第12章 深入模板基础

在本章中，我们将深入探讨本书第一部分中介绍的一些基础知识：模板的声明，模板参数(template paramenters)的限制(restrictions)，模板实参(template arguments)的限制(constraints)等等。

12.1 参数化声明

C++目前支持4种基础模板：类模板、函数模板、变量模板以及别名模板。每一种模板都既可以出现在命名空间作用域，也可以出现在类作用域。在类作用域中，它们作为嵌套的类模板、成员函数模板、静态数据成员模板以及成员别名模板。这些模板的声明与普通类、函数、变量以及类型别名（或者是它们的类成员副本）非常相似，只不过需要一个形如`template<parameters here>`的子句来做前置指引。

请注意，C++17引入了另一种带有参数化子句的结构：推导指引(deduction guides)(参考P42节2.9以及P314节15.12.1)。本书中它们不被称为模板(因为它们没有被实例化)，但是这一语法的选择会让人联想到函数模板。

在下一节中，我们将重返实际的模板参数声明。首先，一些示例用以说明四种类型的模板。它们可以像这样在命名空间作用域（全局或是某个命名空间内）中出现：

details/definitions1.hpp

```
template<typename T> // a namespace scope class template
class Data {
  public:
    static constexpr bool copyable = true;
    …
};

template<typename T> // a namespace scope function template
void log (T x) {
  …
}

template<typename T> // a namespace scope variable template (since C++14)
T zero = 0;

template<typename T> // a namespace scope variable template (since C++14)
bool dataCopyable = Data<T>::copyable;

template<typename T> // a namespace scope alias template
using DataList = Data<T*>;
```

注意到示例中，静态数据成员`Data<T>::copyable`并不是一个变量模板，尽管它是通过类模板`Data`参数所间接参数化的。然而，变量模板可以出现在类作用域中(下一个例子会展示)，彼时它将作为一个静态数据成员模板。

下面展示了定义在所属类中的4种模板，它们都是类的成员：

details/definitions2.hpp

```
class Collection {
  public:
          template<typename T>          // an in-class member class template definition
        class Node {
          ...
        };

        template<typename T>         // an in-class (and therefore implicitly inline)
        T* alloc() {                          // member function template definition
          ...
        }

        template<typename T>         // a member variable template (since c++14)
        static T zero = 0;

        template<typename T>         // a member alias template
        using NodePtr = Node<T>*;
};
```

请注意，在C++17中，变量（包括静态数据成员）以及变量模板都可以是内联的，内联意味着它们的定义可以跨越多个编译单元重复。对于总能定义在多个编译单元中的变量模板来说，这是多余的。但类内定义的静态数据成员不会像成员函数一样内联，因此就要指定inline关键字。

最后，下面的代码演示了如何在类外定义别名模板以外的成员模板：

details/definitions3.hpp

```
template<typename T>        // a namespace scope class template
class List {
  public:
    List() = default;        // because a template constructor is defined

        template<typename U>        // another member class template,
        class Handle;                       // without its defination

        template<typename U>        // a member function template
        List (List<U> const&);        // (constructor)

        template<typename U>        // a member variable template (since C++14)
        static U zero;
};

template<typename T>        // out-of-class member class template definition
template<typename U>
class List<T>::Handle {
  ...
};

template<typename T>        // out-of-class member function template definition
template<typename T2>
List<T>::List(List<T2> const& b)
{
  ...
}

template<typename T>        // out-of-class static data member template definition
template<typename U>
U List<T>::zero = 0;
```

定义在类外的成员模板需要多个`template<... >`参数化子句：每个外围作用域的类模板一个，成员模板本身也需要一个。子句从类模板最外层开始逐行展示。

同时也注意到构造器模板（一种特殊的成员函数模板）会禁用掉隐式声明的默认构造器（因为只有在没有其他构造器被声明时，默认构造器才会被声明）。增加一个默认的声明：

```
List() = default;
```

这确保了`List<T>`的实例可以通过隐式声明的默认构造器构造出来。

联合体模板

联合体模板(union templates)也是可行的（它们被视为一种类模板）：

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

默认调用参数

函数模板可以有默认参数，就如同普通的函数一样：

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

template<typename T>
void fill(Array<T>&, T const& = T{});        // T{} is zero for built-in types
```

第二个声明展示了默认调用参数可以依赖于模板参数。它也可以被定义成如下形式（在C++11之前唯一可行的方式，可以参考P68节5.2）：

```
template<typename T>
void fill(Array<T>&, T const& = T());        // T() is zero for built-in types
```

当`fill()`函数被调用时，如果传入了第二个参数，那么默认参数不会实例化。这保证了如果默认调用参数对特定`T`无法实例化的情景下不会发生错误。例如：

```
class Value {
  public:
    explicit Value(int);        // no default constructor
};

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

  fill(array, zero);        // OK: default constructor not used
  fill(array);                        // ERROR: undefined default constructor for Value is used
}
```

类模板的非模板成员

除了类内定义的4种基础模板以外，你还可以定义普通的类成员作为类的一部分。它们有时（错误地）也称为成员模板(member templates)。尽管它们可以被参数化，但这种定义并非是第一类模板（指上述的几种模板）。它们的参数完全由成员所在的模板本身决定。例如：

```
template<int I>
class CupBoard
{
  class Shelf;                                                  // ordinary class in class template
  void open();                                                  // ordinary function in class templa
  enum Wood : unsigned char;                // ordinary enumeration type in class template
  static double totalWeight;                // ordinary static data member in class template
};
```

对应的定义仅仅只是为所属的类模板指定了参数化子句，但是却并没有为成员本身指定，因为其并非是一个模板（没有参数化子句与最后一个`::`之后出现的名称相关联）。

```
template<int I>                // definition of ordinary class in class template
class CupBoard<I>::Shelf {
  ...
};

template<int I>                // definition of ordinary function in class template
void CupBoard<I>::open()
{
  ...
}

template<int I>                // definition of ordinary enumeration type class in class template
enum CupBoard<I>::Wood {
  Maple, Cherry, Oak
};

template<int I>                // definition of ordinary static member in class template
double CupBoard<I>::totalWeight = 0.0;
```

C++17之后，静态成员`totalWeight`可以在类模板内部使用`inline`关键字初始化。

```
template<int I>
class CupBoard {
  ...
  inline static double totalWeight = 0.0;
};
```

尽管这种参数化定义通常被称作模板，但这里的“模板”一词相当不合适。对于这种情况，有一个经常被推荐的词是"temploid"。C++17之后，C++标准定义了模板化实体(a templated entity)的概念，它包括templates和temploids，以及递归地包含模板化实体中创建或定义的任何实体（这包括，例如，一个类模板内定义的友元函数（参考P30节2.4）或是模板中出现的一个lambda表达式闭包）。不管是temploid还是templated entity目前都没有产生足够的吸引力，但是在未来，需要更精准的沟通C++模板时，这些术语可能会很有用。

12.1.1 虚成员函数

成员函数模板不能被声明为virtual。施加这一限制是因为虚函数调用机制的通用实现会使用一个

固定大小的虚表，其中存储了每一个虚函数条目（译者注：虚函数指针）。然而，成员函数模板直到整个程序被编译之前，实例化的个数都无法固定。因此，成员函数模板支持virtual需要C++编译器和链接器支持一种全新的机制。

相反的，类模板的普通成员函数可以是virtual，因为它们的数量是固定的。

```
template<typename T>
class Dynamic {
  public:
    virtual ~Dynamic();                                  // OK: one destructor per instance of Dynam

        template<typename T2>
        virtual void copy(T2 const&);        // ERROR: unknown number of instances of copy()
                                                                                  // given an instance of Dy
};
```

12.1.2 模板的链接

每个模板都必须有一个名字，并且该名字必须是所属作用域内独一无二的，除了函数模板重载的情景（参考第16章）。特别要注意，与类类型不同，类模板无法与不同类型的实体共享名称：

```
int C;
...
class C;        // OK: class names and nonclass names are in a different "space"

int X;
...
template<typename T>
class X;        // ERROR: conflict with variable X

struct S;
...
template<typename T>
class S;        // ERROR: conflict with struct S
```

模板名称具有链接，但是他们无法拥有C链接。非标准链接可能具有某个依赖于实现体的意义（然而我们并不知道某个实现体支持模板的非标准链接与否）：

```
extern "C++" template<typename T>
void normal();                // this is the default: the linkage specification could be left out

extern "C" template<typename T>
void invalid();               // ERROR: templates cannot have C linkage

extern "Java" template<typename T>
void javaLink();        // nonstandard, but maybe some compiler will someday
                                        // support linkage compatible with Java generics
```

模板通常有外部链接。唯一的一些例外是命名空间作用域中具有静态限定符的函数模板、匿名空间的直接或间接的成员的模板（它们拥有内部链接）以及匿名类的成员模板（它们没有链接）。

举个例子：

```
template<typename T>          // refers to the same entity as a declaration of the
void external();                 // same name (and scope) in another file

template<typename T>         // unrelated to a template with the same name in
static void internal();         // another file

template<typename T>         // redeclaration of the previous declaration
static void internal();

namespace {
  template<typename>         // also unrelated to a template with the same name
  void otherInternal();         // in another file, even one that similarly appears
}                                                 // in an unnamed namespace

namespace {
  template<typename>         // redeclaration of the previous template declaration
  void otherInternal();
}

struct {
  template<typename T> void f(T) {} // no linkage: cannot be redeclared
} x;
```

注意到最后面的成员模板没有链接，它必须在匿名类定义处定义，因为想要在类外部定义是不可

能的。

当前，模板无法在函数作用域或局部类作用域中声明，但是泛化的lambda可以（参考P309节15.10.6），它有一个关联的闭包类型，其中包含了成员函数模板，其可以在局部作用域中出现，这实际上意味着一种局部成员函数模板。

模板实例的链接就是模板的链接。例如，函数`internal<void>()`从上面声明的模板`internal`实例化出来，它会拥有一个内部链接。而对于变量模板来说，这会产生一个有趣的后果。实际上，考虑下例：

```
template<typename T> T zero = T{};
```

`zero`所有实例化的实例都拥有一个外部链接，即使哪怕形如`zero<int const>`也是如此。这可能对既定的拥有一个内部链接的`int const zero_int = int{};`来说是违反直觉的，毕竟它使用了一个`const`类型来做修饰。同样的，模板`template<typename T> int const max_volume = 11;`实例化的所有实例也都拥有外部链接，尽管那些实例同样都是类型`int const`。

12.1.3 主模板

模板的一般性声明声明了主模板(primary templates)。如此声明的模板在模板名后无需书写尖括号模板参数子句。

```
template<typename T> class Box;                 // OK: primary template
template<typename T> class Box<T>;          // ERROR: does not specialize

template<typename T> void translate(T);         // OK: primary template
template<typename T> void translate<T>(T);          // ERROR: not allowed for functions

template<typename T> constexpr T zero = T{};          // OK: primary template
template<typename T> constexpr T zero<T> = T{};          // ERROR: does not specialize
```

非主模板会在声明类模板或变量模板的偏特化时出现。这些将在第16章讨论。函数模板始终必须是主模板（参考P356节17.3，这里讨论了未来语言变化的某种潜在可能）。

12.2 模板参数(Template Parameters)

有三种基本类型的模板参数：

1. 类型参数（目前最常用的）
2. 非类型模板参数
3. 模板模板参数

这些基本类型的模板参数中的任何一种都可以用作模板参数包的基础（参考P188节12.2.4）。

模板参数在模板声明的参数化引导子句中声明，该声明无需命名：

```
template<typename, int>
class X;                // X<> is parameterized by a type and an integer
```

当然，参数是否需要名称取决于模板后面的语句。还要注意，模板参数名可以在后续参数声明中引用（但前置则不行）：

```
template<typename T,                        //the first parameter is used
                T root,                                     // in the declaration of the second
                template<T> class Buf>        // in the declaration of the third one
class Structure;
```

12.2.1 类型参数

类型参数由关键字`typename`或`class`所引导：二者是完全等价的。关键字后必须有一个简单的标识符，并且该标识符后必须带有逗号，以表示下一个参数声明的开始，闭合的尖括号`>`用以指示参数化子句的结束，`=`用以指示一个默认模板参数的起始。

在模板声明内，类型参数的行为与类型别名(type alias)非常相似（参考P38节2.8）。例如，当`T`是模板参数时，即使`T`是被某种类(class)类型替换，也不能使用形如`class T`的详尽名称：

```
template<typename Allocator>
class List {
  class Allocator* allocptr;        // ERROR: use "Allocator* allocptr"
  friend class Allocator;               // ERROR: use "friend Allocator"
  ...
};
```

12.2.2 非类型参数

非类型模板参数表示一个可以在编译期或链接期确定的常量值。这样的参数类型（换句话说，它所代表的值类型）必须是以下之一：

- 整型或枚举型
- 指针类型
- 成员指针类型
- 左值引用类型（既可以是对象引用，也可以是函数引用）
- `std::nullptr_t`
- 包含`auto`或`decltype(auto)`的类型（C++17后支持；可参考P296节15.10.1）

其他类型当前都不支持（尽管浮点数在未来会被支持；可参考P356节17.2）。

也许令人惊讶的是，在某些情况下，非类型模板参数的声明也可以以关键字`typename`开头：

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

template<class X*>                // a nontype parameter of pointer type
class Y;
```

