如：

```
// details/max.cpp
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

int main()
{
    max<double>(1.0, -3.0);      // explicit template arguments
    max(1.0, -3.0);              // template arguments 被隐式推导为 double
    max<int>(1.0, 3.0);          // 明确指定<int> 以抑制推导，从而令自变量类型为
    int
}
```

某些 template arguments 无法被推导获得(见 11 章)你最好把这一类参数放在 template parameter list 的最前面，这样客户端（调用者）只需明白指定编译器无法推导的那些自变量即可，其余自变量 仍可被自动推导而得。例如：

```
// details/implicit.cpp
template <typename DstT, typename SrcT>
inline DstT implicit_cast (SrcT const& x) // SrcT可被推导，但 DstT无法推导。
{                                 // 译注：因为 DstT不出现在自变量列，无法进行自变量推导。
    return x;
```

```
}

int main()
{
    double value = implicit_cast<double>(-1);
}
```

如果不这么写，改而把template parameters的顺序换一下（写成template<typename SrcT,typename DstT>），我们就不得不在调用implicit_cast时把两个 template arguments 都明确写出来。

由于 function templates 可被重载（overloaded），因此即使明确写出一个 function template 的所有自变量，可能也不足以使编译器确定该调用哪一份具体函数，因为某些情况下符合要求的函数可 能有「一群」。下面的例子说明这种情况：

```
template <typename Func, typename T>
void apply (Func func_ptr, T x)
{
    func_ptr(x);
}

template <typename T> void single(T);

template <typename T> void multi(T);
template <typename T> void multi(T*);

int main()
{
    apply(&single<int>, 3); // OK
    apply(&multi<int>, 7);  // ERROR: 符合 multi<int>形式的函数不只一个
}
```

在这个例子中，对 apply()的第一个调用动作是合法的，因为表达式 &single<int> 没有任何歧义，因此 template parameter Func的值可被编译器轻易推导出来。然而在第二个调用动作中，&multi<int> 可以是两个不同类型中的任何一个，这种情况下编译器无法推导出 Func。

不仅如此，明确指定 template arguments 还可能导致建构出不合法的 C++ 类型。考虑下例的重 载函数，其中 RT1和 RT2是未定类型：

```
template<typename T> RT1 test(typename T::X const*);   // (1)
template<typename T> RT2 test(...);                    // (2)
```

算式 test<int>对于第一个 function template 而言是无意义的，因为 int类型并没有 member type X。然而第二个 function template 无此问题，因此算式 &test<int> 可确定出惟一一个函数地址。 于是尽管第一个 template 以 int替换失败，却没有造成&test<int> 随之不合法。

正是「替换失败并非错误」（substitution-failure-is-not-an-error; SFINAE）这一原则，使得 function templates 的重载实际可行。而且我们可以借助这个原则，创造出很多极出色的编译期技术
（compile-time techniques）。例如，假设类型 RT1和 RT2分别定义为：

```
typedef char RT1;
typedef struct { char a[2]; } RT2;
```

我们可以在编译期（也就是经由一个常数算式, constant-expression）判断某给定类型T 是否有 member type X：

```
#define type_has_member_type_X(T) \
        (sizeof(test<T>(0)) == 1)
```

为理解上述宏中的表达式，我们最好由外而内地分析它。首先，如果上页的第一个 test template 被编译器选用，sizeof算式值等于 1（回返类型为 char，大小为 1）。如果第二个 template 被选用，则 sizeof 算式值至少为 2（回返类型是个内含两个 char 的 array）。换言之此宏的功 用是判别编译器实例化 test<T>(0)时选中第一个 function template 或第二个。很明显，如果给 定的类型 T不存在 member type X，第一个 function template 不会被选用（译注：这时 sizeof的 值不为 1）。然而如果给定的类型 T存在 member type X，根据重载解析规则（见附录 B），第一个 function template 会被优先选用（译注：这时 sizeof 结果为 1）；这是因为重载解析规则 会优先考虑「把 0 值转换为一个 null 指针」，而不考虑「把 0 值当作省略号（ellipsis）参数」（「省略号参数」是重载解析规则中最后才考虑是否匹配的参数形式）。第 15 章大量用到了类 似技法。

上述的 SFINAE 原则只是协助你避免创造出非法类型，并不能防止非法算式（invalid expressions）被编译器求值（*evaluated*）。因此下面例子是不合法的：

```
template<int I> void f(int (&)[24/(4-I)]);
template<int I> void f(int (&)[24/(4+I)]);

int main()
{
    &f<4>;   // ERROR：除数为 0(不适用 SFINAE原则)
}
//译注：VC7.1在此例中表现奇怪，它对编译期表达式 24/(4-4)的求值结果居然是 1，这大概是某种
//古怪的错误防护机制作祟(除数为 0时认定商为 1)。VC6/ICL 7.1/g++ 3.2则给出合理的错误信息。
```

例中的 &f<4>将导致编译错误 — 即使第二个 template 被选用时不会出现「除数为 0」的情况。 这一类错误并不出现在「编译器把算式值绑定（*bind*）至 template parameter」的时候，而是出 现在「算式求值过程」中。下面例子是合法的：

```
template<int N> int g() { return N;   }
template<int* P> int g() { return *P; }

int main()
```

```
{
    return g<1>(); // 1无法适用于 int* 参数，然而 SFINAE原则在这里起了作用。
}
//译注：VC7.1和 VC6均无法编译此例。VC7.1总是把 template argument 1匹配至错误的
//重载函数上，VC6则是无法区分两个 g()，认为两者相同。g++ 3.2/ICL 7.1无此问题。
```

关于 SFINAE 原则的进一步应用，请看 15.2.2 节, p.266 和 19.3 节, p.353。

8.3.2 Type Arguments（型别引数）

Template type arguments 是针对 template type parameters 而指定的「值」。我们惯用的大多数 types 都可以作为 template arguments 使用，但有两个例外：

1. local classes 和 local enum types（亦即声明于函数定义内部的 classes 和 enums）不能做为 template type arguments 使用。
2. 如果某个 type 涉及无名的(unnamed)class types 或无名的(unnamed)enum types，这样的 type 不能做为 template type arguments 使用；但如果运用 typedef 使其具名，便可用做 template type arguments。

下面例子展示了上述两种例外情况：

```
template <typename T> class List {
    ...
};

typedef struct {
    double x, y, z;
} Point;          // typedef使某个 unnamed type 有了名称


typedef enum { red, green, blue } *ColorPtr;

int main()
{
    struct Association          //译注：这是一个定义于函数内部的所谓 local type
    {
        int* p;
        int* q;
    };
    List<Association*> error1;  // ERROR：template argument 不能是个 local type
    List<ColorPtr> error2;      // ERROR：template argument 不能是个 unnamed type
    List<Point> ok;             // OK：原本无名的 type 因 typedef而有了名称
}
```

尽管通常各式各样的 types 可被当做 template arguments 使用，但以那些 types 替换 template
parameters 之后，其结果必须是个合法的C++ 构件（constructs）：

```
template <typename T>
void clear (T p)
{
    *p = 0;     // unary operator* 必须可施行于 T身上
}

int main()
{
    int a;
    clear(a);   // ERROR：int并不支持 unary operator*
}
```

8.3.3 Nontype Arguments（非类型引数）

Nontype template arguments 是针对 nontype template parameters 而指定的「值」，它必须符合下列条件之一：

是另一个具有正确类型的 nontype template parameter。

是一个编译期整数型（integer）或列举型（enum）常数，但必须与对应的参数类型匹配，或 者可以被隐式转型为该类型（例如 int参数可以使用 char值）。

以内建一元取址运算符（unary addressof operator&）为前导的外部变量或函数。面对函数或 array 变数，可省略不写 '&'。这一类 template arguments 匹配 *pointer* type nontype parameter。

如上所述但无前导 '&'。匹配的是 *reference* type nontype parameter。

一个 pointer-to-member 常数亦即形如 &C::m的表达式其中 C 是个 class type，m 是个 non-static 成员（函数或变数）。它只匹配 *pointer-to-member* type nontype parameters。

当以一个template argument 匹配一个pointer type 或 reference type 的template parameter 时(1) 使 用者自订转换（包括单自变量构造函数及转型函数）以及(2) derived-to-base 转换都不被编译器考虑，即使它们在其它情境中是合法有效的隐式转换。至于为自变量添加const 饰词或 volatile 饰词是可以的。

下面例子中的 nontype template arguments 都合法：

```
template <typename T, T nontype_param>
class C;

C<int, 33>* c1;     // 整数型（integer type）


int a;
```

```
C<int*, &a>* c2;    // 外部变量的地址


void f();
void f(int);
C<void (*)(int), f>* c3;
                    // 一个函数名称。重载解析规则在此选用 f(int)。& 会被隐寓加入。


class X {
public:
    int n;
    static bool b;
};

C<bool&, X::b>* c4; // static class members 都可接受


C<int X::*, &X::n>* c5;
                    // pointer-to-member 常数亦可接受


template<typename T>
void templ_func();

C<void (), &templ_func<double> >* c6;
                    // function template 具现体也是一种函数
```

译注：VC7.1 无法通过此例中的 c6声明。它认为`&templ_func<double>`在编译期不可求值。g++ 3.2 认为 `void()()`不是编译期常数。只有 ICL7.1 能顺利编译此例。把 `void()`改为 `void(*)()` 后，VC7.1 编译通过，VC6 失败依然。c6的声明语法应是正确的，见C++*standard* 14.1/8。

template arguments 的一个一般性约束条件（general constraint）是：必须能够在编译期或链接期求值。「只在执行期才能求值」的表达式（例如某区域变量的地址）不能作为 nontype template arguments 使用。

即使如此，可能令你感到惊奇的是，至少你目前还不能使用以下各种常数值：

null pointer 常数

浮点数（floating-point numbers）

字符串字面常数（string literals）

不能以字符串字面常数作为 nontype template arguments 的一个技术难题在于：两个内容完全相同 的字符串字面常数可能存在两个不同的地址上。一种稍显笨拙的解法是引入一个字符串 array：

```
template <char const* str>
```

```
class Message;
extern char const hello[] = "Hello World!";
Message<hello>* hello_msg;
```

注意这里必须使用关键词 extern，因为 const array 采用内部链接（internal linkage）。

4.3 节有这个问题的另一个例子。13.4 节讨论了C++ 语言在这个问题上的未来可能变化。

这里还有一些比较不那么令人惊讶的非法实例：

```
template<typename T, T nontype_param>
class C;

class Base {
public:
    int i;
} base;

class Derived : public Base {
} derived_obj;

C<Base*, &derived_obj>* err1;   // ERROR：「derived-to-base 转换」不被考虑

C<int&, base.i>* err2;          // ERROR：成员变数不被考虑


int a[10];
C<int*, &a[0]>* err3;           // ERROR：不能使用 array内某个元素的地址
```

8.3.4 Template Template Arguments（双重模板自变量）

Template template argument 必须是这样一个class template：其参数完全匹配待替换之template template parameter 的参数。Template template argument 的default template argument（预设模板引 数值）会被编译器忽略，除非对应的template template parameter 有预设自变量。下面示范非法情 况：

```
#include <list>
    // 声明：
    // namespace std {
    //     template <typename T,
    //               typename Allocator = allocator<T> >
    //     class list;
    // }


template<typename
```

```
        T1,
        typename
        T2,
        template<typename> class Container>
                        // Container要求只带一个参数
class Relation {
public:
    ...
privat
e:
    Container<T1> dom1;
    Container<T2> dom2;
};

int main()
{
    Relation<int, double, std::list> rel;
        // ERROR：std::list拥有不止一个 template parameters
    ...
}
```

问题出在标准库的 std::list template 拥有不止一个参数。第二参数（是个 allocator, 配置器）有默认值，但编译器把 std::list匹配至 Container时，该默认值被忽略了。

有些时候这种问题可以绕道解决：只需要为template template parameter加上一个带有默认值的 参数即可。对上述例子来说，我们可以把 template Relation重写为：

```
#include <memory>
template<typename T1,
         typename
         T2,
         template<typename T,
                  typename = std::allocator<T> > class Container>
           // Container现在可接受标准库的 container templates 了
class Relation {
public:
    ...
privat
e:
    Container<T1> dom1;
    Container<T2> dom2;
};
```

很明显，这么做并不完全令人满意，但毕竟使我们得以运用标准库的容器（container templates）。13.5 节讨论了C++ 语言对这个问题的未来可能变化。

虽然从语法上说，你只能以关键词 class来声明一个 template template parameter，但这并不意味 template template argument 必须是 class type。事实上以关键词 struct和 union声明的 templates 也都可以当作 template template parameters 的合法自变量。这和我们先前讲过的「任意类型都可作 为由关键词 class声明之 template type parameters 的合法自变量」类似。

8.3.5 等价（Equivalence）

当两组 template arguments 的元素一一对等时，我们称这两组自变量等价。对于 type arguments， typedef的名称并不影响对比过程最终被比较的是 typedef所指代的 type。对于整数型 nontype arguments，比较的是自变量值，与自变量表达方式无关。下面例子阐释了这个概念：

```
template <typename T, int I>
class Mix;
typedef int Int;
Mix<int, 3*3>*
p1;
Mix<Int, 4+5>* p2; // p2和 p1具有相同类型
```

一个由 function template 产生的函数，和一个常规函数，无论如何不会被编译器视为等价，即使它们的类型和名称完全相同。这对 class members 造成两个重要结果：

1. 由 member function template 产生的函数不会覆盖虚拟（virtual）函数。
2. 由 constructor template 产生的构造函数不会被当做 *default copy* 构造函数。（同样道理由 assignment template 产生的 *assignment* 运算符不会被当做一个 *copy-assignment* 运算符。这个问题较少 出现，因为 *assignment* 运算符不像 *copy* 构造函数那样会被隐式调用。）

8.4 Friends

Friend 声明语句的基本概念很简单：指定某些 classes 或 functions，让它们可以对 friend 声明语句所 在的 class 进行特权存取。但是下面两个事实使这个概念复杂化了：

1. friend 声明语句可能是某一物体（entity）的惟一声明（译注：意思是 friend 仅在 class 内声明， 别无其它兄弟）。
2. friend 函数声明可以就是其定义。

friend class 声明语句不能成为一个定义式，这就大大降低了问题的发生。涉及 templates 时，惟一
需要考虑的新增情况是：你可以把某个 class template 的特定具现体（实体）声明为 friend：

```
template <typename T>
class Node;
```

```
template <typename T>
class Tree {
    friend class Node<T>;
    ...
};
```

注意，在 class template 的某一实体（如上述 Node<T>）成为其它 class 或 class template（如上述 Tree）的 friend 之前，class template Node必须已被声明并可见。但对常规 class 来说没有这个限 制。

```
template <typename T>
class Tree {
    friend class Factory;   // OK，即使这是 Factory的首次声明
    friend class Node<T>;   // ERROR(如果此前并未声明 Node)
};
```

9.2.2 节对此有更多讨论。

8.4.1 Friend Functions

Function template 具现体(实体)可以成为别人的一个 friend function，前提是该 function template 名称之后必须紧跟着以角括号括起的自变量列—如果编译器可推导出所有自变量，自变量列可以为空：

```
template <typename T1, typename T2>
void combine(T1, T2);

class Mixer {
    friend void combine<>(int&, int&);
                    // OK：T1 = int&, T2 = int&
    friend void combine<int, int>(int, int);
                    // OK：T1 = int, T2 = int
    friend void combine<char>(char, int);
                    // OK：T1 = char T2 = int
    friend void combine<char>(char&, int);
                    // ERROR：与 combine() template 不匹配
    friend void combine<>(long, long) { ... }
                    // ERROR：不能在此处定义函数
};
```

注意，我们无法定义一个 template 实体（*instance*），最多只能定义一个特化体（*specialization*）。因此一个「令某实体获得名称」的 friend 声明语句，不能够是个定义式。

译注：作者的意思是，template 实体（*instances*）只能由编译器产生，不能由程序员定义；程序员所定义的，称为特化体（*specializations*）。见 p.88。

如果 friend 名称之后没有跟着角括号，有两种可能：

1. 如果这不是个资格修饰名称（亦即不含::），就绝不会引用某个 template 实体。如果编译器 无法在 friend 声明处匹配一个 non-template function，则这个 friend 声明便被当作这个函数的 首次声明。这个声明语句也可以是个定义式。
2. 如果这是一个资格修饰名称（亦即含有::），它就必定引用一个先前已定义的 function 或 function template。编译器会优先匹配常规non-template函数，然后才匹配 function templates。这个 friend 声明语句不能是个定义式。

以下例子有助于理解各种可能情况：

```
void multiply (void*);      // 常规函数（ordinary function）


template <typename T>
void multiply(T);           // function template

class Comrades {
    friend void multiply(int) {}
                        // 以上定义了一个新函数 ::multiply(int)

    friend void ::multiply(void*);
                        // 以上引用先前定义的常规函数，而非 multiply<void*> 实体


    friend void ::multiply(int);
                        // 以上引用 template 的一个实体


    friend void ::multiply<double*>(double*);
                        // 资格修饰名称也可以带角括号，但此时编译器必须见到该 template

    friend void ::error() {}
                        // ERROR：带资格修饰的 friend，不能是个定义
};
```

先前数个例子中，我们将 friend function 声明在常规 class 内。如果把 friend 声明于 class templates 中，先前所说的规则也全部适用，而且 template parameter 可参与到 friend function 之内：

```
template <typename T>
class Node {
    Node<T>* allocate();
```

```
    ...
};

template <typename T>
class List {
    friend Node<T>* Node<T>::allocate();
    ...
};
```

然而把 friend function 定义于 class template 中可能会引发一个有趣的错误因为任何只在 template 中被声明的 object，都是直到 template 被实例化后才能成为具体实体（concrete entity）。考虑下 面例子：

```
template <typename T>
class Creator {
    friend void appear()
                // 定义一个新函数::appear()，但是只有当 Creator被实例化它才存在
        ...
    }
};

Creator<void> miracle; // ::appear()此时被生成
Creator<double> oops;  // ERROR：试图再次生成::appear()
```

在这个例子中，两个不同的具现体产生出两个完全相同的定义，这直接违反了ODR原则（见附 录A）。

因此，我们必须确保 class template 的 template parameters 出现在「定义于该 template 之内的所有 friend function」的类型之中（除非我们想要阻止这个 class template 在一个文件中被多次实例化， 但没有什么人会这样做）。现在让我们对先前的例子进行一些改动：

```
template <typename T>
class Creator {
    friend void feed(Creator<T>*){   // 不同的 T会产生不同的::feed函数定义
        ...
    }
};
Creator<void> one;          // 产生::feed(Creator<void>*)
Creator<double> two;        // 产生::feed(Creator<double>*)
```

在这个例子中 每一个 Creator具现体会产生一个不同的函数。注意 虽然这些函数是在 template 实例化过程中产生的，但它们仍是常规函数，不是某个 templates 的实体。

另外请注意，由于这些函数被定义于 class 定义式内，因此它们暗自成为 inline。而且如果

你在 两个不同的编译单元内产生同一个函数，编译器不会认为错误。9.2.2 节和 11.7 节对此问题有更多论述。

8.4.2 Friend Templates

通常，当我们定义一个 friend 而它是个 function 或是个 class template 时，我们可以明确指定以 哪一个物体entity做为 friend。但有时候把一个 template 的所有实体都指定为某个 class 的 friend 也相当有用。这是经由一种所谓的 friend template 机制实现的。例如：

```
class Manager {
    template<typename T>
        friend class Task;
    template<typename T>
        friend void Schedule<T>::dispatch(Task<T>*);
    template<typename T>
        friend int ticket() {
            return ++Manager::counter;
        }
    static int counter;
};
```

和常规的 friend 声明语句一样，只有当 friend template 产生一个无修饰函数名（unqualified function name），而且该名称之后不紧跟着角括号，这个 friend template 才可以是一个定义式。

Friend template 只能声明 primary templates 及 primary templates 的成员。任何与该 primary template 相关的偏特化体（partial specializations）和明确特化体（explicit specializations）都将自动被编 译器视为 friends。

8.5 后记

C++ templates 的一般概念和语法，自从出现于 1980 年代晚期就基本固定了。Class templates 和 function templates、type parameters 和 nontype parameters 等概念都是 template 最初机制的一部份。

然而由于C++ 标准库的需求所带来的策动，template 原始设计发生了一些重大增建。Member templates 是这些重大增建中最基础的部份。有趣的是，只有 member function templates 被正式票 选解析为C++*Standard* 的一部份，但由于一个文字上的疏漏，member class templates 也成为 C++*Standard* 的一部份。

译注：C++ 标准委员会只投票通过加入 member function templates，C++ 标准文件也随之做出增
补。然而这部份文字写得并不严谨，使得人们误以为 member class templates 也被加入 C++*Standard*。之后不久，某些标准库也在实作中使用了 member class templates。这使得这个误 会更凿实，人们再也无法把它从语言标准规格中剔除。

Friend templates、default template arguments 和 template template parameters 是晚近才加入C++ 语言阵营的。「声明 template template parameters」的能力有时称为更高级泛型（higher-order genericity），它们最初引入 C++ 是为了支持某个配置器模型（allocator model），但之后这项 任务就被另一个并不依赖 template template parameters 特性的模型所取代。由于缺乏一个完备规 格，这一特性在标准化工作后期几乎要被从标准规格中去掉。最终是标准委员会的多数成员投 票通过，留下了这些特性，并完备其规格。

9

Templates 内的名称

Names in Templates

名称是绝大多数编程语言最基础的概念。程序员可以通过名称来引用先前建构的物体（entities）。当 C++ 编译器碰到一个名称，它必须搜寻这个名称引用的是哪个物体。从编译器实作者的角度来看，C++ 名称体系非常复杂。试着考虑述句 `x*y;`，如果 `x`和 `y`是变量名称，则这个述句表示一个乘法；但如果 `x`是个类型名称，则这个述句声明一个指针 `y`，指向「类型为 `x`的物体」。

这个小例子说明 C++（以及 C）是一种所谓「前后脉络敏感」（context-sensitive）的语言：如果没有前后脉络，就不可能知道某一构件（construct）的具体含义。这和 templates 有何关系呢？ 唔，templates 是一种「涉及前后脉络更多更广」的构件，这些前后脉络包括：(1) templates 出现的位置和方式；(2) templates 被实例化的位置和方式；(3)「用以实例化 templates」的那些 template arguments 的前后脉络。因此，如果我说在C++ 中必须小心处理名称，你应该不会感到惊讶。

9.1 名称分类学（Name Taxonomy）

C++ 以各式各样的方法来对名称分类。为帮助你理解如此多的术语，这里提供表 9.1 和表 9.2，描述了这些分类。幸运的是，如果你能够熟悉两个最主要的命名概念，就能够洞察大多数 C++template 问题：

1. 当一个名称被称为「受饰名称」（qualified name）时，其含义是：以 *scope resolution* 运算子（`::`）或 *member access* 运算符（`.` 或 `->`）指明该名称所属作用域。例如 `this->count` 是一个受饰名称，`count`不是（即使两者等价）。
2. 当一个名称被称为「受控名称」（dependent name）时，其含义是：它在某些方面受控（倚赖）于某个 template parameter。如果 `T` 是个 template parameter，`std::vector<T>::iterator` 就是一个受控名称；但如果 `T`是个已知类型（例如 `int`），`std::vector<T>::iterator`是
一个非受控名称（nondependent name）。

分類	解釋和注意
Identifier 標識符號	一個由不間斷的字母、底線和數字組成的序列，不能以數字開頭。某些標識符號保留給編譯器使用，應用程式中無法引用它們（通則之一是：避免以底線或雙底線作爲標識符號的前導）。字母概念較廣，包括特殊的 universal character names（UCNs），亦即「非字母語言」的字型編碼集。
Operator-function-id 運算子函式 標識符號	關鍵字 `operator` 後跟一個「表示某項操作」的符號。例如 `operator new` 和 `operator []`。很多運算子有另一種表示法。例如 `operator &` 和 `operator bitand` 等價（即使做爲一元取址（*address-of*）運算子也如此）。
Conversion-function-id 轉型函式 標識符號	用來表示使用者自定之隱式轉型函式。例如 `operator int&`，它有另一種易招困惑的寫法：`operator int bitand`。
Template-id 模板 標識符號	Template 名稱之後跟著以角括號括起的 template argument list。例如 `List<T,int,0>`。嚴格說來 *C++ Standard* 只允許 template-id 的 template 名稱使用簡單標識符號，這大概是個疏漏。實際上應該也允許使用 Operator-function-id，例如 `operator+< X<int> >`。
Unqualified-id 非受飾 標識符號	這是標識符號的泛化。可以是上述名稱中的任意一個，或是一個「解構式名稱」（例如`~Data` 或 `~List<T,T,N>`）。
Qualified-id 受飾 標識符號	當一個 unqualified-id 被 class 名稱或 namespace 名稱或 global scope operator 修飾後，便成爲 qualified-id。注意，class 名稱或 namespace 名稱本身也可以帶修飾。下面是一些例子： `::X`, `S::x`, `Array<T>::y`, `::N::A<T>::z`。
Qualified name 受飾 名稱	這個術語在 *C++ Standard* 中並無定義，但我們用它表示那些經歷了所謂「受飾查詢」（*qualified lookup*）的名稱。具體而言這是指一個在 member access operator（`.`或`->`）之後的 qualified-id 或 unqualified-id。例如 `S::x`, `this->f` 和 `p->A::m`。然而即使 `class_mem` 在上下脈絡（context）中隱式等價於 `this->class_mem`，它也不是一個受飾名稱，因爲 member access operator 必須明確出現。
非受飾 名稱 Unqualified name	一個並非 qualified name 的 unqualified-id。這並不是一個標準術語。我們用它來表示那些經歷了所謂「非受飾查詢」（*unqualified lookup*）」的名稱。

表9.1 名称分类表

分類	解釋和注意
Name 名稱	既可以是個受飾名稱, qualified name，也可以是個非受飾名稱, unqualified name。
Dependent name 受控名稱	它在某些方面受控（倚賴）於某個 template parameter。顯然，任何包含 template parameter 的受飾名稱或非受飾名稱都是一種受控名稱。一個以 member access operator（.或->）修飾的受飾名稱，如果左側內容倚賴某個 template parameter，便可視爲受控名稱。具體而言，如果 template 程式碼中出現 this->b，那麼 b 是一個受控名稱。最後，出現於呼叫型式 ident(x,y,z)中的標識符號 ident 是個受控名稱 — 若且惟若任一引數型別倚賴某個 template parameter。
Nondependent name 非受控名稱	只要不符合上述受控名稱（dependent name）所描述的，便是此類。

表9.2 名称分类表

一个由不间断的字母、底线和数字组成的序列，不能以数字开头。某 些标识符保留给编译器使用，应用程序中无法引用它们（通则之一 是：避免以底线或双底线作为标识符的前导）。字母概念较广，包 括特殊的 universal character names（UCNs），亦即「非字母语言」的 字型编码集。

关键词 `operator`后跟一个「表示某项操作」的符号。例如 `operator new`和 `operator []`。很多运算符有另一种表示法。例如 `operator &`
和 `operator bitand`等价（即使做为一元取址（*address-of*）运算符 也如此）。

用来表示使用者自定之隐式转型函数。例如 `operator int&`，它有另
一种易招困惑的写法：`operator int bitand`。

Template 名称之后跟着以角括号括起的 template argument list。例如 `List<T,int,0>`。严格说来 C++*Standard* 只允许 template-id 的 template 名称使用简单标识符，这大概是个疏漏。实际上应该也允许使用 Operator-function-id，例如 `operator+< X<int> >`。

这是标识符的泛化。可以是上述名称中的任意一个，或是一个「解 构式名称」（例如`~Data`或 `~List<T,T,N>`）。

当一个 unqualified-id 被 class 名称或 namespace 名称或 global scope operator 修饰后，便成为 qualified-id。注意，class 名称或 namespace 名称本身也可以带修饰。下面是一些例子： `::X`, `S::x`, `Array<T>::y`, `::N::A<T>::z`。

这个术语在C++*Standard* 中并无定义，但我们用它表示那些经历了所 谓「受饰查询」（*qualified lookup*）的名称。具体而言这是指一个在 member access operator（.或->）之后的 qualified-id 或 unqualified-id。 例如 `S::x`, `this->f`和 `p->A::m`。然而即使 `class_mem`在上下脉络（context中隐式等价于 `this->class_mem`它也不是一个受饰名称，因为 member access operator 必须明确出现。

一个并非 qualified name 的 unqualified-id。这并不是一个标准术语。 我们用它来表示那些经历了所谓「非受饰查询」（*unqualified lookup*）」的名称。

既可以是个受饰名称, qualified name，也可以是个非受饰名称, unqualified name。

它在某些方面受控（倚赖）于某个 template parameter。显然，任何包含 template parameter 的受饰名称或非受饰名称都是一 种受控名称。一个以 member access operator（.或->）修饰的 受饰名称，如果左侧内容倚赖某个 template parameter，便可 视为受控名称。具体而言，如果 template 程序代码中出现
`this->b`，那么 `b`是一个受控名称。最后，出现于调用型式 `ident(x,y,z)`中的标识符 `ident`是个受控名称 — 若且惟 若任一自变量类型倚赖某个 template parameter。

只要不符合上述受控名称（dependent name）所描述的，便是此类。

9.2 名称查询（Looking Up Names）

C++ 语言的名称搜寻机制涉及极多细节，我们只集中在数个主要概念上。众多细节只是为了保 证：(1) 通常情况下这些查询机制符合人们的直觉；(2) 特别复杂的情况可以在C++ *Standard* 中 找到解答。

受饰名称（qualified names）的查询范围是在其「修饰构件所意味的作用域」（the scope implied by the qualifying construct）内。如果该作用域是个 class，则其 base class 也将被查询。然而编译器 查询受饰名称时并不考虑其「圈封作用域」（enclosing scopes）。下面的例子阐示这个基本法则：

```
int x;

class B {
public:
    int i;
};

class D : public B { };


void f(D* pd)
{
    pd->i = 3; // 编译器找到 B::i。（译注：pd->i是一个 qualified name）
    D::x = 2;  // ERROR：在 D作用域（包括 B作用域）中找不到 x
}
```

与之对比的是，编译器通常会依次在「更外层的圈封作用域」（more enclosing scopes）中查询非受饰名称（unqualified names）— 尽管在成员函数定义式内编译器会先查询 class 作用域和 base classes 作用域，然后才是其它圈封作用域。这便是所谓的 ***ordinary lookup***（常规查询）。下面 例子展示 *ordinary lookup* 的基本概念：

```
extern int count;              // (1)
```

```
int lookup_example(int count)  // (2)
{
    if (count < 0) {
        int count = 1;          // (3)
        lookup_example(count);  // 非受饰的 count代表的是(3)
    }
    return count + ::count;     // 第一个 count(非受饰) 代表(2)，
                                // 第二个::count(受饰) 代表(1)。
}
```

在*ordinary lookup*（常规查询）之外还有一种作法用来查询非受饰名称（unqualified names）。这种机制有时称为 *argument-dependent lookup*（ADL；「相依于自变量」的查询）。深入ADL细节之前，我先介绍一个引发此机制的例子：

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

现在假设我们需要把这个 template 应用到定义于另一个 namespace 内的类型身上：

```
namespace BigMath {
    class BigNumber {
        ...
    };
    bool operator< (BigNumber const&, BigNumber const&);
    ...
}

using BigMath::BigNumber;

void g (BigNumber const& a, BigNumber const& b)
{
    ...
    BigNumber x = max(a,b);
    ...
}
```

这里的问题在于，max() template 对 BigMath namespace 一无所知，而 *ordinary lookup*（常规查
询）机制无法找到一个 operator< 可施行于 BigNumber 类型。如果没有某种特殊规则，这个 问题就大大缩减了 templates 在 C++ namespaces 情形下的应用性。C++ 对这个问题的

解决办法 就是 ADL。

9.2.1「相依于自变量」的查询（Argument-Dependent Lookup, ADL）

ADL 只适用于这样的非受饰名称（unqualified names）：在函数调用动作中用到的一个非成员函 式名称。如果 *ordinary lookup* 可找到一个成员函数名称或一个类型名称，编译器就不启动 ADL。 如果被调用函数的名称被写进一对小（圆）括号内，ADL 也不起作用。

否则，如果名称之后跟着的是一个以小（圆）括号括起的自变量算式列（argument expressions list），ADL 就会在「与所有 call arguments 类型相关联」的每一个 namespace 和 class 中查询该名称。所 谓「相关联的 namespaces 和 classes」的精确定义将在稍后给出，但你可以直观认为，它们与给 定之类型有十分直接的关系。假设给定的类型是个指针，指向 class X，编译器便认为X以及X隶属的所有 classes 和 namespaces 都与该指针类型相关联（*associated*）。

下面是「与 call arguments 类型相关联之 namespaces 集和 classes 集」的精确定义：

对内建类型而言：空集合。（译注：此条款在实际运用时有些例外情况，请参考 10 章）

对 pointer 类型和 array 类型而言，基础构成类型（underlying type，亦即 pointer 所指类型或 array
元素类型）。

对 enum 类型而言：声明所在之 namespace。如果其成员是 classes，那么圈封类别（enclosing class）就是其关联类别（associated class）。

对 class（含 union）类型而言：「相关联之 class 集」包括(1) class 或 union 自身、(2) 其圈封 类别（enclosing class）、以及 (3) 任何直接或间接的 base classes。「相关联之 namespace 集」 则包括所有相关联之 classes 声明所在的 namespaces。如果我们所讨论的这个 class 是个 class template 具现体，那么所谓「相关联之 class 集」和「相关联之 namespace 集」就还包括 (1) 每
一个 template type argument 类型，以及 (2) 每一个 template template arguments 声明所在的
classes 和 namespaces。

对 function 类型而言：所有「与参数类型和回返值类型相关联」的 classes 和 namespaces。

对pointer-to-member-of-class-X 类型而言：所有「与X相关联」以及「与被指向之成员相关联」的 classes 和 namespaces。如果我们所讨论的是个 pointer-to-member-function 类型，那么就还包括所有「与参数和回返值相关联」的 classes 和 namespaces。

ADL 会在所有「相关联的namespaces」中查询名称，就像该名称被所有这些namespaces修饰过一样（只不过略去 using指令罢了）。下面例子展示了ADL过程：

```
// details/adl.cpp
#include <iostream>

namespace X {
    template<typename T> void f(T);
}
```

```
namespace N {
    using namespace X;
    enum E { e1 };
    void f(E) {
        std::cout << "N::f(N::E) called" << std::endl;
    }
}

void f(int)
{
    std::cout << "::f(int) called" << std::endl;
}

int main()
{
    ::f(N::e1); // 受饰（qualified）函数名称：不使用 ADL。
    f(N::e1);    // ordinary lookup 找到::f，而 ADL 找到 N::f()，编译器会优先考虑后者
}
```

注意这个例子，当编译器动用 ADL 机制时，namespace N中的using指令被忽略。因此main()中的 f()不会被编译器认为是对 X::f()调用。

9.2.2 Friend 名称植入（Friend Name Injection）

friend function 声明语句可以是该函数的第一份声明语句。这种情况下，该函数会被视为声明于封住 class X（内含该 friend 声明）的「最内层 namespace（有可能是 global namespace）」作用域中。 这个声明在「接受其植入」的作用域内是否可见，是人们经常争论的问题。这个问题很大程度
是 templates 带来的。考虑下面例子：

```
template<typename T>
class C {
    ...
    friend void f();
    friend void f(C<T> const&);
    ...
};

void g (C<int>* p)
{
    f();    // 此处是否可见 f()?
    f(*p); // 此处是否可见 f(C<int> const&)?
}
```

问题在于，如果 friend 声明语句在其圈封之命名空间（enclosing namespace）中可见，那么当我们 实例化一个 class template 时会造成常规函数（ordinary function）的声明也可见。有些程序员会对此感到惊讶，因此 C++ *Standard*规定friend声明语句不得造成其常规函数名称在圈封作用域（enclosing namespace）中可见。

然而，有一个有趣的编程技术，依靠「只在friend声明语句中声明或定义函数」来实现（见11.7节），因此C++*Standard* 又规定，当函数的 friend class 是符合 ADL 规则之「相关联 classes」中的一个时，该 friend function 可见。

让 我 们重新考虑上 面 的例子。由 于f()调用 中没有引 数，因此它没有相 关 联 的classes 或 namespaces，所以上例中的它是个非法调用。然而f(*p)确实与class C<int> 相关联（因为后者是*p 的类型），也与global namespace相关联（因为*p 的类型声明于global namespace），于是由于class C<int> 在此调用动作发生之前就已被实例化，因此第二个friend function声明语句是可见的。为确保这一点，一个会造成「在相关联之classes内查询friends」的函数调用动作，会引发「相关联之 classes」被实例化（如果当时它还未被实例化的话）。

9.2.3 植入 Class 名称（Injected Class Names）

Class 名称会被「植入」class 自身作用域中，因此你可以在该作用域中透过未受饰名称（unqualified name）的形式使用该名称。你不能透过受饰名称（qualified name）来存取它，因为受饰名称在 语法上被用来表示该 class 的构造函数。举个例子：

```
// details/inject.cpp
#include <iostream>

int C;

class C {
private:
    int i[2];
public:
    static int f() {
        return sizeof(C);
    }
};

int f()
{
    return sizeof(C);
}

int main()
{
   std::cout << "C::f() = " << C::f() << ","
```

```
            << " ::f() = " << ::f() << std::endl;
}
```

成员函数 C::f()传回类型 C的大小，::f()传回变量 C的大小，两者都表示一个 int object 的大小。

Class templates 也会内植 class 名称，但和常规的「内植 class 名称」比起来有点特别：它们的后 面可以跟着 template arguments（此时称为「内植 class template 名称」）。如果后面不跟着 template arguments，代表的是「以参数做为自变量（对 class template 偏特化来说则是做为特化用自变量）」
的 class。下面的例子可以解说这些规则：

```
template<template<typename> class TT> class X {
};

template<typename T> class C {
   C* a;          // OK：与 C<T>* a; 相同
   C<void> b;     // OK
   X<C> c;        // ERROR：不带 template argument list 的 C并不象征一个 template
   X<::C> d;      // ERROR：<: 是 [ 的另一种写法
   X< ::C> e;     // OK：有必要在 < 和 :: 之间加入空格
};
```

注意这个例子中的非受饰名称（unqualifited name）代表的是「被内植名称」（injected name）。如果它后面不跟着 template argument list，编译器不认为它是 template 名称。为了弥补这个问题，我们可以使用 scope qualifier ::（作用域修饰符号）修饰之，强迫 template 名称被找到。这么做 确实可行，但我们必须小心不要写出「双字符语汇单元」（digraph token）<:，因为它会被编译 器解释为一个 '['。虽然你可能很少使用这种写法，但万一出现这种错误，恐怕很难诊断。