这两种情形很容易辨别，因为第一种的后面跟随了一个简单的标识符，然后是一小段标记（'='用以表示默认参数，','用以指示后面的另一个模板参数，'>'用以闭合模板参数列表）。P67节5.1和P229节13.3.2对第一个非类型模板参数的关键字`typename`做出了解释（译者注：这里的`typename`是用来表示`Allocator`是`T`内的一个类型，而非静态数据成员）。

函数和数组类型可以被指定，但是它们会通过退化(decay)隐式地调整为相应的指针类型：

```
template<int buf[5]> class Lexer;                // buf is really an int*
template<int* buf> class Lexer;                        // OK: this is a redeclaration

template<int fun()> struct FuncWrap;        // fun really has pointer to
                                                                                // function type
template<int (*)()> struct FuncWrap;        // OK: this is a redeclaration
```

非类型模板参数的声明与变量声明非常相似，但是它们不可以有非类型指示符，比如`static`、`mutable`等等。它们可以有`const`和`volatile`限定符，但是如果这种限定符出现在参数类型的最顶层，就会被忽略（译者注：换句话说，对左值引用或指针来说支持底层const）：

```
template<int const length> class Buffer;        // const is useless here
template<int length> class Buffer;                        // same as previous declaration
```

最后，在表达式中使用时，非引用类型的非类型参数始终都是`prvalues`（译者注：pure right values，即纯右值）。它们的地址无法被窃取，也无法被赋值。而另一方面，左值引用类型的非类型参数是可以像左值一样使用的：

```
template<int& Counter>
struct LocalIncrement {
  LocalIncrement() { Counter = Counter + 1; }        // OK: reference to an integer
  ~LocalIncrement() { Counter = Counter - 1; }
};
```

右值引用是不被允许的。

12.2.3 模板模板参数

模板模板参数是类或别名模板的占位符。它们的声明与类模板很像，但是不能使用关键字`struct`或`union`：

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

template<template<typename X> struct C>               // ERROR: struct not valid here
void f(C<int>* p);

template<template<typename X> union C>                // ERROR: union not valid here
void f(C<int>* p);
```

从C++17开始允许使用`typename`替代这里的`class`，驱使这一改动的原因在于：模板模板参数不仅可以由类模板替代，还可以由别名模板（可以实例化为任意类型）替代。因此，在C++17中，我们的上例可以改写成如下形式：

```
template<template<typename X> typename C>               // OK since C++17
void f(C<int>* p);
```

在其声明的作用域内，模板模板参数用起来就像另一个类模板或是别名模板一样。

模板模板参数的参数可以有默认模板参数。在使用模板模板参数而未指定相应的参数时，这些默认参数会生效：

```
template<template<typename T,
                                  typename A = MyAllocator> class Container>

class Adaptation {
  Container<int> storage;        // implicitly equivalent to Container<int,MyAllocator>
  ...
};
```

`T`和`A`都是模板模板参数`Container`的模板参数名称。这些名称仅可以在该模板模板参数的其他参数声明中使用。下面的模板阐释了这一概念：

```
template<template<typename T, T*> class Buf>        // OK
class Lexer {
  static T* storage;        // ERROR: a template template parameter cannot be used here
  ...
};
```

但是，通常在其他模板参数的声明中不需要模板模板参数的模板参数名称，因此常常根本不命名。例如，我们早期的`Adaptation`模板可以按如下声明：

```
template<template<typename,
                  typename = MyAllocator> class Container>
class Adaptation {
  Container<int> storage;        // implicitly equivalent to Container<int,MyAllocator>
  ...
};
```

12.2.4 模板参数包

从C ++ 11开始，可以通过在模板参数名称之前引入省略号（...）来将任何类型的模板参数转换为模板参数包（如果模板参数匿名，那么就在模板参数名称本该出现的位置之前）：

```
template<typename... Types>                // declares a template parameter pack named Types
class Tuple;
```

模板参数包的行为与其基础模板参数类似，但有一个关键的区别：普通的模板参数严格匹配某一个模板实参(template argument)，而模板参数包可以匹配任意数量的模板实参。这意味着上面声明的`Tuple`类模板可以接受任意数量任意类型（很可能彼此不一样）的模板实参：

```
using IntTuple = Tuple<int>;                        // OK: one template argument
using IntCharTuple = Tuple<int, char>;        // OK: two template arguments
using IntTriple = Tuple<int, int, int>;        // OK: three template arguments
using EmptyTuple = Tuple<>;                                // OK: zero templates arguments
```

同样，非类型参数和模板模板参数的模板参数包可以分别接受任意数量的非类型或模板模板实参，分别为：

```
template<typename T, unsigned... Dimensions>
class MultiArray;               // OK: declares a nontype template parameter pack

using TransformMatrix = MultiArray<double, 3, 3>;        // OK: 3x3 matrix

template<typename T, template<typename,typename>... Containers>
void testContainers();        // OK: declares a template template parameter pack
```

`MultiArray`示例需要全部的非类型模板实参均为相同的`unsigned`类型。C++17引入了非类型模板实参的推导，这将允许我们解除这一限制而做一些扩展（参考P298节15.10.1了解更多细节）。

主模板中的类模板、变量模板和别名模板至多只可以有一个模板参数包，且模板参数包必须作为最后一个模板参数。函数模板则少些限制：允许多个模板参数包，只要模板参数包后面的每个模板参数都具有默认值（请参阅下一节）或可以推导（参考第15章）：

```
template<typename... Types, typename Last>
class LastType;               // ERROR: template parameter pack is not the last template paramete

template<typename... TestTypes, typename T>
void runTests(T value);               // OK: template parameter pack is followed
                                                  // by a deducible template parameter
template<unsigned...> struct Tensor;
template<unsigned... Dims1, unsigned... Dims2>
auto compose(Tensor<Dims1...>, Tensor<Dims2...>);        // OK: the tensor dimensions can be deduc
```

最后一个例子使用了返回类型推导——C++14的特性。可以参考P296节15.10.1。

类和变量模板的偏特化声明（参考第16章）可以有多个参数包，这与主模板不同。这是因为偏特化是通过与函数模板几乎相同的推导过程所选择的。

```
template<typename...> Typelist;
template<typename X, typename Y> struct Zip;
template<typename... Xs, typename... Ys>
struct Zip<Typelist<Xs...>, Typelist<Ys...>>;
                        // OK: partial specialization uses deduction to determine
                        // the Xs and Ys substitutions
```

也许不足为奇的是，类型参数包不能在其自己的参数子句中进行扩展。例如：

```
template<typename... Ts, Ts... vals> struct StaticValues {};
        // ERROR: Ts cannot be expanded in its own parameter  list
```

然而，嵌套模板可以实现有效的类似情景：

```
template<typename... Ts> struct ArgList {
  template<Ts... vals> struct Vals {};
};
ArgList<int, char, char>::Vals<3, 'x', 'y'> tada;
```

包含模板参数包的模板被称为可变参数模板(variadic template)，因为它接受可变数量的模板参数。第4章和P200节12.4介绍了可变参数模板的使用。

12.2.5 默认模板实参

非模板参数包的任何类别的模板参数都可以配置默认参数，尽管它必须与相应的参数匹配（例如，类型参数不能有一个非类型默认实参）。默认实参不能依赖于其自身的参数，因为参数的名称直到默认实参之后才在作用域内生效。然而，他可以依赖前面的参数：

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

当且仅当还为后续参数提供了默认参数时，类模板、变量模板或别名模板的模板参数才可以具有默认模板实参。（对默认函数调用参数来说有着相似的限制条件。）通常在同一模板声明中提供后续所有的默认值，但也可以在该模板的先前声明中声明它们。下面的例子可以清楚地做出解释：

```
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 and T5 already have defaults

template<typename T1 = char, typename T2, typename T3,
                 typename T4, typename T5>
class Quintuple;        // ERROR: T1 cannot have a default argument
                                   // because T2 doesn't have a default
```

函数模板的模板参数的默认模板实参，则不受这样的约束：

```
template<typename R = void, typename T>
R* addressof(T& value);        // OK: if not explicitly specified, R will be void
```

默认模板实参不允许重复声明：

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

template<typename T = void>
class Value;        // ERROR: repeated default argument
```

许多上下文不允许使用默认模板实参：

- 偏特化：

```
template<typename T>
class C;
...
template<typename T = int>
class C<T*>;        // ERROR
```

- 参数包：

```
template<typename... Ts = int> struct X;        // ERROR
```

- 类模板成员类外定义：

```
template<typename T>
struct x
{
        T f();
};

template<typename T = int> // ERROR
T X<T>::f() {
  ...
}
```

- 友元类模板声明：

```
struct S {
  template<typename = void> friend struct F;
};
```

- 友元函数模板声明，除非它是一个定义并且它在编译单元的其他任何地方都没有声明：

```
struct S{
  template<typename = void> friend void f();        // ERROR: not a definition
  template<typename = void> friend void g() {        // OK so far
  }
};

template<typename> void g();        // ERROR: g() was given a default template argument
                                                    // when defined; no other declarat
```

12.3 模板实参(Template Arguments)

实例化模板时，模板实参会替换模板参数。模板实参可以被各种不同类型的机制所判定：

- 显式模板实参：模板名称后可以跟随在尖括号内显式指定的模板实参。这种名称被叫做模板ID（template-id）。
- 注入式类名：在具有模板参数`P1, P2 ...`的类模板`X`的作用域内，该模板（`X`）的名称可以等价于模板ID `X<P1, P2, ...>`。可以参考P221节13.2.3了解更多细节。
- 默认模板实参：如果默认模板实参可用，则可以在模板实例化时省略显式的模板实参。然而，对于类模板或别名模板来说，即使模板参数有默认值，尖括号也不能省略（其内可以为空）。
- 实参推导：没有被显式指定的函数模板参数会通过函数调用的实参类型来进行推导。在第15章对细节进行了描述。在一些其他情景中也会完成推导。如果所有的模板实参都可以被推导，那么函数模板的名称后就无需书写尖括号子句。C++17还引入了从变量声明或函数符号类型转换的初始化器中推导类模板实参的能力。可以参考P313节15.12中对此的一个探讨。

12.3.1 函数模板实参

函数模板的模板实参可以被显式地指定，它会按模板被使用的方式来推导，或者直接使用默认模板实参。例如：

details/max.cpp

```
template<typename T>
T max(T a, T b)
{
  return b < a ? a : b;
}

int main()
{
  ::max<double>(1.0, -3.0);        // explicitly specify template argument
  ::max<1.0, -3.0);                      // template argument is implicitly deduce to be double
  ::max<int>(1.0, 3.0);                // the explicit <int> inhibits the deduction;
                                                                  // hence the result has type int
}
```

某些模板实参永远不会被推导，这可能是因为它们所对应的模板参数并没有在函数参数类型中出现或是一些其他原因（参考P271节15.2）。这种参数通常应放在模板参数列表的开头，使其能被显式地指定，而于此同时也让其他参数能够完成推导。例如：

details/implicit.cpp

```
template<typename DstT, typename SrcT>
DstT implicit_cast (SrcT const& x)                // SrcT can be deduced, but DstT cannot
{
  return x;
}

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

如果我们反转示例中模板参数的顺序（换句话说，写成`template<typename SrcT, typename DstT>`），`implicit_cast`的调用就必须同时显式地指定两个参数。

此外，这样的参数不能合法地放在模板参数包之后或在偏特化中出现，因为无法明确地指定或推导它们。

```
template<typename... Ts, int N>
void f(double (&)[N+1], Ts... ps);        // useless declaration because N
                                                                  // cannot be specified or
```

由于函数模板可以重载，为函数模板显式地指定所有的实参可能也无法充分指定某一个特定函数：在某些场景中，它选中了一个函数集。下面的例子阐述了这一现象：

```
template<typename Func, typename T>
void apply(Func funcPtr, T x)
{
  funcPtr(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: no single multi<int>
}
```

本例中，第一个`apply()`调用可以成功是因为表达式`&single<int>`没有歧义。如此，模板实参值`Func`就可以被轻易的推断。在第二个调用中，`&multi<int>`可能是2种不同的类型，因此`Func`无法被推导。

更进一步，在函数模板中替换模板实参可能会导致尝试构造无效的C++类型或表达式。考虑下面的重载函数模板（`RT1`和`RT2`没有指定类型）：

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

表达式`test<int>`对于上述两种函数模板的前者来说都是没有意义的，因为类型`int`并没有成员类型`X`。然而，后者没有这样的问题。因此，表达式`&test<int>`标志了一个特定函数的地址。将`int`替换第一个函数模板失败的事实并不会使表达式无效。这一SFINAE (substitution failure is not an error)原则对函数模板的重载来说是非常关键的一部分，我们会在P129节8.4和P284节节15.7中讨论。

12.3.2 类型实参

模板类型实参是模板类型参数的选定“值”。任何类型（包括`void`，函数类型，引用类型等等）通常来说都可以作为模板实参，但是它们对模板参数的替换构成必须是合法的：