9.3 解析（Parsing）Templates

大多数编程语言编译器会进行两项基础任务：(1) ***tokenization***（语汇单元化，又称扫描 scanning） 或律法分析（lexing），(2) ***parsing***（词法解析）。*tokenization* 会把源码读入一个字符序列（characters sequence），再由其中产生一个语汇单元序列（tokens sequence）。举个例子，当编译器看到字 元序列 int* p = 0;，tokenizer（语汇单元建立器）会产生如下的语汇单元：一个关键词 int，一个运算符（或符号）*，一个标识符（identifier）p，一个运算符（或符号）=，一个整数字 面常数 0，和一个运算符（或符号）;。

之后，*parser*（词法解析器）会在语汇单元序列（tokens sequence）中搜寻已知的 patterns（样式）， 作法是递归缩减 token 或是把先前搜寻到的 patterns 组成更高级别的构件（high level constructs）。例如「token 0」是个合法算式，「* 后跟一个标识符 p」是合法的声明符号，其后再跟一个 '='， 而后

再跟算式 '0'，仍然是个合法声明符号。最后关键词 int是个已知类型，当它后跟一个 *p=0 声明符号时，就得到了 p的初始化声明语句。

9.3.1 Nontemplates 的前后脉络敏感性（Context Sensitivity）

正如你所知道或所猜想的那样，*tokenizing* 要比 *parsing* 容易。幸运的是 *parsing* 理论基础已经相 当稳固，很多语言都可以根据这个理论不困难地进行 *parsing*。这个理论对于「前后脉络无关
（context-free）」的语言最有效，然而我们先前已经说过，C++ 是一个前后脉络敏感的语言。 为了进行 *parsing*，C++ 编译器把一个符号表（symbol table）结合于 *tokenizer* 和 *parser* 身上： 当某个声明被成功解析(*parsed*)后便进入符号表中当 *tokenizer* 找到一个标识符（identifier） 便查询符号表，如果在其中找到对应（同名）类型，就将 *resulting token* 标注出来。

举个例子。如果C++ 编译器看到

```
x*
```

于是 *tokenizer* 查询 x。如果在符号表中找到类型 x，*parser* 会看到

```
identifier, type,
x symbol, *
```

于是推断这是一个声明的开始。然而如果 x不是类型，那么 *parser* 会从 *tokenizer* 获得：

```
identifier, nontype,
x symbol, *
```

于是这一构件(construct)就只能被合法解析为一个乘法操作。这些原则的细节取决于具体实作 策略，但要旨不变。

下面这个例子说明「前后脉络敏感性」（context sensitivity）。考虑以下表达式：

```
X<1>(0)
```

如果 X是个 class template 名称，这个表达式便是将整数 0转型为 X<1> 所指涉的类型。但如果 X不是个 template，这个表达式等价于：

```
(X<1)>0
```

换句话说 X和 1的比较结果（true 或 false）被隐式转型为 1 或 0，然后再和 0比较。虽然这种写 法很少见，但它确实是合法的 C++ 程序代码（也是合法的 C 程序代码）。只有当 C++ *parser* 看到
一个 template 名称时，它才会认为其后跟随的<是个角括号，否则会认为那是个「小于」符号。

这种「前后脉络敏感性」，是当初以角括号作为 template argument list 界定符号的不幸产物。下面是另一个例子：

```
template<bool
B> class Invert
```

```
{ public:
    static bool const result = !B;
};

void g()
{
    bool test = Invert<(1>0)>::result; // 小（圆）括号必须存在!
}
```

如果 `Invert<(1>0)>`中的小括号省略第一个大于符号会被编译器误认为是 template argument list 的结束符号，于是编译器认为这是个非法算式（等价于`((Invert<1>))0>::result`）。

tokenizer 也会碰到角括号的问题。前面我们已经提醒过（见 3.2 节），在嵌套的（nested）template-ids 之间应该加入空格：

```
List<List<int> > a;
             //^-- 这里的空格是必要的
```

两个右角括号之间的空格是必要的：如果没有空格，两个紧邻的 `>` 会合成一个右移运算符 `>>`，不会被当作两个独立的 `>`。这便是所谓 *maximum munch tokenization principle*（语汇单元化之最大吞入原则）的结果：C++ 编译器必须使一个 token（语汇单元）所含的连续字符尽可能地多。

这个问题使得很多 template 初学者相当困惑。因此有些C++ 编译器实作品做了修改，俾能够在特定情况下将`>>`当成两个独立的`>`（同时给出「不符合 C++ *Standard*」的警告）。C++标准委员会也正在考虑把此一行为加入C++*Standard* 的某个修订版本中（见 13.1 节）。

「最大吞入原则」的另一个例子较少引人注意：*scope resolution* 运算符如果和角括号连用，也必须小心：

```
class X {
    ...
};

List<::X> many_X;    // 语法错误!
```

问题在于，字符序列`<:` 是个所谓的「双字符语汇单元」（*digraph*），是 `[` 的另一种写法。编译器实际处理的将是 `List[:X> many_X;`，而这不具合法意义。这个问题同样可以透过空格来解决：

```
List< ::X> many_X;
   //^-- 这里的空格是必要的
```

9.3.2 类型的受控名称（Dependent Names）

Templates内的名称，其问题在于：它们往往不具备足够的确定性。更明确地说，一个template 不能窥见另一个template 内部，因为后者的内容可能因为明确特化（explicit specialization）而变得不合法（细节见第12章）。下面的例子可以说明这一点：

```
template<typename T>
class Trap {
public:
    enum { x };         // (1) 在这里，x并不是类型
};

template<typename
T> class Victim
{ public:
    int y;
    void poof() {
        Trap<T>::x*y;   // (2) 这是一个声明还是一个乘法运算？
    }
};

template<>
class Trap<void> {     // 「恶意」特化
public:
    typedef int x;     // (3) 在这里，x是个类型
};

void boom(Victim<void>& bomb)
{
    bomb.poof();
}
```

当编译器对(2)进行 *parsing* 时，它必须确定(2)是个声明还是个乘法，而这取决于「受控受饰 名称」(dependent qualified name)Trap<T>::x是不是一个类型名称。你可能会想去 template Trap 里头查看，于是从(1)得知 Trap<T>::x 不是类型，这就使我们认为(2)是个乘法。然而稍后的 程序代码打破了这种想法：令 T为 void，将泛化的 Trap<T> 加以特化，于是 Trap<T>::x事实上 成了 int类型。

C++ 语言对此问题做出了明确的规定：通常一个「受控受饰名称」（dependent qualified name）并不指涉某个类型，除非该名称以关键词 typename为前导。如果在 template arguments 替换过 程之后，该名称最终并不是个类型名称，则编译器认为程序不合法，并在具现期（instantiation time）报错。注意此处的 typename 关键词并非用来指涉一个 template type parameter 参数，因 此它和 type parameter 不同你不能把 typename换以 class。下列情况的名称必需加上 typename 前导：

1. 该名称在 template 中出现。
2. 该名称是个受饰名称（qualified name）。
3. 该名称不被用于 base class list，也不被用于构造函数的成员初值列(member initializers)。
   译注：上述说的 base class list 是指这种情况：

```
template <typename T>
class Derived : Base1<typename T::x>, Base2<typename T::y> {
    // Base1和 Base2之前不能加 typename，
    // 但是当 x和 y都是类型时，T::x和 T::y之前必须加 typename
    ...
};
```

4. 该名称受控（依赖）于某个 template parameter。

除非前三种情况都成立，否则你不能使用 typename前导词。考虑下面例子[27]：

```
template<typename1 T>
struct S: typename2 X<T>::Base {
    S(): typename3 X<T>::Base(typename4 X<T>::Base(0)) {}
    typename5 X<T> f() {
        typename6 X<T>::C * p;          // 声明指针 p
        X<T>::D * q;                    // 乘法
    }
    typename7 X<int>::C * s;
};

struct U {
    typename8 X<int>::C * pc;
};
```

每一处 typename 不论正确与错误，都加了下标以便区分。第 1 个 typename 表示 T 是个 template parameter，先前的规则不适用于这里。第 2 和第 3 个 typename错误，因为根据前述第三条规 则，在 base classes list 或 member initialization list 中不能使用 typename。第 4 个 typename是必 要的，因为这里的 base class 名称并无「被初始化」或「被继承」的含义，而是用来以自变量 0建 构一个暂时的 X<T>::Base（你也可以认为这是一种转型）。第 5 个 typename错误，因为 X<T> 不是个受饰名称qualified name)第 6 个 typename所在述句如果是个指针声明，那么 typename 是必需的；其下一行没有使用关键词 typename，编译器因而认为那是个乘法。第 7 个 typename 可有可无，因为它同时满足了前三条规则。第 8 个 typename 错误，因为它没有被用在 template 内。

9.3.3 Templates 的受控名称（Dependent Names）

当 template 之中有个受控名称（*dependent name*）时，你可能会碰到和前面类似的问题。通常如果 < 紧跟于一个 template 名称之后，C++ 编译器会认为 < 代表一个 template argument list 的开 始；其它情况下编译器会认为 < 是个「小于」符号。这就好像面对类型名称时，编译器必须假 设受控名称（*dependent name*）并不表示某个 template，除非你以关键词 template 提供额外资 讯：

```
template<typename T>
class Shell {
public:
```

```
    template<int
    N> class In
    { public:
        template<int
        M> class Deep
        { public:
            virtual void f();
        };
    };
};

template<typename T, int N>
class Weird {
public:
    void case1(typename Shell<T>::template In<N>::template Deep<N>* p) {
        p->template Deep<N>::f(); // 抑制 virtual call
    }
    void case2(typename Shell<T>::template In<N>::template Deep<N>& p) {
        p.template Deep<N>::f();  // 抑制 virtual call
    }
};
```

这个例子有些复杂，它阐示在「可用以修饰（qualify）名称」的所有运算符（`::`, `->`, `.`）之后，可能需要加上关键词 `template`。当修饰运算符（qualifying operator）之前的「名称或算式的型别受控于某个 template parameter，而修饰运算符（qualifying operator之后的名称是个 template-id（亦即 template 名称再加上以角括号括起来的 template argument list）时，就必须这么办。例如以下算式：

```
p.template Deep<N>::f()
```

p的类型受控（取决）于 template parameter T。C++ 编译器无法知道 Deep是否是个 template，我们必须透过 template前导词才能指明 Deep是个 template 名称，否则 p.Deep<N>::f()会被解析（*parsed*）为((p.Deep)<N)>f()。另请注意，对一个受饰名称（qualified name），有时你可能需要添加多个 template 前导词，因为饰词（qualifier）本身也可能被一个受控饰词（dependent qualifier）修饰（前例的 case1和 case2两声明就属于这种情况）。

如果在这些情况下不写 template关键词，左角括号和右角括号就被为编译器视为小于和大于符号。然而如果某处并非绝对需要这个关键词，那就是不需要。你不能到处喷洒（滥用）template饰词。

9.3.4 using声明语句中的受控名称（Dependent Names）

using 声明语句可以把 namespaces 和 classes 内的名称带入当前作用域。这里不考虑 namespaces 的情况，因为并不存在所谓 namespace templates。当你使用 using 声明语句带入

classes 内的名称时， 可以把 derived class 内的名称带入 base class。你可以把这一类 using声明语句看成为 derived class 建立一个通往 base class 的符号链接（symbolic links）或快捷方式（shortcuts）」，如此一来就允许 derived class 的成员可以存取 base class 成员，就好像那些 base class 成员声明于 derived class 一样。下面这个 non-template 的小例子可以很好地说明这个问题：

```
class BX {
public:
    void f(int);
    void f(char const*);
    void g();
};

class DX : private BX {
public:
    using BX::f;
};
```

上例中的 using声明语句把 base class BX内的名称 f带入 derived class DX。此时的名称 f与两个宣告相关联，这正强调了这种机制带入的是一组名称，而不是一组符合该名称的个别声明。注意 这种 using声明语句可以使 derived class 存取它原本无权存取的 base class 成员本例的 base class BX
（及其成员）对 class DX来说是 private，但 BX:f函数被引入成为 DX的 public 接口，于是 class DX 使用者也能取用 BX:f 函数。由于 using 机制具备这个机能，先前的「存取宣告」（access declaration）在C++ 中便作废了（未来C++ 修订版可能不会再支持这种机制）：

```
class DX : private BX {
public:
    BX::f; // 存取声明（access declaration）语法被废除，应以 using BX::f代替
};
```

现在你或许察觉到了，使用 using声明语句将一个受控（dependent）class 内的名称带入当前作用 域会造成什么问题 尽管我们知道某个名称的存在，但我们不知道它是个 type 还是个 template， 或是别的什么东西：

```
template<typename T>
class BXT {
public:
    typedef T Mystery;
    template<typename
    U> struct Magic;
};

template<typename T>
class DXTT : private BXT<T> {
public:
    using typename BXT<T>::Mystery;
```

```
    Mystery* p; // 如果上一行无 typename，此处语法有误。
};
```

如果我们希望以一个 using声明语句将一个受控名称（dependent name）带入当前作用域，并以它 指涉（代表）一个类型，就必须使用关键词 typename明确告知编译器。奇怪的是 C++ 并没有 提供类似机制来指示「受控名称（dependent name）是个 template」。下面例子说明这个问题：

```
template<typename T>
class DXTM : private BXT<T> {
public:
    using BXT<T>::template Magic;     // 错误：无此用法
    Magic<T>* plink;                  // 语法错误：Magic不是个确知的 template
};
```

这大概是C++*Standard* 的疏忽。未来修订版可能会考虑加入这种用法，从而使上述例子合法。

9.3.5 ADL 和 Explicit Template Arguments（明确模板引数）

考虑下面例子：

```
namespace N {
    class X {
        ...
    };

    template<int I> void select(X*);
}

void g (N::X* xp)
{
    select<3>(xp); // 错误：ADL不适用
}
```

此例之中，我们可能希望编译器看到 select<3>(xp)时透过 ADL 找到 template select()。但 情况并非如此，只有在确认 <3>是个 template argument list 时，编译器才能确认 xp是一个 function call argument。反过来说只有当编译器发现 select()是个 template，它才能确认 <3>是个 template argument list。这是个「先有鸡还是先有蛋」的问题；这个算式只能被解析（*parsed*）为 (select<3)>(xp)，而这毫无意义。

9.4 衍生（Derivation）与 Class Templates

Class templates 可以继承其它 classes，也可以被其它 classes 继承。大多数情况下 class templates 和 non-template classes 在这方面并没有什么重大区别。但是有一个微妙而重要的问题，出现在

「由一个受控名称（dependent name）所指涉的 base class，衍生出一个 class template」时。我们先从较简单的 non-dependent base classes 说起。

9.4.1 非受控的（Non-dependent）Base Classes

在 class template 中，所谓 non-dependent base class 是指一个无需知道任何 template arguments 就 可完全确定的类型。换句话说，「用来指涉该 base class」的名称，是个非受控名称（non-dependent name）。例如：

```
template<typename X>
class Base {
public:
    int basefield;
    typedef int T;
};

class D1: public Base<Base<void> > {      // 并不真正是个 template
public:
    void f() { basefield = 3; }           // 以通常方式存取继承而来的成员
};

template<typename T>
class D2 : public Base<double> {          // non-dependent base class
public:
    void f() { basefield = 7; }           // 以通常方式存取继承而来的成员
    T strange;                        // T是 Base<double>::T，不是 template parameter
};
```

non-dependent base templates 的行为和常规的 non-template base classes 非常相似，但是有个令人 惊奇的差异：当编译器在 derived class template 中查询一个未受饰名称（unqualified name）时， 会优先查询 non-dependent base templates，然后才查询 template parameters。这意味上述例子中， class template D2的成员 strange将拥有 Base<double>::T类型（本例为 int）。因此以下函 式非法（续上）：

```
void g (D2<int*>& d2, int* p)
{
    d2.strange = p; // 错误：类型不匹配
}
```

这与直觉相悖，derived template 编写者因此必须特别留心其 non-dependent bases 中的名称 —甚 至即使是间接继承，或名称都是 private。也许较好的作法是把 template parameters 放在被它们模 板化的物体（the entity they templatized）的作用域内。

9.4.2 受控的（Dependent）Base Classes

上面的例子中，base class 是完全确定的。它不取决于某个 template parameter。也就是说C++ 编译器只要见到了 template 的定义，就可以在 base classes 中查询非受控名称（non-dependent names）。另一种作法（不被 C++ 接受）是将这一类名称的查询过程推迟，直到 template 被具现化后才开始。这种作法的缺点是，由于查询过程被推迟至实例化之后，也会把查询过程中可能产生的「找不到符号」错误信息推迟。因此 C++ *Standard* 规定，编译器只要见到了一个非受控名称（non-dependent name），就立即开始查询（looked up）。牢记这一点之后，考虑下面的例子（译注：其中 class `Base`的定义请见 9.4.1 节）：

```
template<typename T>
class DD : public Base<T> {          // dependent base
public:
    void f() { basefield = 0; }      // (1) 有问题…
};

template<>                           // 明确特化（explicit specialization）
class Base<bool> {
public:
    enum { basefield = 42 };         // (2) 小花招
};

void g (DD<bool>& d)
{
    d.f();                           // (3) 喔哦?
}
```

在(1)处我们发现，这里使用了一个非受控名称 `basefield`：编译器一定会立即查询它。假设我们在 template `Base`中查询它，并将它系结为一个（我们所找到的）`int`成员。然而在这之后我们立刻对这个泛型定义进行特化，并在(2)处将 `basefield`改变为我们委任的定义。于是当(3)处实例化 `DD:f`的定义时，会发现(1)处对 `basefield`的类型解释并不准确。(2)处特化的`DD<bool>`之中并没有可变化的 `basefield`，因此编译器会发出一个错误讯息。

为了解决这个问题，C++ *Standard* 规定非受控名称（non-dependent names）不在受控的 base class 中查询（但是见到这种名称仍会立刻展开查询）。因此符合标准的 C++ 编译器会在(1)处报错。欲修正上述程序代码，我们可以令 `basefield`成为一个受控名称（dependent names），而受控名称的查询过程会发生在它被实例化之后此时 `basefield`的特化类型就可以被编译器发现。例如在(3)处，编译器会知道 `DD<bool>`的 base class 是 `Base<bool>`，而 `Base<bool>`已被具现化。据此，我们可以将程序代码修改为：

```
// 修改 1
template<typename T>
class DD1 : public Base<T> {
public:
    void f() { this->basefield = 0; } // 查询过程被推迟
};
```

另一种作法是使用一个受饰名称（qualified name）导入相依性（受控性, dependency）：

```
// 修改 2
template<typename T>
class DD2 : public Base<T> {
public:
    void f() { Base<T>::basefield = 0; }
};
```

你必须小心使用这种解法。如果在 virtual function call 中使用非受饰、非受控名称（unqualified nondependent name），修饰符号会使 virtual call 机制失效，程序代码意义将因此改变。然而也存 在只能用第二种方法，不能用第一种方法的情况：

```
template<typename T>
class B {
public:
   enum E { e1 = 6, e2 = 28, e3 = 496 };
   virtual void zero(E e = e1);
   virtual void one(E&);
};

template<typename T>
class D : public B<T>
{ public:
    void f() {
        typename D<T>::E e; // this->E语法有错
        this->zero();       // D<T>::zero()会抑制 virtual call
        one(e);             // one是受控名称，因为其自变量是受控的
   }
};
```

注意，one(e)中的 one是个受控名称，受到 template parameter 的影响。这是因为该调用中的一 个明确自变量的类型是受控类型（译注：e隶属受控类型 D<T>::E，受到 template parameter T的 影响）。对于被隐式使用的预设自变量来说，如果它会受到 template parameter 的影响，就会被编 译器忽略。这是因为编译器决定查询（lookup）它时，却无法验证（查清, verify）它，这又是一 个鸡和蛋的问题。为了避免此种微妙，我们推荐你在所有情形中使用 this-> 前缀词 — 即使 在书写 non-template 程序代码时亦然。

如果你觉得这么多修饰符号（饰词, qualifier）会弄乱程序代码，可透过 using 声明语句将某个名称 从受控的 base class 中带到当前作用域，于是你就可以肆意使用它。

```
// 修改 3
template<typename T>
class DD3 : public Base<T> {
public:
```

```
    using Base<T>::basefield;     // (1) 现在，basefield在当前作用域中为受控名称
    void f() { basefield = 0; }   // (2) 正确
};
```

(2)处进行的查询过程将会成功，它会找到(1)处的 using声明（亦即找到 basefield名称）。然而直到具现期（instantiation time）之前编译器并不会验证 using 声明语句，因此我们就达到了 目的。这个方案也有一些微妙限制，例如继承自多个 base classes 时，程序员必须明确指定需要 哪一个 base class 的成员。

9.5 后记

可对 template 定义式进行词法解析（*parse*）的第一个编译器是 Taligent 公司于 1990s 年代中期 开发的。在此之前（及之后），大多数编译器都把 templates 当作一个 *tokens sequence*（语汇单 元序列），并在具现期（instantiation time）将它交给 *parser*。因此除了小量情况足以标示「template 定义结束位置」，大部份时候 *parser* 都无法正常工作。Bill Gibbons 是 Taligent 公司在C++ 标 准委员会的代表，他是使 templates 可被精确进行词法解析的主要贡献者。Taligent 的成果一直 都未公开，后来 Hewlett-Packard（HP）获得了它并加以完善，并把它变为 aC++ 编译器。除了 极富竞争力的各种优点外，其诊断信息也被公认为有着极高质量。由于它不把诊断信息推迟至 template 实例化之后，因此获得人们更高的评价。

在 templates 技术发展前期，Tom Pennello（一位公认的 *parsing* 专家，任职于 Metaware）注意 到角括号带来的一些问题。Stroustrup 在 [StroustrupDnE] 中对此作出评论，并认为相对于小（圆） 括号而言，人们更喜欢使用角括号。然而也有人偏爱其它符号作为 template parameter list 的界 定符号，例如 Pennello 在 1991 年于 Dallas（达拉斯）举行的C++ 标准委员会议上[30]，就提议使 用大（花）括号（像是 List{::X}）。当时这个问题不像如今这么凸显，因为当时还不允许把
一个 template 嵌套于另一个 template 之中（成为 member templates），因此 9.3.3 节， p.132 的讨 论大部份在当时都没有意义。最终结果是：委员会否决了以大（花）括号替代角（尖）括号。

C++ Standard 于 1993 年引入非受控名称（non-dependent names）和受控 base classes 的名称查询 规则（name lookup rule，见本书 9.4.2 节，p.136），Bjarne Stroustrup 于 1994 年初在 [StroustrupDnE]
中向公众做了相关描述，但是这个规则的第一个可用实作品直到 1997 年才出现：HP 在它的 aC++ 编译器中引入了这个特性。此后有大量程序代码从受控的 base classes 中继承出 class templates。当
HP 工程师开始进行测试时，他们发现大部分对 template 运用颇深的程序代码再也无法成功编译[31]。

特别是所有 STL 实作品都违犯了这个规则，违犯点成百上千[32]。为了缓解其顾客在过渡期间所遇到的难题，HP 放宽了这个规则：当根据 C++ 标准无法找到一个 class template 作用域中的非 受控名称时，aC++ 会在其受控 base classes 中继续寻找。如果还是找不到这个名称，编译器会 报错，声明失败。然而如果在受控 base classes 中找到了该名称，编译器会给出一个警告，同时 该名称会被标示为受控（dependent），这么一来当该名称被实例化时，编译器会再次查询它。

根据查询规则（lookup rule），非受控 base class 中的名称可能会导致编译器无法见到一个与之 同名的 template parameter（见 9.4.1 节，p.135）。这是一个疏漏，未来C++*Standard* 修订版可能 会解决它。目前你最好避免这种重名情况。

Andrew Koenig 首先提议在运算符函数中使用 ADL（这就是为什么 ADL 有时被称为 *Koenig lookup* 的原因）。这其实是个审美观问题：明显地将运算符名称饰以「圈封之 namespace」的名称，委 实有些笨拙（例如我们不能写 `a+b`，必须写成 `N::operator+(a, b)`）；而且在每一个运算符 前面使用 `using` 声明语句，也使程序代码难看。因此人们决定让编译器在查询运算符时，同时查询 其自变量之相关联 namespaces。后来 ADL 又被扩展，适用于查询常规函数名称，以容许一些有限 的「friend 名称植入」（friend name injection），并支持 templates 的两段式查询（two-phase lookup model）模型和实例化。这种更为泛化的 ADL 规则又称为 *extended Koenig lookup*。

10
实例化/实体化
Instantiation

Template 实例化（instantiation）是「由泛化的 template 定义式产生出实际类型和函数」的过程[33]。C++ templates 实例化是一个基础而复杂的概念。说它复杂的一个原因是，由 template 所产生的 物体（entities）的定义不限于源码的某一单独位置上。template 所在位置、template 被使用位置、 以及 template arguments 定义位置，都在「构成物体意义」一事上扮演各自的角色。

本章讲述如何组织我们的程序代码才能以正确方式运用 templates。我们还将讲述为解决实例化问 题，大多数流行的 C++ 编译器所使用的方法。尽管这些方法应该是语意等价的（semantically equivalent），但了解你的编译器的实例化策略，会对你带来帮助。构筑真实世界中的软件时， 每种机制都有不足，然而C++ 标准规格正是在这些机制基础上才得以成形。

10.1 随需实例化（On-Demand Instantiation）

当C++ 编译器见到程序中使用一个 template 特化体（specialization），便将 template parameters 替换为所需的 argument，以创建出这个特化体[34]。这个过程自动完成，无需借助任何 client 程序 码（或 template 定义）的指示。正是这种「随需实例化」（有需要时便实例化）的特性使得C++template 机制有别于其它语言的类似机制这种机制有时也称为隐式*implicit*或自动*automatic*）实例化。

随需实例化（on-deman instantiation）意味着当程序用到 template 时，编译器需要得知该 template 和其某些成员的完整定义（而不仅仅是 template 的声明）。考虑下面例子：

```
template<typename T> class C;   // (1) 只有声明

C<int>* p = 0;                  // (2) 没问题，不需 C<int> 的定义


template<typename T>
class C {
public:
   void f();                    // (3) 成员声明
};                              // (4) class template 定义完备

void g (C<int>& c)              // (5) 只用到 class template 的声明
{
  c.f();                    // (6) 用到了 class template 的定义；需要 C::f()的完整定义
}
```

在(1)处，编译器只能见到 template 的声明，见不到其定义（这种声明又称为前置声明，*forward declaration*）和常规 classes 的规则一样，不需 class template 的定义你就可以声明该类型的 pointers 或 reference（如(2)）。例如函数 g()的参数类型 C<int>& 并不需要 template C的完整定义。然 而一旦编译器需要知道某个 template 特化体的大小，或程序代码中取用了该特化体的某个成员， 编译器就必须见到 template 的定义。这说明为什么在(6)处，class template 的定义必须可见；如 果见不到这个定义，编译器就无法确认这个成员是否存在，或程序代码是否有权取用它（必须不为 private 或 protected，才能被取用）。

这里有另外一个表达式，需要 class template 具现体，因为此处编译器必须知道 C<void>的大小：

```
C<void>* p = new C<void>;
```

这里必须完成实例化，这么一来编译器才知道 C<void>的大小。你可能会注意到，无论把 template C的参数 T以什么样的类型 X替换，其 template 具现体的大小都不会受到影响：任何情况下 C<X>都是个 empty class。但你不能强求编译器知道这一点。不仅如此，在这个例子中，编译器也需 要以实例化过程来得知 C<void> 是否有一个可被取用的 *default* 构造函数，并确保 C<void> 没有 定义 private operator new和 private operator delete。

有时候，从程序代码本身并不能看出某个 class template 的成员会被取用。例如C++ 的重载解析机 制（overload resolution）便需要得知各候选函数的参数的 class types：

```
template<typename T>
class C {
public:
    C(int);     // 单自变量构造函数（one-argument ctor）有可能被用于「隐式转型」。
};

void candidate(C<double> const&);    // (1)
void candidate(int) {}               // (2)

int main()
{
    candidate(42); // 上面声明的两个函数都可被调用
}
```

编译器会将 candidate(42)解析（resolve）为(2)的声明。然而(1)处的声明也可能被实例化， 用来确定它是否也是上述调用的一个合法候选函数（本例之中实例化可能发生，因为 42 可隐寓 通过「单自变量构造函数」转型为 C<double> 类型的 rvalue）。注意，即使编译器可以不透过具现 化来解析（resolve）某个重载调用，*C++ Standard* 仍然允许（但不要求）编译器进行实例化（放 到本例之中，实例化就非必要了，因为已有一个完全匹配的函数 candidate(int)，这使得编译 器不会选用其它需要隐式转型的候选函数）。注意，将 C<double> 实例化可能会触发一个令你 惊讶的错误。

10.2 缓式实例化（Lazy Instantiation）

目前为止，我们所举的例子中，还没有什么东西与 non-template classes 有本质上的区别。很多 情况下编译器需要一个「完整的」class 类型。运用 template 时，编译器会从 class template 定义 式中产生这个「完整的」定义。

这就引发一个相关问题：template 之中有多少内容需要被实例化？一个模糊的回答是：只有「真正被用到」的那些才需要被实例化。换句话说编译器应该尽量推迟 templates 实例化的进行，这 便是所谓的 lazy instantiation。我们来看看 lazy 的具体是什么含义。

当 class template 被隐式实例化，其中的每一个成员声明也都同时被实例化，然而并非所有对应 的定义也都被实例化，其中有些例外。首先，如果 class template 内含一个不具名的 union，这个 union 的成员也都会被实例化[35]。另一个例外与 virtual 成员函数有关。当 class template 被实例化 时，virtual 成员函数的定义可能被（也可能不被）实例化。事实上很多编译器都会实例化这些 定义，因为 virtual call 机制的内部结构要求：virtual 函数必须作为可链接物（linkable entities）， 存在于某个结构（译注：vtbl/vptr）中。

当 templates 被实例化时 *default* call arguments 通常被个别考虑除非有些调用真正用上了 *default* arguments，否则那些 *default* arguments 不会被实例化。如果调用函数时明确指定了 arguments，那么 *default* arguments 也不会被实例化。

下面的例子涵盖了上述内容：

```
// details/lazy.cpp

template <typename T>
class Safe {
};

template <int
N> class Danger
{ public:
    typedef char Block[N]; // 如果 N<=0，会报错
};

template <typename T, int N>
class Tricky {
public:
    virtual ~Tricky() {
    }
    void no_body_here(Safe<T> = 3);
    void inclass() {
        Danger<N> no_boom_yet;
    }
```

```
    // void error() { Danger<0> boom; }
    // void unsafe(T (*p)[N]);
    T operator->();
    // virtual Safe<T> suspect();
    struct Nested
       { Danger<N>
       pfew;
    };
    union {              // 不具名的 union
      int align;
      Safe<T> anonymous;
    };
};

int main()
{
  Tricky<int, 0> ok;
}
```

首先考虑 main()不存在的情况。符合标准的 C++ 编译器通常会编译 template 定义式以进行语法检查，并进行一般的语意约束（semantic constraints）检查。然而在检查 template parameters 的约束条件时，它会「假设最好情况」。例如 Block 内的 typedef 的参数 N 有可能是 0 或负值（这是非法的），但编译器假设这种情况不会发生。类似情况，class template Tricky的成员函数 no_body_here的声明语句中，预设自变量（=3）颇为可疑，因为 template Safe 无法以一个整数值初始化，但编译器会假定 Safe<T> 的泛化定义并不需要这个预设自变量。另外，如果不把成员函数 error()注释掉，它会引发一个错误，因为它会用到 Danger<0> 特化体，而这个特化体试图以 typedef 定义「0 个元素的 array」，即使成员函数 error()未被调用（因而也未被实例化），这个错误也会在编译期对整个泛型 template进行处理的时候触发但成员函数 unsafe(T(*p)[N]) 并没有这个问题，因为此时的 N仍是个泛型参数，template parameters 的替换并未发生。

现在让我们分析 main()存在时的情况。它会将 template Tricky的参数 T替换为 int，参数 N 替换为 0。编译器并不需要所有成员的定义，但 *default* 构造函数（本例为隐寓声明）和析构函数一定会被叫用，因此编译器必须见到它们的定义。virtual 成员的定义也必须可见，否则会链接错误。如果拿掉 virtual 函数 suspect()的注释符号，此例可能会引发链接错误，因为我们没有给出 suspect()定义式。Danger<0> 的完整类型（前面说过这会产生一个非法的 typedef）也是需要的，因为它在成员函数 inclass()和 struct Nested中出现，但由于这些定义没有被用到，所以它们不会被生成，从而不会引发错误。然而，编译器会生成所有成员的声明语句，并可能因为参数/自变量的替换而在这些声明语句中存在非法类型。例如，如果我们拿掉 unsafe(T (*p)[N]) 声明语句的注释符号，会再次创建一个「0 个元素的 array」，这会引发错误。同样道理，如果成员 anonymous 并非声明为 Safe<T> 类型而是声明为 Danger<N> 类型，由于 Danger<0> 类型并不合法，会引发一个错误。

最后请注意 `operator->`。通常这个运算符必须传回一个 pointer 类型，或一个可被 `operator->` 施行于其上的类型。于是你可能认为 `Tricky<int,0>` 会引发错误，因为 `operator->` 的回返型 别是 `int`。然而由于某些自然的（natural）class template 触发了这种定义[36]，因此 C++ 语言在 此较为灵活。使用者自定的 `operator->` 只需在「被重载解析机制选中」时传回一个「可施行 另一个 `operator->`（例如内建的那个）」的类型即可。这在不涉及 template 的情况时也适用（虽 然那种情况下就没什么太大用处）。因此这里的声明不会引发错误，即使它传回的是个 `int`。

10.3 C++实例化模型（C++ Instantiation Model）

所谓 template 实例化，是「适当地替换 template parameters，以便从 template 获得常规 class 或 常规 function」的过程。听来平淡无奇，但实际上这个过程涉及极多细节。

10.3.1 两段式查询（Two-Phase Lookup）

在第 9 章中我们知道，当编译器对 templates 进行 *parsing*（词法解析）时，它无法解析受控名称（dependent names）。这些受控名称将在具现点被再次查询（lookup）。然而非受控名称（non-dependent names）会较早被查询，如此一来当 template 首次被编译器看到时，就可以较多地诊
断出错误。这就是「两段式查询」概念[37]：第一阶段发生在 template *parsing*（词法解析）时刻， 第二阶段发生在实例化时刻。

在第一阶段编译器会查询非受控名称（non-dependent names），这个过程会用到常规查询（*ordinary lookup*）规则；如果情况适用也会动用 ADL（*argument-dependent lookup*）规则。非受饰的受控 名称（unqualified dependent names；由于它们在一个带有 dependent arguments 的函数调用中看起 来像个函数名称，所以是受控的， dependent）也会以此方式被查询，然而其结果并不完备，因此 会在 template 实例化时再次被查询。

在第二阶段，也就是在一个所谓「具现点」（*point of instantiation*, **POI**）处，编译器会查询受控 受饰名称（dependent qualified names；将特定具现体的 template parameters 代换为 template arguments），也会对非受饰受控名称（unqualified dependent names）额外加以 ADL 查询。

10.3.2 具现点（Points of Instantiation）

我们已经说过，template 程序代码中有这样一些位置点；C++ 编译器在这些点上必须能够取得某
个 template 物体的声明或定义。一旦程序代码需要用到 template 特化体，而编译器需要见到该 template 的定义以创造出该特化体时，就会在这个具现点（POI）上进行具现过程。具现点（POI） 就是替换后之 template 可插入的源码位置点。例如：

```
class MyInt {
public:
```

```
    MyInt(int
    i);
};

MyInt operator - (MyInt const&);

bool operator > (MyInt const&, MyInt const&);

typedef MyInt Int;

template<typename T>
void f(T i)
{
    if (i>0) {
        g(-i);
    }
}
// (1)
void g(Int)
{
    // (2)
    f<Int>(42); // 调用点
    // (3)
}
// (4)
```

当编译器看到调用动作 `f<Int>(42)`，它知道必须将 `T`替换为 `MyInt`来实例化 template `f`。这于 是就产生了一个 POI（具现点）。`(2)`和`(3)`距离调用点很近，但它们不能作为 POI，因为 C++语法不允许我们在这两处插入`::f<Int>(Int)`的定义。`(1)`和`(4)`的本质差异在于，函数 `g(Int)` 在`(4)`处可见 于是 template dependent call `g(-i)`可被解析（*resolved*）出来 如果把`(1)`当做 POI，`g(-i)`就无法被成功解析出来，因为编译器在该处看不到 `g(Int)`。幸运的是，针对 "a referendce to a nonclass specialization", C++ 设计的 POI 是在「最内层之 namespace 作用域（惟需内含该reference）」的声明向或定义式的紧临后方。本例中这个点是`(4)`。

你可能会奇怪为什么本例使用 `MyInt`而不是简单地使用 `int`。答案是在 POI 处进行的第二阶段 查询只动用 ADL（「相依于自变量的查询」）。由于 `int`并无相应的 namespace，不会发生 POI 查询，从而编译器无法找到函数 `g()`。如果你把 `Int`的 `typedef`换成：

```
typedef int Int;
```

上述例子就无法通过编译。

```
template<typename T>
class S
```

```
  { pub
  lic:
  T m;
};
// (5)
unsigned long h()
{
    // (6)
    return (unsigned long)sizeof(S<int>);
    // (7)
}
// (8)
```

函数作用域内的(6)和(7)不能作为 POI，因为 class S<int>（它构成一个 namespace 作用域）的 定义式不能出现在这两个地方（template 不能在函数作用域内出现）。如果我们按照 nonclass 的 方式思考，POI 将是(8)，但这样就造成算式 S<int> 不合法，因为在(8)之前 S<int> 的大小 无从得知。因此对于 "a reference to a template-generated class instance"，其 POI 位置被定义为：「最内层之 namespace 作用域（惟需内含该 class instance）」的声明语句或定义式的紧临前方。本 例中这个点是(5)。

当 template 被实际实例化，有可能引发其它实例化动作。考虑下面例子：

```
template<typename T>
class S {
public:
    typedef int I;
};

// (1)
template<typename
T> void f()
{
    S<char>::I var1 = 41;
    typename S<T>::I var2 = 42;
}

int main()
{
    f<double>();
}
// (2):(2a),(2b)
```

根据先前讨论，f<double> 的 POI 是(2)。function template f()也用到了 S<char> 这个 class 特化体，其 POI 在(1)。还用到了 S<T>，但在(1)处 S<T> 仍然受控（dependent），因此在(1)还不能进行实例化。然而如果我们在(2)实例化 f<double>，我们同时必须实例化 S<double>。这种次级 POI（*secondary* POI，或称 *transitive* POI）的定义稍有不同。针对 nonclass 物体，次级 POI 和主要 POI(*primary* POI)完全相同，但针对 class 物体，次级 POI 紧临于其基本 POI 之前。
上例 f<double>的 POI 在(2b)，其前先的(2a)是 S<double>的二级 POI。请注意这与 S<char>的 POI 有所不同。

在一个编译单元中，同一个具现体往往有多个 POI。针对 class template 实体，编译器只保留头一个 POI，忽略所有后续 POI(编译器并不真正认为它们是 POI)。针对 nonclass 实体，所有 POI 都被保留。不论哪一种情况，ODR（单一定义原则）都要求：被编译器保留的所有 POI 彼此必须等价，但编译器不需要检验是否有违例情况。这就使得编译器可以一个 nonclass POI 进行真正的实例化动作，而不需忧虑其它 POI 可能导致其它具现体。

事实上，大多数编译器都会把对 noninline function templates 的真正实例化过程推迟至编译单元尾端。这也就是把相应之 template 特化体的 POI 移到了编译单元尾端。C++ 编译器实作者认为这是一种合法的实作技术，但C++*Standard* 对此并无明确态度。

10.3.3 置入式（Inclusion）和分离式（Separation）模型

无论何时遇到一个 POI，编译器必须能够取得对应之 template 的定义。对于 class 特化体而言，这意味在当前编译单元中，class template 的定义式必须出现于 POI 之前。对 nonclass POI 的态度虽然也如此，但典型情况是 nonclass template 的定义式被放在表头档中，由当前编译单元以#include 将它包含进来。这种 template 定义式的源码组织方式称为置入式模型（inclusion model），也是截至本书完稿时最被大众采用的模型。

针对 nonclass POI 另还存在一种方式：nonclass template 可被声明为 export，并定义于另一个编译单元。这种方式称为分离式模型（separation model）。下面例子展示这种情况，仍然用的是我们的老朋友 max()：

```
// 编译单元 1
#include <iostream>

export template<typename T>
T const& max (T const&, T const&);

int main()
{
    std::cout << max(7, 42) << std::endl;   // (1)
}

// 编译单元 2
```

```
export template<typename T>
T const& max (T const& a, T const& b)
{
    return a < b ? b : a;                    // (2)
}
```

当第一个文件被编译时，编译器认为(1)是 POI，此时 T被替换为 int。编译系统必须确保编译 单元 2 中的 max()定义式被实例化，从而满足 POI 的需求。

10.3.4 跨越编译单元寻找 POI

假设上述的编译单元 1 被重写为：

```
// 编译单元 1
#include <iostream>

export template<typename T>
T const& max(T const&, T const&);

namespace N
   { class I
   { public:
      I(int i): v(i) {}
      int v;
   };

   bool operator < (I const& a, I const& b) {
      return a.v < b.v;
   }
}

int main()
{
    std::cout << max(N::I(7), N::I(42)).v << std::endl; // (3)
}
```

POI 诞生于(3)，而且编译器需要编译单元 2 中的 max()定义。然而 max()使用重载的 operator<，而后者却是在编译单元 1 中被声明，不可见于编译单元 2。为了正确处理这种情况，编译器显然 必须在实例化过程中参考两个不同的「声明脉络」（declaration contexts）：一是 template 定 义于何处，二是类型 I声明于何处。为涉入这两个脉络（contexts），编译器使用两段式查询（见10.3.1节）来对付 template 中的名称。

第一阶段发生在 templates 被 *parsing*（词法解析）时，换句话说在C++ 编译器首次见到 template 定义式时。在此阶段，编译器会使用 *ordinary lookup* 规则和 *ADL* 规则来查询非受控名称（non- dependent names）。另外编译器还使用 *ordinary lookup* 规则来查询受控函数（dependent

functions； 肇因于其函数自变量受控）的非受饰名称（unqualified names），但它会记住查询结果而不企图进 行重载解析（overload resolution）。重载解析工作是在第二阶段后进行的。

第二阶段发生在 POI（具现点）。在 POI 处，编译器会使用 *ordinary lookup* 规则和 *ADL* 规则来 查询受控受饰名称（dependent qualified names）。至于受控非受饰名称（dependent unqualified names；已于第一阶段以 *ordinary lookup* 规则查询过），编译器只运用 *ADL* 规则去查询，再把 查询结果结合第一阶段的结果。这两个结果构成的集合将被编译器用来完成重载函数的解析过 程（overload function resolution）。

尽管两段式查询机制本质上使分离式模型（separation model）的实现成为可能，但这种机制也 用于置入式模型（inclusion model）。然而在很多早期的内置式模型实作品中，所有查询过程都 被推迟至 POI。

10.3.5 举例

拿出一些实例来，可以较有效地解释我们先前所讲述的内容。第一个例子是置入式模型（inclusion model）的简单情况：

```
template<typename T>
void f1(T x)
{
   g1(x); //(1)
}

void g1(int)
{
}

int main()
{
   f1(7);   // ERROR：找不到 g1()
}           // (2)f1<int>(int)的 POI
```

调用 f1(7)会制造出一个 POI，位于 main()的紧临外侧（(2)处）。这个实例化过程的关键在
于函数 g1()的查询。当 template f1的定义式首次出现，编译器会注意到非受饰名称 g1是受 控的（dependent），因为它是「带有受控自变量（dependent arguments；自变量 x 的类型取决于 template parameter T之函数调用」的函数名称。因此编译器在(1)处使用 *ordinary lookup* 规则来查询 g1， 然而此刻 g1不可见。在(2)处（亦即 POI），编译器再次于相应的 namespaces 和 classes 中查询 g1， 但因 g1()的惟一自变量是 int类型，而 int类型没有相应的 namespaces 和 classes，所以编译器 最终没有找到 g1的完整类型 — 即使发生于 POI 的 *ordinary lookup* 确实找到了 g1的名称。

第二个例子示范：跨编译单元时，分离式模型（inclustion model）可以导致重载歧义（overload ambiguities）。此例由三个文件组成，其中一个是头文件：

```
// common.hpp文件
export template<typename T>
void f(T);

class A {
};
class B {
};

class X {
public:
    operator A() { return A(); }
    operator B() { return B(); }
};

// a.cpp 文件
#include "common.hpp"

void g(A)
{
}

int main()
{
    f<X>(X());
}

// b.cpp文件
#include "common.hpp"

void g(B)
{
}

export template<typename T>
void f(T x)
{
    g(x);
}
```

在文件 a.cpp中，main()调用 f<X>(X())。f是个 exported template，定义于文件 b.cpp，

其内的调用动作 g(x)因而被自变量类型 X 实例化。g()被查询两次：第一次使用 *oridnary lookup* 规则在文件 b.cpp 中查询（当 template 被 *parsing* 时），第二次使用 *ADL* 规则在文件 a.cpp中查询（那是 template 实例化动作所在）。第一次查询找到 g(B)，第二次查询找到 g(A)。两个结果都合理（因为 class X内有对应的两个使用者自定转型运算符），因此这个调用带有歧义性。

注意，文件 b.cpp中的 g(x)调用动作看起来一点问题也没有。完全是因为两段式查询机制才带来额外的候选函数。因此，撰写 export templates 程序代码和其说明文件时，一定要极度小心。

10.4 实作方案（Implementation Schemes）

译注：以下以 obj 表示目标文件（object files），lib 表示库（libraries），exe 表示可执行文件（executable files）。

本节之中我们将回顾普及的 C++ 编译器产品实现置入式模型（inclustion model）的方法。所有这些产品都倚赖两个典型组件：编译器和链接器。编译器用来将源码转译为 obj 文件，其内包含带符号标注（symbolic annotations；用以交叉引用其它 obj 文件和 lib 文件）的机器码。链接器用来合并 obj 文件、解析（*resolving*）obj 文件所含之符号式交叉引用（symbolic cross-references），最终生成 exe 文件或 lib 文件。接下来的内容中，我们假设你的编译系统使用这种模型，当然以其它方式实作C++ 编译系统也是完全可能的但不普及）例如你可以想象一个C++ 直译器interpreter）

当一份 class template 特化体被用于多个编译单元中，编译器会在处理每个编译单元时重复具现化。这不会引发什么问题，因为编译器并不直接根据 class 定义式产生机器码。C++ 编译器只是在其内部用到这些具现体，以便检验或解释其它各种算式和声明。从这个角度说，class 定义式的多次实例化，与 class 定义式在多个编译单元被多次含入（典型情况是透过含入文件）并无本质上的区别。

然而，如果你实例化一个 noninline function template，情况可能有所不同。如果你提供了一个常规的 noninline function 的多份定义，你将违反 ODR（单一定义原则）。假设你要编译并链接一个程序，而它由下面两个文件组成：

```
// a.cpp文件
int main()
{
}

// b.cpp文件
int main()
{
}
```

C++ 编译器会顺利编译两个文件，不会遭遇任何问题，因为它们确实都是合法的 C++ 编

译单 元。然而如果你试图链接两个 obj 档，链接器几乎肯定要发出抗议，因为你在同一个程序中重 复定义了 main()，这是不允许的。

对比地，考虑 template 的情况：

```
// t.hpp文件
// 普通的头文件（置入式模型）
template<typename T>
class S {
public:
    void f();
};

template<typename T>
void S::f()    // 成员定义
{
}

void helper(S<int>*);

// a.cpp文件
#include "t.hpp"

void helper(S<int>* s)
{
    s->f();    // (1) S::f的第一个 POI（具现点）
}

// b.cpp文件
#include "t.hpp"

int main()
{
    S<int> s;
    helper(&s);
    s.f();    //(2) S::f的第二个 POI
}
```

如果链接器以「对待常规函数或常规成员函数的方式」来处理 templates 的实例化成员，那么编译器便需确保在两个POI处只生成一份程序代码：不是(1)处就是(2)处，但不能两处都生成程序码。为了达到这个目标，编译器不得不把某个编译单元的信息带到其它单元，而这在 templates 概念被引入之前，编译器是绝对无需这么做的。接下来的内容将讨论三种主要解决办法，它们 在各种C++ 编译器实作品中运用极广。

注意，同样问题也会出现在 template 实例化过程所产生的所有可链接物（linkable entities）上：实例化的 function templates 和 member function templates，以及实例化的 static 成员变量。

10.4.1 贪婪式实例化（Greedy Instantiation）

头一个广泛运用「贪婪式实例化」机制的编译器，出自于 Borland 公司。后来这种机制逐渐成为各种 C++ 编译系统中最常用的技术，特别是在 Microsoft Windows 开发环境中，它几乎成为所有编译系统使用的机制。

「贪婪式实例化」机制假设，链接器知道某些物体（特别是可链接之 template 具现体）可能在各个 obj 文件和 lib 文件中存在多份。编译器通常会以特殊方法在这些物体上作出标记。当链接器发现存在多个重复具现体时，它只保留一个，并将其它具现体全部丢弃。就是这么简单。

「贪婪式实例化」在理论上有一些严重缺点：

- 编译器可能生成并优化 N 个 template 具现体，最后却只保留一个。这会严重浪费时间。
- 通常链接器并不检查两个具现体是否完全相同（identical），因为一个 template 特化体的多个实体之间往往存在一些不重要的差异，这是合法的。这些小差异不会导致链接失败（只是可能在实例化时造成编译器的内部数据有所不同）。然而这也经常导致链接器注意不到更大的差异，例如某个具现体可能在编译期针对最大效能进行优化，另一个具现体在编译期含入更多除错信息。
- 和其它机制对比，「贪婪式实例化」机制可能会产生总长度大得多的 obj 文件，因为相同的程式码可能重复出现多次。

这些缺点实际并不是什么大问题。这可能是因为「贪婪式实例化」机制在一个重要方面明显优于其它机制：传统的「源码-目标文件」（source-object）依存关系得以保留，特别是：一个编译单元只生成一个 obj 文件，每个 obj 文件都包含对应源码文件中的所有可链接定义（其中包括实例化后的定义）。

最后一个值得注意的问题是，如果链接机制「允许多个可链接物（linkable entities）的定义存在」，这个机制通常可被用来处理重复出现到处散落的 inline 函数和虚拟函数分派表（virtual function dispatch tables）。当然，如果不支持这种机制，也可能使用其它机制以内部链接方式（internal linkage）产生这些东西，代价是产生出来的 obj 档比较大。

10.4.2 查询式实例化（Queried Instantiation）

此类机制最普及的实作品来自 Sun Microsystems 公司，第一个支持此机制的是其编译器 4.0 版。「查询式实例化」概念上极其简单优雅，如果按发生时间排列，它也是我们回顾的这些实例化机制中最晚出现的。在此机制中，编译器维护一个由程序各编译单元共享的数据库（database）。这个数据库可用来追踪「哪些特化体在哪个相关源码文件中被实例化」。生成的所有具现体也会连同这些信息被储存于数据库中。当编译器遇到一个可链接物（linkable entity）的 POI（具现点）时，可能会发生以下三种情况之一：

1. 找不到其特化体：这种情况下，编译器会进行实例化过程，产生的特化体被存入数据库。
2. 找到了其特化体，但已过期（out of date），也就是特化体被生成后源码又被改动过。和 1. 相同，编译器于是进行实例化过程，然后把新生成的特化体存入数据库中。
3. 在数据库中找到了最新（up-to-date）特化体。编译器于是什么都不用做。

虽然概念上看来简单，但实作时这种设计会带来一些难题：

编译器需要正确地根据源码状态来维护数据库中的各种依存关系，而这并非垂手可得。虽然「把第三种情况误当作第二种情况来处理」并不算错误，但这会增加编译器的工作量，也就是增加整个编译时间。

并行（concurrently）编译多份源码如今已是颇为平常的事了。那么，编译系统的实作者必须 提供一个工业强度的数据库，恰当地控制各种并行情况。

尽管存在这些困难，这个机制还是可以实作出极高效率。而且也不存在什么情况会使这种解决 办法无法处理大规模程序。相反地，如果采用「贪婪式实例化」机制，处理大规模程序时会产 生许多任务效（工时）上的浪费。

不幸的是，数据库的使用会带给程序员一些问题。多数问题的根源在于，继承自大多数 C 编译器的传统编译模型，此处不再适用，因为单一编译单元不再产生单一的独立 obj 文件。举个例子，

假设你想要链接你的最终版本的程序，链接操作中不仅需要编译单元相应的各个 obj 文件，也需
要储存于数据库中的 obj 文件。同样道理，如果你建立一个二进制 lib 文件，你必须确保生成该文件 的工具程序（通常是 linker 或 archiver）能够感知数据库的内容。更常见的情况是，所有 obj 文件 操作工具都需要感知数据库的内容。许多这些问题都可以透过「不要将具现体保存于数据库」 而得以缓和，换之以另一种方式：吐露出 obj 文件中「首先引发实例化」的 obj 码。

lib 文件是另一个大难题。很多编译器生成的特化体可能会被打包到一个 lib 文件中。当另外一个专 案使用这个 lib 时，该项目的数据库必须感知到这些既有的特化体。如果无法感知，编译器就会 为此项目产生出它自己的 POI（具现点）。然而实际上这些特化体已经存在于 lib 文件中，于是造 成重复实例化。一个可能解决这个问题的策略是：使用类似「贪婪式实例化」的链接技术，让 链接器可以感知已经生成的特化体，并可清除所有重复特化体（相对于「贪婪式实例化」策略 来说，这种情况较少出现）。使用其它精妙的安排方式来组织源码文件、obj 文件或 lib 文件，还是可 能引发令人沮丧的问题，例如由于「含有所需具现体之 obj 文件未被链接至最终 exe 文件」，导致联 结器抱怨找不到某些具现体。诸如此类的问题不该被看作是「查询式实例化」机制的缺点，实 在是由于程序开发环境过于复杂和微妙。

10.4.3 迭代式实例化（Iterated Instantiation）

第一个支持C++ templates 的编译器是 Cfront 3.0，这是 Bjarne Stroustrup 写来发展C++ 语言的 编译器的直系后裔。Cfront 的一个硬性条件是：必须具备高度移植性。这意味(1)它在所有目 标平台上都以 C 语言为共通表述；(2)它使用平台提供的链接器。更明确地说这意味链接器对 template 一无所知。事实上 Cfront 将 template 具现体生成为常规 C 函数，因此它必须避免产出 重复的具现体尽管 Cfront 的源码模型source model不同于标准的置入式模型inclusion model） 和分离式模型（separation model），但是它的实例化策略可加以改编适

应置入式模型。也因此 它被人们誉为「迭代式实例化」机制的第一个化身。Cfront 的「迭代式实例化」机制描述如下：

1. 编译源码，但不实例化任何所需的可链接特化体（linkable specialization）。
2. 使用一个「预链接器」（prelinker）链接 obj 文件。
3. 预链接器调用链接器，并解析其错误信息以便得知是否遗漏任何具现体。如果真有遗漏，预 链接器会调用编译器，编译那些「内含所需 template 定义」的文件。调用时带一个选项，让 编译器产出先前遗漏的具现体。
4. 如果产出任何具现体定义，则重复 3.。

之所以需要迭代（反复）步骤 3，因为一个可链接物（linkable entity）的实例化过程可能引发另一个物体被实例化。最终，迭代过程会使所有实例化需求都获得满足，从而让链接器成功生成一个完整的可执行文件。

原始的 Cfront 方案有一些严重缺点：

链接时间大大膨胀了，不仅因为预链接器的额外开销，每次必要的再编译和再链接也需要高额的时间代价。以 Cfront 为基础之编译系统的用户们说，如果使用其它实例化机制用掉大约一个小时，那么使用「迭代式实例化」机制会花掉好几天。

诊断信息（错误信息和警告）被推迟至链接期。漫长的链接时间对程序员而言非常痛苦，他 们可能不得不等上数小时之久而最后发现只是 template 定义式有个笔误。

编译器必须特别记住内含特定定义的源码位于何处（前述步骤 1）。明确地说，Cfront 使用 了一个集中式容器，它必须实现出类似「查询式实例化」机制中的数据库机能。而且，最初
的 Cfront 实作品并不支持并行编译（concurrent compilations）。

尽管有这些不足，仍有两个 C++ 编译系统使用改良后的「迭代式实例化」机制，并开拓了更高 级的C++template 特性，它们是 Edison Design Group（EDG）的版本和 HP 的 aC++ 版本。 接下来我将介绍 EDG 开发的技术，展示其C++ 前端（front-end）技术。

EDG 的迭代方式是在预链接器和各个编译步骤之间建立双向通信：预链接器可将「特定编译单元所需的实例化」透过一个「实例化申请档」（instantiation request file）转给编译器处理；藉由「在 obj 文件中嵌入某些信息」或「产生个别的 template 信息文件」，编译器有能力告诉预链接器可 能的 POI 位置。「实例化申请文件」及「template 信息文件」与对应之待编译档同名，只不过分别 带有尾词.ii和.ti。迭代机制按以下步骤工作：

1. 编译某个编译单元的源码时，EDG 编译器读取对应的.ii文件（如果存在的话），并产生需要 的具现体。同时它将「原本可兑现的 POI」（译注：兑现意指实现.ii文件中的实例化请求） 写入当前编译产生的 obj 文件，或写入到一个独立的.ti 文件中。同时也将这个文件如何编译 的信息写入。
2. 链接步骤被预链接器拦截。预链接器检查参与链接的 obj 文件和对应的.ti文件，如果发现有「尚
未产生的具现体」，就把请求写入可兑现此一请求的编译单元所对应的.ii文件中。
3. 如果预链接器察觉到任何.ii 文件被更改过，就重新调用编译器（步骤 1）来处理对应的

源码 文件，然后迭代（反复）预链接步骤。

4. 当所有具现体都被产出完毕（迭代完成），链接器便会进行实际的链接工作。 这个机制解决了并行建造（concurrent build）问题，它把所有用到的信息分置于各编译单元中。
使用这种机制时，链接时间仍然可能远远超过使用「贪婪式实例化」或「查询式实例化」机制所用掉的时间，但由于链接工作实际并未进行（只是「预链接」而已），时间的增长并不如想 像中那般可怕。更重要的是，由于链接器在各个.ii 文件中保留了相关信息，下次编译、链接程 式时，这些文件仍然可用。特别是在对源码进行一些修改后，程序员会重新编译、链接这些修 改后的文件，这时所引发的编译动作就可根据上次编译留存的.ii 文件，立刻进行实例化工作。 如果运气好，链接期需要重新编译的可能性并不大。

现实中 EDG 方案的表现非常好。虽然一个彻头彻尾的编译链接过程仍然比其它机制耗用更多时间，但后续的编译链接速度很有竞争力。

10.5 明确实例化（Explicit Instantiation）

明确指定（产生）某 template 特化体的 POI（具现点）也是可能的，称为「明确实例化」指令。语法上规定，「明确实例化」指令由「关键词 template」及「待实例化之特化体的声明」构成。例如：

```
template<typename T>
void f(T) throw(T)     //译注：最后面那个 throw(T)就是"exception spec."（异常规格）
{
}

// 四个合法的「明确实例化」指令：
template void f<int>(int) throw(int);
template void f<>(float) throw(float);
template void f(long) throw(long);
template void f(char);
```

注意，上述所有实例化指令都合法。template arguments 可被编译器推导而出（见第 11 章），异 常规格（exception specifications）可省略。如果它们没被省略，就必须和 template 的那一个匹配。

class templates 的成员也可以这么地明确实例化：

```
template<typename T>
class S {
public:
    void f() {
    }
};

template void S<int>::f();
template class S<void>;
```

不仅如此，明确实例化一个 class template 特化体，会使其所有成员都被明确实例化。

许多早期C++ 编译系统刚加入对 template 的支持能力时，并不具备自动实例化的能力。某些系 统要求：一个程序用到的 function template 特化体必须在某个独一无二的位置上由程序员手动进 行实例化。手动式实例化通常以编译器实作品专属之#pragma指令完成。

为了这样的原因，C++ *Standard* 以清晰的语法「法定化」这种具现指令。C++ *Standard* 并指出，在一个程序中，每个 template 特化体最多只能有一次明确实例化。而且如果一个 template 特化 体被明确实例化（explicit instantiated），就不能被明确特化（explicit specialized）；反之亦然。

在手动式实例化的原始含义下，上述限制似乎并不会带给人们任何烦恼，然而实际上它们可能让我们吃到苦头。

首先考虑一个库实作者发布了某个 function template 的第一版：

```
// toast.hpp文件
template<typename T>
void toast(T const& x)
{
    ...
}
```

客户端程序代码可以含入这个头文件，并明确实例化这个 template：

```
// 客户端程序代码
#include "toast.hpp"

template void toast(float);
```

不幸的是，如果库作者也将 toast<float> 明确实例化了，便会使客户端程序代码非法。如果这个库是由不同厂家实作出来的标准库，更有可能发生问题。有些实作品明确具现 化了一些标准 templates，另一些实作品没有这么做（抑或它们明确具现化的是另一些标准 templates），于是客户端程序代码无法以一种可移植性的方式来对库组件进行明确实例化。本书撰写之际（2002），C++ 标准委员会似乎倾向于：如果一个明确实例化指令（explicit instantiation directive）接在一个针对相同物体的明确特化（explicit specialization）之后，那么该指令不具任何效果（此议尚未定论，而且有可能被否决，因为技术上似乎不可行）。

Templates 明确实例化的当前限制的第二个挑战是：如何提高编译速度。很多 C++ 程序员都发 觉，template 自动实例化机制对编译时间有不小的负面影响。一种改善编译速度的技术是：在某 些位置上手动实例化某些特定的 template 特化体，而在其它所有编译单元中不做这些实例化工 作。惟一一个具有可移植性并能阻止「在其它单元发生实例化」的方法是：不要提供 template 定义式，除非是在它被明确实例化的编译单元内。例如：

```
// 编译单元 1
template<typename T> void f(); // 无定义：避免在本单元中被实例化
```

```
void g()
{
    f<int>();
}

// 编译单元 2
template<typename T> void f()
{
}

template void f<int>();        // 手动实例化


void g();

int main()
{
    g();
}
```

这个解法运作良好，但程序员必须拥有「提供 template 接口」之源码文件的控制权。通常程序员无法拥有这种控制权。「提供 template 接口」的源码文件无法被修改，而且它们通常会提供 templates 定义式。

一个有时候被用到的技巧是，在所有编译单元中声明某 template 特化体（这可禁止该特化体被 自动实例化），惟有在「该特化体被明确实例化」的编译单元中例外。为示范这种作法，让我们修改前一个例子，用以含入一个 template 定义式：

```
// 编译单元 1
template<typename T> void f()
{
}

template<> void f<int>();       // 声明但不定义


void g() {
    f<int>();
}

// 编译单元 2
template<typename T> void f()
{
```

```
}

template void f<int>();          // 手动实例化


void g();

int main()
{
   g();
}
```

不幸的是这里假设「对一个明确特化之特化体的调用」等价于「对一个相匹配之泛型特化体的调用」。这个假设并不正确。很多 C++ 编译器为这两个物体生成了不同的重整名称（mangled names）。如果使用这些编译器，obj 码就无法被链接成一个完整的 exe 文件。

某些编译器提供了一种扩展语法，用来向编译器指示：一个 template 特化体不应该在其编译单元内被实例化。一个普及（但不符合标准）的语法形式是：在一个「明确实例化指令」之前添加关键词 extern。如果编译器支持这种语法，前例中的第一个源码档可改变如下：

```
// 编译单元 1
template<typename T> void f()
{
}

extern template void f<int>(); // 声明但未定义


void g()
{
   f<int>();
}
```

10.6 后记

本章讨论了两个相关但有区别的问题：C++ template 编译模型和各种C++template 实例化机制。

编译模型（compilation model）用来在程序的各个编译阶段决定一个 template 的意义。更明确地 说，它决定了「template 被实例化」时其内各个构件的意义。名称查询（name lookup）当然是 编译模型中最基本的部份。当我们谈论置入式（inclusion）模型和分离式（separation）模型时， 我们就是在谈论编译模型。这些模型是语言定义的一部份。

实例化机制是定义于 C++ 语言之外的机制，它使 C++ 编译器可以正确产生具现体。这些机制可能受制于链接器或其它开发工具。

然而最早的（Cfront）templates 编译器实作品超越了这两个概念。它为 template 具现体产生新的 编译单元，并使用特殊的源码文件组织方式。这些新编译单元使用「本质上为置入式模型」的 方式编译（虽然其「名称查询规则」与置入式模型所使用者大有不同）。因此尽管 Cfront 没有 实作出 templates 分离式模型，但它使用隐式置入式模型，产生分离式模型的幻觉。很多后来的 实作品（例如 Sun Microsystems）预设使用「隐式置入式模型」机制，或把这种机制作为可选方 式（例如 HP、EDG），给那些使用 Cfront 开发的程序代码提供一定的兼容性。

下面例子展示 Cfront 实作方案的细部：

```
// template.hpp文件
template<class T>     // Cfront不支持关键词 typename
void f(T);

//File template.cpp:
template<class T>     // Cfront不支持关键词 typename
void f(T)
{
}

// app.hpp文件
class App {
    ...
};

// main.cpp文件
#include "app.hpp"
#include "template.hpp"

int main()
{
    App a;
    f(a);
}
```

在链接期，Cfront 的「迭代式实例化」机制会产生一个新编译单元，其中含入一些文件，它期 望那些文件含有它「在头文件中发现的 templates」的实作码（定义式）。Cfront 的习惯是将表 头文件扩展名.h（或类似者）替换为.c（或.C或.cpp）。这时它生成的编译单元变成：

```
// main.cpp文件
#include "template.hpp"
#include "template.cpp"
```

```
#include "app.hpp"

static void _dummy_(App a1)
{
   f(a1);
}
```

这个编译单元会在编译期获得一个特殊参数，禁止为「定义于 included file 中的任何物体」生成 obj code。这将使得「含入 `template.cpp`」这一事实不会造成「`template.cpp` 之中（可能）含有的任何可链接物（linkable entities）产生重复定义」。

函数`_dummy_()`被用来建立一个 reference，指向「必须被实例化」的那个特化体。请注意表头 文件的顺序重新排过了：Cfront 使用「头文件分析程序」（header analysis code）去除新编译单元
中无用的头文件。不幸的是，由于可能存在「跨头文件作用域」的宏，导致这个技术并不稳 固。

与之对比的是，标准「C++ 分离式模型」将两个（或多个）编译单元分别编译，并使用 ADL 跨越编译单元，产生一个「可使用各编译单元内之物体」的具现体。由于这种机制并不倚赖档 案含入关系（*not based on inclusion*），也就不倚赖特殊的头文件组织（排列）方式，而编译单 元中的宏也不会「污染」其它编译单元。然而就像本章先前所说，宏并不是 C++ 中惟一带 来惊奇的东西，「汇出模型」（export model）是另一种形式的「污染」。

11

Template 自变量推导

Template Argument Deduction

如果你在每一个 function template 调用语句中明确指定 template arguments（例如 `concat<std:: string, int>(s, 3)`），程序代码会显得笨拙又难看。幸运的是 C++ 编译器通常可以自动判定 你所要的 template arguments 的类型，这是透过一个名为 template argument deduction（模板自变量 推导）的强大机制完成的。

本章详细解说 template argument deduction 细节。正如其它C++ 特性一样，很多推导规则所产生 的结果都符合人们的直观认知。本章带来的扎实基础可以让我们避免对更多情况感到惊讶。

11.1 推导过程（Deduction Process）

编译器会比对「调用语句内某自变量的类型」和「function template 内对应的参数化类型」，并试图 总结出「被推导的一或多个参数」的正确替换物。每一对「自变量-参数」独立分析。如果推导结 束时结果有异，则推导失败。考虑下面这个例子：

```
template<typename T>
T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

int g = max(1, 1.0);
```

在这里，第一个 call argument 的类型为 `int`，因此 `max()`的 template parameter `T`被暂时推导为 `int`。然而第二个 call argument 的类型为 `double`，因此 `T`应该为 `double`：这与第一个推导结果 冲突。注意，我们说的是「推导失败」而不是「程序非法」；推导程序还是有可能成功找到另一个名为 `max`的 template（因为 function templates 可以重载，就像常规 functions 那样；见 2.4 节和第 12 章）。

即使所有的 template parameters 推导结果前后一致，但如果以自变量替换进去后会导致函数声明的 其余部份变为非法，推导过程还是有可能失败。例如：

```
template<typename T>
typename T::ElementT at (T const& a, int i)
{
    return a[i];
```

```
}

void f (int* p)
{
    int x = at(p, 7);
}
```

在这里 T被推断为 int*（T只在一个参数类型中出现，因此显然不会发生冲突）。然而，将回 返类型 T::ElementT中的 T替换为 int* 显然是非法的，因此推导失败。错误讯息大约会说『找不到与调用语句 at()匹配的东西』。但如果你明确指定所有 template arguments，推导程序就 没有机会成功匹配另一个（可能存在的）template，这时的错误讯息会是『at()的 template argument 非法』。你可以使用你喜欢的编译器，把下面例子的错误诊断讯息和前一例做个对比：

```
void f (int* p)
{
    int x = at<int*>(p, 7);
}
```

我们还需要探究「自变量-参数」匹配是如何进行的。今后的描述将出现符号 A和 P，意义如下： 将类型 A（由 argument type 导出）匹配至一个参数化类型 P（由 template parameter 的声明导出）； 如果参数被声明为以 *by reference* 方式传递，则 P表示被指涉（*referenced*）类型，A 为自变量型 别；否则 P表示声明之参数类型，A是个「由 array 或 function 退化而成[49]」并「去除外围之 const
和 volatile饰词」的 pointer 类型。例如：

```
template<typename T> void f(T);      // P为 T

template<typename T> void g(T&);     // P仍为 T

double x[20];

int const seven = 7;

f(x);        // nonreference parameter：T为 double*
g(x);        // reference parameter：T为 double[20]
f(seven);    // nonreference parameter：T为 int
g(seven);    // reference parameter：T为 int const
f(7);        // nonreference parameter：T为 int
g(7);        // reference parameter：T为 int => 错误：不能把 7当做 int&传递
```

对于调用语句 f(x)，x由 array 类型退化为 double*类型，T被推导为该类型。调用语句 f(seven)
中，const饰词被卸除，从而 T被推导为 int。调用语句 g(x)中，T被推导为 double[20]（并无 发生类型退化）。类似情况，g(seven)的自变量是一个 int const 类型的左值（lvalue），

此外 由于 const 和 volatile 饰词在匹配 reference parameters 时不被卸除，因此 T 被推导为 int const。然而请注意调用语句 g(7)中 T被推导为 int（因为 nonclass rvalue 表达式不能含有带 const或 volatile饰词的类型），因此调用失败，因为自变量 7不能传给一个 int& 类型的参数。

当字符串字面常数（string literals）类型的自变量被系结至 reference parameters 时，不会发生退化， 这可能使你惊讶。重新考虑先前的 max() template：

```
template<typename T>
T const& max(T const& a, T const& b);
```

很多人希望在 max("Apple", "Pear")表达式中，T 被推导为 char const*，这合乎常情。然而 "Apple" 的类型却是 char const[6]，"Pear" 的类型是 char const[5]。array 类型至 pointer 类型的退化过程并未发生（因为 max的参数使用 *by reference* 传递方式），因此如果推导成功，T必须既是 char[6]又是 char[5]，那当然不可能。请参考 5.6 节，那里有更多相关讨论。

11.2 推导之前后脉络（Deduced Contexts）

比 T更复杂的参数化类型也可以被匹配到一个给定的自变量类型上。下面是一些基本例子：

```
template<typename T>
void f1(T*);

template<typename E, int N>
void f2(E(&)[N]);

template<typename T1, typename T2, typename T3>
void f3(T1 (T2::*)(T3*));   // 译注：灰色部分即是 pointer-to-member type
     //译注：意思是：f3的参数是个 T2成员函数，该函数的回返类型为 T1，接受一个 T3*参数。


class S {
public:
   void f(double*);
};

void g (int*** ppp)
{
   bool b[42];
   f1(ppp);      // T被推导为 int**
   f2(b);        // E被推导为 bool，N被推导为 42
   f3(&S::f);    // 推导结果：T1=void，T2=S，T3=double
}
```

复杂的类型声明是由基本构件（pointer、reference、array、function 声明符号、pointer-to-member 声明符号、template-id 等等）搭建而成的，匹配过程从上层构件开始，在各组成元素中递归进行。公正地说，大多数「类型声明构件」（type declaration constructs）都可以用这种方式进行匹配，这些构件被称为 deduced contexts（推导之前后脉络）。有些构件并不是 deduced contexts：

> 受饰类型（qualified type）名称。例如 `Q<T>::X`不会被用以推导 template parameter `T`。
>
> 「非单纯只是个非类型参数（nontype parameter）」的一个非类型算式（nontype expression）。例如类型名称 `S<I+1>` 不被用于 `I` 的推导；推导机制也不认为 `T` 能够与 `int(&)[sizeof(S<T>)]` 的参数匹配。

译注：pointer-to-member 类型包括两个组成：(1) member class 和(2) member type。例如在 `int* C::*` 这一 pointer-to-member 类型中，(1) 是 `C`而 (2) 是 `int*`。两者都是 deducible contexts。

不必对这些限制太过惊讶，因为通常说来推导结论并不惟一（甚至不是有限个），虽然受饰型别（qualified）的名称有时很容易被忽视。一个 nondeduced context 并不意味程序有错，甚至不意味正被分析的参数不能用于类型推导。考虑下面这个更复杂的例子：

```
// details/fppm.cpp

template <int N>
class X {
public:
    typedef int I;
    void f(int) {
    }
};
template<int N>
void fppm(void (X<N>::*p)(typename X<N>::I));
     // 译注：意思是：fppm的参数是个「X<N> 成员函数」，
     //       该函数的回返类型为 void，接受一个 X<N>::I参数。


int main()
{
    fppm(&X<33>::f); // 正确：N被推导为 33
}
```

在 function template `fppm()`中，子构件 `X<N>::I`是个 nondeduced context。然而 pointer-to-member 类型中的 `X<N>` 却是个 deducible context，而且当由此推导出的参数 `N`被塞入 nondeduced context 时，我们将获得一个与实元`&X<33>::f`相容的类型。于是「自变量-参数」推导成功。

对一个「完全以 deduced contexts 构建而成」的参数类型进行推导，反而可能导致矛盾结果。以

下例子假设 class templates X和 Y都已正确声明：

```
template<typename T>
void f(X<Y<T>, Y<T> >);

void g()
{
    f(X<Y<int>, Y<int> >());   // OK
    f(X<Y<int>, Y<char> >()); // ERROR：推导失败
}
```

第二个 function template f()调用语句的问题在于，根据两个自变量推导出不同的 template parameter T，这是不合法的。上述两种情况中，call arguments 都是「由 class template X的 *default* 构造函数建 立」的一个暂时对象（temporary object）。

11.3 特殊推导情境（Special Deduction Situations）

有两种情形使得**(A,P)**并非由 function template 的调用自变量和 function template 的参数获得。第一 种情况发生在对 function template 取址时。这时 **P**是 function template 声明语句中的参数化类型，**A** 是被初始化或被赋值之指针的「台面下（underlying）函数类型」。例如：

```
template<typename T>
void f(T, T);

void (*pf)(char, char) = &f;    //译注：取 function template 的地址
```

这个例子中，**P**是 void(T, T)，**A**是 void(char, char)，推导成功，并将 T替换为 char，pf 被初始化为「f<char>特化体的地址」。

另一种特殊情形发生在 conversion operator templates（转型运算符模板）。例如：

```
class S {
public:
    template<typename T, int N> operator T&();
};
```

这时**(P,A)**的获得犹如涉及某自变量（其类型等于「吾人企图转换过去之目标类型」）及某参数型别（也就是「转型运算符之回返类型」）。下面例子展示了上述情况的一个变异：

```
void f(int (&)[20]);

void g(S s)
{
    f(s);
}
```

这里我们试图把 S转型为 int(&)[20]。因此类型 A为 int[20]，类型 P为 T。推导成功，T被替换为 int[20]。

11.4 可接受的自变量转型（Allowable Argument Conversions）

通常，template 推导机制会试图寻找 function template parameters 的替换品，使参数化类型 P与类型 A完全相同。然而当它无法达到这个目标时，推导机制容许以下差异：