```
template<typename T>
void clear(T p)
{
  *p = 0;         // requires that the unary * be applicable to T
}

int main()
{
  int a;
  clear(a);        // ERROR: int doesn't support the unary *
}
```

12.3.3 非类型实参

非类型实参是指那些替换非类型模板参数的值。这种值必须是以下其中一项：

- 另一个具有正确类型的非类型模板参数。
- 整型（或枚举）类型的编译器常量。只有在相应的参数具有一个匹配该类型或是一个无需塌缩(narrowing)而可以被隐式转换到该类型的值的时候才可以接受。例如，`char`值可以提供给`int`参数，但是`500`对于`char`这一8位参数来说却是无效的。
- 外部变量或函数的名称，其前面带有内置的一元`&`（“取址”）运算符。对于函数和数组变量，可以省略`&`。此类模板实参与指针类型的非类型参数匹配。 C++17放宽了此要求，允许任何的常量表达式产生一个指向函数或变量的指针。
- 对于引用类型的非类型参数，前一种（但不带`&`运算符）实参是有效实参。同样地，C++17在这里也放宽了约束，允许任意的常量表达式`glvalue`应用于函数或变量。
- 成员指针常量；换句话说，表达式形如`&C::m`，其中`C`是类类型，`m`是非静态成员（数据或函数）。这只会匹配成员指针类型的非类型参数。同样的，在C++17中，实际的语法形式不再受限制：对匹配的成员指针常量的任何常量表达式求值都会被允许。
- 空指针常量对指针或成员指针的非类型参数来说都是合法的。

对整型类型的非类型参数来说（可能也是最常用的非类型参数），到这一参数类型的隐式转换是可行的。随着C++ 11中constexpr转换函数的引入，这意味着转换前的参数可以具有类类型。

C++17之前，将实参与作为指针或引用的参数进行匹配时，不会考虑用户定义的转换（单参数

构造函数和转换运算符）和派生类到基类的转换，即使在其他情况下它们是有效的隐式转换。使得实参更`const`和/或更`volatile`的隐式转换是可行的。

下面是一些有效的非类型模板实参的例子：

```
template<typename T, T nontypeParam>
class C;

C<int, 33>* c1;                // integer type
int a;
C<int*, &a>* c2;         // address of an external variable

void f();
void f(int);
C<void (*)(int), f>* c3;         // name of a function: overload resolution selects
                                                        // f(int) in this case; the & is implied

template<typename T> void templ_func();
C<void(), &templ_func<double>>* c4;         // function template instantiations are functions
struct X {
  static bool b;
  int n;
  constexpr operator int() const { return 42; }
};

C<bool&, X::b>* c5;         // static class members are acceptable variable/function names

C<int X::*, &X::n>* c6;         // an example of a pointer-to-member constant

C<long, X{}>* c7;                 // OK: X is the first converted to int via a constexpr conversion
                                                    // function and then to long via a standard intege
```

模板实参的一个通用限制在于编译器或链接器必须在程序构建时有能力表示它们的值。在程序运行前无法知晓的值（例如，局部变量的地址）在程序构建时与模板实例化的概念是不相容的。

尽管如此，还是有着一些常量目前是无效的，这可能会令人惊讶：

- 浮点数
- 字符串字面量（C++11之前，空指针常量也不行）

字符串字面量的一个问题在于两个相同的字面量可以存储在不同的地址上。对常量字符串做模板实例化有另一种迂回的方法（但麻烦），这涉及了引入一个附加变量来保存字符串：

```
template<char const *str>
class Message {
  ...
};

extern char const hello[] = "Hello Wolrd!";
char const hello11[] = "Hello World!";

void foo()
{
  static char const hello17[] = "Hello World!";

  Message<hello> msg03;                // OK in all versions
  Message<hello11> msg11;        // OK since C++11
  Message<hello17> msg17;        // OK since C++17
}
```

必要条件是声明为引用或指针的非类型模板参数必须是一个在C++全版本中拥有外部链接的常量表达式，自C++11起内部链接亦可，而C++17之后则只要求有任意的某个链接就行。

参考P354节17.2对这一领域未来可能发生变化的一个讨论。

这里有些（少得可怜）非法的示例：

```
template<typename T, T nontypeParam>
class C;

struct Base {
  int i;
} base;

struct Derived : public Base {
} derived;

C<Base*, &derived>* err1;                // ERROR: derived-to-base conversions are not considered

C<int&, base.i>* err2;                      // ERROR: fields of variables aren't considered to b

int a[10];
C<int*, &a[0]>* err3;                     // ERROR: addresses of array elements aren't acceptab
```

12.3.4 模板模板实参

模板模板实参通常必须是一个严格匹配类模板或别名模板的模板参数的实参替换。C++17之前，模板模板实参的默认参数会被忽略（但是如果模板模板参数有默认参数，它们会在模板实例化时被考虑）。C++17放宽了这一匹配规则，它只需要模板模板参数至少被相应的模板模板实参特化（参考P330节16.2.2）。

在C++17之前下面的例子是非法的：

```
#include <list>
        // declares in namespace std:
        // template<typename T, typename Allocator=allocator<T>>
        // class list;
template<typename T1, typename T2, template<typename> class Cont> // Cont expects one parameter
class Rel {
  ...
};

Rel<int, double, std::list> rel;        // ERROR before C++17: std::list has more than
                                                                  // one template parameter
```

示例中的问题在于`std::list`这一标准库模板拥有多于一个的模板参数。第二个参数（描述一个`allocator`）拥有一个默认值，但是在C++17之前，在匹配`std::list`为`Container`参数时这并不会被考虑。

可变模板模板参数是C++17之前上述描述的“严格匹配”规则的一个例外，同时它也有一个解除这一限制的方案：它们对模板模板实参启用更通用的匹配。模板模板参数包可以匹配零到多个模板模板实参中的相同种类的模板参数。

> 译者注：这里相同种类不是指狭义的数据类型，而是指类型参数、非类型参数、函数模板参数、模板模板参数这些不同的类别（也就是12.3分开讨论的这些）。

```
#include <list>

template<typename T1, typename T2,
                 template<typename... > class Cont>                // Cont expects any number of
class Rel {                                                                                  // type
  ...
};

Rel<int, double, std::list> rel;                                  // OK: std::list has two template
                                                                                                //
```

模板参数包只能匹配相同种类的模板参数。例如，下面的类模板可以使用仅有一个模板参数类型的任意类模板或别名模板实例化，因为模板类型参数包在这里传递的`TT`可以匹配零到多个模板类型参数：

```
#include <list>
#include <map>
        // declares in namespace std;
        // template<typename Key, typename T,
                                typename Compare = less<Key>,
                                typename Allocator = allocator<pair<Key const, T>>>
        // class map;
#include <array>
        // declares in namespace std;
        // template<typename T, size_t N>
        // class array;

template<template<typename... > class TT>
class AlmostAnyTmpl {
};

AlmostAnyTmpl<std::vector> withVector;        // two type parameters
AlmostAnyTmpl<std::map> witMap;                        // four type parameters
AlmostAnyTmpl<std::array> withArray;        // ERROR: a template type parameter pack
                                                                                // doesn't match a
```

在C++17之前，声明模板模板参数只能使用关键字`class`，但这并不代表仅允许将用关键字class声明的类模板用作替换参数。实际上，`struct`，`union`以及别名模板也都是模板模板参数的合法实参（别名模板是C++11后才出现并支持）。这类似于这一现象：任何类型都可以用作关键字`class`声明的模板类型参数的实参。

12.3.5 等价性(equivalent)

当两组模板实参的每一对参数值都相同时，它们被视为等价的。对于类型参数，类型别名无关紧要：最终比较的是类型别名所声明的底层类型。对于整型非类型实参，参数的值会被比较；这个值如何表示无关紧要。下面的例子阐释了这一概念：

```
template<typename T, int I>
class Mix;

using Int = int;

Mix<int, 3*3>* p1;
Mix<int, 4+5>* p2;        // p2 has the same type as p1
```

(正如这一示例所澄清，无需模板定义即可确定模板参数列表的等价性。)

在模板依赖上下文中，模板实参的”值“却是无法一直被明确确定的，且对等价性来说这里的规则更加复杂。考虑下例：

```
template<int N> struct I {};

template<int M, int N> void f(I<M+N>);        // #1
template<int N, int M> void f(I<N+M>);        // #2

template<int M, int N> void f(I<N+M>);        // #3 ERROR
```

谨慎声明#1和#2，你将注意到它们仅仅是交换重命名了的`M`和`N`，你得到了相同的声明：二者是等价的，它们声明了相同的模板`f`。表达式`M+N`和`N+M`在这两个声明中被视为等价的。

然而#3的声明，确是有着巧妙的不同：只有操作数被翻转。这会让表达式`N+M`与前两者都不等价。然而，对于任意的模板参数值，最终产生的都是相同的结果，因此，这些表达式在功能上也是等价的（functionally equivalent）。以此差别而声明模板是错误的行径，尽管它们不等价但在功能上确是等价的。然而，编译器无需诊断此类错误。这是因为某些编译器可能，举例来说，在内部将`N+1+1`表示为等同的`N+2`，但其他编译器则不然。C++标准没有强行规定某种特定的实现方式，而是两者皆允，同时要求程序员对这一领域保持谨慎。

函数模板生成的函数与普通的函数永远不是等价的，尽管他们可能有相同的类型和名称。这对类成员来说产生了两个重要影响：

1. 成员函数模板生成的函数永远不会覆盖(override)虚函数。
2. 构造器模板生成的构造器永远不会是拷贝或移动构造器。类似的，赋值操作符模板生成的赋值操作符函数也永远不会是拷贝赋值或是移动赋值操作符函数。（然而，由于隐式调用拷贝赋值或移动赋值操作符函数的情景相对少，所以这一般不会引起

问题。）这一事实各有优劣。可以参考P95节6.2和P102节6.4了解更多细节。

12.4 可变模板

在P55节4.1中介绍的可变模板参数，是指那些至少包含一个模板参数包（参考P188节12.2.4）的模板。当模板的行为可以泛化为任意数量实参时可变模板将非常有用。P188节12.2.4引入的`Tuple`类模板就是一个可变模板，因为一个tuple可以有任意数量的元素，它们被同等对待。我们也可以想象一个简单的`print()`函数，它携带任意数量的参数并按顺序打印每一个。

当可变模板的模板实参被确定时，可变模板的每个模板参数包都将匹配连续的零到多个模板实参。我们将此模板实参序列称为实参包(argument pack)。下面的例子阐述了模板参数包`Types`是如何根据`Tuple`所提供的模板实参而匹配不同的实参包的。

```
template<typename... Types>
class Tuple {
        // provides operations on the list of types in Types
};

int main() {
  Tuple<> t0;                                  // Types contains an empty list
  Tuple<int> t1;                            // Types contains int
  Tuple<int, float> t2;          // Types contains int and float
}
```

由于模板参数包代表了若干个而不是单一的模板实参，它必须在实参包中所有参数都被应用的相同语法结构上下文中使用。其中之一就是`sizeof...`操作符，它会对实参包中实参的个数进行计数。

```
template<typename... Types>
class Tuple {
  public:
    static constexpr std::size_t length = sizeof...(Types);
};

int a1[Tuple<int>::length];                                          // array of the integer
int a3[Tuple<short, int, long>::length];        // array of three integers
```

12.4.1 包展开(Pack Expansions)

`sizeof...`表达式是包展开的一个例子。包展开是一种把一个实参包展开成独立实参的结构。`sizeof...`执行这一展开只是为了去计数独立实参的个数，其他形式的实参包——那些在C++渴望一个列表的场合——可以将列表展开成多个元素。这样的包展开由列表中元素右侧的省略号（...）标识。这里有一个简单的例子，我们创建了一个新的类模板`MyTuple`，它传递实参给`Tuple`的同时也从`Tuple`类继承：

```
template<typename ...Types>
class MyTuple : public Tuple<Types..> {
        // extra operations provided only for MyTuple
};

MyTuple<int, float> t2;        // inherits from Tuple<int, float>
```

模板实参`Types...`是一个包展开，它产生了一个模板实参序列，实参包中的每个实参都用于取代`Types`。如例子中所展示，实例化的类型`MyTuple<int, float>`的模板类型参数包`types`被实参包`int, float`所取代。当出现在参数展开`Types...`时，我们得到一个模板实参`int`和另一个模板实参`float`，因此`MyTuple<int, float>`从`Tuple<int, float>`处继承。

理解包展开的一种直观方法是根据语法展开来思考它们，模板参数包将被正确数量的（非包）模板参数替换，并且包展开被写为单独的参数，每个非包类型的模板参数各一个。例如，`MyTuple`被展开成两个参数应该长这个样子：

```
template<typename T1, typename T2>
class MyTuple : public Tuple<T1, T2> {
  // extra operations provided only for MyTuple
};
```

三个参数则长这样子：

```
template<typename T1, typename T2, typename T3>
class MyTuple : public Tuple<T1, T2, T3> {
  // extra operations provided only for MyTuple
};
```

然而请注意，你无法直接通过名字来访问参数包中的独立元素，因为`T1, T2`等名字并没有在可

变模板中定义。如果你需要类型，唯一可以做的事就是传递它们(非递归地)给另一个类或函数。

每个包展开都有一个模式(pattern)，它是一个被实参包的每个实参所替换的类型或表达式，并且通常出现在表示包展开的省略号之前。我们前面的例子都只有些无关紧要的模式——参数包的名称——但是模式可以更为复杂。例如，我们可以定义一个新类型`PtrTuple`，它继承于实参类型的指针所构成的`Tuple`：

```
template<typename... Types>
class PtrTuple : public Tuple<Types*...> {
  // extra operations provided only for PtrTuple
};

PtrTuple<int, float> t3;        // Inherits from Tuple<int*, float*>
```

包展开`Types*...`的模式是`Types*`。该模式产生了一个模板类型实参替换的序列，每个实参的类型都被其对应的指针类型所取代，并应用于Types中。在包展开的语法解释下，这是如果将`PtrTuple`扩展为三个参数时看起来的样子：

```
template<typename T1, typename T2, typename T3>
class PtrTuple : public Tuple<T1*, T2*, T3*> {
  // extra operations provided only for PtrTuple
};
```

12.4.2 包展开可以在哪里出现？

我们目前的例子都是聚焦于使用包展开来产生一个模板实参序列。实际上，包展开基本上可以在语法提供逗号分隔列表的任何位置使用，这包括：

- 基类列表
- 构造器中的基类初始化列表(initializer)
- 调用实参列表(模式就是实参表达式)
- 初始化列表(例如，在花括号初始化列表(initializer list))
- 类、函数或别名模板的模板参数列表
- 函数可以抛出的异常列表(自C++11起不建议使用、C++17后不再允许)
- 在属性内，如果属性本身支持包展开（尽管在C++标准中没有定义这样的属性）
- 指定某个声明的对齐方式时
- 指定lambda表达式捕获列表时

- 函数类型的参数列表
- 在`using`声明中（自C++17起支持；参考P65节4.4.5）。我们已经提到过`sizeof...`作为一种包展开机制，它并不会真正产生一个列表，C++17也增加了表达式折叠(fold expressions)，这是另一种不产生逗号分隔的列表的机制（参考P207节12.4.6）

上述包展开所在的某些上下文只是为了归纳的完整性，因此，我们仅将注意力集中在那些在实践中往往有用的包展开上下文上。毕竟包展开在所有上下文中都遵循相同的原则和语法，你大可从此处给出的示例推断出是否需要更深奥的包展开上下文。

在基类列表中的包展开会扩展成多个直接基类。这种扩展对于通过mixins聚合外部提供的数据和功能很有用，mixins是旨在“混合到”类层次结构中以提供新行为的类。例如，下面的`Point`类在多个不同上下文中使用了包展开以允许任意的mixins：

```
template<typename... Mixins>
class Point : public Mixins... {        // base class pack expansion
    double x, y, z;
  public:
    Point() : Mixins()... { }               // base class initializer pack expansion
        template<typename Visitor>
        void visitMixins(Visitor visitor) {
          visitor(static_cast<Mixins&>(*this)...);        // call argument pack expansion
        }
};

struct Color { char red, green, blue; };
struct Label { std::string name; };
Point<Color, Label> p;                // inherits from both Color and Label
```

`Point`类使用包扩展来获取每个提供的mixin，并将其扩展为公有继承的基类。`Point`的默认构造器在类初始化列表中使用了包展开，对mixin机制引入的每个基类进行了值初始化。

成员函数模板`visitMixins`最有趣，它使用了包展开的结果作为调用参数。通过转换`*this`为每一种mixin类型，包展开生成了每个基类对应mixin类型的调用参数。P204节12.4.3中介绍了实际上与`visitMixins`一起使用而编写的visitor，它可以使用任意数量的函数调用参数。

包展开也在模板参数列表中创建非类型模板参数包时使用：

```
template<typename... Ts>
struct Values {
  template<Ts... Vs>
  struct Holder {
  };
};

int i;
Values<char, int, int*>::Holder<'a', 17, &i> valueHolder;
```

注意一旦`Values<..>`的类型实参被确定，`Values<...>::Holder`的非类型实参列表就是固定的尺寸；参数包`Vs`就不是一个变长参数包。

`Values`是一个非类型模板参数包，其中每个真实的模板实参都可以是不同的类型，它们由模板类型参数包`Types`提供的类型所指定。请注意，`Values`声明中的省略号起着双重作用，既将模板参数声明为模板参数包，又将该模板参数包的类型声明为一个包展开。这种模板参数包在实践中非常罕见，而在一个更加常见的上下文——函数参数中这种规则同样生效。

12.4.3 函数参数包

函数参数包(function parameter pack)是一个匹配零到多个函数调用实参的函数参数。与模板参数包相似，函数参数包通过在函数参数名前使用前置省略号引入，同样地，函数参数包在使用时必须由包展开来扩展。模板参数包和函数参数包被统一称作参数包(parameter packs)。

与模板参数包不同的是，函数参数包始终都是包展开，因此它们声明的类型必须包含至少一个参数包。下面的例子中，我们引入一个新的`Point`构造器，使用提供的构造器实参来拷贝初始化每一个mixin：

```
template<typename... Mixins>
class Point : public Mixins...
{
    double x, y, z;
  public:
    // default constructor, visitor function, etc. elided
        Point(Mixins... mixin)        // mixins is a function parameter pack
                : Mixins(mixins)...{ }        // initialize each base with the supplied mixin valu
};

struct Color { char red, green, blue; };
struct Label { std::string name; };
Point<Color, Label> p({0x7F, 0, 0x7F}, {"center"});
```

函数模板的函数参数包可能依赖于模板中声明的模板参数包，这使得函数模板可以接受任意数量的调用实参而不会损失类型信息：

```
template<typename... Types>
void print(Types... values);

int main
{
  std::string welcome("Welcome to ");
  print(welcome, "C++", 2011, '\n');        // calls print<std::string, char const*,
                                                                                //
}
```

当使用多个实参调用函数模板`print()`时，实参的类型将放置在参数包中，以取代模板类型参数包`Types`，而实参本身则放入参数包中，以代替函数参数包`Values`。调用实参被确定的过程在第15章对细节进行了描述。当前，只要了解`Types`中的第`i`个类型对应`Values`的第`i`个值即可，并且这些参数包的每一对在函数模板`print()`内都是可用的。

`print()`的真正实现使用了递归的模板实例化，这是一种模板元编程技术，在P123节8.1和第23章中有所描述。

在参数列表末尾出现的匿名函数参数包与C样式的“ vararg”参数之间在语法上存在歧义。例如：

```
template<typename T> void c_style(int, T...);
template<typename... T> void pack(int, T...);
```

前者的`T`被视为`T, ...`：一个匿名参数类型`T`跟着一个C风格的vararg参数。后者的`T...`结构被视为一个函数参数包，因为`T`是一个合法的展开模式。可以通过在省略号前强制添加一个逗号（这保证了省略号被认作C风格vararg参数）或在省略号后跟随一个标识符——这意味着它是一个命名函数参数包来消除歧义。请注意，在通用的lambda中，如果紧随其后的类型（没有中间逗号）包含auto，则尾随的`…`将被视为表示参数包。

12.4.4 多重与嵌套包展开

包展开的模式可以随意复杂且可以包含多重、不同的参数包。当实例化包含多重参数包的包展开时，所有的参数包都必须有相同的尺寸。从每个参数包的第一个实参开始进行模式替换，然后是每个参数包的第二个实参，以此类推，最终组织成类型或值的序列。例如，下面的函数在转发所有实参给函数对象`f`之前，对他们进行了拷贝：

```
template<typename F, typename... Types>
void forwardCopy(F f, Types const&... values) {
  f(Types(values)...);
}
```

调用实参包展开命名了两个实参包，`Types`和`values`。当实例化该模板时，`Types`和`values`参数包的逐个元素会产生一系列对象构造体，它们使用`Types`的第i个类型创建了`values`的第i个值。在包展开的语法解析下，三个实参的`forwardCopy`可能长这个样子：

```
template<typename F, typename T1, typename T2, typename T3>
void forwardCopy(F f, T1 const& v1, T2 const& v2, T3 const& v3) {
  f(T1(v1), T2(v2), T3(v3));
}
```

包展开本身也可以嵌套。此时，每个参数包都可以由最近的一个闭合的包展开所扩展（也只能是这个包展开）。下面的例子阐释了引入3个不同参数包的嵌套包展开：

```
template<typename... OuterTypes>
class Nested {
  template<typename... InnerTypes>
  void f(InnerTypes const&... innerValues) {
    g(OuterTypes(InnerTypes(innerValues)...)...);
  }
};
```

`g()`的调用中，模式`InnerTypes(innerValues)`的包展开是最内层的，它扩展了`InnerTypes`和`innerValues`并为`OuterTypes`表示的对象产生了一个函数调用实参序列。外层的包展开模式包含内层包展开，为函数`g()`产生了一个调用参数集，它们由内层包展开生成的函数调用实参序列所形成的`OuterTypes`中的每一种实例化类型所创造。在这种包展开的语法解析下，当`OuterTypes`有2个实参，`InnerTypes`和`innerValues`都有3个实参时，嵌套会变得更加明显：

```
template<typename O1, typename O2>
class Nested {
  template<typename I1, typename I2, typename I3>
  void f(I1 const& iv1, I2 const& iv2, I3 const& iv3) {
    g(O1(I1(iv1), I2(iv2), I3(iv3)),
          O2(I1(iv1), I2(iv2), I3(iv3)));
  }
};
```

> 这里作者多写了一行O3

多重与嵌套包展开是一个非常强力的工具（例如，参考P608节26.2）。

12.4.5 零尺寸包展开

包展开的语法解析对于理解不同实参数量的可变模板实例化的方式非常有用。然而，对于零尺寸实参包来说语法解析经常会失败。为了说明这一点，请考虑P202节12.4.2中的`Point`类模板，该模板在语法上用零个实参替换：

```
template<>
class Point : {
  Point() : { }
};
```

上面编写的代码格式不正确，因为模板参数列表现在为空，并且空的基类和基类初始化器列表每个都有一个冒号。

包展开实际上是语义结构，任意尺寸实参包的替换并不会影响包展开（或其封闭的可变参数模板）的解析。当包扩展展开成一个空列表时，程序的表现（语义上）就好像该列表不曾存在。实例化`Point <>`最终没有基类，并且其默认构造函数没有基类初始化程序，但其格式正确。这一语法规则使得即使是零尺寸的包展开也可以被完美定义（但有所区别）。例如：

```
template<typename T, typename... Types>
void g(Types... values) {
  T v(values...);
}
```

可变函数模板`g()`创造了一个值`v`，它使用传入的`values`一系列值来直接初始化。如果`values`是空的，那么`v`在语法上看起来就好像是一个函数声明`T v()`。然而，因为包展开的替换是一种语法且解析时不会产生影响其他类型的实体，`v`会通过零个实参进行初始化，也就是说，这依然还是值初始化。

12.4.6 折叠表达式

对一连串的值进行同一模式的递归处理被称做操作的折叠。例如，对序列`x[1],x[2],...,x[n-1],x[n]`进行函数`fn`右折叠会得到`fn(x[1],fn(x[2], fn(...,fn(x[n-1],x[n])...)))`。在探索一种新的语言特性时，C++委员会遇到了需要特殊处理的结构：应用于包展开的二元逻辑运算符(即`&&`或`||`)。在没有额外的语法特性时，我们需要编写下面的代码来实现`&&`操作：

```
bool and_all() { return ture; }
template<typename T>
bool and_all(T cond) { return cond; }
template<typename T, typename... Ts>
bool and_all(T cond, Ts... conds) {
  return cond && and_all(conds...);
}
```

C++17引入了一种新的特性——折叠表达式(fold expressions)（参考P58节4.2）。它可以应用于除了`.`，`->`和`[]`以外的所有的二元操作符。

给定一个未展开表达式模式`pack`和一个非模式表达式`value`，C++17允许我们使用任意操作符`op`写出：

```
(pack op ... op value)
```

作为一个操作符右折叠（称作二元右折叠），或者写出：

```
(value op ... op pack)
```

作为一个操作符左折叠（称作二元左折叠）。参考P58节4.2了解更多基本示例。

折叠操作应用于一个序列，对包进行展开并从最后一个（右折叠）或第一个（左折叠）序列中的元素施加`value`。

有了这一特性，如下代码：