如果原始参数使用 *by reference* 传递方式，则被替换类型 P比类型 A可以多带上 const饰词 或 volatile饰词。

如果类型 A是一个 pointer 类型或 pointer-to-member 类型，A可以透过（添加）一个 const

饰词或 volatile饰词，转型为被替换类型 P。

除非推导程序因 conversion operator template 而发生，否则被替换类型 P可以是类型 A的 base class 类型；或者当 A是个 pointer-to-X 类型时，P可以是个 pointer-to-base-class-of-X。例如：

```
template<typename T>
class B {
};

template<typename T>
class D : public B<T> {
};

template<typename T> void f(B<T>*);

void g(D<long> dl)
{
    f(&dl); // 推导成功：T被替换为 long
}
```

当推导机制无法进行完全匹配时，才会考虑这种宽松的匹配条件。即使如此，惟有当推导机制找到惟一一个「使类型 A符合被替换类型 P」的替换方式时，推导才会成功。

11.5 Class Template Parameters（类别模板参数）

Template argument deduction 专只用于 function templates 和 member function templates。更明确地说，class template 的自变量不从外界对其某一构造函数的 call arguments 中推导出来。例如：

```
template<typename T>
```

```
class S {
public:
   S(T b) : a(b) {
   }
privat
   e:
   T
   a;
};

S x(12);   // 错误：class template parameter  T不从构造函数的 call argument  12中推导出来
```

11.6 预设的调用自变量（Default Call Arguments）

在 function templates 中你也可以指定预设的函数调用自变量，就像在常规函数中一样：

```
template<typename T>
void init (T* loc, T const& val = T())
{
   *loc = val;
}
```

事实上，正如这个例子所展示，预设的 funcion call arguments 也可以倚赖某个 template parameter。这种受控预设自变量（dependent default argument）只在调用端没有使用明确自变量时才会被实例化。正是这个法则使得下面的例子合法：

```
class S {
public:
   S(int, int);
};

S s(0, 0);

int main()
{
  init(&s, S(7, 42));
      // 当 T=S时 T()不合法，但由于给定了一个明确自变量，因此不需将预设自变量 T()实例化
}
```

即使预设的 funcion call arguments 并非受控的（dependent），它也无法被用来推导 template arguments。这意味下面这个例子不合法：

```
template<typename T>
void f (T x = 42)
```

```
{
}

int main()
{
    f<int>();      // OK: T = int
    f();           // ERROR: 无法从预设的 call argument 推导 T
}
```

11.7 Barton-Nackman Trick

1994 年 John J. Barton 和 Lee R. Nackman 展现一种 template 技术，他们称之为 restricted template expansion（受限的模板扩展技术）。此技术的动机源于一个事实：当时的 function template 不能 被重载，并且大多数编译器不支持 namespaces。

为阐示这种技术，假设我们有一个 class template Array，并为它定义一个 *equality* 运算子
operator==。一种可能是将这个运算符定义为 class template 的成员，但这不是个好办法，因为 第一自变量（系结于 this 指针）的类型如果和第二自变量不同，两者的转型规则也不同。由于 operator== 应该符合交换律，把它定义为 namespace 作用域下的函数会更好些。一个很自然的 实作法可能看起来这个样子：

```
template<typename T>
class Array {
public:
    ...
};

template<typename T>
bool operator == (Array<T> const& a, Array<T> const& b)
{
    ...
}
```

如果 function templates 不能被重载，这就引发一个问题：在此作用域中不能定义其它的
operator== template，然而其它 class templates 很可能也需要定义这样一个 operator==。Barton 和 Nackman 解决这个问题的办法是：把 operator==定义为 class 内的一个普通的 friend 函数：

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

```
    friend bool operator == (Array<T> const& a,
                             Array<T> const& b) {
        return ArraysAreEqual(a, b);
    }
};
```

假设这个 Array被实例化为 float类型，导致 friend operator function 被声明，但是注意，这个
函数本身并非是某个 function template 的具现体。它只是个普通的 non-template function，因具现 化过程的影响而被插入 global 作用域中。由于它是个 non-template function，可以和其它重载形 式的 operator== 共存，即使当时 C++ 语言尚未能够支持 function template 重载。Barton 和 Nackman 把这种技术称为 restricted template expansion（受限的模板扩展），因为它可以避免使 用一个适用于所有类型的 operator==(T,T)（也就是一个 unrestricted expansion, 毫不受限的扩 展）。

由于 operator== (Array<T> const&, Array<T> const&)定义在 class 定义式内，也就隐隐 成为一个 inline 函数，因此我们将这个函数的实作委托（*delegate*）给一个 function template ArraysAreEqual()，后者不必须为 inline，不致于和同名的其它 templates 起冲突。

Barton-Nackman trick 的最初目的已经消失了,但研究它仍然很有趣因为它允许我们在生成 class template 具现体的同时，生成 non-template functions。由于后者并非由 function templates 产生， 所以无需 template argument 的推导，但是仍然符合正常的重载解析规则（见附录 B）。理论上 这意味当把一个 friend 函数匹配到一个特定调用场所（call site）时，匹配过程会考虑进行隐式 转型。然而这并没带来什么明显好处，因为在标准 C++ 中（这和 Barton 及 Nackman 发明该手 法时的C++ 已是今非昔比），被植入到 global 作用域中的 friend 函数并非无条件地在其外围作 用域中可见：只有 ADL 适用时它们才可见。这意味调用语句中的自变量必须将「含有该 friend 函数
之 class」做为相关 class。如果函数自变量是个不相关的 class 类型，但可被转型为「含有该 friend 函数之 class」，这个 friend 函数也是不可见的。例如：

```
class S {
};

template<typename
T> class Wrapper
{ private:
    T object;
public:
    Wrapper(T obj) : object(obj) {   // 可从 T隐式转型到 Wrapper<T>
    }
    friend void f(Wrapper<T> const& a) {
    }
```

```
};

int main()
{
    S s;
    Wrapper<S> w(s);
    f(w);     // OK：Wrapper<S>是 w的相关联 class
    f(s);     // ERROR：Wrapper<S>与 S不相关联
}
```

在这个例子中，调用语句 `f(w)`是合法的，因为函数 `f()`是声明于 `Wrapper<S>` 内的一个 friend 函 式而 `Wrapper<S>`与自变量 `w`相关联。然而调用 `f(s)`时friend 函数声明 `f(Wrapper<S>const&)` 并不可见因为 class `Wrapper<S>`与类型为 `S`的自变量 `s`不相关联即使存在一个从 `S`到 `Wrapper<S>` 的隐式转换（透过 `Wrapper<S>` 构造函数），这个转换也不会被考虑，因为候选函数 `f` 并没有先 被找到。

结论是，与「简单定义一个常规的 function template」相较，「在 class template 内定义一个 friend 函数」并没有多大好处。

11.8 后记

针对 function template 而做的 template argument deduction 是C++ 原始设计的一部份。事实上直到很多年后，由 explicit template arguments 带来的另一种自变量推导机制才成为C++ 的一部份。

「Friend 名称植入（name injection）」被许多C++ 语言专家认为有害，因为它使「程序合法性」过于受到「实例化的顺序」的影响。Bill Gibbons（当时致力编写 Taligent 编译器）代表解决这 个问题的主要声音，因为去除了「实例化顺序」的依存关系后，会使得新颖而有趣的 C++ 开发 环境成为可能（听说 Taligent 正为此而努力）。然而 Barton-Nackman trick 需要某种形式的「friend 名称植入（name injection）」，正是因此才使得后者以目前的（弱化后的）形态留存于C++ 语 言至今。

有趣的是，很多人都听说过 Barton-Nackman trick，但很少有人知道它的实际描述。于是你会发现很多涉及 friends 及 templates 的技巧都被误称为 Barton-Nackman trick（例如 16.5 节）。

12
特化与重载

Specialization and Overloading

目前为止，我们已经学习了 C++ 如何使一个泛型定义被展开为一族系相关的 classes 或 functions。虽然这个机制强劲有力，在许多情形下对某些操作而言，「以一个特定替换物取代泛 化的 template paramters」远远达不到优化的要求。

就各流行编程语言对泛型编程的支持程度而言，C++ 可说是相当独特，因为它提供了一组丰富 的特性供程序员透通地将一个泛型定义替换为一个更特定（更专门）的定义。本章中我们将学 习两种机制，它们将纯粹的泛型导向更多实用价值，它们是：(1) template 的特化 (2) function templates 的重载。

12.1 当泛型码（Generic Code）不合用...

考虑下面例子：

```
template<typename
T> class Array
{ private:
    T* data;
    .
..
publi
c:
    Array(Array<T> const&);
    Array<T>& operator = (Array<T> const&);

    void exchange_with (Array<T>* b)
        { T* tmp = data;
        data = b->data;
        b->data = tmp;
    }
    T& operator[] (size_t k) {
        return data[k];
    }
    ...
};
```

```
template<typename T>
inline void exchange (T* a,
T* b)
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}
```

对于简单类型，exchange()的泛型实作码可以有效运作。然而对于「*copy* 操作代价高昂」的型 别而言，这个泛型实作码可能带来更为高昂的代价：既增加 CPU 负担，也占用更多内存。这 远不如一个量身而制的实作码来得高效。在这个例子中，泛型实作码需要一个对 Array<T> *copy* 构造函数的调用，以及两个对其 *copy assignment* 运算符的调用。对大型数据结构来说，这些 *copy* 过程往往涉及海量存储器复制。然而 exchange()的功能往往可以利用「交换两个内部指针」就 达成，就像成员函数 exchange_with()所作所为那样。

12.1.1 透通订制（Transparent Customization）

前面的例子中，成员函数 exchange_with()为泛化的 exchange()函数提供一个有效的特殊实 作，但是从数种理由来看，使用另一个函数很不方便：

1. class Array的使用者不得不记住它有个额外的界面，并且得花心思在必要时使用它。
2. 实作泛型算法时，你通常无法区分何时该使用这个额外接口，何时不用它。例如：

```
template<typename T>
void generic_algorithm(T* x, T* y)
{
    ...
    exchange(x, y); // 我们如何选择正确的算法？
    ...
}
```

出于这些考虑，C++ templates 提供了一些办法，让我们得以透通地订制 function templates 和 class templates。对于 function templates 来说，你可以借助重载机制达到目的。例如我们可以写出一组 重载的 quick_exchange() function templates 如下：

```
template<typename T> inline
void quick_exchange(T* a, T* b)                    // (1)
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}
```

```
template<typename T> inline
void quick_exchange(Array<T>* a, Array<T>* b)      // (2)
{
    a->exchange_with(b);
}

void demo(Array<int>* p1, Array<int>* p2)
{
    int x, y;
    quick_exchange(&x, &y);                       // 使用(1)
    quick_exchange(p1, p2);                       // 使用(2)
}
```

对 quick_exchange()的第一次调用有两个 int* 自变量，推导机制发现只有第一个 template（声明 于(1)处）适用，于是将 T 替换为 int，不存在「究竟该调用哪个函数」的疑虑。第二次调用匹 配两份 templates：「把第一个 template 中的 T替换为 Array<int>」或是「把第二个 template 中的 T 替换为 int」都可行。不仅如此，两种替换方法引发的函数参数类型都与第二次调用中的 自变量类型匹配。通常这会使我们认为这个调用有歧义性，然而（稍后会有讨论）C++ 语言认为 第二个 template 比第一个 template 更特别（more specialized），因此在其它因素平手的情况下， 更特别（更特化）的 template 胜出。

12.1.2 语意的透通性（Semantic Transparency）

上一节所说的重载机制可以帮助我们实现实例化过程的透通订制，但认识到以下这一点也是很 重要的：这种透通性 倚赖诸多实作细节。为说明这个问题，请考虑先前的 quick_exchange() 方案。尽管两个泛型算法以及我们为 Array<T> 订制的算法最终都能交换两个指针所指的值，但不同的操作带来的副作用差异很大。

考虑下面这个有点过于戏剧化的例子。请比较「struct objects 之间的互换」和「Array<T> 之间的互换」：

```
struct S {
    int x;
} s1, s2;

void distinguish (Array<int> a1, Array<int> a2)
{
    int* p = &a1[0];
    int* q = &s1.x;
    a1[0] = s1.x = 1;
    a2[0] = s2.x = 2;
    quick_exchange(&a1, &a2); // 调用后*p仍然为 1
```

```
    quick_exchange(&s1, &s2); // 调用后*q为 2
}
```

这个例子显示，在调用 quick_exchange()之后，「指向第一个 Array」的指针 p 改而「指向 第二个 Array」。然而调用 quick_exchange()之后，「指向 non-Array s1」的指针仍然指向 s1， 只是指针所指的值互换了。这个差别已经足够迷惑这个 template 的用户。前导词 quick_ 倒是 颇能吸引人们注意力：有一条快捷方式可以实现你所渴求的操作。然而最初的泛化 exchange() template 也可以对 Array<T> 进行优化：

```
template<typename T>
void exchange(Array<T>* a, Array<T>* b)
{
    T* p = &(*a)[0];
    T* q = &(*b)[0];
    for (size_t k = a->size(); k-- != 0; ) {
        exchange(p++, q++);
    }
}
```

这个版本相对泛化版本的优势在于，不需要一个大型的瞬时 Array<T>。exchange() template 被递归调用，因此即使面对 Array<Array<char> > 它也能有不错的表现。另外请注意，这个 较为特化的 template 版本并没有被声明为 inline，因为它内部做了相当数量的工作；而原始的泛 化版本是 inline，因为它只进行少数几个操作（但每个操作的代价都有可能很高）。

12.2 重载 Function Templates

前一节中我们知道，同名的两个 function templates 可以共存，即使它们可能在被实例化后拥有 完全相同的参数类型。下面是个小例子：

```
// details/funcoverload.hpp

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}
//译注：对于调用语句 f((int*)0)，VC7.1/g++ 3.2可以正确解析为
```

```
//        调用第二个 function template；然而 VC6认为此一调用模棱两可（带有歧义）。
```

当第一个 template 中的 T被替换为 int*，如果把第二个 template 中的 T替换为 int，我们将得 到「参数类型和回返类型完全相同」的两个函数。不光是这些 templates 可以共存，它们的具现 体（即使拥有完全相同的参数和回返类型）也可以共存。

下面这个例子展示：以明确指定的 template argument 语法生成两个完全相同的函数：

```
// details/funcoverload.cpp

#include <iostream>
#include "funcoverload.hpp"

int main()
{
    std::cout << f<int*>((int*)0) << std::endl;
    std::cout << f<int>((int*)0) << std::endl;
}
```

输出结果为：

```
1
2
```

为了搞清楚这个例子，让我们详细分析调用语句 f<int*>((int*)0)[52]。语法 f<int*>表示我们要
将 template f的第一个 template parameter 替换为 int*,而不借助于自变量推导机制本例的 template f 不止一个，于是编译器产出一个「内含两个函数」的重载集合：从第一个 template 中产出 f<int*>(int*)，从第二个 template 中产出 f<int*>(int**)。call argument (int*)0 的类型为 int*，只匹配重载集合中的第一个函数，因此最终调用的是第一个函数。

你可以使用类似方法来分析第二个调用语句。

译注：以下三个小标题：12.2.1 Signatures, 12.2.2 Partial Ordering of Overloaded Function Templates 和 12.2.3 Formal Ordering Rules，均未译。这些原文名词在英文书籍、杂志、网站、新闻组或论 坛上被大量使用，硬译并无好效果。但盼谅解。

12.2.1 Signatures（署 名 式 ）

两个函数可以共存于一个程序中，只要彼此的 signatures 不同。我们定义函数的 signature 包含以下信息[53]：

1. 函数的非受饰（*unqualified*）名称（或生成该函数之 function template 的名称）

2. 该名称的 class/namespace 作用域；以及它所声明于其中的编译单元（如果该名称是内部链接 方式）
3. 该函数所带的 const、volatile 或 const volatile 饰词（如果它是一个成员函数并带有上述饰词）
4. 函数参数的类型（如果该函数是从一个 function template 生成，此处指的是 template paramters 被替换前的类型）
5. 回返类型（如果函数从一个 function template 生成）
6. template parameters 和 template arguments（如果函数从一个 function template 生成） 这意味着下面的 templates 和其具现体原则上可共存于一个程序中：

```
template<typename T1, typename T2>
void f1(T1, T2);

template<typename T1, typename T2>
void f1(T2, T1);

template<typename T>
long f2(T);

template<typename T>
```

char f2(T); 然而，当这些函数被声明于同一个作用域（scope）中，它们无法被使用。因为只要实例化其中 任意两个，就会造成重载歧义性（overload ambiguity）。例如：

```
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
    std::cout << "f1(T1, T2)" << std::endl;
}

template<typename T1, typename T2>
void f1(T2, T1)
{
    std::cout << "f1(T2, T1)" << std::endl;
}

// 目前还好


int main()
{
    f1<char, char>('a', 'b'); // 错误：有歧义
}
```

在这里，函数 f1<T1=char, T2=char>(T1,T2)可以和函数 f1<T1=char,T2=char>(T2,T1)共存，但是重载解析（overload resolution）机制永远无法得知哪一个更好。如果这些 templates 出现在不同的编译单元中，那么两个具现体可共存于同一个程序中（链接器不会抱怨有重复定义，因为这些具现体的 signatures 是可以区分的）：

```
// 编译单元 1
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
    std::cout << "f1(T1, T2)" << std::endl;
}

void g()
{
    f1<char, char>('a', 'b');
}

// 编译单元 2
#include <iostream>

template<typename T1, typename T2>
void f1(T2, T1)
{
    std::cout << "f1(T2, T1)" << std::endl;
}

extern void g(); // 定义于编译单元 1

int main()
{
    f1<char, char>('a', 'b');
    g();
}
```

这个程序合法，其输出为：

```
f1(T2, T1)
f1(T1, T2)
```

12.2.2 Partial Ordering of Overloaded Function Templates

（重载化函数模板的偏序规则）

重新考虑先前的例子：

```
#include <iostream>

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}

int main()
{
   std::cout << f<int*>((int*)0) << std::endl;
   std::cout << f<int>((int*)0) << std::endl;
}
```

我们发现，在替换了给定之 template argument list（<int*> 和 <int>）后，重载机制最终正确 选择了它要调用的函数。然而即使不提供明确的 template arguments，重载机制也会选中某个函 式；这种情况下自变量推导机制加入战局。我们对 main()函数稍做修改来讨论这种机制：

```
#include <iostream>

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}

int main()
{
```

```
    std::cout << f(0)       << std::endl;
    std::cout << f((int*)0) << std::endl;
}
```

考虑第一个调用语句 `f(0)`：自变量类型为 `int`，与第一个 template 的参数类型匹配（如果将 `T` 替换
为 `int`）。至于第二个 template 的参数类型是个指针，不可能匹配。因此推导程序完成后惟一 的候选函数是第一个 template 产生的具现体。在此情况下，重载解析机制的作用不大。

第二个调用语句 `f((int*)0)`比较有趣：自变量推导机制成功作用于两个 templates 身上，并产生 `f<int*>(int*)`及 `f<int>(int*)`两个函数。从传统的重载解析（overload resolution）角度看， 两个函数对于「以 `int*` 为引数」的呼叫式匹配程度相同，这也许使你认为这里发生歧义
（ambiguous）。然而在这种情况下，另一个重载解析准则参与了进来：「更特化」的 template 所产出的函数将被选用。在这里，第二个 template 被认为更特化（稍后讨论），因此，这个例 子的输出结果仍然是：

```
1
2
```

12.2.3 Formal Ordering Rules（正序规则）

上个例子中你可以直观看出第二个 template 比第一个更特化，因为第一个 template 几乎可以容 纳任意类型，而第二个 template 只接受指针类型。然而其它情形就未必这么直观。接下来将讲 述在一组重载函数中如何「判断某个 function template 比另一个更特化」。然而亦请注意所谓的 **partial ordering rules**（偏序规则）：给定的两个 templates 有可能谁也不比谁更特化。如果重载 机制必须在这样的两个 templates 之间作出选择，那它做不了任何决定：程序带有歧义性错误。

假设比较两个同名的 function templates ft_1 和 ft_2，它们都适用于某个给定的调用语句。被「预设 自变量」和「省略号参数（ellipsis parameters）」涵盖的那些参数不在 **ordering rules** 考虑之内。 接下来我们合成两组自变量类型（针对转型运算符则还包括一个回返类型），并逐一以下面方式 替换 template parameters：

1. 将每个 type template parameter 替换为一个独一无二的「编造类型」（"made up" type）。
2. 将每个 template template parameter 替换为一个独一无二的「编造出来的」class template。
3. 将每个 nontype template parameter 替换为一个独一无二且在相称类型之下的编造值」("made up" value）。

如果「以第一组合成类型对第二个 template 进行自变量推导」产生出一个完全匹配，反之不然， 我们便说第一个 template 比第二个更特化。相反地，如果「以第二组合成类型对第一个 template 进行自变量推导」产生出一个完全匹配，反之不然，我们便说第二个 template 比第一个更特化。 当两个推导都成功或都失败时，两个 templates 之间无顺序可言（no ordering）。

让我们更具体说明上面所讲的内容，把它运用到前例的两个 templates 身上。在那两个 templates 中我们合成两组自变量类型，并以上述所谈方式去替换 template parameters。两组类型分别为

(A1) 和(A2*)，这里的 A1和 A2都是独一无二编造得来的类型。很明显，将 T替换为 A2* 后，「以 第二组自变量类型替换第一个 template」的推导是成功的。但我们无法「将第二个 template 的 T*匹 配于第一列自变量中的非指针类型 A1」。因此我们得出结论：第二个 template 比第一个更特化。

最后，考虑一个更复杂的例子，涉及多个函数参数：

```
template<typename T>
void t(T*, T const* = 0, ...);

template<typename T>
void t(T const*, T*, T* = 0);

void example(int* p)
{
    t(p, p);
}
```

首先，因为实际调用语句中并未用到第一个 template 的「省略号参数」，而第二个 template 的最
后一个参数被其预设自变量顶替因此在 partial ordering 中这些参数都被忽略注意第一个 template 的预设自变量未被使用，因此对应的参数会参与到 ordering 之中。

合成的自变量类型为(A1*,A1 const*)和(A2 const*,A2*)。前者对第二个 template 的推导过程 成功，以 A1 const替换 T，但引发的匹配并不完全，因为以类型(A1*,A1 const*)调用 t<A1 const>(A1 const*,A1 const*,A1 const*=0)时，需要一个 const 饰词。类似情况， 以第二组自变量类型(A2 const*,A2*)推导第一个 template 的过程中，也无法找到完全匹配。因 此两个 templates 之间没有顺序关系（no ordering），这个调用带有歧义性。

Formal ordering rules（正序规则）通常可以导致直观的选择。然而，曾经有一个例子显示，这些规则偶尔也会与你的直观选择相悖。C++ 标准委员会将来很有可能修订这些规则，使它们与人们的直觉更相符。

12.2.4 Templates 和 Nontemplates

Function templates 可以被 non-template function 重载。在其它因素相同的情形下，编译器会优先 选择 non-template function。下面例子展示这一点：

```
// details/nontmpl.cpp
#include <string>
#include <iostream>

template<typename T>
```

```
std::string f(T)
{
    return "Template";
}
std::string f(int&)
{
    return "Nontemplate";
}
int main()
{
    int x = 7;
    std::cout << f(x) << std::endl;
}
```

程序输出：

```
Nontemplate
```

12.3 明确特化（显式特化；Explicit Specialization）

我们可以重载 function templates，并结合 partial ordering rules 以选择最匹配的 function templates，这就使我们可以为一个泛型实作品加入更多的特化 templates，以便透通实现程序代码的优化，得到更高效能。然而 class templates 无法被重载，它的订制可采用另一种机制：明确(显式)特化。标准术语「明确特化」也就是某些人所谓的「全特化」(full specialization)语言特性。它提供
一种 template 实作方式：将所有 template parameters 替换掉，不留任何 template parameters。class templates 和 function templates 都可以被全特化，定义于 class 定义式之外的 class template 成员(例如成员函数、嵌套类别（nested classes）和 static 成员变量）也可以被全特化。

稍后我们将描述偏特化（partial specialization）。它与全特化类似，但与「替换所有 template parameters」不同的是，允许在 template 的某一份实作品中保留部份参数化特性。全特化和偏特化在程序代码中都是「明确指定」的，这就是为什么我们避免使用「明确特化」这个术语的原因。全特化和偏特化都不会引入全新的 template 或 template 具现体。这些构件为「泛化（非特化）template 之中已被隐寓声明的实体提供另一种定义这是一个相当重要的概念也是与 overloaded templates 的一个关键区别。

12.3.1 Class Template 全特化（Full Specialization）

「全特化」系藉由「三个符号所组成的序列」引入：template, < 和 >[54]。紧跟在 class 名称宣告之后的是 template arguments。下面例子展示这一点：

```
//译注：以下是泛化（非特化）
```

```
template<typename T>
class S {
public:
   void info() {
      std::cout << "generic (S<T>::info())" << std::endl;
   }
};

//译注：以下全泛化
template<>
class S<void> {
public:
   void msg() {
      std::cout << "fully specialized (S<void>::msg())" << std::endl;
   }
};
```

注意，全特化的实作不必与泛化定义有任何关联。这使我们得以拥有不同名称的成员函数（`info` 和 `msg`）。两者（全特化与泛化）之间只靠 class template 名称相连。

全特化所指定的 template argument list 必须与 template parameter list 对应。如果你对一个 template type parameter 指定一个 nontype 值，是不合法的。然而，对于带有 default template argument 的 那些 template parameters，你可以不予理会：

```
template<typename T>
class Types {
public:
   typedef int I;
};

template<typename T, typename U = typename Types<T>::I>
class S;                              // (1)

template<>
class S<void> {                       // (2)
public:
   void f();
};

template<> class S<char, char>;       // (3)

template<> class S<char, 0>;          // ERROR：不能以 0替换 U
```

```
int main()
{
    S<int>*       pi;       // OK：使用(1)，无需定义
    S<int>        e1;       // ERROR：使用(1)，但找不到其定义
    S<void>*      pv;       // OK：使用(2)
    S<void,int>   sv;       // OK：使用(2)，其定义可用
    S<void,char>  e2;       // ERROR：使用(1)，但找不到其定义
    S<char,char>  e3;       // ERROR：使用(3)，但目前找不到其定义
}

template<>
class S<char, char> {      // (3)的定义
};
```

这个例子展示的是，全特化 template 的声明语句不必须为定义式。然而当一个全特化体被声明后，对于给定之 template arguments 集合，泛化定义式就不再会被使用。因此如果没有提供所需定义，程序会出错。撰写 class template 特化码时，前置声明（forward declaration）很有用，使你得以 声明彼此相依（mutually dependent）的类型。全特化声明语句与常规 class 声明语句完全相同（也就 是说前者并不被视为一个 template 声明），惟一区别是声明的语法，而且该声明必须与先前的 template 声明匹配。由于它不是一个 template 声明，所以 class templates 全特化体可以使用常规
的 out-of-class 成员定义语法（但也就是说你不能使用 template<> 前导词）：

```
template<typename T>
class S;

template<> class S<char**> {
public:
    void print() const;
};

// 以下定义式的前端不能出现 template<>
void S<char**>::print() const
{
    std::cout << "pointer to pointer to char" << std::endl;
}
```

下面这个稍复杂一些的例子也许可以强化你的印象：

```
template<typename
T> class Outside
{ public:
```

```
    template<typename U>
    class Inside {
    };
};

template<>
class Outside<void> {
    // 下面的 nested class 不必与其泛化版 template 的定义式有任何关连
    template<typename
    U> class Inside
    { private:
        static int count;
   };
};

// 以下定义式的前端不能出现 template<>
template<typename U>
int Outside<void>::Inside<U>::count = 1;
```

所谓全特化就是某确凿之泛化 template 具现体』因此同一个程序中不能同时存在一个template 明确具现体」和一个由前者产生的具现体。如果同时使用两者，编译器会抓出你的小辫子：

```
template <typename T>
class Invalid {
};

Invalid<double> x1;    // 引发 Invalid<double> 被实例化


template<>
class Invalid<double>; // 错误：Invalid<double> 已被实例化
```

不幸的是，如果你在不同的编译单元中这么写，编译器很难找出问题。下面是一个非法的 C++ 例子，由两个文件组成。它可以顺利通过很多编译器和链接器，但它不但非法而且危险：

```
// 编译单元 1
template<typename
T> class Danger
{ public:
    enum { max = 10 };
};

char buffer[Danger<void>::max]; // 使用泛化值
```

```
extern void clear(char const*);

int main()
{
    clear(buffer);
}

// 编译单元 2
template<typename T>
class Danger;

template<>
class Danger<void> {
public:
    enum { max = 100 };
};

void clear(char const* buf)
{
    // array 边界不匹配（不吻合）
    for (int k = 0; k < Danger<void>::max; ++k) {
        buf[k] = '\0';
    }
}
```

这个例子刻意维持短小，但它说明：你必须小心确保「特化体的声明对泛化 template 的所有使用者都可见」。现实而言，这意味特化体的声明通常应该在头文件中紧跟着其泛化 template 的 声明。但是当泛化实作源自一个外部源码文件（因此你无法修改其对应头文件），上述说法就不 太实际。不过你还是可以建立一个头文件，含入泛化 template，然后是其特化体的声明，这就可 以避免这些难以发现的错误。我们发现，最好避免特化「外来源码所含的 templates」，除非对 方明确标示其设计目标就是如此。

12.3.2 Function Template 全特化（Full Specialization）

function template（明确）全特化的语法和原则非常类似 class template 全特化，只是你需要附加考虑重载（overloading）和自变量推导（argument deduction）。

当被特化之 template 可经由自变量推导（亦即将声明语句中的「参数类型」视为「自变量类型」）决 定时，全特化声明语句中可以省略不写明确的 template arguments：

```
template<typename T>
```

```
int f(T)                    // (1)
{
    return 1;
}

template<typename T>
int f(T*)                   // (2)
{
    return 2;
}

template<> int f(int)       // OK：(1)的特化
{
    return 3;
}

template<> int f(int*)      // OK：(2)的特化
{
    return 4;
}
```

function template 的全特化体不能含有预设自变量值。然而明确特化时，仍然适用 template 的预设自变量：

```
template<typename T>
int f(T, T x = 42)
{
    return x;
}

template<> int f(int, int = 35) // 错误！全特化体不能含有预设自变量值。
{
    return 0;
}

template<typename T>
int g(T, T x = 42)
{
    return x;
}
```

```
template<> int g(int, int y)
{
    return y/2;
}

int main()
{
    std::cout << g(0) << std::endl; // 输出 21
}
```

全特化声明在很多方面类似于一个普通声明（或说「再声明」，redeclaration）。更明确地说，它并不是声明一个 template，因此程序中的 noninline function template 全特化定义只能出现一份。但我们仍然必须确保全特化声明语句紧跟在 template 之后，以求防范使用该 template 产生的函数。前例中的 template g因此往往声明于两个文件中。接口文件（头文件）可能看来像这样：

```
#ifndef TEMPLATE_G_HPP
#define TEMPLATE_G_HPP

// template 定义式应该写在头文件中
template<typename T>
int g(T, T x = 42)
{
    return x;
}

// 特化体的声明将会抑制 template 的实例化
// 不应该将定义写在这里，用以避免重复定义
template<> int g(int, int y);

#endif //TEMPLATE_G_HPP
```

对应的实作档是这样：

```
#include "template_g.hpp"

template<> int g(int, int y)
{
    return y/2;
}
```

另外，特化体可以声明为 inline。在这种情况下，其定义可以（并且应该被）写在头文件中。

12.3.3 Member 全特化（Full Specialization）

不仅是 member templates，连 class templates 内的 static 成员变量/成员函数也可以被全特化。语 法规定你必须在每个圈封的（enclosing）class template 前加上 template<> 前导词。如果你特化
一个 member template，也必须使用 template<> 指出它是被特化的。为了更佳说明这些规定，假设有以下定义：

```
template<typename T>
class Outer {                           // (1)
public:
    template<typename U>                // member template
    class Inner {                       // (2)
    private:
        static int count;               // (3)
    };
    static int code;                    // (4)
    void print() const {                // (5)
        std::cout << "generic";
    }
};

template<typename T>
int Outer<T>::code = 6;                 // (6)

template<typename T>
template<typename U>
int Outer<T>::Inner<U>::count = 7;      // (7)

template<>
class Outer<bool> {                     // (8)
public:
    template<typename U>
    class Inner {                       // (9)
    private:
        static int count;               // (10)
    };
    void print() const {                // (11)
    }
};
```

泛型 template Outer（位于(1)）的常规成员 code（位于(4)）和 print()（位于(5)）只有一
个圈封的（enclosing）class template，因此需要以 template<>为前导，针对一组特定的 template

arguments 进行全特化：

```
template<>
int Outer<void>::code = 12;          //译注：这是一个定义


template<>
void Outer<void>::print() const      //译注：这是一个定义
{
    std::cout << "Outer<void>";
}
```

这些定义被用于(4)和(5)泛化版本中特定的 Outer<void>。Outer<void> 的其它成员仍将由泛化版本（位于(1)）产生。请注意，提供这些定义之后，你就不能再提供 Outer<void> 的一个明确特化体（explicit instantiation）了。

和 function templates 全特化一样，我们需要一种方式用以声明 class template 的常规成员的特化 体，但不指定任何定义（以防止重复定义）。虽然C++ 不允许常规 class 的成员函数和 static 成 员变量存在着「非定义之 out-of-class 声明语句」但是当特化 class templates 成员时它们是合法的。 前面的定义可以被声明为：

```
template<>                    //译注：这是一个非定义声明语句（nondefining declaration）
int Outer<void>::code;

template<>                    //译注：这是一个非定义声明语句（nondefining declaration）
void Outer<void>::print() const;
```

用心的读者可能会指出， Outer<void>::code 全特化体的「非定义声明语句」（ nondefining declaration），和一个「以 *default* 构造函数进行初始化」的定义式，语法相同。的确如此，但这 类声明总是会被编译器理解为「非定义声明语句」。

因此，我们无法提供一个「只能以 *default* 构造函数初始化」的类型，又提供一个 static 成员变量 的全特化定义。

```
//译注：下面是个「只能以 default 构造函数初始化」的类型
class DefaultInitOnly {
public:
    DefaultInitOnly() {     //这是个 default ctor
    }
private:
    DefaultInitOnly(DefaultInitOnly const&);
            // 不允许 copying（因为刻意把 copy ctor 声明于 private 区）
};
```

```
template<typename
T> class Statics
{ private:
    static T sm;
};

// 下面是个声明：不存在任何 C++语法可为它提供一份定义
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm;
```

member template Outer<T>::Inner也可以经由一个给定的 template argument 进行特化，不会影 响某特定之 Outer<T> 具现体（我们正是在其中特化 member template Inner）中的其它成员。 由于此处只有一层圈封的 template，我们只需要一个 template<> 前导词。程序代码可被写为：

```
template<>
   template<typename X>
   class Outer<wchar_t>::Inner {
   public:
      static long count;   // 成员的类型改变了（译注：原为 int，现改为 long）
   };

template<>
   template<typename X>
   long Outer<wchar_t>::Inner<X>::count;
```

template Outer<T>::Inner 也可以被全特化，但只能在某个给定的 Outer<T> 实体中对它全特化。我们现在需要两个 template<> 前导词：一个是针对外层 class，另一个是针对内层 template：

```
template<>
    template<>
    class Outer<char>::Inner<wchar_t> {
    public:
        enum { count = 1 };
    };

// 下面程序代码不合法：template<>不能跟在一个 template parameter list 之后
template<typename X>
template<> class Outer<X>::Inner<void>; // ERROR!
```

你可以把这段程序代码和 Outer<bool>的 member template 特化体进行对照。由于后者已经被完全
特化，没有外层圈封之 template，因此只需要一个 template<> 前导词：

```
template<>
class Outer<bool>::Inner<wchar_t> {
public:
    enum { count = 2 };
};
```

12.4 Class Template 偏特化（Partial Specialization）

template 全特化非常有用，但如果我们想针对「一整个族系的 template arguments」而不是「特 定某一组 template arguments」来特化 class template，也是很自然的想法。例如，假设我们有个 class template，实作出一个 linked list：

```
template<typename T>
class List {       // (1)
public:
   ...
   void append(T const&);
   inline size_t length() const;
   ...
};
```

在使用上述 template 的大型项目中，可能会将其成员实例化为多种类型。对那些不是 inline 的成 员函数（例如 List<T>::append()）而言，这将导致 obj 码的不小膨胀。然而从底层角度看， List<int*>::append()的 obj 码和 List<void*>::append()的 obj 码是一样的。换句话说我 们希望所有「以指针为元素」的 Lists 共享一份实作码。尽管无法以C++表达这样的需求，但 我们可以指定所有「以指针为元素」的 Lists 都以另外一个 template 定义式进行实例化：

```
template<typename T>
class List<T*> {   // (2)
private:
    List<void*> impl;
    .
..
publi
c:
    ...
    void append(T* p) {
        impl.append(p);
    }
```

```
    size_t length() const {
        return impl.length();
    }
    ...
};
```

在本例情境中，(1)处的原始 template 称为 primary template，后一个定义称为 partial specialization（偏特化体；部份特化体），因为你只指定了这个 template 的一部份 template arguments。偏特化的语法是：template parameter list 的声明（template<...>）加上 class template 名称，再加上一组明确指定的 template arguments（本例为<T*>）。

上述程序代码有个问题：List<void*> 之中包含另一个 List<void*> 成员，构成递归声明（译注：当 T是 void时）。我们可以在偏特化之后加以一个全特化，终结这个递归圈：

```
template<>
class List<void*> {   // (3)
    ...
    void append (void* p);
    inline size_t length() const;
    ...
};
```

这种方法之所以可行，乃是因为全特化会被优先选用，然后才是偏特化。于是所有「以指针为 元素」的 Lists 的成员函数都经由 inline 函数转调用 List<void*> 实作码。这是一个有效对抗
所谓「程序代码膨胀」的办法。

偏特化声明语句的 parameter list（参数列）和 argument list（自变量列）存在诸多限制。其中一些如
下：

1. 偏特化 template 的自变量必须与其 primary template 的参数种类（type, nontype 或 template）逐一匹配。
2. 偏特化 template 的 parameter list（参数列）不能拥有预设自变量；它将使用 primary class template
   的预设自变量。
3. 偏特化 template 的 nontype arguments（非类型自变量）只能是非受控数值（non-dependent value）或简朴的（plain）nontype template parameters，不能是较复杂的受控运算式（dependent expressions）如 2*N（N是某个 template parameter）。
4. 偏特化 template 的 template argument list 不能与 primary template 的 template argument list 完全 相同（无论 template parameter 是否以其它名称出现）。