```
template<typename... T> bool g() {
  return and_all(trait<T>()...);
}
```

（`and_all`在上面代码中定义），就可以被替换写成：

```
template<typename... T> bool g() {
  return (trait<T>() && ... && true);
}
```

如你所愿，折叠表达式是包展开。注意即使包为空，折叠表达式的类型仍然可以借由非包操作数(上例中是`value`)来确定。

然而，这一特性的设计者还希望增加一个摆脱`value`操作数的选项。在C++17中还支持另外两种形式：一元右折叠`(pack op ...)`和一元左折叠(`... op pack`)。

此时小括号依然是必须的。很明显对于空展开来说这产生了一个问题：如何确定它们的类型或是值呢？答案就是对于一元折叠表达式来说，空展开通常来说会导致一个错误，除了以下三种特例：

- 单一折叠`&&`对空展开产生一个值`true`。
- 单一折叠`||`对空展开产生一个值`false`。
- 单一折叠`,`会产生表达式`void`。

注意，如果你重载上述某个特殊的操作符时（通常不太常见），可能会出乎意料，例如：

```
struct BooleanSymbol {
  ...
};

BooleanSymbol operator||(BooleanSymbol, BooleanSymbol);

template<typename... BTs> void symbolic(BTs... ps) {
  BooleanSymbol result = (ps || ...);
  ...
}
```

假设我们用从`BooleanSymbol`继承的类型来调用`symbolic`。对所有展开来说，除了空展开以外，都会产生一个`BooleanSymbol`值（空展开产生的是布尔值）。我们要注意一元折叠表达式的使用，并推荐以二元折叠表达式作为替代（显式地指定空展开值）。

12.5 友元

声明友元的初衷非常简单：在某个类中标记友元函数或友元类以使其获得访问特权。由于以下两个因素，事情变得有些复杂：

1. 友元的声明必须是唯一的。
2. 友元函数声明时可以直接定义。

12.5.1 类模板的友元类

友元类声明时不能定义，因此很少出问题。在模板的上下文中，友元类声明的唯一新奇之处是在于能够将类模板的特定实例声明为友元：

```
template<typename T>
class Node;

template<typename T>
class Tree {
  friend class Node<T>;
  ...
};
```

请注意，类模板必须在其实例之一成为类或类模板的友元时是可见的。对普通类来说，则没有这种要求：

```
template<typename T>
class Tree {
  friend class Factory;        // OK even if first declaration of Factory
  friend class Node<T>;        // error if Node isn't visible
};
```

P220节13.2.2对此有更多描述。

P75节5.5引入了一个例子，给出了其他类模板实例做友元时的声明：

```
template<typename T>
class Stack {
  public:
    ...
        // assign stack of elements of type T2
        template<typename T2>
        Stack<T>& operator=(Stack<T2> const&);
        // to get access to private members of Stack<T2> for any type T2:
        template<typename> friend class Stack;
        ...
};
```

C++11也增加了让模板参数作友元的语法：

```
template<typename T>
class Wrap {
  friend T;
  ...
};
```

对任何类型`T`来说这都是合法的，如果`T`不是一个类类型的话，友元就会被忽略（译者注：基础类型不需要声明为友元）。

12.5.2 类模板的友元函数

函数模板的实例可以作为友元，只要保证友元函数名称后跟着一个尖括号子句即可。尖括号子句可以包含模板实参，但是如果实参可以被推导，那么尖括号就可以留空：

```
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: doesn't match combine() template
  friend void combine<>(long, long) { ... }        // ERROR: definition not allowed!
};
```

请注意，我们无法定义模板实例（最多可以定义一个特化体），因此友元声明不能是一个定义。

如果名称后没有跟尖括号子句，那么有两种可能：

1. 如果名字没有限定符（换句话说，不包含`::`），它永远不会是一个模板实例。如果友元声明时不存在可见的匹配的非模板函数，此处的友元声明就作为该函数的第一次声明。该声明也可以是一个定义。
2. 如果名称带有限定符（包含`::`），该名称必须可以引用到一个此前声明过的函数或函数模板。非模板函数会比函数模板优先匹配。然而，这里的友元声明不能是一个定义。这里有个例子来说明这一区别：

```
void multiply(void*);        // ordinary function

template<typename T>
void multiply(T);               // function template

class Comrades {
  friend void multiply(int) { }        // defines a new function ::multiply(int)

  friend void ::multiply(void*); //refers to the ordinary function above,
                                                                  // not the the multiply<void*> i
  friend void ::multiply(int);        // refers to an instance of the template
  friend void ::multiply<double*>(double*);        // qualified names can also have angle brackets
                                                                                  // but a

  friend void ::error() { }        // ERROR: a qualified friend cannot be a definition
};
```

在前例中，我们在一个普通的类中声明了友元函数。在类模板中声明友元函数规则也是如此，只不过模板参数可以参与到函数声明中：

```
template<typename T>
class Node {
  Node<T>* allocate();
  ...
};

template<typename T>
class List {
  friend Node<T>* Node<T>::allocate();
};
```

函数模板也可以在类模板中定义，此时只有在它真正被使用到时才会实例化。通常，这要求友元函数以友元函数的类型使用类模板本身，这使得在类模板上表示函数变得更容易，就好像它们在命名空间中可见一样：

```
template<typename T>
class Creator {
  friend void feed(Creator<T>) { //every T instantiates a different function ::feed()
    ...
  }
};

int main()
{
  Creator<void> one;
  feed(one);                // instantiates ::feed(Creator<void>)
  Creator<double> two;
  feed(two);                // instantiates ::feed(Creator<double>)
}
```

示例中，`Creator`的每个实例都会生成一个不同的函数。请注意，即使这些函数是作为模板实例化的一部分生成的，这些函数本身也只是普通的函数，并不是模板的实例。然而，这种情况被视为模板实体(templated entities, 参考P181节12.1)，它们仅在被使用到时才会被定义。同时也注意到由于这些函数的函数体在类定义域内被定义，所以它们是内联(inline)的。因此，两个不同编译单元生成该相同的函数并不会引起错误。可以参考P220节13.2.2和P497节21.2.1来了解该话题的更多信息。

12.5.3 友元模板

通常在声明一个函数或类模板的实例为友元时，我们可以严格地表示哪个实体才是友元。尽管如此，有些时候对某种模板的所有实例都设为友元也是很有用的。这就需要使用友元模板(friend template)。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;
};
```

与普通的友元声明一样，当名称是不含限定符的函数名时友元模板也可以是一个定义，函数名后不接尖括号子句。

友元模板只能定义主模板和主模板的成员。主模板的偏特化和显式特化也都会被自动的视作友元。

12.6 后记

C++模板的通用语法和概念自80年代起就相对保持稳定。类模板和函数模板是最开始时构成模板的两部分。类型模板和非类型模板也是。

然而，受C++标准库的需求所驱动，后来新增了一些重大的特性。成员模板可能是这些添加中最基础的。搞笑的是，只有成员函数模板被正式票入C++标准。成员类模板在社论监督下才成为标准的一部分。

友元模板，默认模板实参，模板模板参数是在C++98标准化后出现的。声明模板模板参数的能力有时被成为高阶泛型(higher-order genericity)。引入它们原本是为了支持一个已有的C++标准库的分配器(allocator)模型，然而这个分配器模型后来被另一个不依赖模板模板参数的模型取代了。后来，模板模板参数距离被踢出语言标准越来越近，因为它们的规范并不完整，直到非常晚才出现的1998标准化进程。最终大多数委员会成员投票表示保留它们，而它们的规范也完整制定。

别名模板是在2011标准引入的。别名模板为需要`typedef templates`特性而简化书写模板的场合提供了相同的服务，它仅仅是一个现有类模板的另一种拼写。规范(N2258)（作者是 Gabriel Dos Reis 和 Bjarne Stroustrup;）把它加入到标准。 Mat Marcus 也贡献了这一提议的一些早期草稿。Gaby 还为C++14(N3651)的可变模板提议处理了很多细节内容。本来，该提议仅仅想要支持`constexpr`变量，但是这一限制在标准制定阶段被解除了。

可变模板由C++11标准库和Boost库所驱动，C++模板库此前一直使用一种递进型高级技巧来支持接受任意数量模板参数。 Doug Gregor, Jaakko J¨arvi, Gary Powell, Jens Maurer, 和 Jason Merrill 为标准化提供了初始的规范(N2242)。当这一规范问世时，Doug 还开发了这一特性的原始实现代码(在GNU的GCC中)，为标准库使用这一特性提供了极大助力。

折叠表达式是 Andrew Sutton 和 Richard Smith 的作品：它们通过N4191文献引入到C++17。

第13章 模板中的名称

在大多数编程语言中，名称是一个基本的概念。借助名称，程序员可以引用前面已经构造完毕的实体。当C++编译器遇到一个名称时，它会查找该名称，来确认它所引用的是哪个实体。从实现者角度来看，就名称而言，C++在这方面相当棘手。譬如C++语句`x * y;`，如果`x`和`y`都是变量的名称，那么这一语句就是一个乘法表达式，但是如果`x`是类型的名称，该语句就是在声明一个`y`变量实体，其类型是`x`类型实体的指针。

这一小小的例子阐释了C++（类C）是一门上下文敏感型语言(context-sensitive language)：对于C++的一个结构，我们无法脱离上下文来理解它。而这又与模板有什么关联呢？事实上，模板也是一种结构，它也必须处理多种上下文相关信息：（1）模板出现的上下文；（2）模板实例化的上下文；（3）用于模板实例化的模板实参的上下文。因此，在C++中，“名称”需要被小心的处理这一事实就不足为奇了。

13.1 名称的分类

C++对名称的分类有多种多样的方式。为了理解名称的众多术语，我们提供了表13.1和表13.2，对这些分类进行了描述。幸运的是，熟悉下面两种主要的命名概念，就可以深入理解大多数的C++模板话题：

1. 如果名称的作用域由域操作符（`::`）或是成员访问操作符(`.`或`->`)显式指定，我们就称该名称为限定名称(qualified name)。例如，`this->count`是一个限定名称，

但是`count`本身则不是（尽管字面上`count`实际上指代的也是一个类成员）。

2. 如果一个名称以某种方式依赖于模板参数，那么该名称就是一个依赖型名称(dependent name)。例如，当`T`是一个模板参数时，`std::vector<T>::iterator`是一个依赖型名称；但如果`T`是一个已知的类型别名时（比如`using T = int`），那么`std::vector<T>::iterator`就不是一个依赖型名称。

分类	
标识符(Identifier)	仅由不间断的字母、下划线和数字组成的名称。不能以数字开头，并且某
操作符函数id(Operator-function-id)	关键字`operator`后紧跟的操作符符号。例如，`operator new`和`operator`
类型转换函数id(Conversion-function-id)	用于表示用户自定义的隐式转换运算符，例如`operator int &`（也可以以
字面操作符id(Literal-operator-id)	用于表示一个用户定义的字面操作符——例如，`operator ""_km`，可以用
模板id(Template-id)	由闭合的尖括号子句内的模板实参构成的模板名称。例如，`List<T, int`
非限定id(Unqualified-id)	广义的标识符。可以是上述的任何一种（标识符、操作符函数id、类型转
限定id(Qualified-id)	对非限定id使用类、枚举、命名空间的名称做限定或是仅仅使用全局作用
限定名称(Qualified-name)	标准中并没有定义这一概念，但是我们一般用它来表示经过限定查找的名
非限定名称(Unqualified-name)	除限定名称以外的非限定id。这并非标准中的概念，我们只是用它来表示
名称(Name)	一个限定或非限定名称

表13.1 名称分类（第一部分）

分类	
依赖型名称(Dependent name)	通过某种方式依赖于模板参数的名称。一般来说，显式包含模板参数的
非依赖型名称(Nondependent name)	不满足上述描述中“依赖型名称”的名称即是一个非依赖型名称

表13.2 名称分类（第二部分）

通读该表会更加熟悉C++模板话题中的这些概念，但是也没有必要去记住每个定义的精准含义。什么时候需要，就什么时候通过索引来查阅。

13.2 名称查找

在C++中，名称查找有非常多的小细节，但是我们这里只关注一些主要概念。只有在下面两种情景中我们才有必要确认名称查找的细节：（1）按直觉处理会犯错的一般案例（2）C++标准给出的错误案例。

限定名称在限定结构所隐含的作用域中进行查找。如果该作用域是一个类，则还可以向上搜索基类。然而，在查找限定名称时不会考虑封闭作用域(enclosing scopes)。下面的例子阐释了这一基本原则：

```
int x;

class B {
  public:
    int i;
};

class D: public B {
};

void f(D* pd)
{
  pd->i = 3;        // finds B::i
  D::x = 2;                 // ERROR: does not find ::x in the enclosing scope
}
```