下面的例子展示上述限制：

```
template<typename T, int I = 3>
```

```
class S;                    // primary template

template<typename T>
class S<int, T>;            // 错误：参数种类不匹配


template<typename T = int>
class S<T, 10>;             // 错误：无预设自变量


template<int I>
class S<int, I*2>;          // 错误：不能是 nontype 表达式


template<typename U, int K>
class S<U, K>;              // 错误：与 primary template 无显著区别
```

所有的偏特化体和全特化体一样，与其 primary template 相关联。当你使用一个 template 时，编 译器总会首先查询 primary template，但随后各个自变量会与其关联的各个特化体进行匹配，决定 到底选用哪一份 template 实作码。如果有多个匹配结果，最特化（从 function templates 的重载 解析机制看来）那个会胜出。如果无法比较哪一份最特化，程序便被视为模棱两可（带有歧义）。

最后我们要指出，class template 的偏特化完全有可能比其 primary template 拥有更多或更少的参 数。再次考虑我们的泛化 template List（声明于(1)）。我们已经讨论过如何对「以 pointer 为 元素」的 Lists 进行优化，但我们也可能想对 pointer-to-member 类型做同样的事情。下面程序 码针对 pointer-to-member-pointers 达到同样的优化目的：

```
template<typename C>
class List<void* C::*> { // (4)
public:
    // 对任何 pointer-to-void*  member 的偏特化
    // 任何其它 pointer-to-member-pointer 类型将使用这一份偏特化定义
    typedef void* C::*ElementType;
    ...
    void append(ElementType pm);
    inline size_t length() const;
    ...
};

template<typename T, typename
C> class List<T* C::*> {// (5)
```

```
private:
    List<void* C::*> impl;
    .
..
publi
c:
    // 针对「先前处理过之 pointer-to-void* member 类型」以外
    //   的任何 pointer-to-member-pointer 类型进行特化
    // 这个特化体有两个 template parameters，而 primary template 只有一个 template parameter
    typedef T* C::*ElementType;
    ...
    void append(ElementType pm) {
        impl.append((void* C::*)pm);
    }
    inline size_t length() const {
        return impl.length();
    }
    ...
};
```

除了关注 template parameters 的个数，我们还要注意，定义于(4)的通用实作版本，是其它版本（如定义于(5)）的转调用（forwarding）目标，本身就是个偏特化（对简单指针而言则是个全特化）。然而很明显(4)要比(5)更特化，因此程序并没有歧义性。

12.5 后记

一开始template 全特化就是C++template 机制的一部份function template 的重载和 class template 的偏特化则是晚近才加入 C++。HP 的 aC++ 编译器最早实作出 function template 重载机制，EDG 的C++ 前端系统最早实作出 class template 偏特化机制。本章描述的 partial ordering rules（偏序 原则）源自 Steve Adamczyk 与 John Spicer 的发明（他们都在 EDG 工作）。

人们知道「如何终结一个无限递归之 template 定义」（例如之前出现之 List<T*>，见 12.4 节, p.200）已有一段不短的时间。然而 Erwin Unruh 可能是意识到「这可以引领出一种有趣的所谓 template metaprogramming 编程技法」的第一人；这种技法利用 template 实例化机制，在编译期 进行甚具意义的计算。第 17 章专门讲述这个主题。

你完全有理由感到奇怪：为什么只能偏特化 class templates? 这基本上是历史原因。技术上其实完全有可能为 function template 定义相同机制（请参考第 13 章）。某种程度上 function templates 的偏特化与 class templates 的偏特化类似，但两者之间仍有精微的差异。这些差异大多与以下事 实有关：当一个 template 被使用时，编译器只需查询 primary template；各个特化体是在后来才 被考虑，用以决定使用哪一份实作码。对比的是所有重载的 function templates 都会被带入一个 集合之中，供编译器查询；而这些 function templates 可能来自不同的 namespaces 或 classes。这 在某种程度上增加了「无意中重载一个 template 名称」的可

能性。

相反地，我们也可以想象出一种重载 class templates 的形式。下面是个例子：

```
// 非法重载 class templates
template<typename T1, typename T2> class Pair;
template<int N1, int N2> class Pair;
```

然而人们对这种机制似乎并没有迫切的需求。

13
未来发展方向
Future Directions

从 1988 年的最初设计，到 1998 年的标准化（其中技术工作于 1997 年 11 月完成），C++ templates 有长足的发展。此后数年 C++ 的语言定义趋于稳定。但是在那期间，人们对于 C++ templates 又有了各种新需求。其中有些是对语言的一致性（consistency）和正交性（orthogonality）的更高期望。例如，既然允许 class templates 有预设自变量，为什么不允许 function templates 也有预设自变量？其它的期待主要来自日渐复杂的 template 编程手法的要求，这些手法往往要求既有的编译器扩展自身能力。

后续我们将描述 C++ 语言及编译器设计者多次畅议的 C++ 语言扩展机能。这些扩展常常源自各种高级 C++ 库（包括 C++ 标准库）设计者的促使与提示。我们并不保证这里所列的任何一条将成为C++ *Standard* 的一部份，不过我们知道已有某些C++ 编译器实作品包含了其中一些扩展机能。

13.1 Angle Bracket Hack（角括号对付法）

一个最使 template 编程初学者感到惊讶的地方是：连续两个右角括号之间必须加入空格。例如：

```
#include <list>
#include <vector>

typedef std::vector<std::list<int> > LineTable;   // OK

typedef std::vector<std::list<int>> OtherTable; // 语法错误
```

第二个 typedef 是错误的，因为两个连续中间无空格的右角括号会形成一个右移运算符（>>），而那对本段程序代码并无意义。

然而「检测出这个错误，并悄悄地将 >>运算符当成两个右角括号」（这个特性有时被称为 **angle bracket hack**）比起C++ 源码词法解析器（parser）的其它功能来说算是相对简单的。事实上已有很多编译器可以辨识这种情形并正确分析程序代码（带着一个警告讯息）。

因此，未来版本的 C++ 语言很可能认为前例中的 `OtherTable` 声明语句合法。然而我们应该注意 angle bracket hack 中隐藏着一些隐微问题。确实存在这样的情形：>>运算符在 template argument list 中是合法记号。以下展示这一点：

```
template<int N> class Buf;

template<typename T> void strange() {}
```

```
template<int N> void strange() {}

int main()
{
    strange<Buf<16>>2> >(); // >>记号并不是错误
}
```

另一个某种程度上有所关连的问题是双字符记号 `<:` 的偶尔误用。这个记号等价于中括号(方括号)`'['`(见 9.3.1 节, p.129)。考虑下面程序代码片段:

```
template<typename T> class List;
class Marker;

List<:Marker>* markers; // 错误
```

例中最后一行会被当作 `List[:Marker>* markers;`,这将不具任何意义。编译器完全有理由注意到一个像 `List` 这样的 `template` 不可能后面跟一个 `'['`,这种情况下不应该把 `<:` 当作中括号(方括号)对待。

13.2 宽松的 typename使用规则

某些程序员和语言设计者认为关键词 `typename` 的使用规则(见 5.1 节和 9.3.2 节)过于严苛。例如下面程序代码中,`typename Array<T>::ElementT`里面的 `typename`必不可缺,但是却严禁在 `typename Array<int>::ElementT`里头使用 `typename`(否则就报错):

```
template <typename T>
class Array {
public:
    typedef T ElementT;
    ...
};

template <typename T>
void clear (typename Array<T>::ElementT& p);     // OK

template<>
void clear (typename Array<int>::ElementT& p);    // ERROR
```

这种例子令人惊讶。对 C++ 编译器实作品而言,忽略这个额外的 `typename` 关键词毫无困难,因此,语言设计者正考虑允许在「未被 struct/class/union/enum 等关键词标注之受饰型别名称
(qualified typename)」前使用关键词 `typename`。如果对此做出定论,也将有益于澄清「何时允许使用`.template`, `->template`和`::template`构件」的问题(见9.3.3节)。

从编译器实作者的角度来看，忽略这些额外的 typename 和 template 相对简单。有趣的是也存 在这种情形：语言要求某处必须使用这些关键词，然而如果程序员没有照做，某些编译器实作 版本也顺利放行。例如前面的 function template clear() 中，编译器知道 Array<T>::ElementT 不能是别的什么，只能是个类型名称（此处不允许任何表达式），这种情形下关键词 typename 可有可无。C++ 标准委员会也在检讨是否能够减少「必须使用关键词 typename 和 template」 的情形。

13.3 Function Template 的预设自变量

templates 最初加入C++ 语言时明确指定 function template arguments 就不被语言所容许Function template arguments 总是必须由调用语句推导得出。这就似乎没有什么理由允许 function template 有 预设自变量，因为默认值总是会被推导值覆写（overridden）。

既然如此，我们总该可以为「无法被推导得出」之自变量明确指定 function template arguments 吧。 因此，为那些「无法被推导得出」的 template arguments 指定默认值，也就成了很自然的想法。 考虑下面例子：

```
template <typename T1, typename T2 = int>
T2 count (T1 const& x);

class MyInt {
    ...
};

void test (Container const& c)
{
    int i = count(c);
    MyInt j = count<MyInt>(c);
    assert(j == i);
}
```

此例之中我们遵循一条约束条件：如果某个 template parameter 有预设自变量值，则它后面的所有 template parameters 也必须拥有预设自变量值。这个限制对 class templates 而言是必需的，否则就 无法在一般情况下指定吊尾（trailing）的自变量。下面的错误程序代码展示了这一点：

```
template <typename T1 = int, typename T2>
class Bad;

Bad<int>* b; // 这个 int是用来替换 T1还是 T2?
```

然而对 function templates 来说，吊尾自变量可由推导机制得出。因此不难改写本例如下：

```
template <typename T1 = int, typename T2>
T1 count (T2 const& x);

void test (Container const& c)
{
    int i = count(c);
    MyInt j = count<MyInt>(c);
    assert(j == i);
}
```

本书撰写之际，C++ 标准委员会正考虑在这方面对 function templates 进行扩展。

有些程序员事后注意到，不使用明确指定的（explicit）template arguments 也可以达到相同目的。

```
template <typename T = double>
void f(T const& = T());

int main()
{
    f(1);              // OK：推导出 T为 int
    f<long>(2);        // OK：T为 long，不需推导
    f<char>();         // OK：与 f<char>('\0')相
    同 f();            // 与 f<double>(0.0)相同
}
```

这里透过 default template argument 也实现了 default call argument，然而不需明确指定（explicit）template arguments。

13.4 以字符串字面常数（String Literal）和浮点数（Floating-Point）作为 Template Arguments

在 nontype template arguments 的限制中，最可能令 template 初学者和高级使用者感到惊讶的是，他们无法使用字符串字面常数作为 template argument。

下面是个看起来十分直观可行的例子：

```
template <char const* msg>
class Diagnoser {
public:
    void print();
};
```

```
int main()
{
    Diagnoser<"Surprise!">().print();
}
```

然而其中有些潜在问题。在标准 C++ 中， Diagnoser 的两个具现体只有当它们拥有相同的 template arguments 时，其类型才相同。本例的 template arguments 是个指针，或说是个地址。然 而位于不同源码文件中的两个内容相同的字符串字面常数并不一定落在同一地址上。我们可能会 因此尴尬地发现，Diagnoser<"X"> 与 Diagnoser<"X"> 是两个不同而且不兼容的类型！（注意 "X" 的类型是 char const[2]，当它被用作 template argument 时会退化为 char const*）

基于这些考虑（及其相关考虑），*C++ Standard* 禁止使用字符串字面常数作为 template arguments。 然而某些编译器会以扩展形式提供这项机能 — 他们使用字符串字面值作为 template 具现体的内 部表述（internal representation）。虽然这显然是可行的，但有些C++ 语言评论者认为，「可被 替换为字符串字面常数」之 nontype template parameter 的声明方式，应该与「可被替换为地址」之 nontype template parameter 的声明方式有所不同。本书撰写之际，并没有哪一种声明语法获 得压倒性的支持。

我们应该注意到这个问题中的一个小小的技术缺陷。考虑下面的 template 声明语句，并假设语言
已被扩展为「允许以字符串字面常数作为 template arguments」：

```
template <char const* str>
class Bracket {
public:
    static char const* address();
    static char const* bytes();
};

template <char const* str>
char const* Bracket<str>::address()
{
    return str;
}

template <char const* str>
char const* Bracket<str>::bytes()
{
    return str;
}
```

上述程序代码中，两个成员函数除了名称不同，其它完全相同（这种情形并不少见）。假设编译 器实作品使用类似宏（macros）展开的方式来实例化 Bracket<"X">，那么如果两个成

员函数 被实例化于不同的编译单元，它们会传回不同的值。有趣的是，我们测试了一些支持本节扩展
的C++ 编译器，发现它们也受到这种令人惊讶的行为的折磨。

一个与之相关的问题是使用浮点字面常数（以及简单的浮点常数运算式）作为 template arguments。例如：

```
template <double
Ratio> class Converter
{ public:
    static double convert (double val) {
        return val*Ratio;
    }
};

typedef Converter<0.0254> InchToMeter;
```

也有一些 C++ 编译器实作品提供了这个特性，它并不带来任何技术上的困难，不像以字符串字面 常数作为 template arguments 那样困难。

13.5 Template Template Parameters 的宽松匹配规则

一个 template 如果被用来「替换某个 template template parameter」，两者的 parameter list 必须匹 配。有时这会引发令人惊讶的结果，例如以下面例子所展示：

```
#include <list>
    // 声明:
    // namespace std {
    //    template <typename T,
    //              typename Allocator = allocator<T> >
    //    class list;
    // }

template<typename
        T1, typename
        T2,
        template<typename> class Container>
                                // Container期望的是仅带单一参数的 template
class Relation {
public:
    ...
privat
e:
```

```
    Container<T1> dom1;
    Container<T2> dom2;
};

int main()
{
    Relation<int, double, std::list> rel;
        // 错误：std::list拥有不止一个 template parameter
    ...
}
```

这个程序并不合法，因为我们的 template template parameter Container要求的是「仅有一个参数」的 template，然而 std::list除了有一个标示元素类型的参数外，还有个 allocator参数。

然而由于 std::list 的 allocator 参数有一个预设的 template argument，因此如果说 Container 被认为与 std::list匹配而且 Container的每一个具现体都将使用 std::list的预设 template argument（见 8.3.4 节, p.112），也是可能的。

赞成「维持现状」（不企图匹配）的一个论点是：function 类型的匹配也如此。然而在这个情况中，预设自变量并非总是能够被推导而出，因为一个 function 指针值必须直到运行期才能确定下 来。与之对比的是，并不存在所谓 template 指针，而且推导所需的所有信息在编译期都可取得。

某些编译器以扩展形式提供了宽松匹配原则（relaxed matching rule）。这个问题和 typedef
templates 有关（下一节讨论）。考虑以下面的程序代码替换前例的 main()定义：

```
template <typename T>
typedef std::list<T> MyList;

int main()
{
    Relation<int, double, MyList> rel;
}
```

typedef template 引入一个新的 template，它与 Container的 template pamameter list 完全匹配。这种情况究竟是强化还是弱化宽松匹配规则，尚有争议。

这个议题已向C++ 标准委员会提出，但委员会并不倾向于把宽松匹配规则加入C++ 语言。

13.6 Typedef Templates

Class templates 常以十分精巧的方式组合在一起，以求获得其它的参数化类型。当这种参数化型 别在源码中反复出现时，我们很自然会想要以一种简短形式（shortcut）来表示它，就像 typedef 为一般非参数化类型提供简短形式一样。

所以，C++ 语言设计者正在考虑这样一种构件：

```
template <typename T>
typedef std::vector<std::list<T> > Table;
```

有了这个声明后，Table将成为一个新的 template，可被实例化成为一个具体类型定义。这种 template 被称为 typedef template（相对于 class template 或 function template）。例如：

```
Table<int> t;       // t的类型为 vector<list<int> >
```

目前我们可以运用 class templates 的 member typedefs 来暂时取代 typedef templates。对于前面例 子我们可以这么写：

```
template <typename T>
class Table {
public:
    typedef std::vector<std::list<T> > Type;
};

Table<int>::Type t; // t将是一个 vector<list<int> > 类型
```

由于 typedef templates 也是一种具有完全特性的 templates，因此也可以像对待 class templates 一样地加以特化：

```
// primary typedef template
template<typename T> typedef T Opaque;

// 偏特化
template<typename T> typedef void* Opaque<T*>;

// 全特化
template<> typedef bool Opaque<void>;
```

typedef templates 的肌理并不是那么直观。例如人们尚未清楚在推导过程中它将如何作用：

```
void candidate(long);

template<typename T> typedef T DT;

template<typename T> void candidate(DT<T>);
```

```
int main()
{
    candidate(42); // 应该调用哪个 candidate()?
}
```

目前还不清楚这种情况下是否应该让推导成功。可以肯定的是，推导机制不能对任意的 typedef patterns 进行推导。

13.7 Function Templates 偏特化（partial specialization）

第 12 章我们讨论过，class templates 可以被偏特化，而 function templates 只能被重载。这两种机 制在某种程度上是不同的。

偏特化并不引入一个全新的 template：它只是既有之 template（所谓 primary template）的扩展。 当编译器查询一个 class template 时首先只考虑 primary templates. 如果编译器选中了某个 primary template 之后发现它有一个偏特化体，其 argument list 与具现体的 argument list 匹配，编译器会 实例化这个偏特化体（的定义），而不实例化 primary template。Template 全特化工作方式与此 完全相同）。

与之相反的是，重载化的 function templates 是彼此完全独立的 templates。当编译器准备选择某个 template 进行实例化时，所有重载化的 function templates 都被考虑，并由重载解析机制选择
一个最适任者。虽然这似乎也是个不错的第二选择，但实际上存在不少限制：

你可以特化一个 class 的 member templates，而无需修改这个 class 的定义。然而如果你增加一个重载成员，就必须改动 class 的定义。许多情况下我们之所以无法选用这个方法，因为 我们可能无权修改 class。而且目前的C++*Standard* 并不允许我们往 `std` namespace 添加新的 templates，但却允许我们特化 `std` namespace 中的 templates。

若要重载 function templates，各重载函数的参数就必须有实质上的差异。考虑一个 function template `R convert(T const &)`，这里 `R`和 `T`是 template parameters。我们可能非常想针对`R`为`void`的情况对它进行特化，然而这无法透过重载来实现（译注：因为 `R`不在参数列中）。

程序代码即使对非重载函数合法，有可能在函数被重载后不再合法。更具体地说，给定两个 function templates `f(T)`和 `g(T)`（这里 `T`是 template parameter），只有当 `f`未被重载时，运 算式 `g(&f<int>)`才是合法的（否则无法判断 `f`的意义）。

friend 声明语句可引用一个特定的 function template 或一个特定 function template 的具现体。然而 function template 的重载版本不具有 primary template 所可能拥有的特权（privileges）。 把这些加在一起，就构成了「加入 function templates 偏特化支持」的强有力论据。

function templates 偏特化语法是 class template 偏特化语法的泛化：

```
template <typename T>
T const& max (T const&, T const&);              // primary template

template <typename T>
T* const& max <T*>(T* const&, T* const&);       // 偏特化
```

有些语言设计者担心这种偏特化方式将干扰 function template 的重载机制，例如：

```
template <typename T>
void add (T& x, int i);     // 一个 primary template

template <typename T1, typename T2>
void add (T1 a, T2 b);      // 另一个(重载的)primary template

template <typename T>
void add<T*> (T*&, int);    // 这一行打算特化哪一个 primary template?
```

然而我们希望这种情形被视为一种错误，这才不致于对这个语言特性的效用有太大的影响。本书撰写之际，C++ 标准委员会还在考虑这一项扩展提议。

13.8 typeof运算符

撰写 templates 程序代码时，如果能够表达「与 template 相依之表达式」的类型，将非常有用。或 许最具代表性的例子就是：一个针对数值型 array template 而设计的算术运算符，且其操作数(都是 array 的元素）的类型是混杂的。下面的例子展示这一点：

```
template <typename T1, typename T2>
Array<???> operator+ (Array<T1> const& x, Array<T2> const& y);
```

如你所料，这个运算符产生一个 array，由 array x和 array y对应元素之和构成。因此运算成果 的类型应该是 x[0]+y[0] 的类型。不幸的是C++ 并没有提供什么方法让我们以 T1 或 T2来表 达这个成果类型。

某些编译器以扩展形式提供了一个 typeof运算符来解决这个问题。它让人联想到 sizeof运算 子：它们都接受一个表达式，并从该表达式中产生一个编译期物体（compile-time entity）。只 不过在这里，和 sizeof 不同的是，这个编译期物体起到一个「类型名称」作用。我们可以利用 这项扩展机能改写前面的例子为：

```
template <typename T1, typename T2>
Array<typeof(T1()+T2())> operator+ (Array<T1> const& x,
                                    Array<T2> const& y);
```

这样很好，但还不够理想，因为这会要求那些给定类型必须可以被其默认值初始化。我们可以 用一个辅助的 template 解决这个问题：

```
template <typename
```

```
T> T makeT(); //不需
定义


template <typename T1, typename T2>
Array<typeof(makeT<T1>()+makeT<T2>()
)> operator+ (Array<T1> const& x,
             Array<T2> const& y);
```

我们其实更想以 x和 y作为 typeof的自变量，但是不能这样做，因为在 typeof构件处，x和 y 尚未声明。解决这个问题的一个激进作法是引入另一个函数声明语法，把回返类型写到参数型 别之后：

```
// operator function template
template <typename T1, typename T2>
operator+ (Array<T1> const& x, Array<T2> const& y)
  -> Array<typeof(x[0]+y[0])>;
//译注：以上是作者假设的一种新式声明语法，实际并不存在。

// regular function template
template <typename T1, typename T2>
function exp(Array<T1> const& x, Array<T2> const& y)
  -> Array<typeof(exp(x[0], y[0]))>
//译注：以上是作者假设的一种新式声明语法，实际并不存在。
```

正如本例所示，对于非运算符（non-operator）函数来说，这个新语法需要引入一个新关键词（上述的 function）。对于运算符函数，关键词 operator已经能够给予词法解析器（parser）足够 导引。

注意，typeof必须是一个编译期运算符。更明确地说，typeof将不考虑协变回返类型（covariant return types），如下所示：

```
class Base {
public:
   virtual Base* clone();
};

class Derived : public Base {
public:
   virtual Derived* clone(); // 协变回返类型（covariant return types）
};
```

```
void demo (Base* p, Base* q)
{
   typeof(p->clone()) tmp = p->clone();   // tmp的类型总是为 Base*
   ...
}
```

请参考 15.2.4 节， p.271，那里介绍的 promotion traits 可以使目前缺少 typeof运算符的情况获得 部份抒解。

13.9 Named Template Arguments（具名模板引数）

16.1 节描述一种技术，可以让我们为特定的 template parameter 提供非预设的（nondefault）template argument，而无需为其它已有默认值的 template parameters 指定 template arguments。尽 管这是一项很有趣的技术，但很明显我们杀鸡用了牛刀。因此我们很自然地希望 C++ 能够提供
一个机制让我们对 template arguments 命名。
在这一点上，我们应该注意，早在C++ 标准化过程中，Roland Hartinger 就提交过一个类似的扩展机能（请参考 [StroustrupDnE], 6.5.1 节）。虽然技术上没什么问题，但是基于数种原因，这 个提议最终没有被接纳。目前看来没什么理由相信 named template arguments 会被纳入 C++ 语 言。

然而，出于完备性的考虑，以下将简述数种语法设计中的一种：

```
template<typename T,
         Move: typename M = defaultMove<T>,
         Copy: typename C = defaultCopy<T>,
         Swap: typename S = defaultSwap<T>,
         Init: typename I = defaultInit<T>,
         Kill: typename K = defaultKill<T> >
class Mutator {
   ...
};

void test(MatrixList ml)
{
   mySort (ml, Mutator <Matrix, Swap: matrixSwap>);
}
```

注意自变量名称（在一个冒号之前）是如何与参数名称区分的。这使我们实作程序代码时可以针对 参数使用短名，并因为自变量名称的存在而使得自变量的作用一目了然。由于这在某些编码风格中 可能显得过于累赘，任何都可能想象，当自变量名称与参数名称相同时，自变量名称可省略不写：

```
template<typename T,
         : typename Move = defaultMove<T>,
         : typename Copy = defaultCopy<T>,
         : typename Swap = defaultSwap<T>,
         : typename Init = defaultInit<T>,
         : typename Kill = defaultKill<T> >
class Mutator {
   ...
};
```

13.10 静态属性（Static Properties）

第 15 章和第 19 章将讨论数种在编译期间归类（categorize）类型的方法。如果有必要「根据型 别的静态属性来选择 templates 的特化方式」，那两章提到的 type traits（类型特征萃取机制）非 常重要。例如你可以参考 15.3.2 节的 `CSMtraits` class，它可以选择「最优」或「接近最 优」的策略（policies），对隶属其自变量类型的元素进行 *copy*、*swap* 或 *move* 操作。

有些语言设计者观察到，如果这类「选择特化方式」（specialization selections）的需求非常普遍，C++ 语言就不应该要求使用者提供各种精巧复杂而煞费苦心的「自定程序代码」，用以探寻编译 器内部已知的属性。C++ 语言实在可以提供一些内建的 type traits。如果有了这项扩展，下面的C++ 程序就完全合法：

```
#include <iostream>

int main()
{
    std::cout << std::type<int>::is_bit_copyable << std::endl;
    std::cout << std::type<int>::is_union << std::endl;
}
```

尽管我们可以为这一类构件发展出个别而独特的语法，但如果让它配合一个「使用者可自定」的语法，便可获得一种更平滑的过渡 — 从「当前语言」过渡到「包含这种特性的语言」。但 是某些 C++ 编译器可轻易提供的静态属性，却无法透过 traits 传统技术获得（例如判断一个型 别是否为 union）。这正是「让这个特性成为语言元素之一」的主要支撑论据。另一个论据是： 如果拥有这种语言元素，当我们编译「倚赖这些属性」的程序时，可以显著降低编译器所耗用 的内存及所消耗的机器时间。

13.11 订制的实例化诊断讯息（Custom Instantiation Diagnostics）

很多 templates 对其参数都有隐式要求。当你以「无法满足这些要求」的自变量来进行 templates 具 现化时，要么引发一个一般性错误，要么实例化之后的函数无法正常运作。早期C++ 编译器中，实例化 template 所引发的一般性错误往往极度晦涩（p.75 有个好例子）。晚近的编译器中，

这 些错误讯息已经足够清晰易读，有一定经验的程序员可以很快追查到问题所在。但是人们仍希 望这种情形能够再加改善。考虑下面这个刻意制造的例子（展示真实 template 库中所发生 的情况）：

```
template <typename T>
void clear (T const& p)
{
    *p = 0; // 假设 T是一个「类似指针」的类型
}

template <typename T>
void core (T const& p)
{
    clear(p);
}

template <typename T>
void middle (typename T::Index p)
{
    core(p);
}

template <typename T>
void shell (T const& env)
{
    typename T::Index i;
    middle<T>(i);
}

class Client {
public:
    typedef int Index;
    ...
};

Client main_client;

int main()
{
    shell(main_client);
}
```

这个例子展示了典型的软件开发分层策略高层 function templates 如 shell()倚赖诸如 middle()的组件，后者又使用 core()之类的基础设施。当我们实例化 shell()时，其每一个下层都需要 被实例化。本例的问题出在最底层 core()：它被实例化为 int类型（middle()之中调用 core() 时的自变量类型是 Client::Index），而它试图对该类型取值（*dereference*），这是错误的。一 个表现良好的通用诊断讯息将会回溯引发问题的各层，但如此大量的讯息通常难以被使用。

另一个经常被提倡的方法是：在最高层安插一个装置，使得如果来自更底层的需求无法获得满足，就抑制底层的实例化。已有各种「运用现有 C++ 构件」而进行的尝试（例如 [BCCL]）试 图达到这个目标，但它们并不总是已获得实战效能。也因此，那么多人提议增加这项语言扩展， 也就不让人惊讶了。这项扩展显然可以构筑于先前讨论的静态属性设施基础上。例如我们可以 想象把 shell() template 修改如下：

```
template <typename T>
void shell (T const& env)
{
    std::instantiation_error(
          !std::type<T>::has_member_type<"Index">,
          "T must have an Index member type");
    std::instantiation_error(
          !std::type<typename T::Index>::dereferencable,
          "T::Index must be a pointer-like type");
    typename T::Index i;
    middle(i);
}
```

伪函数（pseudo-function）instantiation_error()应该会导致编译器中断实例化过程（这就避免了由 middle()实例化触发的诊断讯息），并使得编译器报出给定的诊断讯息。

虽然这是可行的，但这个方法有一些缺陷。例如，描述类型的所有属性既笨重又不必要。有些人提议使用所谓「哑码」（dummy code）构件来当作中断实例化的条件。下面是众多提议中的一种（不引入新关键词）：

```
template <typename T>
void shell (T const& env)
{
    template try {
        typename T::Index p;
        *p = 0;
    } catch "T::Index must be a pointer-like type";
    typename T::Index i;
```

```
    middle(i);
}
```

这里呈现的想法是，template try子句将被暂时实例化，但不真正生成 obj 码。如果此处发生 错误，子句之后的诊断讯息会被报出。不幸的是这种机制难以实现，因为即使可以免除 obj 码 的生成，也会给编译器的内部数据带来难以避免的负面作用。换句话说，这个相当小的特性很 可能需要对现有编译技术进行大规模重新规划。

大多数此类方案都有各种其它限制。例如许多 C++ 编译器可以使用各种语言（英文、德文、日文等等）报出诊断讯息，但要求在源码中提供各语种的诊断讯息就有些过份了。不仅如此，如 果实例化过程被中断后，这些预先处理没有在诊断讯息中被清晰而系统化地说明，程序员或许 还不如面对那些一般的诊断讯息（虽然笨拙难用）。

13.12 经过重载的（Overloaded）Class Templates

完全可以想象，class templates 可透过其 template parameters 进行重载。例如我们可以想象这样一种方式：

```
template <typename T1>
class Tuple {
   // singleton
   ...
};

template <typename T1, typename T2>
class Tuple {
   // pair
   ...
};

template <typename T1, typename T2, typename T3>
class Tuple {
   // three-element tuple
   ...
};
```

下一节将讨论这种重载方式的一个应用。

重载方式不必局限于 template parameters 个数（这样的重载可使用偏特化来模拟，请参考第 22 章实作的 FunctionPtr）。参数种类也可以是多样的：

```
template <typename T1, typename T2>
class Pair {
   // pair of fields
```

```
    ...
};

template <int I1, int I2>
class Pair {
    // pair of constant integer values
    ...
};
```

虽然某些语言设计者曾经非正式地讨论过这个想法，但它还未被提交到C++ 标准委员会。

13.13 List Parameters（一系列参数）

有时人们需要将一整串类型（a list of types）当作单一 template argument 传递。通常这是出于下面两种目的之一：(1) 宣告一个函数，其参数个数是参数化的；(2) 定义一个类型结构（type structure），其成员个数是参数化的。

举个例子，我们也许想要定义一个 template，用来计算任意数值列中的最大值。一个可能的声明语法是：使用省略符号（ellipsis）表示「最后一个 template parameter 可匹配任意数量的自变量」：

```
#include <iostream>

template <typename T, ... list>      //译注：...就是所谓的 list template parameter
T const& max (T const&, T const&, list const&);

int main()
{
    std::cout << ::max(1, 2, 3, 4) << std::endl;
}
```

这样一个 template 可以有多种实作可能。以下办法不需要引入新关键词，但它必须对 function
template 重载机制加入一条新规则：优先选用「不含 list template parameter」的 function template：

```
template <typename T> inline
T const& max (T const& a, T const& b)
{
    // 这是惯用的二元 max()
    return a<b ? b : a;
}

template <typename T, ... list> inline
```

```
T const& max (T const& a, T const& b, list const& x)
{
   return ::max (a, ::max(b,x));
}
```

让我们分析调用语句`::max(1,2,3,4)`的各个执行步骤 由于这个调用语句有四个自变量 二元的`max()`
函数与之不匹配，但是当 `T=int` 而 `list=int,int` 时，第二个 template 与之匹配。这使得我们 调用二元 function template `max()`，第一个自变量为 1，第二个自变量是`::max(2,3,4)`的求值结果。 面对后者二元 `max()`再次不匹配于是调用 list parameter 版的 `max()`,并使 `T=int` 而 `list=int`。 这一次子表达式 `max(b,x)`被展开为 `max(3,4)`，于是选用二元 function template 并结束递归。

感谢 function templates 的重载能力，它运作得极好。当然实际情况比我们讲的要复杂一些。例 如我们必须精确指出 `list const&`在当前情境（前后脉络, context）下的意义。

有时人们希望引用参数列中的某个特别元素或某个子集。我们可以使用下标运算符（subscript operator; 方括号）满足需要。下面例子展示如何建构一个 metaprogram（超程序），并使用这种 技术统计 list 中的元素个数：

```
template <typename
T> class ListProps
{ public:
    enum { length = 1 };
};

template <...
list> class
ListProps
{ public:
    enum { length = 1+ListProps<list[1 ...]>::length };
};
```

这个例子说明 list parameters 对于 class templates 也相当有用。它也可以结合前面讨论的 class template 重载概念，使我们得以增强各种 template metaprogramming（模板超编程）技术。

list parameter 也可以用来声明一串字段（fields, 成员变量）：

```
template <... list>
class Collection {
    list;
};
```

在这种设施基础上，我们可以发展出数量惊人的基础工具（dundamental utilities）。有关这方面的更多构想，我们推荐你阅读《*ModernC++Design*》（请见 [AlexandrescuDesign]），书中告诉你大量的 templates-based 和 macros-based 超编程（metaprogramming），取代目前缺乏的 list parameters。

13.14 布局控制（Layout Control）

template 编程中一个颇为普遍的困难是声明一个够大（但不过份大）的 bytes array，用以容纳一个「类型未知的 object」（或说某个 template parameter）。一个应用是所谓的 discriminated unions（有分辨能力的 unions；又称为 variant types（易变类型）或 tagged unions（带标记的 unions））：

```
template <...
list> class
D_Union { public:
    enum { n_bytes };
    char bytes[n_bytes]; // 最终将容纳 template arguments 所描述之各类类型中的一种
    ...
};
```

常数值 n_bytes不能强定为 sizeof(T)，因为 T可能比 bytes 有更严厉的齐位（alignment）条件。很多这方面的探讨都考虑到了齐位方式，但它们通常过于复杂，或是存在某种程度上的武断假设。

对于此类应用，我们实际需要的是「以常数表达式表述某个类型的齐位方式」的能力，以及反过来能够「对某个类型或某个字段（field）或某个变量进行齐位」的能力。很多 C 和C++编译器提供__alignof__ 运算符，它传回某给定类型或表达式的齐位方式。这几乎完全和 sizeof 运算符相同，只是它传回的是齐位方式（alignment）而不是类型大小。很多编译器也提供 #pragma 指令或类似装置来设定某个物体（entity）的齐位方式。一个可能的作法是引入关键词 alignof，它既可用于在声明中设定齐位方式，也可用于在表达式中求取齐位方式。

```
template <typename
T> class Alignment
{ public:
    enum { max = alignof(T) };
};

template <...
list> class
Alignment
{ public:
    enum { max = alignof(list[0]) > Alignment<list[1 ...]>::max
                ? alignof(list[0])
                : Alignment<list[1 ...]>::max }
```

```
};

// 也可运用同样道理设计一组 Size templates，用以获得一大串类型中的体积（尺码）最大者。

template <...
list> class
Variant { public:
    char buffer[Size<list>::max] alignof(Alignment<list>::max);
    ...
};
```

13.15 初始式的推导（Initializer Deduction）

人们常说程序员懒惰，有时这句话指的是我们总想在程序中少写些标记。考虑下面的声明：

```
std::map<std::string, std::list<int> >* dict
  = new std::map<std::string, std::list<int> >;
```

这很啰嗦。现实之中我们往往（也应该）引入该类型的一个 typedef 同义词。然而这个声明似乎有些多余：我们指定了类型 dict，而它应该隐寓成为其初始式（initializer）中的类型。如果可以写个等价声明，其中只需指定一个类型符号，那将多么优雅。例如：

```
dcl dict = new std::map<std::string, std::list<int> >;
```

上面这个声明语句中变量类型是从其初始式推导出来的这里需要一个关键词上例用的是 dcl，但也有其它提议如 var、let甚至 auto）使它异于常规赋值操作（ordinary assignment）。

到目前为止，这不是一个只与 template 相关的议题。事实上一个相当早期的 Cfront 版本就已经出现过这个构件（时值 1982 年，早在 templates 出现之前）。但正由于许多 template-based 类型过于冗长，人们才如此迫切需要这个特性。

人们也想象了局部推导（partial deduction）机制。此机制适用于「只有 template arguments 需要推导」的情况：

```
std::list<> index = create_index();
```

这个机制的另一个变体是由构造函数自变量推导出 template arguments。例如：

```
template <typename
T> class Complex
{ public:
    Complex(T const& re, T const& im);
    ...
};

Complex<> z(1.0, 3.0); // 推导出 T为 double
```

由于构造函数乃至构造函数模板（constructor templates）都可能存在重载，因此这种推导机制的精
确规格描述变得更为复杂。假设我们的 Complex template 包含一个 constructor template，以及一个常规的 *copy* 构造函数：

```
template <typename
T> class Complex
{ public:
    Complex(Complex<T> const&);

    template <typename T2> Complex(Complex<T2> const&);
    ...
};

Complex<double> j(0.0, 1.0);
Complex<> z = j; // 究竟想用哪一个构造函数?
```

在第二个初始式声明中，使用者应该是想调用常规的 *copy* 构造函数；因此 z应该与 j具有相同型 别。然而如果制订一个「忽略 constructor templates」的隐式规则，似乎又过于呆板了。

13.16 Function Expressions（函数运算式）

本书第 22 章说明一个主题：将小型函数（或仿函数, functors）作为参数传递给其它函数，常常 能够带来方便。第 18 章也提到 expression template 技术可用来搭建小型仿函数（functors），无 需承受因明确声明而可能带来的额外开销（见 18.3 节, p.340）。

例如我们可能需要针对标准 vector的每一个元素，调用元素类型所提供的某个特定成员函数，以便进行各个元素的初始化：

```
class BigValue {
public:
    void init();
    ...
};

class Init {
public:
    void operator() (BigValue& v) const {
        v.init();
```

```
    }
};

void compute (std::vector<BigValue>& vec)
{
    std::for_each (vec.begin(), vec.end(),
                   Init());
    ...
}
```

为了这个目的而定义出一个个别的 class `Init`，似乎有些笨拙。或许我们可以写出一个（不具名）函数，作为表达式的一部份：

```
class BigValue {
public:
    void init();
    ...
};

void compute (std::vector<BigValue>& vec)
{
    std::for_each (vec.begin(), vec.end(),
                   $(BigValue&) { $1.init(); } );
    ...
}
```

这里的构想是引入一个所谓的 function expression（函数表达式），语法如下：一个特殊记号`$`，后跟着以小（圆）括号括起来的参数类型，然后是一个大括号括起来的函数本体。在这样一个 构件中，我们可以透过特殊记号`$n`引用某个函数参数，此处 `n`是个常数，代表参数序号。

这种形式与其它语言所谓的 lambda expressions（或称 *lambda functions*）和 *closures* 很有关（译 注：*lambda* 是希腊字母中的第 11 个字母）。其它解法也是可能的，例如我们可以使用不具名内 层（annoymous inner）class，像 Java 那样：

```
class BigValue {
public:
    void init();
    ...
};

void compute (std::vector<BigValue>& vec)
{
    std::for_each (vec.begin(), vec.end(),
                   class {
```

```
                    public:
                      void operator() (BigValue& v) const
                                          v.init
                                            ();
                      }
                  };
                 );
    ...
}
```

虽然这些构件方式经常在语言设计者之间被讨论，但很少有人提出具体的提案。这或许是因为这种扩展的设计难度相当可观，远远超出前面例子所涵盖的范围。待解决的问题包括回返类型 的详细规范、如何决定 function expressions（函数表达式）中哪些物体可用…等等。举个例子， 外层函数的 local 变量可以被取用吗？function expressions 也应该可以是个 templates，因为其参 数类型可由用户调用语句中推导而得。这种处理方式可以使前面的例子更加简练（我们可以完全 省略 parameter list），但这又给 template argument 的推导系统带来了新的挑战。

目前还不完全清楚C++ 是否会包含 function expressions 概念。然而由 Jaakko Järvi 和 Gary Powell 实作出来的 Lambda 库（见 [LambdaLib]）在这方面取得了长足进展 — 尽管它会耗费相当 多的编译时间。

13.17 后记

在大多数C++ 编译器才刚刚能够符合 1998C++ *Standard*（*C++98*）的今天，谈论对这个语言的扩展似乎为时过早。然而正是在努力符合标准的过程中，我们才得以洞察 C++ 真正的局限（特别是 templates 的局限）。

为了迎合C++ 程序员的新需求，C++ 标准委员会（或名 ISO WG21/ANSI J16 或简称 WG21/J16）已经开始了新标准的征途：*C++0x*。2001/04 于哥本哈根召开预备陈述会后，WG21/J16 开始着 手审查具体的库扩展提案。

审查目标尽可能限于C++ 标准库的扩展，然而人们都理解某些扩展需要语言核心的搭配。 预计 C++ 语言核心所进行的许多修改都将与 templates 有关，就像 1990 年代 STL 被引入C++ 标准库促进了 template 技术的发展。

最后要说的是，*C++0x* 也致力于解决 *C++98* 中的一些困窘。人们希望这么做可提高 C++ 的可 用性。本章也对这个方向上的某些扩展做了一些讨论。

第三篇 模板与设计

Templates and Design

一般而言，程序通常使用「与选定之编程语言所提供的机制有良好映像关系」的某种设计来加 以建构（constructed）。由于 templates 是一种全新的语言机制，它需要新的设计元素，也就不 足为奇。本篇将探索这些元素。

Templates 不同于传统语言之处在于，它允许我们将程序代码中的类型和常数参数化（parameterize）。当它结合 (1) 偏特化（partial specialization）、(2) 递回实例化（recursive instantiation），会产生让人目瞪口呆的威力。此点将于接下来的数章中以如下的众多设计技术 加以说明：

Generic programming（泛型编程）

Traits（特征、特征萃取）

Policy classes（策略类别）

Meta-programming（超编程）

Expression templates（算式模板） 我们的目标并不仅止于列出形形色色的设计元素，还要传达当初激发这些设计的本源，从而让
大家更有可能创造出新技术。

14

Templates 的多型威力

The Polymorphism Power of Templates

所谓多型（Polymorphism），是一种「以单一泛化记号（generic notation）表述多种特定行为」的能力。多型是面向对象编程思维模型(object-oriented programming paradigm)的基石，C++ 主要透过 class 的继承和虚拟函数（virtual functions）支持多型。由于这些机制（全部或部份）生效于执行期，所以称为动态多型（**dynamic polymorphism**）。谈论C++ 多型时，所指的通常便是动态多型。然而 templates 也允许我们以单一泛化记号表述多种特定行为，只不过一般发生在编译期，因此称为静态多型（**static polymorphism**）。本章将检阅上述两种多型，并讨论各自适用的场合。

14.1 动态多型（Dynamic Polymorphism）

从历史上看，C++ 一开始只是透过「继承机制与虚拟函数的结合运用」来支持多型。这种背景下的多型设计艺术是：在彼此相关的 object types 之间确认一套共通能力，并将其声明为某共通基础类别（common base class）的一套虚拟函数接口。

这种设计最具代表性的例子就是一个「管理若干几何形状」的程序，这些几何形状可以某种方式着色例如在屏幕上着色)。在这样的程序中，我们可以定义一个所谓的抽象基础类别(abstract base class, ABC）`GeoObj`，在其中声明适用于几何对象的一些共通操作（operations）和共通属性（propertie）每一个针对特定几何对象而设计的具象类别（concrete class）都衍生自 `GeoObj`（见图 14.1）

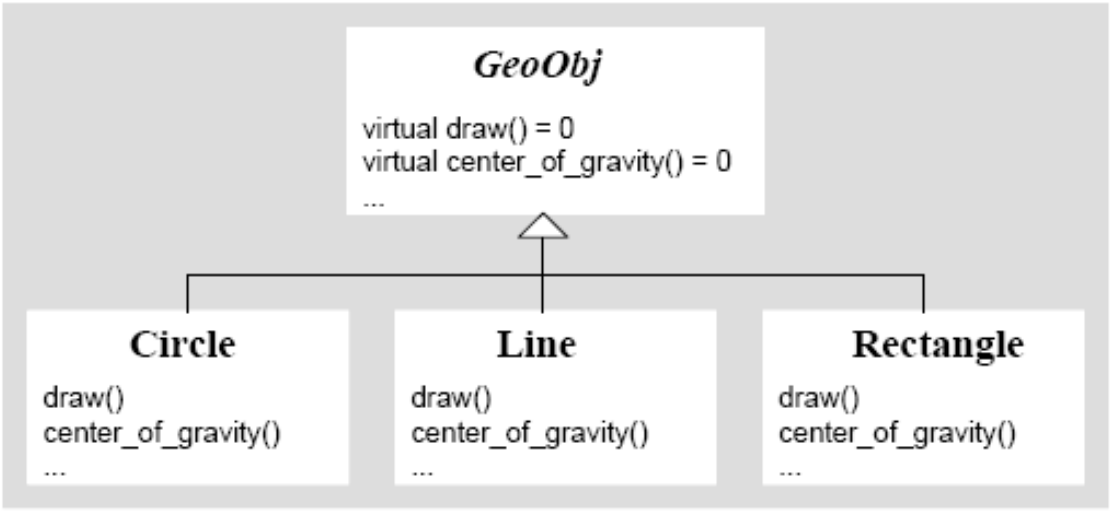


图 14.1 多型（polymorphism）透过继承（inheritance）实现出来

```
//poly/dynahier.hpp
#include "coord.hpp"
```

```
// 针对几何对象而设计的共通抽象基础类别（common abstract base class）GeoObj
class GeoObj {
  public:
    // 绘制几何对象:
    virtual void draw() const = 0;
    // 传回几何对象的重心（center of gravity）:
    virtual Coord center_of_gravity() const = 0;
    //...
};

// 具象几何类别（concrete geometric class）Circle
// - 衍生自 GeoObj
class Circle : public GeoObj {
  public:
    virtual void draw() const;
    virtual Coord center_of_gravity() const;
    //...
};

// 具象几何类别 Line
// - 衍生自 GeoObj
class Line : public GeoObj {
  public:
    virtual void draw() const;
    virtual Coord center_of_gravity() const;
    //...
};
//...
```

建立具象对象（concrete objects）之后，客端程序代码可以透过「指向基础类别」的 references 或 pointers 来操纵这些对象，这会启动虚拟函数分派机制（virtual function dispatch mechanism）。当我们透过一个「指向基础类别之子对象（subobject）」的 references 或 pointers 来调用某虚拟 函数，会调用「被指涉（*refered*）的那个特定具象物件」的相应成员函数。

以本例而言，具体程序代码大致描绘如下：

```
//poly/dynapoly.cpp

#include "dynahier.hpp"
#include <vector>

// 绘制任何 GeoObj
```

```
void myDraw (GeoObj const& obj)
{
    obj.draw();          // 根据对象的类型调用 draw()
}

// 处理两个 GeoObjs重心之间的距离
Coord distance (GeoObj const& x1, GeoObj const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs();       // 传回坐标绝对值
}

// 绘出 GeoObjs异质群集（heterogeneous collection）
void drawElems (std::vector<GeoObj*> const& elems)
{
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i]->draw(); // 根据对象的类型调用 draw()
    }
}

int main()
{
    Line l;
    Circle c, c1, c2;

    myDraw(l);             // myDraw(GeoObj&) => Line::draw()
    myDraw(c);             // myDraw(GeoObj&) => Circle::draw()

    distance(c1,c2);       // distance(GeoObj&,GeoObj&)
    distance(l,c);         // distance(GeoObj&,GeoObj&)

    std::vector<GeoObj*> coll;  // 异质群集（heterogeneous collection）
    coll.push_back(&l);         // 插入一个 line
    coll.push_back(&c);         // 插入一个 circle
    drawElems(coll);            // 绘出不同种类的
    GeoObjs
}
```

上述程序的关键性多型接口元素是draw()和center_of_gravity()，两者都是虚拟函数。程式示范了它们在 myDraw()、distance()和 drawElems()函数内被使用的情况— 由于这三个函数使用共通基础类别 GeoObj作为表达手段因而无法在编译期决定使用哪一个版本的 draw()或center_of_gravity()。然而在执行期，「调用虚拟函数」的那个对象的完整动态类型会被 取得，

以便对调用语句进行分派（dispatch）。于是，根据几何对象的实际类型，程序得以完成适 当操作：如果对一个 Line对象调用 myDraw()，函数内的 obj.draw()就调用 Line::draw()；对 Circle物件调用的则是 Circle::draw()。同样道理，对 distance()而言，调用的将是「与自变量物件相应」的那个 center_of_gravity()。

或许「动态多型」最引人注目的特性就是处理异质对象群集（heterogeneous collections of objects） 的能力。drawElems()阐示了这样一个概念：下面的简单表达式

```
elems[i]->draw()
```

将根据「目前正被处理的元素」类型，调用不同的成员函数（译注：都名为 draw()）。

14.2 静态多型（Static Polymorphism）

Templates 也可以用来实作多型，然而它们并不倚赖「分解及抽取 base classes 共通行为」。在这里，共通性是指：应用程序所提供的不同几何形状，必须以共通语法支持其操作（也就是说，

相关函数必须同名）。具象类别（concrete classes）之间彼此独立定义（见图 14.2）。当 templates 被具象类别「实例化」，便获得（被赋予）多型的威力。

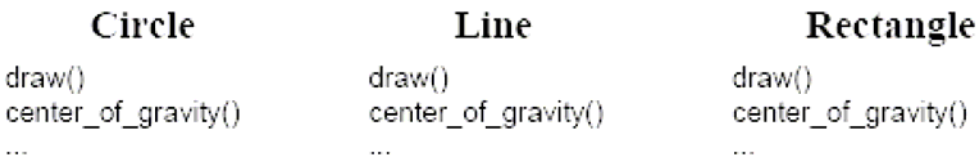


图 14.2 多型（polymorphism）透过模板（templates）实现出来

例如前一节中的函数 myDraw()：

```
void myDraw (GeoObj const& obj)    //GeoObj 是抽象基础类别（abstract base class）
{
    obj.draw();
}
```

可设想改写如下：

```
template <typename GeoObj>
void myDraw (GeoObj const& obj)    //GeoObj是模板参数（template parameter）
{
    obj.draw();
}
```

比较前后两份 myDraw()实作码，我们可以得出一个结论：两者的主要区别在于 GeoObj 如今是个 template parameter 而非一个 common base class。然而在此表象背后，还有一些更根本的区别。比方说，如果使用动态多型，执行期只会有一个 myDraw()函数，但如果使用 templates，我们

会 拥有不同的函数如 myDraw<Line>()和 myDraw<Circle>()。

让我们尝试使用静态多型机制改写前一节的例子。首先不再使用几何类别阶层体系，而是编写若干个独立的几何类别：

```
//poly/statichier.hpp

#include "coord.hpp"

// 具象的几何类别 Circle
// - 不衍生自任何类别

class Circle {
  public:
    void draw() const;
    Coord center_of_gravity() const;
    //...
};

// 具象的几何类别 Line
// - 不衍生自任何类别
class Line {
  public:
    void draw() const;
    Coord center_of_gravity() const;
    //...
};
//...
```

现在，这些 classes 的应用程序看起来像这样：

```
//poly/staticpoly.cpp

#include "statichier.hpp"
#include <vector>

// 绘出任何给定的 GeoObj
template <typename
GeoObj>
void myDraw (GeoObj const& obj)
{
    obj.draw();    // 根据对象的类型调用 draw()
}
```

```
// 处理两个 GeoObjs重心之间的距离
template <typename GeoObj1, typename GeoObj2>
Coord distance (GeoObj1 const& x1, GeoObj2 const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs();    // 传回坐标绝对值
}

// 绘出 GeoObjs同质群集（homogeneous collection）
template <typename GeoObj>
void drawElems (std::vector<GeoObj> const& elems)
{
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i].draw();    // 根据元素的类型调用 draw()
    }
}

int main()
{
    Line l;
    Circle c, c1, c2;

    myDraw(l);       // myDraw<Line>(GeoObj&) => Line::draw()
    myDraw(c);       // myDraw<Circle>(GeoObj&) => Circle::draw()

    distance(c1,c2); // distance<Circle,Circle>(GeoObj1&,GeoObj2&)
    distance(l,c);    // distance<Line,Circle>(GeoObj1&,GeoObj2&)

    // std::vector<GeoObj*> coll;    // ERROR: 不可以是异质群集
    std::vector<Line> coll;          // OK: 同质群集没问题
    coll.push_back(l);               // 安插一条直线
    drawElems(coll);                 // 画出所有直线
}
```

distance()不可再以 GeoObj 为具体参数类型，myDraw()的情况也一样。我们改而提供两个 template parameters GeoObj1和 GeoObj2。透过两个不同的 template parameters，一对「几何型别组」可被拿来计算两几何对象间的距离：

```
distance(l,c);   //distance<Line,Circle>(GeoObj1&,GeoObj2&)
```

然而，异质群集（heterogeneous collections）就不再能够被透通地处理了。这是「静态多型」

的 静态性质所带来的限制：所有类型都必须在编译期决定。不过我们可以轻易为不同的几何类型 引入不同的群集，而且无需再将群集内的元素局限于 pointers（译注：如果使用动态多型，异质 群集内就一定得放置 pointers），这在执行效率和类型安全方面都有显著的优势。

14.3 动态多型 vs.静态多型

让我们对动态和静态两种多型加以归类和比较。

术语

动态多型和静态多型分别支持不同的C++ 编程手法（idioms）：

经由继承而实现的多型是 bounded（绑定的、已系结的）和 dynamic（动态的）：

bounded 意味「参与多型行为」的类型，其接口系透过 common base class 的设计而预 先定妥。此概念的另一些描述术语为 invasive（侵略性的）或 intrusive（侵入式的）。

dynamic 意味界面的系结（绑定; binding）动态完成于执行期。

经由 templates 而实现的多型是 unbounded（非绑定的）和 static（静态的）：

unbounded 意味「参与多型行为」的类型，其接口并非预先决定好。此概念的另一些描 述术语有 noninvasive（非侵略性的）或 nonintrusive（非侵入式的）。

static 意味界面的系结（绑定; binding）静态完成于编译期。

因此，严格说来（以 C++ 行话来说），动态多型和静态多型分别是绑定之动态多型（bounded dynamic polymorphism）和未绑定之静态多型（unbounded static polymorphism）的另一说法。其他语言另有其他结合方式，例如 Smalltalk 提供未绑定之动态多型（unbounded dynamic polymorphism）。「动态多型」和「静态多型」两个简练术语在C++ 环境中并不会招致混淆。

优点和缺点

C++ 的动态多型表现出如下优点：

可优雅处理异质群集（Heterogeneous collections）。

可执行码的体积（executable code size）可能比较小。因为只需一个多型函数。静态多型则必 须产生不同的 template 实体以处理不同的类型。

程序代码可被完全编译，因而无需非得发布实作源码不可。如果你发行的是 template 库，通常还需释出其实作源码。

相较之下，以下的优点属于C++ 静态多型：

内建型资料群集（collections of built in types）可被轻松实作出来。更一般地说，共通性接口 无需透过一个共通基础类别（common base class）来表达。

生成的程序代码可能执行速度较快。因为无需透过 pointers 进行间接操作，而且非虚拟函数

（nonvirtual functions）被 *inlined*（内联化）的可能性比较大。

即使只提供部份接口的具象类型（concrete types）仍然可用 — 如果最终只有这一部份接口
被应用程序用上的话。

通常静态多型被认为比动态多型更具类型安全性（type safe），因为所有系结（binding）都被检查于编译期。举个例子，这种程序不可能存在这样的危险：将一个类型不匹配的物件插入一个从 template 具现出来的容器内。然而如果容器期望获得的元素是 pointer to common base class， 这个 pointer 有可能无意中指向不同类型的对象身上。

现实之中，如果不同的「语意假设」隐藏于外观完全一致的接口背后，template 具现体也可能招致某些麻烦。例如当 template 将 operator+ 针对某类型实例化，但该类型实际上与 + 操作并无关联，人们可能会大感惊讶。在现实经验中，此类「语意不匹配」很少发生于以继承为基础
的 class 阶层体系，这或许是因为在那种情况下接口的规格有更严明的指定。

将两种形式结合起来

当然了，你可以结合使用静态多型和动态多型。例如你可以从一个 common base class 衍生出不同种类的几何对象，以便经营一个异质几何对象群集，而同时仍可针对某些种几何物件以 templates 编写其程序代码。

第 16 章将就继承机制和 templates 的结合运用作进一步讨论。我们将看到成员函数的虚拟性如何被参数化、以继承为基础的 CRTP（curiously recurring template pattern, 奇特递归模板范式）又是如何为静态多型提供额外的弹性。

14.4 Design Patterns（设计范式）的新形式

「静态多型」为 design patterns（设计范式）带来了新的实作方式。举个例子，*bridge* 范式在 C++程序中扮演重要角色，其运用目标之一是「在同一接口的不同实作之间进行切换」。根据 [DesignPatternsGoV] 的描述，通常的达成方式是：以一个指针指向实际实作品，并将所有调用动作都委托（*delegating*）给该 class（见图 14.3）。

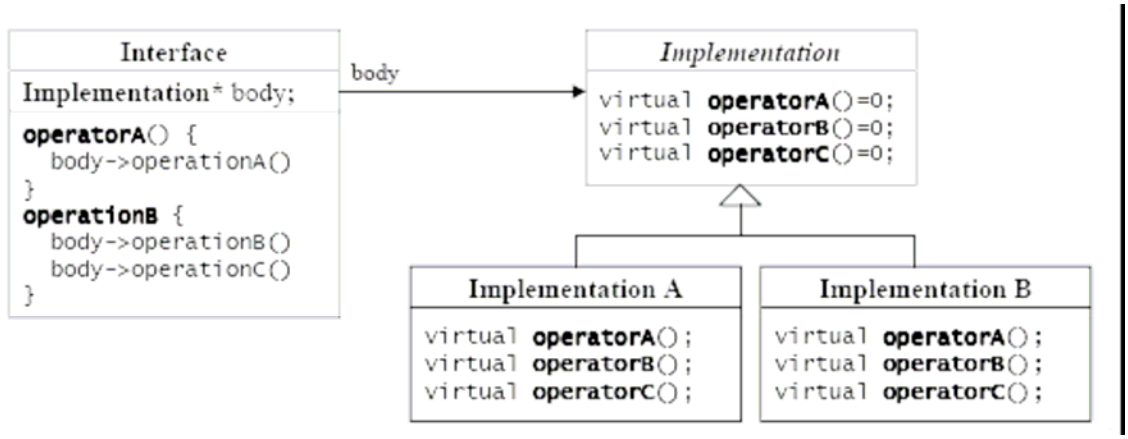


图 14.3 运用继承（inheritance）实作出 *Bridge* 范式

然而如果编译期就能够知道实作品的类型，你就能够透过 templates 使用这一方案（见图 14.4）。

这会带来更好的类型安全性（type safety），又可避免运用指针，而且执行时应该会更快速。

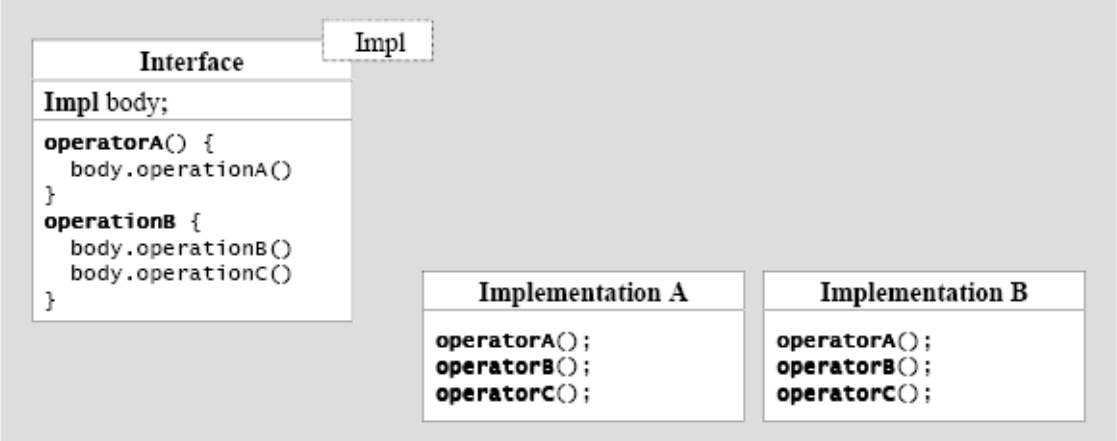


图 14.4 运用 templates 技术实作出 *Bridge* 范式

14.5 泛型编程（Generic Programming）

静态多型带来了「泛型编程」概念。然而世上并不存在一个被普遍认同的泛型编程定义（就像 不存在被普遍认同的面向对象编程定义一样）。按 [CzarneckiEisenecker GenProg] 的描述，众多 定义的根源是：「使用泛化参数（generic parameters）进行编程，以便找出高效算法（efficient algorithms）之最抽象表述（most abstract representation）」。那本书最后总结如下：

> 泛型编程是一个计算器科学分支，适合用来找出（发现）高效算法、数据结构、 其它软件概念…之抽象表述，并用以处理它们的系统化组织方式。泛型编程主要用于表现「领域概念」族系（families of domain concepts）（p.169 和 p.170）。

在 C++ 环境中，泛型编程有时被定义为「以 templates 进行编程」（而面向对象编程则被认为 是「以虚拟函数（virtual functions）进行编程」）。从这个意义上说，只要使用了C++templates，任何编程行为都可被视为泛型编程。不过专业人士通常认为泛型编程还具有其它某些本质要素：为进行大量有意义的联合运用，templates 应该被设计于框架（framework）之中。

迄今为止，这个领域中最耀眼的贡献是 STL（Standard Template Library，后被编入C++ 标准程 式库内）。STL 是个框架（framework），提供大量有用操作，称为算法（**algorithms**），以 及大量用于处理对象群集（collections of objects）的线性数据结构，称为容器（**containers**）。算法和容器都是 templates。不过最关键的还在于：算法并非容器的成员函数，而是以某种泛化方式编写而成，因此可被任何容器（以及由元素组成的任何线性群集）所用。为了做到这一点，STL 设计者确认了一个抽象概念：迭代器（**iterators**），它可被用于任何种类的线性群集 身上。事实上容器的各项操作中，凡与群集相关的部分（译注：例如取前一元素、取下一元素、 取第一元素，取最后元素…），都已被抽取（分解）为迭代器的机能。

因此，计算「序列内的最大值」这般操作，便可在不了解序列实值（values）储存细节的情况下 完成：

```
template <class Iterator>
Iterator max_element (Iterator beg,   //指向（代表）群集的头
                      Iterator end)   //指向（代表）群集的尾
{
    //只需使用 Iterator的某些操作，在群集的每个元素身上来回移动，
    //找出带有最大值的那个元素，并将其位置包装为一个 Iterator传回。
    ...
}
```

每个线性容器不再需要提供诸如 max_element()这样的操作。容器只需提供迭代器类型（iterators type）用以巡访（遍历）容器所含的实值序列（sequence of values），并提供成员函数建立这样 的迭代器即可：

```
namespace std {
    template <class T, ...>
    class vector {
      public:
        typedef ... const_iterator;        //与实作相依（implementation-specific）
        ...                                //的 iterator 类型，用于 const
        vectors. const_iterator begin() const;   //传回一个 iterator，指向
        群集头部 const_iterator end() const;     //传回一个 iterator，指向
        群集尾端
        ...
    };

    template <class T, ...>
    class list {
      public:
        typedef ... const_iterator;        //与实作相依（implementation-specific）
        ...                                //的 iterator 类型，用于 const lists.
        const_iterator begin() const;      //传回一个 iterator，指向群集头部
        const_iterator end() const;        //传回一个 iterator，指向群集尾端
        ...
    };
}
```

现在，只要以群集起点和终点元素为自变量，调用泛化后的 max_element()，就可以找出任何群
集的最大值（这儿暂略对空群集的特别处理）：

```
//poly/printmax.cpp

#include <vector>
#include <list>
```

```
#include <algorithm>
#include <iostream>
#include "MyClass.hpp"

template <typename T>
void print_max (T const& coll)
{
    // 声明群集的区域迭代器（local iterator）
    typename T::const_iterator pos;

    // 计算最大值的位置
    pos = std::max_element(coll.begin(),coll.end());

    // 打印出 coll的最大元素（如果有的话）:
    if (pos != coll.end()) {
        std::cout << *pos << std::endl;
    }
    else {
        std::cout << "empty" << std::endl;
    }
}

int main()
{
    std::vector<MyClass> c1;
    std::list<MyClass>  c2;
    //...
    print_max (c1);
    print_max (c2);
}
```

由于操作式对迭代器（iterators）实施了参数化，STL 得以避免操作式的数量爆炸。如今我们不再需要为每个容器实作每一个操作式，只需算法实作一次便可用于所有容器。迭代器是一种泛化胶水（generic glue），由容器提供，被算法运用。这种机制之所以有效运作，原因在于 迭代器具有某种接口，该接口由容器提供并为算法所用。这种界面常被称为 **concept**（概念），标记出一套约束条件（constraints）— 每一个 templates 都必须实现它们才能符合 STL 框架要求。

原则上，动态多型也可以实现 STL 这样的机能，然而实际很少那么做。因为和虚拟函数调用机 制相比，迭代器概念（iterator concept）显得轻巧许多。如果添加一个虚拟函数接口层，操作速 度极有可能被拖慢一个数量级（甚至不止）。

泛型编程之所以实用，正是因为它倚赖「于编译期间对接口完成解析（*resolves*）」的静态多

型 机制。另一方面，「于编译期间对接口完成解析」这一需求，也声声呼唤着新的设计原理，这 些新原理在很多方面都不同于面向对象设计原理。本书剩余章节将描述许多极其重要的泛型设 计原理（generic design principles）。

14.6 后记

容器类型（container types）是将 templates 引入C++ 编程语言的首要动机。templates 出现前，多型阶层体系（polymorphic hierarchies）是处理容器的流行方式。一个广为流传的例子是 NIHCL（National Institutes of Health Class Library），它很大程度「翻译」了 Smalltalk 的容器类别阶层体系，参见图 14.5。

NIHCL 和 C++ 标准库很像，也支持丰富而多样的容器和迭代器。然而此一实作品沿袭 Smalltalk 的动态多型 — 其 `Iterators` 使用抽象基础类别 `Collection`来操作不同种类的群集

```
Bag
c1;
Set
c2;
...
Iterator
i1(c1);
Iterator
i2(c2);
...
```

遗憾的是，不论从执行时间或内存用量来看，这种方式的代价都太高。其执行时间比 C++ 标 准库的等价程序代码相差数个数量级，因为绝大多数操作最终都调用虚拟函数（而 C++ 标准 库内的许多操作都是 *inlined*，且其迭代器和容器的界面也不涉及虚拟函数）。此外，由于 其接口是 bounded（已系结/绑定）（这和 smalltalk 不同），所以内建类型（built-in types）不得不被包装为较大的多型类别（polymorphic classes；NIHCL 就提供了这样的包装器; wrapper），从而导致储存空间的需求呈戏剧性增长。

有些人在宏（macros）中寻求慰藉，然而即使到了今天这样的 templates 时代，许多项目只是做出次佳的多型选择（译注：意思可能是说应该选用带有类型检测能力的 templates，却选用了无任何检测能力的 macros）。毫无疑问许多时候动态多型的确是正确的选择 — 异质对象迭代
（Heterogeneous iterations）就是个明显例子。然而在此同时，许多编程任务也可以自然而高效地以 templates 解决 — 同质容器（homogeneous containers）就是个例子。

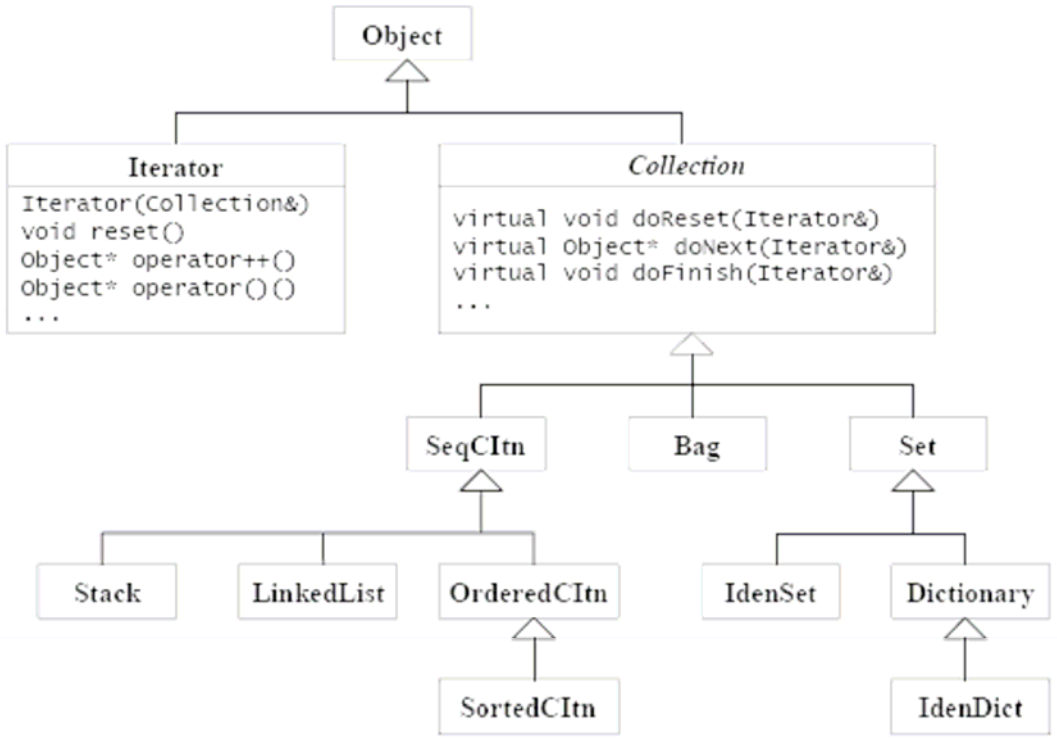


图 14.5 NIHCL 的类别阶层体系（class hierarchy）

「静态多型」适合编写十足的基础计算结构。相形之下，「动态多型」需要选择一个共通基础 类型（common base type），意味其库往往是领域专属的（domain-specific）。因此，毫不 奇怪，C++ 标准库中的 STL 从未包含「多型容器」（译注：意指可放置任意类型的元素，类似 Java Collection classes），而是包含了一大套使用静态多型机制的容器和迭代器（正如 14.5
节, p.241 所示）。

中大型 C++ 程序通常同时需要处理本章讨论的两种多型。某些情况下甚至需要将二者紧密结 合。按照我们的讨论，许多情况下的最佳设计其实十分明显。不过，花一点时间考虑可能的长 远演进，也几乎总是能够获得良好的回报。

15

Traits 和 Policy Classes

「特征萃取」和「策略类别」

Templates 使我们能够以形形色色的类型对 classes 和 functions 进行参数化。或许存在这么一种
诱惑：为程序引入尽可能多的 templates parameters，俾使能够订制（*customize*）类型或算法 的方方面面。然后我们的 templatized 组件就可以被实例化，满足客端程序代码的确切需求。然而 从实践观点来看，很少为了求取最大限度的参数化而引入大量 templates parameters。如果客端 程序代码必须一一指定所有相应自变量，那可真是无趣至极。

幸运的是，我们引入的大多数附加参数（extra parameters）都可以有合情合理的默认值。某些情 况下，附加参数完全由少量主参数（main parameters）决定，而且我们发现，这样的附加参数完 全可以省略。其它某些参数可以倚赖主参数而获得默认值。默认值往往足以满足大多数场合的 需要，不过有时候（针对特殊应用）它们也必须被覆写。另有一些附加参数和主参数并不相干， 从某种意义上来说它们也是主参数，只是存在着「几乎总是能够满足需求」的默认值。

译注：把参数分为「主要」和「附加」两类，并不是严谨或常见的说法。在这儿，作者的意思是，一定得由调用端给值（给予自变量）者，称为「主参数」，可由主参数推导者，称为「附加 参数」。额外参数应可于函数重新编写后被去除。

Policy classes 和 **traits**（或 traits templates）是这样一种C++ 编程装置（programming device）：它们可为「工业强度之 templates 设计」中的附加参数带来很大方便的管理。本章展示了许多情 况，证明它们很有用，并示范形形色色的技术。你可以凭借这些技术编写一些强固而威力强大 的装置（devices）。

15.1 示例：序列的累计（Accumulating a Sequence）

计算数值序列（sequence of values）的总和，是十分常见的任务。这个看似简单的问题为我们提 供了优秀的示例，让我们得以介绍 policy classes 和 traits 可发挥的不同层面。

15.1.1 Fixed Traits（固定式特征）

译注：此处所谓 "fixed"，意指「不经由 template parameters 传递」。

首先假设，求和对象（所有数值）都被储存于一个 array 内，而且我们手上有两个指针，分别可以存取首元素和尾元素。由于这是一本 templates 相关书籍，所以我们希望编写出一个适用多种

类型的 template。就目前而言，以下程序代码看起来直截了当：

```
//traits/accum1.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

template <typename T>
inline
T accum (T const* beg, T const* end)
{
    T total = T(); // 假设 T()产生一个零值
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif // ACCUM_HPP
```

这里惟一有点微妙的地方在于：如何正确地为某个类型产生一个零值，以便开始求和运算。这 儿用的是算式 T(),它对 int和 float这样的内建数值类型而言可以有效运作见 5.5 节）

为促成我们的第一个 traits template，看看下面程序如何使用 accum()：

```
//traits/accum1.cpp

#include "accum1.hpp"
#include <iostream>

int main()
{

    // 建立一个带有 5个整数值的 array
    int num[] = { 1, 2, 3, 4, 5 };

    // 打印平均值
    std::cout << "the average value of the integer values is "
              << accum(&num[0], &num[5]) / 5
              << '\n';

    // 建立一个带有 char元素值的 array
```

```
    char name[] = "templates";
    int length = sizeof(name)-1;

    // 尝试打印 char平均值
    std::cout << "the average value of the characters in \""
              << name << "\" is "
              << accum(&name[0], &name[length]) / length
              << '\n';
}
```

程序的前半部份以 accum()求 5 个整数的和：

```
int num[] = { 1, 2, 3, 4, 5 };
...
accum(&num[0], &num[5])
```

只要将求和结果除以 array 的元素个数，就可以得到平均值。

程序后半段试图对单字 "templates" 中的所有字母做同样的事前提是在一个实际字符集中，字符 a到字符 z构成一个连续序列这对 ASCII 字符集是成立的对 EBCDIC 字符集就不然）按照推测，计算结果应该位于 a和 z之间。在今天的大多数平台上，实际值由 ASCII 码决定：a被编码为 97，z被编码为 122，因此我们预期计算结果落在 97 和 122 之间。然而在我们的平台
上，程序输出如下：

```
the average value of the integer values is 3
the average value of the characters in "templates" is -5
```

问题在于我们的 template 乃是针对 char类型而实例化，这个类型的数值范围太小，以至于即使面对相当小的数值都不足以完成累计动作。显然，我们可以引入额外的 templates parameterAccT 来描述变量 total 的类型（同时也是回返值类型），解决这个问题。然而这会造成此一template 所有用户的一份额外负担：每次调用这个 template 都不得不多指定一个类型。我们的例 子因而必须改写为：

```
accum<int>(&name[0],&name[length])
```

这种约束（或说负担）虽不过份，毕竟可以避免。

下面是上述「附加参数」解法的一个替代方案：为调用 accum()时的「累积物类型 T」和容纳累 计结果的「返回类型 R」产生一份关联（association），这份关联可被视为类型 T 的一种特征（characteristic）。因此返回类型（总量类型）有时称为 T的一个 **trait**（特征、特性）。这个关联性可被编写为 template specializations：

```
//traits/accumtraits2.h
pp template<typename T>
class AccumulationTraits;
```

```
template<>
class AccumulationTraits<char> {
  public:
    typedef int AccT; //译注：一个 int通常可以容纳两个 char相加
};

template<>
class AccumulationTraits<short> {
  public:
    typedef int AccT; //译注：一个 int通常可以容纳两个 short相加
};

template<>
class AccumulationTraits<int> {
  public:
    typedef long AccT; //译注：一个 long通常可以容纳两个 int相加
};

template<>
class AccumulationTraits<unsigned int> {
  public:
    typedef unsigned long AccT;
                    //译注：一个 unsigned long通常可以容纳两个 unsigned int 相加
};

template<>
class AccumulationTraits<float> {
  public:
    typedef double AccT;    //译注：一个 double通常可以容纳两个 float相加
};
```