非限定名称的查找则恰恰相反，它可以(由内到外)在所有外围类中逐层地进行查找（但在某个类内部定义的成员函数定义中，它会优先查找该类和基类的作用域，然后才查找外围类的作用域），这种查找方式被称为一般性查找(ordinary lookup)。下面是一个用于理解一般性查找的基本示例：

```
extern int count;                                   // #1
int lookup_example(int count)         // #2
{
        if (count < 0) {
                int count = 1;                          // #3
                lookup_example(count);         // unqualified count refers to #3
        }
        return count + ::count;                 // the first (unqualified) count refers to #2;
                                                                        // the second (qualified) count re
}
```

对于非限定名称的查找，最近的一种变化是除了普通的查找之外，它们可能还会经历参数依赖查找(argument-dependent lookup, ADL)。在展开叙述ADL之前，我们先用前面的`max()`模板来说明这一机制的动机：

```
template<typename T>
T max(T a, T b)
{
  return b < a ? a : b;
}
```

假设我们现在需要让”在另一个命名空间所定义的某个类型“来使用这一模板：

```
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()`模板不认识`BigMath`命名空间，一般性查找无法找到类型`BigNumber`适用的`operator <`。如果没有特殊规则的话，这种限制大大降低了C++命名空间中模板的应用性。而ADL正是这个“特殊规则”，也正是解决这种限制的关键之处。

13.2.1 ADL

ADL主要适用于在函数调用或运算符调用中看起来像非成员函数名称的非限定名称。如果一般性查找找到了以下信息，ADL就不会发生：

- 成员函数名称
- 变量名称
- 类型名称
- 块作用域函数声明名称

如果把被调用函数的名称用圆括号括起来，ADL也会被禁用。

否则，如果名称后的括号里面有实参表达式列表，则ADL将会查找这些实参“关联”的命名空间和类。对这些关联的命名空间(associated namespace)和关联类(associated class)的精准定义会在后文给出，但在直觉上它们可以被认为是与给定类型相关联的所有命名空间和类。例如，如果某一类型是一个`class X`的指针，那么关联的类和命名空间就包括`X`和`X`所属的任何命名空间或

类。

对给定类型，关联命名空间和关联类所组成的集合的精准定义，我们可以通过下列规则来确定：

- 对内置类型，该集合为空集。
- 对指针和数组类型，该集合就是其底层所引用类型的关联类和关联命名空间。
- 对枚举类型，关联命名空间就是枚举声明所在的命名空间。
- 对类成员，关联类就是其所在的类。
- 对类类型（包括联合体类型），关联类集合包括其类型本身、它的外围类型、所有的直接或间接基类。关联命名空间集合是每个关联类所在的命名空间。如果类是一个类模板实例，那么类模板实参的类型以及声明模板的模板实参所在的类和命名空间也将包含在内。
- 对函数类型，关联命名空间和类的集合包含每一个参数类型和返回值所关联的命名空间和类。
- 对指向类`X`的成员指针类型，关联的命名空间和类包括`X`以及成员类型本身的关联。（如果是指向成员函数的类型，那么参数和返回类型也算数。）
  至此，ADL会在所有的关联命名空间和关联类中依次地查找，就好像依次地直接使用这些命名空间进行限定一样。唯一的例外情况是：它会忽略`using`指示符(using-directives)。下面的例子说明了这一点：

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\n";
  }
}

void f(int)
{
  std::cout << "::f(int) called\n";
}

int main()
{
  ::f(N::e1);        // qualified function name: no ADL
  f(N::e1);               // ordinary lookup finds ::f() and ADL finds N::f(),
                          // the latter is preferred
}
```

我们可以看出：在这个例子中，当执行ADL时，命名空间`N`中的`using-directive`被忽略了。因此，在这个`main()`函数内部的调用中，`X::f()`甚至永远都无法作为一个候选者。

13.2.2 友元声明的ADL

在类中友元函数的声明可以是该友元函数的首次声明。在此场景中，对于包含这个友元函数的类，假设它所属的最近的命名空间作用域（可能是全局作用域）为作用域A，我们就可以认为该友元函数是在作用域A中声明的。然而，这样的友元声明在该作用域中并不是直接可见的。考虑下面的例子：

```
template<typename T>
class C {
  ...
  friend void f();
  friend void f(C<T> const&);
  ...
};

void g(C<int>* p) {
  f();                // is f() visible here?
  f(*p);        // is f(C<int> const&) visible here?
}
```

如果友元声明在封闭命名空间中可见，那么实例化一个类模板可能会使一些普通函数的声明也变为可见的（比如f()）。这可能会产生一些令人惊讶的行为：函数调用`f()`会导致编译错误，除非类C的实例化在程序更早的地方进行过！

另一方面，仅仅通过友元函数声明（并定义）一个函数非常有用（参考P497节21.2.1依赖于这种行为的某个技巧）。当友元函数所在的类属于ADL查找过程的关联类时，该友元函数就是可见的。

再次考虑上面的例子，`f()`没有关联类或关联命名空间，因为它并没有任何参数：在这个例子中该调用是无效的。然而，`f(*p)`调用有着关联类`C<int>`（因为它是`*p`的类型），并且全局命名空间也是关联的（因为这是`*p`的类型声明所在的命名空间）。因此，只要我们在调用之前完全实例化`class C<int>`，就可以找到这一第二个友元函数。为了确保这一点，我们可以假设：对于涉及在关联类中友元查找的调用，实际上会导致该（关联）类被实例化（如果还没有实例化的话）。

ADL查找友元声明和定义的能力有时候也被称为友元名称注入(friend name injection)。然而，这一术语有些误导性，因为它是一个前标准C++特性的名称，该特性会确实地把友元声明的名称“注入”到封闭作用域中，使得它们在一般性名称查找中可见。对上例来说，这就意味着两个调用都有效。本章的后续内容会详述友元名称注入的历史。

13.2.3 注入的类名称

类的名称会被注入到类本身的作用域中，因此在该作用域中作为非限定名称可访问。（然而，它作为限定名称不可访问，因为这种符号表示用于表示构造函数。）例如下面的例子：

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() << '\n';
}
```

成员函数`C::f()`返回了`class`类型`C`的尺寸，而`::f()`则返回了`int`变量`C`的尺寸。

类模板也可以有注入的类名称。然而，相比较一般的注入的类名称来说，二者有些区别：它的后面可以紧跟模板实参（在此场景，它们也被称为注入的类模板名称）。但是，如果后面没有紧跟模板实参，那么它们代表的就是用参数来代表实参的类（例如，对于偏特化，还可以用特化实参代表对应的模板实参）。下述代码解释了这一情景：

```
template<template<typename> class TT> class X {
};

template<typename T> class C {
  C* a;                         // OK: same as "C<T>* a;"
  C<void>& b;         // OK
  X<C> c;                 // OK: C without a template argument list denotes the template C
  X<::C> d;                 // OK: ::C is not the injected class name and therefore always
                                //                 denotes the template
};
```

注意看非限定名称是如何引用注入的名称的，并且，如果名称后没有跟随模板实参列表的话，它们不会被认作模板名称。为了补偿，我们可以在模板名称前强制使用`::`限定符。

可变模板的注入的类名称还有一个额外的特点：如果注入的类名称是通过使用可变模板的模板参数直接组成的，那么注入的类名称也将包含尚未展开的模板参数包（参考P201节12.4.1了解包展开的细节）。因此，在为可变参数模板形成注入的类名时，与模板参数包对应的模板参数是一个模板参数包的展开，其模式就是那个模板参数包：

```
template<int I, typename... T> class V {
  V* a;                         // OK: same as "V<I, T...>* a;"
  V<0, void> b;         // OK
};
```

13.2.4 当前实例

类或类模板的注入的类名称实际上是类型定义的一个别名。对非模板类来说，这一特性是显然的，因为类本身就是其作用域内其名称的唯一类型。然而，在类模板或是类模板嵌套的类中，每个模板实例都会产生一个不同的类型。在这一上下文中，该特性就非常有趣了，因为这意味着注入的类名称指向类模板的相同实例而非类模板的其他实例（对类模板的嵌套类来说也一样）。

在类模板中，类或类模板范围内的注入的类名称或是其他等价于注入的类名称的类型（包括类型别名的声明）都被称为一个当前实例(current instantiation)。依赖于模板参数但并不指代一个当前实例的类型被称为一个未知的特化(unknown specialization)，它可以从相同的类模板或某些全然不同的类模板实例化。下面的例子阐释了这一区别：

```
template<typename T> class Node {
  using Type = T;
  Node* next;                          // Node refers to a current instantiation
  Node<Type>* previous;        // Node<Type> refers to a current instantiation
  Node<T*>* parent;                // Node<T*> refers to an unknown specialization
};
```

在嵌套类和类模板中辨别某个类型是否指代一个当前实例往往扑朔迷离。类和类模板范围内的注入的类名称（或者等价于它们的类型）是一个当前实例，而其他嵌套的类或类模板中的名称则不是一个当前实例：

```
template<typename T> class C {
  using Type = T;

  struct I {
    C* c;                          // C refers to a current instantiation
        C<Type>* c2;         // C<Type> refers to a current instantiation
        I* i;                          // I refers to a current instantiation
  };

  struct J {
    C* c;                          // C refers to a current Instantiation
        C<Type>* c2;         // C<Type> refers to a current instantiation
        I* i;                          // I refers to an unknown specialization,
                                       // because I does not enclose
        J* j;                          // J refers to a current instantiation
  };
};
```

当类型指代的是一个当前实例时，实例化的类的内容可以保证是由当前定义的类模板或嵌套类所实例化的。当解析模板（下一节的主题）时这对名称查找有着意义，但与此同时它也引导了另一种方案，一种更像游戏的方式来决定类模板中的类型`X`的定义指代的是一个当前实例还是一个未知的特化：如果另一个程序员可以写出一个显式特化（在第16章描述细节）使得`X`指向该特化体，那么`X`就指代一个未知的特化。例如，考虑上例上下文中类型`C<int>::J`的实例：我们知道`C<T>::J`的定义用于实例化特定的具体类型（也就是我们所实例化的类型）。此外，由于显式特化无法在不同时特化范围内所有模板或成员的情况下，特化某一个模板或模板成员，`C<int>`会在类定义范围内被实例化。因此，`J`和`C<int>`的引用在J所在范围内均指代一个

当前实例。而另一方面，我们可以写出一个`C<int>::I`的显式特化，如下文：

```
template<> struct C<int>::I {
  // definition of the specialization
};
```

这里，`C<int>::I`的特化提供了一个与`C<T>::J`所可见的定义完全不同的定义，因此定义`C<T>::J`中定义的`I`指代的是一个未知的特化。

13.3 模板解析

大多数程序设计语言的编译都包含两个最基本的步骤——token化（也称作扫描或词法解析）和（语法）解析。Token化过程会按字符顺序读取源代码，然后生成一个token序列。例如，当看到字符序列`int* p = 0;`时，扫描器会为关键字`int`、符号/操作符`*`、标识符`p`、符号/操作符`=`、整型字面量`0`和符号/操作符`;`生成token。

解析器会通过将token或先前发现的模式(pattern)递归地归约为更高级别的结构，从而在token序列中找到已知的模式。例如，token `0`是一个合法的表达式，`*`后跟随的标识符`p`是一个合法的声明器(declarator)，该声明器后接`=`再接表达式`0`是一个合法的初始化声明器(init-declarator)。最后，关键字`int`是一个已知的类型名称，并且当后面跟着初始化声明器`*p = 0`时，就归约为`p`的初始化声明。

13.3.1 非模板中的上下文相关性

如你所闻与所愿，token化过程比解析要简单得多。幸运的是，解析已经是一门理论发展得相当成熟的学科，使用这一理论对于理解大多数语言的解析都不算困难。然而，这一理论在上下文无关语言中表现最佳，而我们已经知道了C++是一门上下文敏感语言。为此，C++编译器会使用一张符号表来把标记器(tokenizer)和解析器(parser)结合起来：当解析到声明时，会把它灌入到符号表中。当标记器找到一个标识符时，它会进行查找，如果找到的是一个类型的话，就对生成的token进行注解。

例如，如果C++编译器看到`x*`，标记器会查找`x`。如果找到了一个类型，解析器就会看到：

```
identifier, type, x
symbol, *
```

并得出一个结论：这是要开始声明了。然而，如果没有找到类型`x`，那么解析器会从标记器处接收这样的信息：

```
identifier, nontype, x
symbol, *
```

此时该结构按合法性只能被解析成一个乘法表达式。这些原则的细节要依赖于编译器的具体实现策略，但大同小异。

另一个上下文敏感的案例在下面的表达式中阐释：

```
X<1>(0)
```

如果`X`是类模板的名称，那么前面的表达式就是将整型`0`强制类型转换到类型`X<1>`（由该模板产生的）。如果`X`不是一个模板，那么上面的表达式等价于