上述的 template AccumulationTraits即是所谓的 traits template，因为它持有其参数类型的一个 trait（特征）。通常会有一个以上的 traits 和一个以上的参数。此例并不为这个 template 提供
一般性定义（译注：也就是不定义一个定义式通吃各种类型 T），因为如果我们不知道元素型 别是什么，就没有很好的办法为它选取合适的「成果类型」。不过元素类型 T 本身通常可以成 为「成果类型」的优秀人选（尽管先前的例子显然没有这么做）。

有了这样的想法，我们将 accum()template 改写如下：

```
//traits/accum2.hpp
```

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits2.hpp"

template <typename T>
inline
typename AccumulationTraits<T>::AccT accum (T const* beg,
                                            T const* end)
    //译注：以上灰底程序代码的意思是：把 T丢进某个「特征萃取机制」中，取其 Acct特征。
{
    // 令回返类型（return type）为「元素类型的特征（traits）」
    typedef typename AccumulationTraits<T>::AccT AccT;

    AccT total = AccT(); // 假设 AccT()真的产生一个零值
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif // ACCUM_HPP
```

于是程序的输出结果就成了我们所期望的：

```
the average value of the integer values is 3
the average value of the characters in "templates" is 108
```

大体上，由于增加了一个「可协助我们订制算法」的极有用机制，因此任何修改也就不怎么引人注目了。更进一步说，若有任何新类型打算用于 accum()，我们可以将一个适当的 AccT与之产生关联 — 只要为 AccumulationTraits template 声明另一份显式特化（explicit specialization）即可。任何类型都可以如法炮制，包括基础类型（fundamental types）、声明于 其它库中的类型…等等。

15.1.2 Value Traits（数值式特征）

到目前为止，我们已经看过以 traits 来表达「与某给定主类型有所关联」的类型附加信息。本节 将告诉你，这种附加信息并不仅仅局限于类型（译注：上一节的 Acct）。事实上常数和其它种 类的数值（译注：本节将出现的 zero）一样可以和某个类型产生关联。

accum() template 最初版本乃是利用「回返类型（return type）之 *default* 构造函数」，将用以储存 结果的那个变量初始化为「类零值」（zero-like value）（译注：之所以说「类零值」乃是因为 并非全为零，例如 bool的初值是 false）：

```
AccT total = AccT(); //假设 AccT()真的产生一个零值（zero value）
...
return total;
```

但显然这并不保证能够产生一个得以开始进行「累计循环」的良好值，因为 AccT说不定根本就 没有 *default* 构造函数。

这时候 traits 可以再次充当救兵。就本例而言，我们可以对 AccumulationTraits加入一个新的 value trait：

```
//traits/accumtraits3.hpp

template<typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
  public:
    typedef int AccT;
    static AccT const zero = 0;
};

template<>
class AccumulationTraits<short> {
  public:
    typedef int AccT;
    static AccT const zero = 0;
};
```

15.1 示例：序列的累计（Accumulating a Sequence）

```
template<>
class AccumulationTraits<int> {
  public:
    typedef long AccT;
    static AccT const zero = 0;
};
//...
```

这种情况下，新的 trait 是一个可于编译期核定的常数。于是 accum()可改写为：

```
//traits/accum3.hpp
```

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits3.hpp"

template <typename T>
inline
typename AccumulationTraits<T>::AccT accum (T const* beg,
                                            T const* end)
{
    // return type is traits of the element type
    typedef typename AccumulationTraits<T>::AccT AccT;

    AccT total = AccumulationTraits<T>::zero; //译注：使用前述三个 zero之一
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif // ACCUM_HPP
```

在这段程序代码中，用以储存累计值的那个变量，其初始化动作还是十分简单明了：

```
AccT total = AccumulationTraits<T>::zero;
```

这种作法的缺点之一在于，面对 class 内的 static const 成员变量，只有当它是整数（integral）或
enum 时，C++ 才允许我们对它进行初始化动作；我们自己定义的 classes 被排除在外，浮点数 也被排除在外，不得直接设定初值。因此下面的特化版本是错误的：

```
...
template
<>
class AccumulationTraits<float> {
public:
    typedef double AccT;
    static double const zero = 0.0; //错误：不是整数类型（integral type）
};
```

一个简单的替代方案是：不要在 class 内「定义」value trait：

```
...
```

```
template
<>
class AccumulationTraits<float> {
public:
    typedef double AccT;
    static double const zero;   //译注：不再定义其值
};
```

这么一来初值设定器（initializer）就会到某个源码文件中寻找类似下面的句子：

```
...
double const AccumulationTraits<float>::zero = 0.0;
```

尽管这样可以有效运作，但它有个缺点：对编译器而言更加不透明了。编译器处理客户端（client）文件时往往察觉不出「定义式」位于其它文件中。这种情况下编译器无法运用「数值 zero其实 就是 0.0」这一事实。

因此，我们更倾向于实作这样的 value traits：不强求以 *inlined* 成员函数传回一个整数值（integral values）。 比方说我们可以将 AccumulationTraits改写如下：

```
// traits/accumtraits4.hpp

template<typename T>
class AccumulationTraits;

template<>
class AccumulationTraits<char> {
  public:
    typedef int AccT;
    static AccT zero() {

       return 0;
    }
};

template<>
class AccumulationTraits<short> {
  public:
    typedef int AccT;
    static AccT zero() {
       return 0;
    }
};
```

```
template<>
class AccumulationTraits<int> {
  public:
    typedef long AccT;
    static AccT zero() {
       return 0;
    }
};

template<>
class AccumulationTraits<unsigned int> {
  public:
    typedef unsigned long AccT;
    static AccT zero() {
       return 0;
    }
};

template<>
class AccumulationTraits<float> {
  public:
    typedef double AccT;
    static AccT zero() {
       return 0.0;
    }
};

...
```

对应用程序而言，新作法造成的惟一区别是：改用函数调用语法（而不是较简单的「static 成员变量存取语法」）：

```
AccT total = AccumulationTraits<T>::zero();
```

显然 traits 的作用不仅仅局限于「提供额外类型」。在我们的例子中，它可以作为一种机制，提供 accum()需要知道的「元素类型的所有必要信息」。这正是 traits 概念的关键：它提供一种途 径，用以为具体元素（通常是某些类型）设定某些信息，供泛型计算时使用。

15.1.3 Parameterized Traits（参数式特征）

先前小节中的 accum()所使用的 traits 被称为 **fixed**（固定式），因为一旦 decoupled trait（解耦 用的特征萃取机制）定义完毕，你就不可以在算法中覆写（overridden）它。不过某

些情况下 这样的覆写可能是我们想要的，例如当我们碰巧知道某一套浮点数可被安全地累计放进一个同 型的（浮点数）变量中，「覆写 traits」就有可能使我们的开发更高效。

原则上，解决方案是这样：添加一个带有默认值的 template parameter，默认值由我们的 traits template 决定。这么一来大部分用户可以忽略额外的 template argument，而特殊需求的用户则可 以覆写(override)预先设定的累计类型。此方案惟一令人不快的是：function templates 不能拥 有预设的 template arguments[61]。

目前就让我们透过「把算法规划为一个 class」来智取问题。这也说明了一个事实：traits 可用于 class templates 之中，而且至少像用于 function templates 那样容易。我们的应用程序不足之处 在于：class templates 无法进行 template argument deduction（模板自变量推导），因此自变量必须被 明确提供出来。因此我们需要以下形式：

```
Accum<char>::accum(&name[0], &name[length])
```

来使用新修订的「累计用 template」：

```
// traits/accum5.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits4.hpp"
template <typename T,
          typename AT = AccumulationTraits<T> >
class Accum {
  public:
    static typename AT::AccT accum (T const* beg, T const* end) {
        typename AT::AccT total = AT::zero();
        while (beg != end) {
            total += *beg;
            ++beg;
        }
        return total;
    }
};

#endif // ACCUM_HPP
```

上述 template 的大多数用户很可能从来都不需要明确提供第二个 template argument，因为根据第一个 template argument 就可以获得适当默认值。

通常我们会导入一些便捷函数（convenience functions）用以简化界面：

```
template <typename T>
```

```
inline
typename AccumulationTraits<T>::AccT accum (T const* beg,
                                            T const* end)
{
   return Accum<T>::accum(beg, end);
}

template <typename Traits, typename T>
inline
typename Traits::AccT accum (T const* beg, T const* end)
{
   return Accum<T, Traits>::accum(beg, end);
}
```

15.1.4 Policies（策略）和 Policy Classes（策略类别）

到目前为止，我们一直都将累计（accumulation）与求和（summation）混为一谈。显然我们其 实可以设想其它种类的累计。例如我们可以求给定之实值序列的乘积；如果被操作的实值是字 串，我们可以将它们串接起来；甚至「寻找序列中的最大值」也可被归结为累计问题。在所有 情况中，`accum()`惟一需要修改的就是 `total += *beg`。这个操作可被称为「累计运算」过程中的一个 **policy**（策略）。因此所谓 **policy class** 就是这样的 class：提供一个接口，从而得以在算法中运用一或多个 policies。

下面是个例子，示范如何在 `Accum` class template 中引入上述界面：

```
// traits/accum6.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits4.hpp"
#include "sumpolicy1.hpp"

template <typename T,
          typename Policy = SumPolicy,
          typename Traits = AccumulationTraits<T> >
class Accum {
  public:
    typedef typename Traits::AccT AccT;
    static AccT accum (T const* beg, T const* end)
       { AccT total = Traits::zero();
       while (beg != end) {
          Policy::accumulate(total, *beg);
```

```
            ++beg;
        }
        return total;
    }
};

#endif // ACCUM_HPP
```

其中用到的 SumPolicy可以这样定义：

```
// traits/sumpolicy1.hpp

#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP

class SumPolicy {
  public:

    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const & value) {
        total += value;
    }
};

#endif // SUMPOLICY_HPP
```

在这个例子中，我们选择让 policy 成为常规 class（而不是个 class template），它具有一个 static member function template（而且是隐式内联, implicitly inline）。另一个替代方案将于稍后讨论。

透过「指定不同的 policy」来进行累计动作，我们就可以完成多种计算。例如下面这个程序，其 目的是计算某些数值的乘积（product）：

```
// traits/accum7.cpp

#include "accum6.hpp"
#include <iostream>

class MultPolicy {
  public:
    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const& value) {
        total *= value;
    }
```

```
};

int main()
{
    // 产生一个 array，内有 5个整数值
    int num[] = { 1, 2, 3, 4, 5 };

    // 打印所有数值的乘积
    std::cout << "the product of the integer values is "
              << Accum<int,MultPolicy>::accum(&num[0], &num[5])
              << '\n';
}
```

然而上述程序的输出结果不如预期：

```
the product of the integer values is 0
```

问题在于我们对初始值的不当选择：初始值 0 对「求和」而言很合适，对「乘积」来说就不妥了（初始值为 0，注定乘积结果也是 0）。这也说明了不同的 traits 和 policies 可能相互影响，同 时也强调了谨慎设计 templates 的重要性。

从这个例子中我们可以意识到，accumulation loop（累计循环）的初始化应该成为 accumulation policy 的一个成份。这个 policy 可以使用也可以不使用 trait `zero()`。其它替代方案也不该被遗忘 — 并不是什么事都非得使用 traits 和 policies 解决不可。C++ 标准库的 `accumulate()`就是接受第三个 call argument 做为初始值。

15.1.5 Traits 和 Policies 有何差异？

我们可以举出相当合理的例子来支持一个事实：policies 只是 traits 的特例，反过来说 traits 只不 过是「对 policy 的编码（encode）」而已。

《*New Shorter Oxford English Dictionary*》（新简洁牛津英文字典；[NewShorterOED]）这样说：

> **trait**, n. ... *a distinctive feature characterizing a thing*
>
> 某种独特特征，用以刻画（描绘）事物。
>
> **policy**, n. ... *any course of action adopted as advantegous or expedient*
>
> 为了优势或权宜（方便）而采纳的任何行动方向（方针）

基于此，我们往往将所谓的 policy classes 限定为这样的 classes：对某种行为进行编码（encode），该种行为很大程度与同组其它任何 template arguments 不相干（亦即正交，orthogonal）。这和 Andrei Alexandrescu 在其《*Modern C++ Design*》（[AlexandrescuDesign], p.8）书中的陈述一致[63]：

> *Policies have much in common with traits but differ in that they put less emphasis on type and more on behavior.* （Policies 和 traits 之间有许多共同点，但前者更强调行为而非类型。）

traits 技术的引入者 Nathan Myers，提出了更开放的定义（见 [MyersTraits]）：

Traits class: A class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that "extra level of indirection" that solves all software problems. （Traits class：一种被用来取代 template parameters 的 class。作为一个 class，它聚合了有用的类型和常数；作为一个 template，它为「可用来解决所有软件 难题」的所谓「额外间接层」提供了一条康庄大道。）

通常我们倾向于使用如下（稍微模糊）的定义：

Traits 表现一个 template parameter 的自然附加属性（natural additional properties）。
Policies 表现泛化函数和类型间的「可设置行为」（configurable behavior；通常带有默认值）。

为了进一步阐述两种概念之间的可能区别，我们列出对 traits 的观察：

Traits 作为 fixed traits（也就是说不经由 template parameters 传递）是有用的。

Traits 参数通常具有十分自然的默认值（它们很少被覆写，或不可被覆写）。

Traits 参数倾向于紧密依赖一或多个主参数（main parameters）。

Traits 通常与类型和常数（而非成员函数）结合使用。

Traits 倾向被收集于所谓的 traits templates 中。

对于 policy classes，我们列出如下观察：

如果不以 template parameters 传递的话，policy classes 作用不大。

Policy 参数无需拥有默认值，而且常被明确指定（虽然许多泛型组件有常用的默认值）。

Policy 参数通常和 templates 的其它参数「不相干」（正交; orthogonal）。

Policy classes 通常和成员函数结合使用。

Policies 可被收集于 "plain"（单纯的、简朴的）classes 或 class templates 中。

当然，两个术语之间的界限并不是那么泾渭分明。例如C++ 标准库的 **character traits** 也定义了诸如比较、搬移、搜寻字符的功能性行为。只要替换这些 character traits，你就可以在保持 相同字符类型（译注:意指设计上无太大改变）的情况下定义一个可区分大小写的 string classes（见 [JosuttisStdLib] 11.2.14 节）。因此尽管它们被称为 traits，也具有一些 policies 相关属性。

15.1.6 Member Templates vs. Template Template Parameters

为实作一个 accumulation policy（译注：用以决定累计动作是「加」或「乘」或其它…），我们选择将 `SumPolicy`和 `MultPolicy`表现为「拥有一个 member function template」的一般性 classes（译注: 正如 p.256 和 p.257 所列）。另一种作法是以 class templates 来设计 policy class interface，然后可以被拿来作为 template template arguments 使用。例如我们可以将 `SumPolicy`改写为一个 template：

```
// traits/sumpolicy2.hpp

#ifndef SUMPOLICY_HPP
```

```
#define SUMPOLICY_HPP

template <typename T1, typename T2>
class SumPolicy {
  public:
    static void accumulate (T1& total, T2 const & value) {
        total += value;
    }
};

#endif // SUMPOLICY_HPP
```

然后我们可以修改 Accum接口，俾能使用一个 template template parameter：

```
// traits/accum8.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits4.hpp"
#include "sumpolicy2.hpp"

template <typename T,
          template<typename,typename> class Policy = SumPolicy,
          typename Traits = AccumulationTraits<T> >
class Accum {
  public:
    typedef typename Traits::AccT AccT;
    static AccT accum (T const* beg, T const* end)
        { AccT total = Traits::zero();
        while (beg != end) {
            Policy<AccT,T>::accumulate(total, *beg);
            ++beg;
        }
        return total;
    }
};

#endif // ACCUM_HPP
```

对 traits 参数也可以如法炮制。（关于这个主题，亦存在其它可能方案。例如不再「将 AccT 明显传递给 policy」，改为传递 accumulation trait 并让 policy 根据 traits 参数来决定其成果类型。）

透过 template template parameter 来存取 policy classes，主要的优点是，这使得 policy class 更容易携带若干状态信息（亦即 static 成员变量）——只需利用一个取决于 template parameters 的型别即可。（在我们给的第一种作法中，static 成员变数不得不嵌于一个 member class template 内）然而 template template parameter 作法亦有不利的一面，policy classes 如今必须被写成 template，携带一组由我们的接口所定义的精确 template parameters。不幸的是这将造成 policies 不得带有任何额外的 template parameters。举个例子，我们可能希望为 SumPolicy添加一个 boolnontype template parameter，由它决定「求和动作」该使用 += 运算符或 + 运算符。我们只需将原本的 SumPolicy（译注：p.256）修改如下即可：

```
// traits/sumpolicy3.hpp

#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP

template<bool use_compound_op = true>
class SumPolicy {
  public:
    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const & value) {
        total += value;
    }
};

template<>
class SumPolicy<false> {
  public:
    template<typename T1, typename T2>
    static void accumulate (T1& total, T2 const & value) {
        total = total + value;
    }
};

#endif // SUMPOLICY_HPP
```

但对于使用 template template parameters 的 Accum实作品来说，这样的修改就不再可能。

15.1.7 联合多个 Policies 和/或 Traits

先前的开发显示，traits 和 policies 并没有完全消除「对多个 template parameters 的需求」，然而它们的确使参数数量减少了，使事情变得好管理了。接下来又出现一个有意思的问题：如何安排这些参数的顺序呢？

一个简单的策略是：根据其默认值被选用的可能性，以递增顺序安排参数顺序。通常这意味 traits 参数跟在 policy 参数后面，因为 policy 参数更常被客端程序代码覆写。细心的读者可能已

经注意 到了，我们的范例程序就是采用这种策略。

如果打算对程序代码添加大量复杂玩意儿，另一种作法是本质上允许以任何顺序指定非预设自变量（non-default arguments）。详见 16.1 节。第 13 章也讨论了将来可能出现、可简化这方面设计的一些 templates 新性质。

15.1.8 以泛型迭代器（General Iterators）进行累计

结束这段对 traits 和 policies 的介绍前看一个加入泛型迭代器非仅指针处理能力的 accum() 版本，应该颇具启发意义。这种能力正是工业强度泛型组件所期望拥有的。有趣的是，这个版本仍然允许我们在调用 accum()时使用指针，因为C++ 标准库提供了所谓的 **iterator traits**（traits 真是无处不在！），于是我们可以将最初版本的 accum()定义如下（不考虑后来对它的 强化）：

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

#include <iterator>

template <typename Iter>
inline
typename std::iterator_traits<Iter>::value_type
accum (Iter start, Iter end)
{
    typedef typename std::iterator_traits<Iter>::value_type VT;

    VT total = VT(); // assume VT() actually creates a zero value
    while (start != end) {
        total += *start;
        ++start;
    }
    return total;
}

#endif // ACCUM_HPP
```

iterator_traits 将迭代器的所有相关属性封装起来。由于它有一个「针对指针」的偏特化版 本，所以这些 traits 可以很方便地和任何一般指针一块儿被使用。下面展示标准库实作品对 于这种支持能力的可能实现手法：

```
namespace std {
    template <typename T>
    struct iterator_traits<T*> {
        typedef T                               value_type;
```

```
        typedef ptrdiff_t                       difference_type;
        typedef random_access_iterator_tag    iterator_category;
        typedef T*                              pointer;
        typedef T&                              reference;
    };
}
```

然而目前并不存在现成可用于「数值累计」类型可供迭代器指涉（*refers*），因此我们还是需要设计自己的 `AccumulationTraits`。

15.2 Type Functions（译注：对比于 value function） 译注：本节（含小节）较多保留了 types（类型）, values（数值）, functions（函数）等英文词。

早先那个 traits 例子展示一个事实：你可以定义「因 types 而异」的行为。这有别于你通常在程 式中实作的东西。在 C 和C++ 中，functions 可被更确切地称为 value functions：它们接受某些 values 做为自变量并传回一个 value 做为结果现在我们可以运用 templates 技术实作出所谓的 type functions：接受一些 types 作为自变量，并产生一个 type 或 constant 作为结果。

`sizeof` 就是一个非常有用的内建 type function。它传回一个常数，记录特定自变量（某个 type） 的体积（以 bytes 为单位）。class templates 也可以充当 type functions，此时的参数就是 template parameters，其结果则被萃取为一个 member type 或 member constant。例如，运用 `sizeof`运算 子可制作出如下接口：

```
// traits/sizeof.cpp

#include <stddef.h>
#include <iostream>

template <typename
T> class TypeSize
{ public:
    static size_t const value = sizeof(T);
};

int main()
{
    std::cout << "TypeSize<int>::value = "
              << TypeSize<int>::value << std::endl;
}
```

稍后我们将开发一些更通用的 type functions，可根据上述方式被视为 traits classes 使用。

15.2.1 决定元素类型（Elements Types）

看看另一个例子。假设我们有许多 container templates（容器模板）如 vector<T>、list<T> 和 stack<T> 等等。我们希望有个 type function，在接受一个 container type（容器类型）之后可以 输出 element type（元素类型）。这可透过偏特化达成：

```
// traits/elementtype.cpp

#include <vector>
#include <list>
#include <stack>
#include <iostream>
#include <typeinfo>

template <typename T>
class ElementT;                          // 主模板（primary template）


template <typename T>
class ElementT<std::vector<T> > {    // 偏特化（partial specialization）
  public:
    typedef T Type;
};

template <typename T>
class ElementT<std::list<T> > {      // 偏特化（partial specialization）
  public:
    typedef T Type;
};

template <typename T>
class ElementT<std::stack<T> > {     // 偏特化（partial specialization）
  public:
    typedef T Type;
};

template <typename T>
void print_element_type (T const & c)
{
    std::cout << "Container of "
              << typeid(typename ElementT<T>::Type).name()
```

```
              << " elements.\n";
}

int main()
{
    std::stack<bool> s;
    print_element_type(s);
}
```

透过偏特化我们得以实现这项功能而无需要求container types 必须对 type function 有所了解」然而在很多场合中，type function 往往和可施行于其上的 types 一块设计，让实作码得以简化。举个例子，假设 container types 定义了一个 member type value_type（一如标准容器所为），我们就可以编写这样的程序代码：

```
template <typename
C> class ElementT
{ public:
    typedef typename C::value_type Type;
};
```

这可以作为一份预设实作品，它并不排斥那些「未曾定义 member type value_type」的任何 container types 特化体。尽管如此，为 template type parameters 提供类型定义（typedef），使它们（因为简短或因为别具命名意义）得以更容易在泛型程序代码中被取用，往往是明智之举。下面程序代码勾勒出这种想法：

```
template <typename T1, typename T2, ... >
class X {
  public:
    typedef T1 ... ;
    typedef T2 ... ;
    ...
};
```

type function 的价值怎么体现出来呢？它允许我们以 container type 来参数化某个 template，而且并不要求「必须一并提供 element type 的相关参数及其它特征」。例如我们不再这么写：

```
template <typename T, typename C>
T sum_of_elements (C const& c);
```

因为如果那么写，就需要以 sum_of_elements<int>(list)之类的语法来明确指定 element type。我们可以改而这么声明：

```
template<typename C>
typename ElementT<C>::Type sum_of_elements (C const& c);
```

如此一来，element type 就可以由 type function 来决定了。

注意，traits 可被实作为「对现有 types 的一个扩展」。因此你甚至可以定义 type functions 来处 理基本类型（fundamental types）和封闭型库中的类型（types of closed libraries）。

这种情况下 ElementT被称为一个 traits class，因为它被用来取得某给定之 container type C的 trait（通常这样的 class 会容纳不止一个 trait）。因此 traits classes 并不仅仅局限于描述容器参数特 征（characteristics of container parameters），还可以描述任何一种主参数（main parameters）。

15.2.2 确认是否为 Class Types

下面的 type function 可以协助我们判断某个 type 是不是个 class type：

```
// traits/isclasst.hpp

template<typename
T> class IsClassT
{ private:
    typedef char One;
    typedef struct { char a[2]; } Two;
  template<typename C> static One test(int C::*);
  template<typename C> static Two test(...); public:
    enum { Yes = sizeof(IsClassT<T>::test<T>(0)) == 1 };
    enum { No = !Yes };
};
```

这个 template 使用 8.3.1 节， p.106 所说的 SFINAE（substitution-failure-is-not-an-error；替换失败 并非错误）原理。SFINAE 的运用关键是找到一个对 class types 无效但对其他 types 有效的类型 构件（type construct）；反之亦可。对于 class types 我们可以倚赖这样的观察结论：只有当 C是
个 class type 时，pointer-to-member（像是 int C::*）这样的类型构件才有效。

以下程序使用了这个 type function 来测试一些 types 和 objects 究竟是不是 class types：

```
// traits/isclasst.cpp

#include <iostream>
#include "isclasst.hpp"

class MyClass {
};
```

```
struct MyStruct {
};

union MyUnion {
};

void myfunc()
{
}

enum E { e1 } e;

// 把类型当做模板自变量（template argument）传进去检验
template <typename T>
void check()
{
    if (IsClassT<T>::Yes) {
        std::cout << " IsClassT " << std::endl;
    }
    else {
        std::cout << " !IsClassT " << std::endl;
    }
}

// 把类型当做函数调用自变量（function call argument）传进去检验
template <typename T>
void checkT (T)
{
    check<T>();
}

int main()
{
    std::cout << "int:    ";
    check<int>();

    std::cout << "MyClass: ";
    check<MyClass>();

    std::cout << "MyStruct:";
    MyStruct s;
    checkT(s);
```

```
    std::cout << "MyUnion: ";
    check<MyUnion>();

    std::cout << "enum:   ";
    checkT(e);

    std::cout << "myfunc():";
    checkT(myfunc);
}
```

程序输出如下：

```
int:
            !IsClas
sT MyClass:
            IsClass
T MyStruct:
            IsClass
T MyUnion:
            IsClass
T enum:
            !IsClas
sT myfunc():
            !IsClas
sT
```

15.2.3 References（引用）和 Qualifiers（饰词）

考虑下面的 function templates 定义：

```
// traits/apply1.hpp

template <typename T>
void apply (T& arg, void (*func)(T))
{
    func(arg);
}
```

再考虑以下程序代码（试图使用上面那个 function templates）：

```
// traits/apply1.cpp
```

```
#include <iostream>
#include "apply1.hpp"

void incr (int& a)
{
   ++a;
}

void print (int a)
{
   std::cout << a << std::endl;
}

int main()
{
   int x = 7;
   apply (x, print);
   apply (x, incr);
}
```

程序中的调用动作：

```
apply (x, print)
```

没问题，T被 int取代，apply()的参数类型分别是 int& 和 void(*)(int)，对应于所得的引数类型。但是下面这个调用动作：

```
apply (x, incr)
```

就不那么简单了。要想匹配第二参数，T 必须被替换为 int&，这就意味第一参数的类型必须是 int& &，但这并不是个合法的C++ type。的确，在原始C++ *Standard* 中这种替换并不合法，但 正因为上述这一类例子，后来的一份技术勘误（对C++*Standard* 的一些小修改，见[Standard02]） 指出：T& 中的 T如果被替换为 int&，那么 T& 等价于 int&。

面对尚未实现此一新式reference 替代规则的C++ 编译器我们可以为它建立一个 type function 充当「*reference* 运算符」，若且惟若（*if and only if*）type 不是个 reference。我们还可以提供相 反的操作：将「*reference* 运算符」剥除，若且惟若讨论中的 type 是个 reference。如果我们兴致 够高，还可以添加或剥除 const 饰词。这些都可以使用以下泛型定义之偏特化版本达成：

```
// traits/typeop1.hpp
template <typename T>
class TypeOp {          // primary template（主模板）
  public:
    typedef T         ArgT;
    typedef T         BareT;
    typedef T const   ConstT;
```

```
    typedef T &       RefT;
    typedef T &       RefBareT;
    typedef T const & RefConstT;
};
```

首先，以一个偏特化版本处理 const types：

```
// traits/typeop2.hpp
template <typename T>
class TypeOp <T const> {   // 针对 const types 而设计的偏特化
  public:
    typedef T const   ArgT;
    typedef T         BareT;
    typedef T const   ConstT;
    typedef T const & RefT;
    typedef T &       RefBareT;
    typedef T const & RefConstT;
};
```

这个用来处理 reference types 的偏特化版本也有能力处理 reference-to-const types。于是，必要 时可递归执行 TypeOp 以取得 "bare type"（剥除各种饰词之后的 type）。对比于此，即使某个 template parameter 被「本身已是 const」的某个 type 替换，C++ 也允许我们将 const饰词施行 于该 template parameter 身上。因此无论我们怎么使用 const，都无需操心是否需要剥除 const 饰词。

```
// traits/typeop3.hpp
template <typename T>
class TypeOp <T&> {      // 针对 references 而设计的偏特化
  public:
    typedef T &                          ArgT;
    typedef typename TypeOp<T>::BareT    BareT;
    typedef T const
                                         const
    T;

    typedef T &                          RefT;
    typedef typename TypeOp<T>::BareT &  RefBareT;
    typedef T const &
                                         RefConstT
    ;
};
```

注意，references-to-void types 是不被允许的。不过有时候能够处理诸如「无任何饰词的 void」这类 types 还是颇有用处。下面的偏特化版本便是用来处理此事：

```
// traits/typeop4.hpp
template<>
```

```
class TypeOp <void> {     // 针对 void 而设计的偏特化
  public:
    typedef void           ArgT;
    typedef void           BareT;
    typedef void const     ConstT;
    typedef void           RefT;
    typedef void           RefBareT;
    typedef void
                           RefConst
    T;
};
```

有了这些，我们就可以将 apply template 改写如下：

```
template <typename T>
void apply (typename TypeOp<T>::RefT arg, void (*func)(T))
{
   func(arg);
}
```

记住，现在的 T不再可从第一自变量推导而来，因为它如今出现于一个名称饰词（name qualifier）之中。现在这个 T只能从第二自变量推导而来，而后才被用来建立第一参数的 type。

15.2.4 Promotion Traits（类型晋升之特征萃取）

到目前为止，我们已经研究和开发「针对单一 type」的 type functions：只要给定一个 type，其
他相关的 types 或 constants 便即获得定义。通常我们还可以开发「与多个自变量相依」的 type functions。举个例子，当我们编写所谓 **promotion traits** 这样的 operator templates 时，这就非常 有用。为了刺激这个构想，让我们编写一个可将两个 Array容器相加的 function template：

```
template<typename T>
Array<T> operator+ (Array<T> const&, Array<T> const&);
```

这很好，但由于语言允许我们将一个 char和一个 int相加，我们十分希望诸如此类的混合操作 也能实施于 arrays 身上。这么一来我们就必须决定新版本的回返类型（return type）应该是什么：

```
template<typename T1, typename T2>
Array<???>
operator+ (Array<T1> const&, Array<T2> const&);
```

promotion traits template 使我们能够以如下方式填充上述声明中的问号：

```
template<typename T1, typename T2>
```

```
Array<typename Promotion<T1, T2>::ResultT>
operator+ (Array<T1> const&, Array<T2> const&);
```

或使用如下方式：

```
template<typename T1, typename T2>
typename Promotion<Array<T1>, Array<T2> >::ResultT
operator+ (Array<T1> const&, Array<T2> const&);
```

此种想法在于为 template Promotion 提供大量特化版本，以创建出一个满足我们所需的 type function。另一个应用由 max() template 激发出来：当我们希望指出类型不同的两数值中的较大 者时，应该采用「较大」的那个类型（见 2.3 节, p.13）。

这个 template 并不存在真正的泛型定义。最好的选择是：别去定义 primary class template（主模 板）：

```
template<typename T1, typename T2>
class Promotion;
```

另一个必须采取的措施是，如果某个 type 比另一个 type 大，我们应该提升至较大的那个 type。 这可利用一个专门的 template IfThenElse（其中带有一个 bool nontype template parameter）对 type parmeters 进行二选一：

```
// traits/ifthenelse.hpp   （译注：VC6无法编译，因不支持偏特化）
#ifndef IFTHENELSE_HPP
#define IFTHENELSE_HPP

// primary template: 根据第一自变量导出第二或第三变量
template<bool C, typename Ta, typename Tb> class
IfThenElse;

// 偏特化: true 导出第二自变量
template<typename Ta, typename Tb>
class IfThenElse<true, Ta, Tb>
{ public:
    typedef Ta ResultT;
};

// 偏特化: false 导出第三自变量
template<typename Ta, typename Tb>
class IfThenElse<false, Ta, Tb>
{ public:
    typedef Tb ResultT;
};

#endif // IFTHENELSE_HPP
```

有了这些东西，我们就可以根据需要提升之 types 的体积大小，在 T1、T2 和 void 之间创造出一个「三向选择」：

```
// traits/promote1.hpp

// type promotion 的主模板（primary template）
template<typename T1, typename T2>
class Promotion {
  public:
    typedef typename
           IfThenElse<(sizeof(T1)>sizeof(T2)
                      ), T1,
                      typename IfThenElse<(sizeof(T1)<sizeof(T2)),
                                          T2,
                                          v
                                          o
                                          i
                                          d
                                         >::Resu
                                           ltT
                     >::ResultT ResultT;
};
```

primary template 所使用的「以大小为根据的探索法」（size-based heuristic）有时可以有效运作，但需要检验。如果它选择了一个错误 type，那么必须有一个适当的特化版本用来推翻这个选择。 从另一方面说，如果两个 types 完全一样，我们可以安全地令它成为 promoted type。下面的偏特化版本考虑了这一点：

```
// traits/promote2.hpp
// 针对「两个 types 完全相同」而设计的偏特化版本。
template<typename T>
class Promotion<T,T>
{ public:
    typedef T ResultT;
};
```

我们需要很多特化版本来记录语言基本(内建)类型的晋级(promotion)原则。以下宏（macro） 多少可以减少一些源码数量：

```
// traits/promote3.hpp
#define MK_PROMOTION(T1,T2,Tr)                \ template<> class Promotion<T1, T2> {   \ public:
    };                                        \
                                             \
    template<> class Promotion<T2, T1> {      \ public:   \ typedef Tr ResultT;    \
    };
```

这么一来，就可以这样加入各种晋级规则：

```
// traits/promote4.hpp

MK_PROMOTION(bool, char, int)
MK_PROMOTION(bool, unsigned char, int)
MK_PROMOTION(bool, signed char, int)
//...
```

这种方式相当直截了当，却需要列举数十个可能组合。各式各样的替代方案也都可能存在，例 如可以修改 `IsFundaT` template 和 `IsEnumT` template 来为整数（integral）类型和浮点数类型定义 promotion type。这么一来 `Promotion`就只需针对「成果基础类型」（resulting fundamental types） 和用户自定类型（user-defined types）完成特化即可。关于后者，你马上就会看到。

一旦为基础（内建）类型（必要的话再加上 enum 类型）定义好 **Promotion**，其它晋级规则通常 可以利用偏特化来表达。以先前的 `Array`为例：

```
// traits/promotearray.hpp
template<typename T1, typename T2>
class Promotion<Array<T1>, Array<T2> > {
  public:
    typedef Array<typename Promotion<T1,T2>::ResultT> ResultT;
};

template<typename T>
class Promotion<Array<T>, Array<T> > {
  public:
    typedef Array<typename Promotion<T,T>::ResultT> ResultT;
};
```

最后这个偏特化版本有若干特别值得注意的地方。乍见之下，好像先前针对类型一致（identical types）而设计的偏特化版本（`Promotion<T,T>`）已经处理了这种情况。不幸的是与偏特化版本 `Promotion<T,T>`相比，偏特化版本 `Promotion<Array<T1>, Array<T2> >` 的偏特化工作既没 多做也没少做（见 12.4 节， p.200）[66]。为避免选择 templates 时发生模棱两可（歧义）情况，最 后那个偏特化版本被加了进来，它比前两个偏特化版本做了更多偏特化工作。

当更多 types 被加入时，为了使 promotion（类型晋升）有意义，我们需要为 `Promotion` template 加入更多的特化版本和偏特化版本。

15.3 Policy Traits

截至目前，我们所给的 traits templates 实例都用以决定 template parameters 的属性（properties），例如它们表示哪一种 type？在混合类型操作（mixed-type operations）中应被提升为哪一种 type？ 这样的 traits 被称为 property traits。

另有一些 traits 负责定义某些 types 应该如何被处置，我们称此为 policy traits。这使我们联想先 前讨论过的 policy classes 概念（我们也早已指出，traits 和 policies 之间的区别并非绝对地泾渭 分明），但 policy traits 倾向于和 template parameter 之间有更多彼此关联的独特属性，而 policy classes 通常和其它 template parameters 保持独立。

Property traits 往往以 type functions 实现，policy traits 则通常将 policy 封装于成员函数。让我们 看第一个例子，那是一个 type function，定义了一个 policy 用以传递惟读参数（read-only parameters）。

15.3.1 惟读的参数类型（Read-only Parameter Types）

在 C 和C++ 中，函数的 call arguments 预设采用 *by value*（传值）方式来传递，这意味「调用者 手上的自变量值」被拷贝到「被调用者所控制的位置」。大多数程序员都知道此举对大型结构而 言代价高昂 — 大型结构更适合采用 *by reference-to-const*（C 语言则是 *by pointer-to-const*）来 传递自变量。不过，对于较小结构来说，整个分际并不总是那么清晰，而且从效率观点来看，最 佳机制取决于程序代码所处的实际架构（exact architecture）。虽然这件事情在大多数情况下并非 多么关键，但有时候即使面对小型结构也必须谨慎处理。

面对 templates，事情变得更为棘手。我们不知道将被用来替换 template parameters 的那个 type 到底有多大。此外，最终决定也不仅仅取决于大小：一个伴有「代价高昂之 *copy* 构造函数」的小 型结构可能会让你醒悟：以 *by reference-to-const* 传递惟读参数还是比较有道理的。

正如先前所暗示，使用一个「身为 type function」的 policy traits template，可以相当方便地处理 这个问题。此一 type function 将一个预期的自变量类型 T映射到一个最佳参数类型 T或 T const&。 初步规划时，你可以在 primary template 中以 *by value* 方式传递「尺码不大于两个指针」的 types， 并以 *by reference-to-const* 方式传递其它所有东西：