```
(X<1)>0
```

换句话说，`X`会和1比较，然后根据结果——`true`或`false`，隐式转换成`1`或`0`——再与0进行比较。尽管这样的代码非常罕见，但它也是一个合法的C++代码（也是合法的C代码）。C++解析器会查找`<`前出现的名称，只有在该名称是一个模板名称时，才会把`<`看成是左尖括号；否则，`<`就被视为普通的小于操作符。

令人遗憾的是，这类上下文敏感性都是由于选择尖括号来界定模板参数列表所造成的。下面是另一个案例：

```
template<bool B>
class Invert {
  public:
    static bool const result = !B;
};

void g()
{
  bool test = Invert<(1>0)>::result;        // parentheses required!
}
```

如果`Invert<(1>0)>`的小括号被省略，大于等于符号就会被误认为是模板参数列表的闭合尖括号。这会使得代码无效，因为编译器会把它读作`((Invert<1>))0>::result`。

尖括号带给标记器的问题还不止这些。例如，在语句：

```
List<List<int>> a;
                        //^-- no space between right angle brackets
```

两个`>`字符组合成了一个右移操作符`>>`，因此它们不再被视为两个独立的符号。这要归因于所谓的maximum munch tokenization原则：C++实现体必须让一个token能捕获尽可能多的连续字符。

如P28节2.2所提及，在C++11之后，C++标准特别指出了这一情景——嵌套的模板id紧跟着右移符号`>>`——解析器会将模板id紧邻的右移符号视为两个独立的右尖括号`>`。有趣的是，此变更项会默默地更改某些程序（公认的程序）的含义。考虑下面的例子：

names/anglebrackethack.cpp

```
#include <iostream>

template<int I> struct X {
  static int const c = 2;
};

template<> struct X<0> {
  typedef int c;
};

template<typename T> struct Y {
  static int const c = 3;
};

static int const c = 4;

int main()
{
  std::cout << (Y<X<1> >::c >::c>::c) << ' ';
  std::cout << (Y<X< 1>>::c >::c>::c) << '\n';
}
```

这是一个合法的C++98程序，输出`0 3`。它也是合法的C++11程序，但是尖括号变革使得括号内的两个语句是等价的，最终输出`0 0`。

由于`<:`是字符`[`的两字符替代(某些传统键盘是不支持的)，还存在一个类似的问题，考虑下面的案例：

```
template<typename T> struct G {};
struct S;
G<::S> gs;               // valid since C++11, but an error before that
```

C++11之前，最后一行代码等价于`G[:S>gs;`，这显然是不合法的。另一个词法hack技术被引入来解决该问题：当编译器看到字符序列`<::`没有紧跟着`:`或`>`时，前导`<:`字符对不再被视为`[`等价的两字符符号。这一两字符hack技术使得以前合法的程序变得不再合法：

```
#define F(X) X ## :

int a[] = {1, 2, 3}, i = 1;
int n = a F(<::)i];               // valid in C++98/C++03, but not in C++11
```

想要理解它，就要注意到两字符hack应用于预处理符号，对预处理器来说变成了截然不同的符号，它们在宏展开完成前被确定。因此，C++98/C++03会无条件转换`<:`到`[`，因而定义展开成`int n = a[ :: i];`，显然这是没问题的。而C++11则不会进行字符转换，因为在宏展开前，序列`<::`没有跟随`:`或`>`而是`)`时，两字符转译不会进行，因此连接操作符`##`会试图连接`::`和`:`成为一个新的预处理符号`:::`，但显然这是一个不合法的符号。这一标准会导致UB行为(undefined behavior)，也就意味着放任编译器自由处理。某些编译器会诊断出这一问题，但也有些不会：它们会保持两个预处理符号分离，然后导致语法错误，因为对`n`的定义最终展开成如下语句：

```
int n = a < :: : i];
```

13.3.2 类型的依赖型名称

模板中名称的问题在于它们无法始终被充分地分类。具体来讲，一个模板无法引用另一个模板的名称，因为其他模板的内容可能因显式特化而使原本的名称失效。下面的例子阐释了这一概念：

```
template<typename T>
class Trap {
  public:
    enum { x };        // #1 x is not a type here
};

template<typename T>
class Victim {
  public:
    int y;
        void poof() {
          Trap<T>::x * y;        // #2 declaration or multiplication?
        }
};

template<>
class Trap<void> {        // evil specialization!
  public:
    using x = int;        // #3 x is a type here
};

boid boom(Victim<void>& bomb)
{
  bomb.poof();
}
```

编译器解析行#2时，它必须确定这是一个声明语句还是一个乘法表达式。这一决定取决于依赖型限定名称`Trap<T>::x`是否是一个类型名称。编译器此时会尝试在模板`Trap`中查找，并且发现根据行#1，`Trap<T>::x`并不是一个类型，从而让我们相信行#2是一个乘法表达式。然而，在后面`T`取`void`的特化中，我们改写了（泛型的）`Trap<T>::X`，让它变成了一个类型，这完全违背了前面的源码。在特化场景中，`Trap<T>::x`实际上是一个`int`类型。

本例中，类型`Trap<T>`是一个依赖型类型，因为类型取决于模板参数`T`。此外，`Trap<T>`指代的是一个未知的特化（在P223节13.2.4中描述），这意味着编译器无法安全的在模板中查找以判定名称`Trap<T>::x`是否是一个类型。当`::`前的类型指代的是一个当前实例时——例如，`Victim<T>::y`——编译器才可以在模板定义中查找，这是因为它已经确定不会有其他的特化来干预。因此，如果`::`前的类型指代的是一个当前实例，那么模板中限定名称的查找与非依赖类型的限定名称查找表现得非常相似。

然而，如上例所阐释，未知特化中的名称查找始终是一个问题。C++语言通过下面的规定来解决这个问题：通常来说，一个依赖型限定名称并不代表一个类型，除非在名字的前面加上了一个关键字`typename`前缀。对于类型而言，如果不加上`typename`前缀，那么在替换模板实参后，就不会被看成是一个类型名称，从而导致程序是无效的，你的C++编译器还会抱怨在实例化过程中出现了错误。另一方面，我们应该知道`typename`的这种用法和前面用于表示模板类型参数的用法是不同的：在这里你不能使用关键字`class`来等价替换`typename`。

总之，当类型名称具有以下性质时，就应该在名称前面添加`typename`前缀：

1. 名称是限定的，且本身没有后跟`::`组成一个更为限定的名称。
2. 名称不是详细类型说明符（elaborated-type-specifier）的一部分（例如，以`class`，`struct`，`union`，或`enum`起始的关键字）。
3. 名称不在指定基类继承的列表中，也不在引入构造函数的成员初始化列表中。
4. 名称依赖于模板参数。
5. 名称是某个未知特化的成员，这意味着由限定器命名的类型指代一个未知的特化。

此外，除非至少满足前两个条件，才能使用`typename`前缀。下面的错误案例为此予以解释：

```
template<typename T>                                    // 1
struct S : typename X<T>::Base {        // 2
  S() : typename X<T>::Base(typename X<T>::Base(0)) {        // 3 4
  }

  typename X<T> f() {                                    // 5
    typename X<T>::C * p;                          // declaration of pointer p        // 6
        X<T>::D *q;
  }

  typename X<int>::C *s;                          // 7

  using Type = T;
  using OtherType = typename S<T>::Type;        // 8
}
```

每个出现的`typename`，不管正确与否，都被标了号。第一个`typename`表示一个模板参数。前面的规则没有应用于此。第二个和第三个`typename`由于上述规则的第三条而被禁止。这两个上下文中，基类的名称不能用`typename`引导。然而，第四个`typename`是必不可少的，因为这里基类的名称既不是位于初始化列表，也不是位于派生类的继承列表，而是为了基于实参`0`构造一个

临时`X<T>::Base`表达式（也可以是某种强制类型转换）。第5个`typename`同样不合法，因为它后面的名称`X<T>`并不是一个限定名称。对于第6个`typename`，如果期望声明一个指针，那么这个`typename`是必不可少的。下一行省略了关键字`typename`，因此也就被编译器解释为一个乘法表达式。第7个`typename`是可选（可有可无）的，因为它符合前面的两条规则，但不符合后面的两条规则。第8个`typename`也是可选的，因为它指代的是一个当前实例的成员（不满足最后一条规则）。

最后一条判断`typename`前缀是否需要的规则有时候难以评估，因为它取决于判断类型所指代的是一个当前实例还是一个未知特化这一事实。在这种场景中，最简单安全的方法就是直接添加`typename`关键字来表明限定名称是一个类型。`typename`关键字，尽管它是可选的，也会提供一个意图上的说明。

13.3.3 模板的依赖型名称

当一个模板的名称是依赖型名称时，我们将会遇到类似上一小节的问题。通常而言，C++编译器会把模板名称后面的`<`看作模板实参列表的开始，否则的话`<`就会被视为小于操作符。与类型名称一样，除非程序员使用关键字`template`提供了额外的信息，编译器是不会把依赖性名称视作模板的：

```
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();               // inhibit virtual call
        }
        void case2 (typename Shell<T>::template In<N>::template Deep<N>& p) {
          p.template Deep<N>::f();               // inhibit virtual call
        }
};
```

这个多少有些复杂的例子展示了所有可以限定名称的操作符是如何需要在操作符前添加关键字`template`的。明确来讲，如果限定符号前面的名称或表达式的类型需要依赖于某个模板参数，并且紧跟在限定符后面的是一个模板id(template-id)（换句话说，就是指一个后面带有闭合尖括号实参列表的模板名称），那么就应该使用关键字`template`。例如，在下面的表达式中：

```
p.template Deep<N>::f()
```

`p`的类型依赖于模板参数`T`。因此，C++编译器并不会查找`Deep`来判断它是否是一个模板，并且我们必须显式地通过插入`template`前缀来指定`Deep`是一个模板名称。如果没有该前缀，`p.Deep<N>::f()`就会被解析成`((p.Deep)<N)>f()`。还要注意在一个限定名称内部，可能需要多次使用关键字`template`，因为限定符本身可能还会受限于外部的依赖型名称（可以从上例的case1和case2的参数中看到）。

如果例子中的关键字`template`被省略了，那么左尖括号和右尖括号会被解析为小于和大于操作符。由于使用了`typename`关键字，我们可以安全的添加`template`前缀来指明后面的名称是一个模板id(template-id)，即使`template`前缀并不是严格需要的。

13.3.4 Using声明中的依赖型名称

Using声明会从两个地方引入名称：命名空间和类。命名空间这一部分与本文不相干，因为并没有诸如命名空间模板(namespace templates)这样的东西。而对于类来说，using声明只能把基类的名称引入到继承类。这样的using声明看起来像继承类访问基类的“符号链接”或是“快捷方式”，就好像是继承类自身声明的成员一样。千言万语不及一个小小示例，我们用一个非模板示例来阐述：

```
class BX {
  public:
    void f(int);
        void f(char const*);
        void g();
};

class DX : private BX {
  public:
    using BX::f;
};
```

类`DX`使用using声明将名称`f`从基类`BX`中引入。本例中，该名称关联了两个不同的声明，但我们这里强调的是一种名称机制，而不是关注该名称是否是一个单一的声明。此外，using声明可以让以前不能访问的成员变成可访问的。从示例代码中可以看到，基类和它的成员对派生类`DX`是私有的（因为私有继承），只有函数`BX::f`是个例外，它因被using引入到了`DX`的公有接口而能够访问。

现在，你可能已经发现了当使用using声明从依赖类中引入名称的问题所在。尽管我们知道该名称，我们还是不知道这个名称到底是一个类型，还是一个模板，或是其他什么东西：

```
template<typename T>
class BXT {
  public:
    using Mystery = T;
        template<typename U>
        struct Magic;
};

template<typename T>
class DXTT : private BXT<T> {
  public:
    using typename BXT<T>::Mystery;
        Mystery* p;                // would be a syntax error without the earlier typename
};
```

如果我们想要使用using声明引入依赖型名称来指定类型时，我们必须显式地插入`typename`关键字前缀。奇怪的是，在这样的名称是一个模板时，C++标准并没有提供一个类似的机制来标记。下面的代码片段揭示了这个问题：

```
template<typename T>
class DXTM : private BXT<T> {
  public:
    using BXT<T>::template Magic;        // ERROR: not standard
        Magic<T>* plink;                                  // SYNTAX ERROR: Magic is not a known temp
};
```

标准委员会至今没有考虑这个议题。然而，C++11别名模板提供了一个迂回解决方案：

```
template<typename T>
class DXTM : private BXT<T> {
  public:
    template<typename U>
          using Magic = typename BXT<T>::template Magic<T>;        // Alias template
          Magic<T>* plink;
};
```

这可能看起来有点笨，但是对类模板的场景它满足了需求。不幸的是，函数模板的情景目前还没

有解决（可以说非常少见）。

13.3.5 ADL与显式模板实参

考虑下面的示例：