```
template<typename
T> class RParam
{ public:
    typedef typename IfThenElse<sizeof(T)<=2*sizeof(void*),
                                T,
                                T const&>::ResultT Type;
};
```

另一方面，对于某些容器类型，虽然其 sizeof返回值可能并不大（译注：因为 sizeof传回的 只是容器本身状态，不含元素），但可能带有代价高昂之 *copy* 构造函数，所以我们需要许多像下 面这样的特化和偏特化版本：

```
template<typename T>
class RParam<Array<T> > {
  public:
    typedef Array<T> const& Type;
```

```
};
```

这一类 types 在C++ 中司空见惯，为求安全最好是在 primary template 中将 nonclass types 标示为以 *by value* 方式传递，然后在需要更佳效率表现时选择性地加入 class types（primary template 可使用 p.266 所示的 `IsClassT<>` 来鉴定 class types）：

```
// traits/rparam.hpp

#ifndef RPARAM_HPP
#define RPARAM_HPP

#include "ifthenelse.hpp"
#include "isclasst.hpp"

template<typename
T> class RParam
{ public:
    typedef typename IfThenElse<IsClassT<T>::No,
                                T,
                                T const&>::ResultT Type;
};

#endif // RPARAM_HPP
```

不论哪一种方式，policy 现在可集中于 traits template 定义式中，客端程序可用它达到良好的效果。举个例子，假设我们有两个 classes，其中之一具体指明「惟读自变量较适合以 *call by value* 传递」：

```
// traits/rparamcls.hpp
#include <iostream>
#include "rparam.hpp"

class MyClass1
  { public:
  MyClass1 () {
    }
    MyClass1 (MyClass1 const&) {
        std::cout << "MyClass1 copy constructor called\n";
    }
};

class MyClass2
  { public:
  MyClass2 () {
    }
```

```
    MyClass2 (MyClass2 const&) {
        std::cout << "MyClass2 copy constructor called\n";
    }
};

// 运用 RParam<>告诉大家：MyClass2 objects 将以 by value 方式传递
template<>
class RParam<MyClass2> {
  public:
    typedef MyClass2 Type;
};
```

现在你可以声明一些函数，针对惟读自变量使用 RParam<>，然后试着调用那些函数：

```
// traits/rparam1.cpp
#include "rparam.hpp"
#include "rparamcls.hpp"

// 以下函数允许参数以 by value 或 by reference 方式传递
template <typename T1, typename T2>
void foo (typename RParam<T1>::Type p1,
          typename RParam<T2>::Type
          p2)
{
    //...
}

int main()
{
    MyClass1
    mc1;
    MyClass2
    mc2;
    foo<MyClass1,MyClass2>(mc1,mc2); //译注：注意，如下所说，无法自动自变量推导。
}
```

不幸的是 使用 RParam会出现一些重大缺点。首先函数的声明明显较为凌乱。更让人讨厌的是：像上述 foo()这样的函数，无法经由自变量推导（argument deduction）被调用，因为 template parameters 只出现在函式参数的饰词（qualifiers）上，因此调用端必须明确指定 templates arguments。

一个笨拙而庞大的解决办法是：使用 inline wrapper function template，而我们假设这个 inline function 会被编译器的优化机制除掉（译注：之所以说「假设」，乃因 inline 成功与否完全由编 译器决定）。例如：

```
// traits/rparam2.cpp

#include "rparam.hpp"
#include "rparamcls.hpp"

// 以下函数允许参数以 by value 或 by reference 方式传递
template <typename T1, typename T2>
void foo_core (typename RParam<T1>::Type p1,
               typename RParam<T2>::Type
               p2)
{
   //...
}

// 下面是个 wrapper，用以免除客户端「明确（显式）指定 template parameter」的责任
template <typename T1, typename T2>
inline
void foo (T1 const & p1, T2 const & p2)
{
   foo_core<T1,T2>(p1,p2);
}

int main()
{
   MyClass1
   mc1;
   MyClass2
   mc2;
   foo(mc1,mc2); // 此式效果相当于 foo_core<MyClass1,MyClass2>(mc1,mc2)
}
```

15.3.2 拷贝（Copying）、置换（Swapping）和搬移（Moving）

让我们继续效率方面的话题。本节介绍一个 policy traits template，用在 (1) 拷贝(copying)或 (2) 置换（swapping）(3) 搬移（moving）元素时，根据其类型选择最佳操作机制。

通常我们会认为，「拷贝操作」以 *copy* 构造函数和 *copy-assignment* 运算符处理即可。这对单一 元素来说无疑是正确的。但也可能存在这样的事实：拷贝某类型的大量元素时，存在一种比「不 断重复调用该类型之 *copy* 构造函数或 *copy-assignment* 运算符」更显著高效的作法。

同样道理，对置换（swapping）或搬移（moving）而言，某些类型的元素可以采用比以下传统 操作更高效的作法：

```
T tmp(a);
a = b;
b = tmp;
```

容器类型（container types）通常便是属于此类。

有时候拷贝是不被允许的，置换或搬移则没有问题。第 20 章谈到 Utilities（工具程序）时，我们开发了一个所谓的 smart pointer（灵巧指针），就具有这个属性。

因此，将此领域的决策集中于一个便利的 traits template 会很有用。针对泛型定义，我们区分 class types 和 nonclass types 两大类，面对后者我们无需挂心「用户自定的 *copy* 建构式和 *copy assignment* 运算符」。这次我们使用继承机制，在两个 traits 实作品之间进行选择。

```
// traits/csmtraits.hpp
template <typename T>
class CSMtraits : public BitOrClassCSM<T, IsClassT<T>::No > {
};
```

这么一来，实作任务就完全委托给 `BitOrClassCSM<>` 的特化版本"CSM" 代表 "copy"、"swap"、"move"）。第二个 template parameters 用以指示「位逐一拷贝」（bitwise copying）能否安全 实作出不同种类的操作。这个泛型定义保守地假设 class types 不能被安全地「位逐一拷贝」，但如果已知某特定的 class type 是个 POD（plain old data）type，那就很容易将 `CSMtraits` class 特化以获得更佳效率：

```
template<>
class CSMtraits<MyPODType>
 : public BitOrClassCSM<MyPODType, true> {
};
```

预设情况下 `BitOrClassCSM` template 包含两个偏特化版本。primary template 以及「不进行位 逐一拷贝」的偏特化安全版本展示如下：

```
// traits/csm1.hpp
#include <new>
#include <cassert>
#include <stddef.h>
#include "rparam.hpp"

// primary template
template<typename T, bool Bitwise>
class BitOrClassCSM;

// 偏特化版本，用以对对象进行安全拷贝（safe copying）
template<typename T>
class BitOrClassCSM<T, false> {
  public:
```

```
static void copy (typename RParam<T>::ResultT src, T* dst) {
    // 将一份数据拷贝（copy）到另一份身上
    *dst = src;
}

static void copy_n (T const* src, T* dst, size_t n) {
    // 将 n份数据拷贝（copy）到另外的 n份数据身上
    for (size_t k = 0; k<n; ++k) {
        dst[k] = src[k];
    }
}

static void copy_init (typename RParam<T>::ResultT src,
                       void* dst) {
    // 将一份数据拷贝（copy）到未初始化的储存空间上
    ::new(dst) T(src);
}

static void copy_init_n (T const* src, void* dst, size_t n) {
    // 将 n份数据拷贝（copy）到未初始化的储存空间上
    for (size_t k = 0; k<n; ++k) {
        ::new((void*)((char*)dst+k)) T(src[k]);
    }
}

static void swap (T* a, T* b) {
    // 置换（swap, 对调）两份资料
    T tmp(*a);
    *a = *b;
    *b = tmp;
}

static void swap_n (T* a, T* b, size_t n) {
    // 置换（swap, 对调）n份资料
    for (size_t k = 0; k<n; ++k) { T
        tmp(a[k]);
        a[k] = b[k];
        b[k] = tmp;
    }
}

static void move (T* src, T* dst) {
```

```
        // 搬移（move）一份数据到另一份身上
        assert(src != dst);
        *dst = *src;
        src->~T();
    }

    static void move_n (T* src, T* dst, size_t n) {
        // 搬移（move）n份数据到另 n份身上
        assert(src != dst);
        for (size_t k = 0; k<n; ++k)
            { dst[k] = src[k];
            src[k].~T();
        }
    }

    static void move_init (T* src, void* dst) {
        // 搬移（move）一份数据到未初始化储存空间上
        assert(src != dst);
        ::new(dst) T(*src);
        src->~T();
    }

    static void move_init_n (T const* src, void* dst, size_t n) {
        // 搬移（move）n份数据到未初始化储存空间上
        assert(src != dst);
        for (size_t k = 0; k<n; ++k) {
            ::new((void*)((char*)dst+k)) T(src[k]);
            src[k].~T();
        }
    }
};
```

术语 *move*（搬移）在这儿的意思是「一个 value 从某地被移转（*transferred*）到另一地」；原 value 不复存在（或更准确地说，原位置可能被销毁）。至于 *copy*（拷贝）则是保证操作后的来源位 置和目标位置同时存在有效且相同的 value。请不要和标准 C 库中的 `memcpy()` 和 `memmove()` 搞混淆了 — 在那个情况下 *move* 意味来源区和目标区有可能重迭，*copy* 则否。我们的 CSM traits总是假设来源区和目标区不重迭。在一个工业强度库中，或许还应该加入 *shift*（平移）操作，用以表达「在一块连续记忆区内移动对象」的 policy（也就是 `memmove()` 所支持的那种操 作）。但是为求简化，我们节略了这一点。

我们的 policy traits template 成员函数都是 static。任何情况下几乎总是如此，因为成员函数意图施行于「隶属参数类型」的物件上，而非「隶属 traits class type」的物件上。

另一个偏特化版本令此 traits 作用于「得以进行位逐一拷贝」的类型身上：

```
// traits/csm2.hpp
#include <cstring>
#include <cassert>
#include <stddef.h>
#include "csm1.hpp"

// 偏特化版本，针对 fast bitwise copying 撰写
template <typename T>
class BitOrClassCSM<T,true> : public BitOrClassCSM<T,false> {
  public:
    static void copy_n (T const* src, T* dst, size_t n) {
        // 将 n份数据拷贝（copy）到另外的 n份数据身上
        std::memcpy((void*)dst, (void*)src, n);
    }

    static void copy_init_n (T const* src, void* dst, size_t n) {
        // 将 n份数据拷贝（copy）到未初始化储存空间上
        std::memcpy(dst, (void*)src, n);
    }

    static void move_n (T* src, T* dst, size_t n) {
        // 搬移（move）n份数据到另 n份身上
        assert(src != dst);
        std::memcpy((void*)dst, (void*)src, n);
    }

    static void move_init_n (T const* src, void* dst, size_t n) {
        // 搬移（move）n份数据到未初始化储存空间上
        assert(src != dst);
        std::memcpy(dst, (void*)src, n);
    }
};
```

以上多用了一层继承，用以简化此类 traits 的实作。这些当然不是惟一可能的作法，事实上某些特定平台（例如）可能希望引入 inline 汇编语言码（以便运用硬件优势）。

15.4 后记

Nathan Myers 是第一个将 traits parameters 观念正式化的人。他最初将此想法提呈给C++标准 委员会时，是将它当作「在标准库组件（如输入数据流和输出数据流; I/O streams）中如何 处理字符类型」的一种定义工具。当时他称之为 baggage templates，并注明它们相当于

traits。 然而有些C++ 委员会成员不喜欢 baggage 这个术语反而提倡以 traits 为名从那时起术语 traits 便获得了广泛的使用。

客端程序代码通常根本无需处理 traits，因为预设的 traits classes 可以满足绝大多数常见需求，而 且由于它们是预设的 template arguments，它们根本无需出现在客端程序代码之中。这将有利于为 default traits templates 提供较长的描述性名称。当客端程序代码透过「提供一个订制的 traits 自变量」 来修改 template 行为时，将 resulting specializations（最后所得的特化体）以 typedef定出一个
「和订制行为相称」的名称，是个好习惯。这种情况下我们可以赋予 traits class 一个较长的、而且不会把源码搞得很难看的描述性名称。

我们的讨论似乎显示 traits templates 非得是 class templates 不可。事实并非如此。如果只需提供 单一 policy trait，其实可使用 function template 传入。例如：

```
template <typename T, void (*Policy)(T const&, T const&)>
class X;
```

不过，traits 的最初目标就是为了减轻 secondary template arguments（次要模板自变量）的包袱。如 果一个 template parameter 内只封装一个 trait，就达不到效果了。这足以说明 Myers 当初对术语 baggage 的偏爱是有道理的。第 22 章提供排序准则（ordering criterion）时我们将重启这个问题。

标准库定义了一个 class template std::char_traits，它被用作 policy traits parameter。此外为了让算法很容易适应于它所获得的 STL iterators，标准库提供了一个非常简明的 std::iterator_traits property traits template（并被用于标准程式库接口）。template std::numeric_limits也可被拿来当作一个 property traits template，但在标准库中它显然 没有得到适当地运用。class template std::unary_function 和 std::binary_function归属于 相同分类，都是非常简单的 type functions：仅仅只是将自变量重新定义（typedef）为成员名称， 从而使这些名称对 functors（或称 function objects，见 22 章）有意义。最后要说的是，标准容器 的内存配置问题是以一个 policy traits class 来处理的，而 template std::allocator正是标准 库为此目的而提供的一个标准组件。

译注：如果不曾研究过 STL 源码，几乎可以断定无法理解上述文字，因为上述文字所描述的组件都是底层的、不与客户端接触的。任何 C++ 编译器均附带 STL 源码，市面上也有 STL 源码 剖析相关书籍，建议一观。

很多程序员和不少书籍作者都开发了 policy classes 技术，Andrei Alexandrescu 更使得术语 policy classes 脍炙人口。相比于本书简短的一节来说，Andrei 的著作《*Modern C++ Design*》更加细致地探讨了这个主题（见 [AlexandrescuDesign]）。

C 参考书目和资源 Bibliography

C.2 书籍和 Web 网站

[AlexandrescuDesign]

Andrei Alexandrescu

ModernC++Design 《C++ 设计新思维》,侯捷/于春景合译, 碁峰 2003

Generic Programming and Design Patterns Applied

Addison-Wesley, Reading, MA, 2001

[AusternSTL]

Matthew H. Austern

Generic Programming and the STL 《泛型程序设计与 STL》,侯捷/黄俊尧合译, 碁峰 2000

Using and Extending theC++Standard Template Library

Addison-Wesley, Reading, MA, 1999

[BCCL]

Jeremy Siek

The Boost Concept Check Library

`http://www.boost.org/libs/concept_check/concept_check.htm`

[Blitz++]

Todd Veldhuizen

Blitz++: Object-Oriented Scientific Computing

`http://www.oonumerics.org/blitz`

[Boost]

The Boost Repository for Free, Peer-ReviewedC++Libraries

`http://www.boost.org`

[BoostCompose]

Boost Compose Library

`http://www.boost.org/libs/compose`

[BoostSmartPtr]

Smart Pointer Library

`http://www.boost.org/libs/smart_ptr`

[BoostTypeTraits]

Type Traits Library

`http://www.boost.org/libs/type_traits`

[CargillExceptionSafety]

Tom Cargill

Exception Handling: A False Sense of Security

Available at: http://www.awprofessional.com/meyerscddemo/demo/magazine/index.htm

C++ Report, November-December 1994

[CoplienCRTP]

James O. Coplien

Curiously Recurring Template Patterns

C++ Report, February 1995

[CoreIssue115]

Core Issue 115 of theC++Standard

`http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_toc.html`

[CzarneckiEisenecker GenProg]

Krzysztof Czarnecki, Ulrich W. Eisenecker

Generative Programming

Methods, Tools, and Applications

Addison-Wesley, Reading, MA, 2000

[DesignPatternsGoV]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Design Patterns 《面向对象设计模式》,叶秉哲译, 培生 2001

Elements of Reusable Object-Oriented Software

Addison-Wesley, Reading, MA, 1995

[EDG]

Edison Design Group

Compiler Front Ends for the OEM Market

`http://www.edg.com`

[EllisStroustrupARM]

Margaret A. Ellis, Bjarne Stroustrup

The AnnotatedC++Reference Manual (ARM)

Addison-Wesley, Reading, MA, 1990

[JosuttisAutoPtr]

Nicolai M. Josuttis
auto_ptr and auto_ptr_ref
`http://www.josuttis.com/libbook/auto_ptr.html`

[JosuttisOOP]
Nicolai M. Josuttis
Object-Oriented Programming in C++
John Wiley and Sons Ltd, 2002

[JosuttisStdLib]
Nicolai M. Josuttis
TheC++Standard Library 《C++ 标准库》,侯捷/孟岩合译, 碁峰 2002
A Tutorial and Reference
Addison-Wesley, Reading, MA, 1999

[KoenigMooAcc]
Andrew Koenig, Barbara E. Moo
Accelerated C++
Practical Programming by Example
Addison-Wesley, Reading, MA, 2000

[LambdaLib]
Jaakko J··arvi, Gary Powell
LL, The Lambda Library
`http://www.boost.org/libs/lambda/doc`

[LippmanObjMod]
Stanley B. Lippman
Inside theC++Object Model 《深度探索C++ 对象模型》,侯捷译, 碁峰 1998
Addison-Wesley, Reading, MA, 1996

[MeyersCounting]
Scott Meyers
Counting Objects In C++
C/C++ Users Journal, April 1998

[MeyersEffective]
Scott Meyers
Effective C++ 《*Effective*C++ 中文版》,侯捷译, 培生 2000
50 Specific Ways to Improve Your Programs and Design (2nd Edition)
Addison-Wesley, Reading, MA, 1998

[MeyersMoreEffective]

Scott Meyers

More Effective C++ 《*More EffectiveC++* 中文版》,侯捷译, 培生 2000

35 New Ways to Improve Your Programs and Designs

Addison-Wesley, Reading, MA, 1996

[MTL]

Andrew Lumsdaine, Jeremy Siek

MTL, The Matrix Template Library

`http://www.osl.iu.edu/research/mtl`

[MusserWangDynaVeri]

D. R. Musser, C. Wang

Dynamic Verification ofC++Generic Algorithms

IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997

[MyersTraits]

Nathan C. Myers

Traits: A New and Useful Template Technique

`http://www.cantrip.org/traits.html`

[NewMat]

Robert Davies

NewMat10, A Matrix Library in C++

`http://www.robertnz.com/nm_intro.htm`

[NewShorterOED]

Leslie Brown, et al.

The New Shorter Oxford English Dictionary (fourth edition)

Oxford University Press, Oxford, 1993

[POOMA]

POOMA: A High-PerformanceC++Toolkit for Parallel Scientific Computation

`http://www.pooma.com`

[Standard98]

ISO

Information Technology-Programming Languages-C++

Document Number ISO/IEC 14882-1998

ISO/IEC, 1998

[Standard02]

ISO
Information Technology-Programming Languages-C++
(as amended by the first technical corrigendum)
Document Number ISO/IEC 14882-2002
ISO/IEC, expected late 2002

[StroustrupC++PL]
Bjarne Stroustrup
TheC++Programming Language, Special ed. 《C++ 程序语言经典本》,叶秉哲译, 儒林
Addison-Wesley, Reading, MA, 2000

[StroustrupDnE]
Bjarne Stroustrup
The Design and Evolution of C++
Addison-Wesley, Reading, MA, 1994

[StroustrupGlossary]
Bjarne Stroustrup
Bjarne Stroustrup'sC++Glossary
`http://www.research.att.com/~bs/glossary.html`

[SutterExceptional]
Herb Sutter
Exceptional C++ 《*ExceptionalC++* 中文版》,侯捷译, 培生 2000
47 Engineering Puzzles, Programming Problems, and Solutions
Addison-Wesley, Reading, MA, 2000

[SutterMoreExceptional]
Herb Sutter
More Exceptional C++
40 New Engineering Puzzles, Programming Problems, and Solutions
Addison-Wesley, Reading, MA, 2001

[UnruhPrimeOrig]
Erwin Unruh
Original Metaprogram for Prime Number Computation
`http://www.erwin-unruh.de/primorig.html`

[VandevoordeSolutions]
David Vandevoorde
C++ Solutions
Addison-Wesley, Reading, MA, 1998

[VeldhuizenMeta95]

Todd Veldhuizen

UsingC++Template Metaprograms

C++ Report, May 1995

[VeldhuizenPapers]

Todd Veldhuizen

Todd Veldhuizen's Papers and Articles about Generic Programming and Templates

`http://osl.iu.edu/~tveldhui/papers`

D

词汇/术语表

Glossary

这份词汇/术语表是本书所论主题之最重要术语的汇编。[StroustrupGlossary] 也提供了一份供 C++ 程 式员使用的全面术语表。

abstract class（抽象类别）

一种无法据以创建*create*; 产生/建立/创造)具象对象实体)的 class。这种 classes 适合将不同 classes 的共通属性集结于单一类型之中，亦适合定义多型接口（polymorphic interface）。由于这种 classes 被当做 base classes 使用，所以有时候会出现缩写字 ABC，代表 abstract base class（抽象基础类别）。

ADL

argument-dependent lookup（相依于自变量的查询）缩写。所谓 ADL 是这样一个过程：在 namespaces 和

classes 内查询某个函数或运算符的名称这些名称出现于调用语句中而接受查询的 namespaces 和 classes 则是以某种方式与函数调用自变量相关联。由于历史因素，这个术语有时被称为 extended Koenig lookup 或干脆简称为 Koenig lookup（后者也用于「只针对运算符的 ADL」）。

angle bracket hack

一种非标准特性，允许编译器接受两个连续的 > 字符作为两个右角括号（它们之间原本应该插入至少一个空格）。例如算式 `vector<list<int>>` 原本不合法，经由 angle bracket hack 的处理，它被 视同合法的 `vector<list<int> >`。

angle brackets（角括号）
字符 < 和 > 被用于 template 自变量列或参数列的边界符号时，即称为角括号。（译注：而不是指「大 于」或「小于」符号）

ANSI
美国国家标准学会（American National Standard Institute）的缩写。这是一个非官方、非盈利组织，努力生产各类标准规格。其中一个名为 J16 的小组委员会是C++ 标准化的背后驱动力。ANSI 和国际标 准组织（international standards organization，ISO）紧密合作。

argument（自变量）
一个用以替换「程序性物体（programmatic entity）之参数」的（广义）值。例如在调用语句 `abs(-3)` 中，`-3`便是自变量。某些编程社群将自变量称为实参（actual parameters），而将参数（parameters）称为 形参（formal parameters）。

argument-dependent lookup（相依于自变量的查询） 见 ADL。

class（类别）
对某种类型之对象的描述。class 为对象（*objects*）定义了一套特征，包括数据（或称属性 *property*、成员变数 *data members*）和操作（*operations*；或称方法 *methods* 或成员函数 *member functions*）。在C++ 中 classes 是带有成员的一种结构其成员可以是函数，并服从某种取用限制它们以关键词 `class` 或 `struct`进行声明。

class template（类别模板）
一种构件，用来表示「一整群类别族系」（a family of classes）。它规定某种样式（pattern），只要 以特定物体（entities）替换其 template 参数，便可从中生成真正的 class。Class template 有时被称为 参数化类别（"parameterized" classes），尽管后者具有更一般化的含义。

class type（类别类型）
任何以关键词 `class`、`struct`或 `union`进行声明的C++ 类型。

collection class（群集类别）
用于管理「一组对象」的 class。在C++ 中群集类别也称为容器（containers）。

constant-expression（常数-算式） 计算值（value）由编译器在编译期计算而得。有时被称为真常数（true constant），以避免和常数算 式（constant expression；中间无连字号）混淆。后者包括「不能由编译器在编译期计算结果」的常数 算式。

`const` member function（const 成员函数） 只可由常数对象和暂时对象调用的成员函数，它通常并不修改 `*this`对象的成员。

container（容器）
参见 collection class（群集类别）

conversion operator（转换运算符；转型运算符）
一种特殊的成员函数，它定义对象如何被隐式（或显式）转换为另一种类型。其声明型式为 `operator type()`。

CRTP
curiously recurring template pattern（奇特递归模板范式）的缩写。指的是这样一种程序写法：class X 衍生自一个「以 X为 template 自变量」的 base class。

curiously recurring template pattern（奇特递归模板范式）
参见 CRTP。

decay（退化）
从一个 array 或 function 隐式转换至一个 pointer。例如字符串字面值 "Hello" 的类型为 `char const[6]`，但在很多C++ 情境中它被隐式转换为一个 `char const*` 指针（指向该字符串的首字符）。

declaration（声明）
一种C++ 构件，用以将某个名称引入（或重新引入）C++ 作用域（scope）。参见 definition。

deduction（推导）
从 template 被使用的情境中「暗自（径自）决定 template argument」的过程。完整的术语是 template argument deduction（模板自变量推导）。

definition（定义）
一种 declaration，使已声明物体（declared entity）的细节曝光。对变量而言，会迫使保留（获得）其 储存空间。对 class 和 function 而言，是指其 declarations 带有「以大括号围起来的本体（body）」。 对外部变数（external variable）而言，意味一个不带 extern 关键词的 declaration，或是一个具有初值 设定器（initializer）的 declaration。

dependent base class（受控基础类别）
「与某个 template parameter 相依」的 base class。程序中存取此种 class 的成员时要特别小心。参见
two-phase lookup（两段式查询）。

dependent name（受控名称）
「含义取决于某 template parameter 的识别名称。例如，当 A或 T是 template parameter 时，`A<T>::x`

就是个 dependent name。如果调用语句中某个自变量的类型取决于某个 template parameter，那么该调用 式中的函数名称也是 dependent。例如在调用语句 `f((T*)0)`中，如果 `T`是个 template parameter，`f`就
是 dependent。然而 template parameter 名称并不被认为是 dependent。参见 two-phase lookup（两段式 查询）。

digraph（双字符语汇单元）
两个连续字符的组合体，等价于 C++ 程序代码中另一个单字元。digraphs 的用途在于允许「缺少某些 字符的键盘」得以输入C++ 源码。当一个左角括号（`<`）后跟一个 *scope resolution* 运算符（`::`）而 没有插入必要空格时，就会意外形成双字符语汇单元 `<:` — 尽管相对来说这很少被用到。

dot-C file（dot-C 文件）
变量和 noninline 函数定义的座落文件。程序的大多数「可执行的」（而非只是「声明的」; declarative） 程序代码正常来说都应该被放进 dot-C 文件。它们之所以被命名为 dot-C 文件，因为通常
以 .cpp, .C, .c, .cc
或 .cxx 为扩展名。参见 header file（头文件）和 translation unit（编译单元）。

EBCO
empty base class optimization（空基础类别优化）的缩写。大多数现代编译器都实作有此优化机制。有了它，一个 "empty" base class subobject（「空」基础类别的子对象）就不会占用任何储存空间。

empty base class optimization（空基础类别优化）
参见 EBCO。

explicit instantiation directive（显式具现指令）
一个C++ 构件，惟一用途是建立一个 POI（point of Instantiation; 具现点）。

explicit specialization（显式特化）
一种 C++构件，为某个 template 声明或定义一个替代性（alternative）定义式。最初的那个最泛化的 版本称为 primary template（主模板）。如果此一替代性定义仍需倚赖一或多个 template parameters， 它就被称为 partial specialization（偏特化），否则就称为 full specialization（全特化）。

expression template（算式模板）
一种 class template，用以表现某算式的一部分。该 template 本身表示某种特定操作（译注：例如加 减乘除等等）， template parameter 则表示操作施行对象（操作数）的种类。

friend name injection（friend 名称植入）
当函数被声明为 friend 时，「使该函数名称可见」的一个处理过程。

full specialization（全特化）
参见 explicit specialization（显式特化）。

function object（函数物件）

参见 functor（仿函数）。

function template（函数模板）
一种构件(construct)，用以表示「一整群函数族系」(a family of functions)，它规定某种样式(pattern)，只要以特定物体（entities）替换其 template 参数，便可从中生成真正的函数。注意，function template 是一个 template 而非一个 function。有时它也被称为参数化函数（"parameterized" functions），尽管后者具有更一般化的含义。

functor（仿函数）
可以「函数调用语法」进行调用的对象（也称为 function object; 函数物件）。在C++ 中指的是 pointers to functions 或 references to functions，或是任何拥有 `operator()`成员的 class。

header file（头文件）
意欲「透过`#include`指令成为编译单元一部分」的文件。这样的文件通常包含变量和函数的声明（它们可被不只一个编译单元引用），以及 types（类型）、inline 函数、templates（模板）、constants（常数）和 macros（宏）等定义。它们通常以 .hpp, .h, .H, .hh 或 .hxx 做为扩展名。又称为含入文件（include files）。参见 dot-C file 和 translation unit（编译单元）。

include file（含入文件）
参见 header file（头文件）。

indirect call（间接调用） 函数调用形式之一。未到调用动作真正发生（执行期），不知道该调用哪一个函数。

initializer（初值设定器）
一种构件（construct），用以规定如何初始化一个具名对象（named object）。例如以下的式子：

```
std::complex<float> z1 = 1.0, z2(0.0, 1.0);
```

初值设定器是 `=1.0`和`(0.0, 1.0)`。

initializer list（初值列，初始值列表）
一个封闭于小括号内部、以逗号分隔的算式列表，用以初始化对象或数组。在构造函数中可用以定义
members（成员）和 base class（基础类别）的初值。

injected class name（植入类别名称）
class 名称在它自己的作用域（scope）中是可见的。对 class template 来说，在 template 作用域内，如果其名称之后没有跟着一个 template argument list 的话，该 template 名称将被视为一个 class 名字。

instance（实体）
在C++ 编程领域中，instance 有两层含义：面向对象方面的意思是「某个 class 的实体」，也就是 object（对象），是某 class 的实现结果。例如C++ 中的 `std::cout`是 class `std::ostream`的一个实体。另一层意思（本书所采用）是 "a template instance"：透过「以特定值替换所有 template parameters」而得到的 classes、functions 或 member functions。在这个意义下 instance 也被称为 specialization（特

化 体），后者常被误以为是 explicit specialization（显式特化体）。

instantiation（实例化）
以实际值替换 template parameters 从而自一个 template 创建出常规的non-template classes、functions 或 member functions 的过程 另一层意思本书不采用是：创建 class 的一份实体/对象 参见 instance）

ISO
国际标准组织（International Organization for Standardization）的全球通用缩写。名为 WG21 的 ISO 工作小组正是 C++标准化和持续发展的背后驱动力。

iterator（迭代器）
一种知道如何巡访（遍历; traverse）元素序列的对象。通常那些元素属于一个群集（参见 collection class）。

linkable entity（可链接物）
包括 non-inline 函数或成员函数、global 变量或 static 成员变量，以及从 template 生成的任何此类东 西。

lvalue（左值）
在最初的 C 语言中，如果算式（expression）可出现于 *assignment*（赋值）运算符左侧的话，它就被 称为一个左值。反过来说，只能出现于 *assignment* 运算符右侧者，就称为 rvalue（右值）。这个定 义不再适用于新一代 C/C++。如今 lvalue 可被认为是一个 locator value，意指「透过名称或地址（可 以是 pointers、references 或 array access）」而非透过纯粹计算来标示出某对象」的算式。lvalue 不必 是可修改的（例如常数对象就是一种不可修改的 lvalue）。所有不是 lvalue 的算式都是 rvalue。透过 `T()`显式创建出来的暂时对象，或作为函数调用结果的暂时对象，都是 rvalue。

member class template（成员类别模板）
一种构件，用来表示「一整个族系的成员类别」（a family of member classes）。这是一种「声明于另一个 class 或 class template 内」的 class template，它有自己的一套 template parameters，此点与「class template 内的 member class」不相同。

member function template（成员函数模板）
一种构件，用来表示「一整个族系的成员函数」（a family of member functions），它有自己的一套 template parameters，此点与「class template 内的 member function」不同，这种东西非常类似 function template，但其所有 template parameters 被替代后产出的是个 member function（而不是个 ordinary function）。 member function template 不可以是 virtual。

member template（成员模板）
意指 member class template 或 member function template。

nondependent name（非受控名称）
一个不取决于 template parameters 的名称。参见 dependent name 和 two-phase lookup。

ODR

one-definition rule（单一定义规则）的缩写。这个规则对 C++ 程序内的定义式（definitions）施加了一些约束。详见 7.4 节, p.90 和附录 A。

one-definition rule（单一定义规则）

参见 ODR。

overload resolution（重载解析） 当函数候选者多于一个时（它们通常拥有相同名称），「选择/决定 调用哪个函数」的过程。

parameter（参数）

一种占位物体(placeholder entity)，会于程序的某一点被实际值一个自变量替换。macro 参数和 template 参数的替换发生于编译期，function call 参数的替换发生于执行期。某些编程社群将参数称为形参（formal parameters），将自变量（arguments）称为实参（actual parameters）。参见 argument。

parameterized class（参数化类别）

意指 class template，或「嵌套（nested）于某 class template 内」的 class。它们都是参数化的，因为 除非 template argument 获得指定，否则它们不会对应至某个独一无二的 class。

parameterized function（参数化函数）

意指 function template 或 member function template 或 class template 的成员函数。它们都是参数化的，因为除非 template argument 获得指定，否则它们不会对应至某个独一无二的函数（或成员函数）。

partial specialization（偏特化）

一种构件（construct），用以为 template 定义一个替代性定义式，而其中仍具有某些可替代参数。最 初那个最泛化的版本称为 primary template（主模板）。这一份替代性定义仍然相依于某些 template parameters。这种构件目前仅对 class templates 有效。参见 explicit specialization（显式特化）。

POD（简朴旧式资料）

plain old data (type) 的缩写。POD 是这样一种类型：其定义不牵扯某些C++ 特性诸如 virtual（虚拟）成员函数、存取关键词 public, private 等等。每个 C `struct`都是个 POD。

POI（具现点）

point of instantiation（具现点）的缩写。POI 是源码中的某个位置，该处将以 template arguments 替换 template parameters，从而将 template（或其成员）概念性地展开。现实中并非每个 POI 都需要发生 展开行为。参见 explicit instantiation directive（显式具现指令）。

point of instantiation（具现点）

参见 POI。

policy class（策略类别）
一个 class 或 class template，其成员用以描述泛型组件（generic component）之可组态行为（configurable behavior）。此物通常透过 template arguments 传递。例如一个排序用的 template 可能需要一个 "ordering policy"。Policy classes 亦被称为 policy templates 或简称为 policies。参见 traits template。

polymorphism（多型）
「将某一（根据名称确认之）操作施行于不同种类的对象身上」的能力。C++ 传统面向对象概念中
的 polymorphism（亦称为 run-time polymorphism 或 dynamic polymorphism）是透过「在 derived class
中覆写虚拟函数」实现出来的。C++ templates 还支持所谓的 static polymorphism（静态多型）。

precompiled header（预编译表头）
一种经过处理的源码形式，可以迅速被编译器加载（loaded）。预编译表头底部的（underlying）程序 码必须是编译单元（translation unit）的第一部分（换句话说不能从编译单元的某个中间点开始）。通 常一个预编译表头对应多个头文件（header files）。使用预编译表头，可大幅改善建置（build）C++ 大型程序的时间。

primary template（主模板/原始模板）
一个「非偏特化版本」（not a partial specialization）的 template。

qualified name（资格限定名称/资格修饰词）包
含 *scope*（作用域）修饰符号（::）的名称。

reference counting（引用计数）
一种资源管理策略，记录（计数）当下有多少个物体（entities）正在引用（*referring to*）某特定资源。一旦数目降为 0，资源就可被释放掉。

rvalue（右值）
参见 lvalue（左值）。

source file（源码文件） 意指头文件
（header file）或 dot-C 文件。

specialization（特化体/特化版本）
以实值替换 template parameters 的结果。特化体可经由实例化（instantiation）或显式特化（explicit specialization）创造出来。这个术语有时被错误地和 explicit specialization（显式特化版本）混为一谈。参见 instance（实体）。

template（模板）
一种构件（construct），用以表示「一整个家族的 class 或 function」。它规定某种样式（pattern），只要以特定物体（entities）替换其 template 参数，便可从中生成真正的 classes 或 functions。本书中 这个术语不包括仅仅由于是某个 class template 的成员而被参数化」的 functions、classes 和 static data members。参见 class template（类别模板）、parameterized class（参数化类别）、function

template（函 式模板）和 parameterized function（参数化函数）。

template argument（模板自变量）
「用以替换 template parameters」的一个值(value)。此「值」通常是一种类型(type)，尽管 constant values（常数）和 templates（模板）也是合法的 template arguments。

template argument deduction（模板自变量推导）
参见 deduction（推导）。

template-id（模板识别符号）
template 名称连同「紧随其后之角括号内的 template arguments」的组合（例如 `std::list<int>`）。

template parameter（模板参数）
templates 中的泛型占位符号(generic placeholder)，最常见的 template parameters 是 type parameters，代表类型；nontype parameters 则用以表示某特定类型的常数值；template template parameters（双重 模板参数）表现的是 class templates（类别模板）。

traits template（「特征萃取」模板）
一种 templates 其成员用以描述 template arguments 的特征(characteristics; traits)，通常 traits templates 的用途是避免 template parameters 过量。参见 policy class。

translation unit（编译单元）
一个 dot-C 文件，带有以`#include`指令含入的所有「头文件」和「标准库头文件」，并扣除被 条件编译指令（例如 `#if`）排除掉的程序段落。简单地说，它可以被视为「对一个 dot-C 文件进行预 处理」后的结果。参见 dot-C file 和 header file。

true constant（真常数）
参见 constant-expression（常数-算式）。

tuple（三部合成构件）
C `struct`概念的一种泛化版本，其成员可经由编号来存取。

two-phase lookup（两段式查询）
template 内的名称查询机制。所谓两段式是指（1）当 templates 定义式被编译器第一次遇上时（2）当 template 被实例化时。Nondependent names（非受控名称）只在第一阶段被查询，但此阶段不考虑 nondependent base classes（非受控基础类别）。至于 dependent names（受控名称），拥有 *scope* 修饰 符号（`::`）者只在第二阶段被查询，不带 *scope* 修饰符号（`::`）者可能在两阶段都被查询，但第二 阶段只执行 argument-dependent lookup（相依于自变量的查询）。

user-defined conversion（用户自定转换） 由程序员定义的类型转换动作，可以是个「以单自变量调用」的构造函数，也可以是个 *conversion*（转 型）运算符。除非带有关键词 `explicit`，否则「单自变量构造函数」可能会发生隐式类型转换。

whitespace

在 C++中这是对源程序代码之各种记号包括标识符 identifiers、字面值 literals、语汇单元符号 symbols 等等）进行分界的区隔物。除了传统的空格（blank space）、换行（new line）和水平定位（horizontal tabulation）等字符外，还包括注释（comments）。其它的空格字符（例如 page feed control character）有时也是合法的。