```
namespace N {
  class X {
    ...
  };

  template<int I> void select(X*);
}

void g(N::X* xp)
{
  select<3>(xp);        // ERROR: no ADL!
}
```

我们期望在调用`select<3>(xp)`中模板`select()`可以通过ADL来找到。然而事与愿违，这是因为编译器直到确定`<3>`是一个模板实参列表之前，它都无法确定`xp`是一个函数调用参数。更进一步，编译器直到确定`select()`是一个模板之前它都无法确定`<3>`是一个模板实参列表。由于这个先有鸡还是先有蛋的问题无法被解决，表达式就会被解析成一个毫无意义的表达式：`(select<3)>(xp)`。

这个例子可能会给你一种ADL对模板id(template-id)没有发挥作用的假象，但事实并非如此。我们可以通过在调用前引入`select`的函数模板声明来解决这个问题：

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

尽管对于调用`select<3>(xp)`来说这没有任何意义，但这一函数模板的存在确保了`select<3>`会被解析成一个模板id(template-id)。ADL就可以顺势找到函数模板`N::select`，然后成功调用。

13.3.6 依赖型表达式

与名称相似，表达式本身也可以依赖于模板参数。依赖于模板参数的表达式彼此之间有着较大差异——例如，选择一个不同的重载函数或是产生一个不同的类型或常量。不依赖于模板参数的表

达式，其所有的实例提供相同的行为。

依赖于模板参数的表达式多种多样。最常见的是类型依赖表达式(type-dependent expression)，表达式的类型本身可以因实例的变化而不同——例如，函数参数类型为模板参数的表达式：

```
template<typename T> void typeDependent1(T x)
{
  x;        // the expression type-dependent, because the type of x can vary
}
```

具有类型依赖子表达式的表达式，通常来说，其本身也是类型依赖的——例如，使用实参`x`调用函数`f()`：

```
template<typename T> void typeDependent2(T x)
{
  f(x);                // the expression is type-dependent, because x is type-dependent
}
```

这里请注意`f(x)`的类型可能因实例的变化而有所不同，因为`f`本身依赖于参数类型，而该参数类型又依赖于模板，因此，两阶段查找（在P249节14.3.1讨论）会在不同的实例中找到完全不同的函数名`f`。

并非所有涉及模板参数的表达式都是类型依赖的。例如，涉及模板参数的某个表达式可以在不同的实例中产生不同的常量`values`。这种表达式被称为值依赖表达式(value-dependent expression)，最简单的一种就是指向非依赖类型的非类型模板参数。例如：

```
template<int N> void valueDependent1()
{
  N;        // the expression is value-dependent but not type-dependent;
            // because N has a fixed type but a varying constant type
}
```

正如类型依赖表达式那样，如果一个表达式是由其他值依赖表达式所组成的，那么通常来说它也是一个值依赖表达式，因此`N + N`或是`f(N)`都是值依赖表达式。

有趣的是，一些操作符，诸如`sizeof`，拥有一个已知的结果类型，因此它们可以把一个类型依赖操作数转换成一个值依赖表达式（也就不是类型依赖的）。例如：

```
template<typename T> void valueDependent2(T x)
{
  sizeof(x);        // the expression is value-dependent but not type-dependent
}
```

不论输入什么，`sizeof`操作符总是产生一个类型为`std::size_t`的值，因此`sizeof`表达式永远不会是类型依赖的，即使——在本例中——它的子表达式是类型依赖的。然而，计算得到的结果常量值会因不同的实例而有所变化，因此`sizeof(x)`是一个值依赖表达式。

那么如果我们对一个值依赖表达式使用`sizeof`操作符会发生什么呢？

```
template<typename T> void maybeDependent(T const& x)
{
  sizeof(sizeof(x))
}
```

这里，正如前文所述，内层的`sizeof`表达式是值依赖的。然而，外层的`sizeof`表达式永远会计算`std::size_t`的尺寸，因此它的类型和常量值对所有的模板实例来说都是一致的，尽管最内层的表达式(`x`)是类型依赖的。涉及模板参数的任何表达式都是一个实例依赖表达式(instantiation-dependent expression)，即使它的类型和常量值对所有有效的实例来说都是不变的。然而，实例依赖表达式可能在实例化过程中变得无效。例如，使用不完整类类型去实例化`maybeDependent()`会触发一个错误，因为`sizeof()`不能应用于这种类型。

类型、值和实例依赖性可以被认为是一系列表达式更为广义的分类。任何类型依赖表达式也可以被认为是值依赖的，因为因不同实例而变化的表达式类型自然而然地会有不同的常量值。类似地，类型或值因不同实例而变化的表达式在某种意义上依赖于模板参数，因此类型依赖表达式和值依赖表达式都是实例依赖的。它们的关系如图13.1所示。

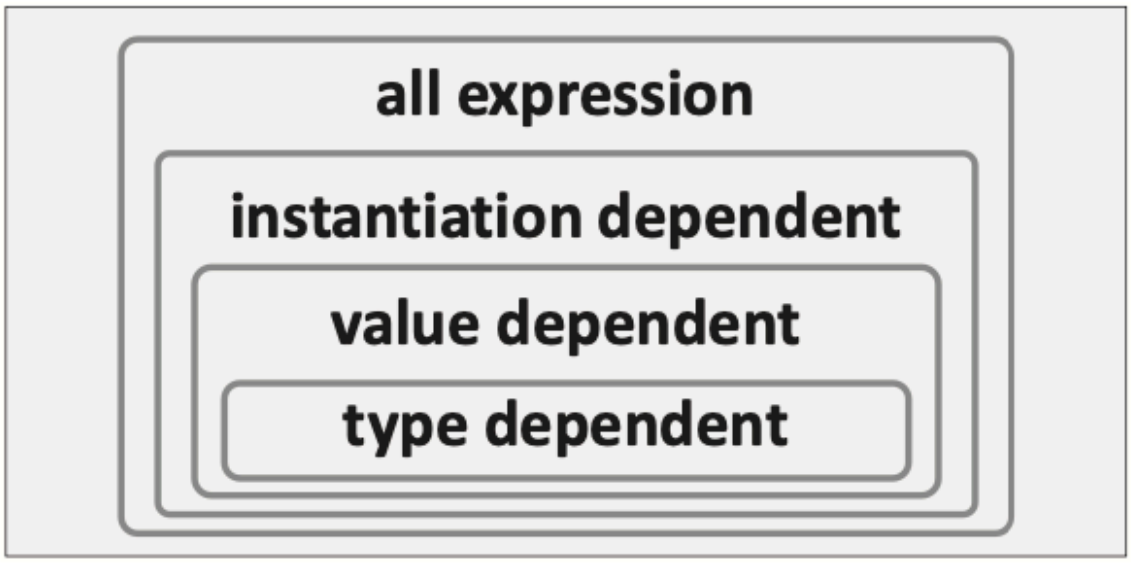

图13.1 类型、值、实例依赖表达式的关系

因为上下文都是由内（类型依赖表达式）向外推进，更多模板行为会在模板解析时确定，因而无法因不同实例而变化。例如，对于调用`f(x)`：如果`x`是类型依赖的，那么`f`就是依赖型名称，它会面临两阶段查找（P249节14.3.1）；而当`x`是值依赖而并非类型依赖时，`f`就不是一个依赖型名称，它的名称在模板被解析的那一刻就已经完全被确定了。

13.3.7 编译错误

当所有的模板实例都将产生错误时，C++编译器被允许（但没被要求）在解析模板时可以忽略该错误。让我们扩展一下前文`f(x)`这一例子：

```
void f() { }

template<int x> void nondependentCall()
{
  f(x);                // x is value-dependent, so f() is nondependent;
                       // this call will never succeed
}
```

函数调用`f()`在每个（模板）实例中都会产生一个错误，因为`f`是一个非依赖型名称，而唯一可见的`f`却接受零个参数，而非一个。C++编译器可以在解析该模板时或者等到模板进行第一个实例化时产生一个错误：常用的编译器对该案例的表现并不一致。你可以构造相似的例子：表达式是实例依赖的，但并不是值依赖的。

```
template<int N> void instantiationDependentBound()
{
  constexpr int x = sizeof(N);
  constexpr int y = sizeof(N) + 1;
  int array[x - y];               // array will have a negative size in all instantiations
}
```

13.4 派生和类模板

类模板可以继承或被继承。对多数情况来说，模板和非模板的继承没有显著区别。然而，当从一个依赖型名称基类派生一个类模板时，二者有着微妙而又重要的区别。让我们先来看一个非依赖型基类的例子。

13.4.1 非依赖型基类

在类模板中，非依赖型基类是指拥有一个完整类型而无需模板实参即可确定的基类。换句话说，这种基类使用的是非依赖型名称。例如：

```
template<typename X>
class Base {
  public:
    int basefield;
        using T = int;
};

class D1 : public Base<Base<void>> {        // not a template case really
  public:
    void f() { basefield = 3; }                        // usual access to inherited member
};

template<typename T>
class D2 : public Base<double> {                // nondependent base
  public:
    void f() { basefield = 7; }                        // usual access to inherited member
        T strange;                                                  // T is Base<double>::T,
};
```

非依赖型模板基类的表现和普通的非模板基类没什么差别，但是有一个细微的区别（可能有些惊奇）：当非限定名称在模板继承中被找到时，非依赖型基类中会优先考虑该名称而后才轮到模板参数列表。这意味着在上面的例子中，成员`strange`始终是对应`Base<double>::T`（也就是`int`）类型。因此，下面的函数就是非法的C++代码：

```
void g(D2<int*>& d2, int* p)
{
  d2.strange = p;                // ERROR: type dismatch!
}
```

这可能有点反直觉，它需要编写者意识到继承的非依赖型模板基类名称的存在——即使这种派生是间接的或者名称是私有的情况。事实上，在参数化实体的(如上面的`D2`)作用域中，可能往往倾向于先查找模板参数，只可惜事与愿违。

13.4.2 依赖型基类

在前面的例子中，基类都是完全确定的，它并不依赖于模板参数。这意味着一旦模板定义是可见的，那么C++编译器就可以在那些基类中查找非依赖型名称。有一种替代品（一种不被C++标准所允许的）会延迟这类名称的查找，直到模板被实例化。这种替代品的缺陷在于：它同时也将诸如漏写了某个符号而导致的错误信息延迟到了模板实例化的时候才产生。因此，C++标准规定模板中出现的非依赖型名称，会在出现的第一时间进行查找。有了这一概念后，我们看看下面的例子：

```
template<typename T>
class DD : public Base<T> {                // dependent base
  public:
    void f() { basefield = 0; }        // #1 problem
};

template<>        // explicit specialization
class Base<bool>{
  public:
    enum { basefield = 42 };        // #2 tricky!
};

void g(DD<bool>& d)
{
  d.f();                                                    // #3 oops?
}
```

在`#1`处我们发现了一个非依赖型名称`basefield`：它必须即刻进行查找。假设我们在模板`Base`中找到了它，并且把它与该`int`型成员进行绑定。然而，紧随其后，我们在一个`Base`的显式特化中覆盖了这一泛型定义。于是，这一特化改变了刚刚确定好的`basefield`的意义！因此，当我们在`#3`处实例化`DD::f`的定义时，就会发现我们在`#1`处过早地绑定了非依赖型名称，然而，在`DD<bool>`中并没有可供修改的`basefield`（`#2`处特化的枚举值），因此这里本应该抛出一个错误信息才对。

为了解决这个问题，C++标准声明：非依赖型名称不会在依赖型基类中进行查找（但仍然是在出现的第一时间查找）。因此，符合C++标准的编译器会在`#1`处给出一个诊断信息。为了修正这段代码，只需要将`basefield`这个名称变为依赖型名称即可，这是因为依赖型名称只在实例化的时候才被查找，而此时此刻基类的实例就已经确定了。比如说，在`#3`处，编译器就会知道`DD<bool>`的基类是`Base<bool>`，并且这个基类是程序员自己显式特化的一个实例。本例中，我们推荐的方式就是让名称转成依赖型：

```
template<typename T>
class DD1 : public Base<T> {
  public:
    void f() { this->basefield = 0; }        // lookup delayed
};
```

还可以使用限定名称来引入依赖性：

```
template<typename T>
class DD2 : public Base<T> {
  public:
    void f() { Base<T>::basefield = 0; }
};
```

如果使用后一个解决方法，我们要格外小心，因为如果（原来的）非限定的非依赖型名称是被用于虚函数调用的话，那么这种引入依赖性的限定将会禁止虚函数调用，从而也会改变程序的含义。因此，当遇到第2种解决方案不适用的情况，我们可以使用方案1：

```
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 would not be valid syntax
          this->zero();                        // D<T>::zero() would inhibit virtuality
          one(e);                                // one is dependent because its argument is depen
        }
};
```

注意看我们这里是如何用`D<T>::E`来取代`B<T>::E`的。对本例来说，二者皆可。然而在多重继承场景中，我们可能无法知道哪一个基类提供了这一想要的成员（在这种情况下，使用派生类进行资格审查），也有可能多个基类同时声明了相同的名称（在这种情况下，我们不得不使用特定的基类名称来消除歧义）。

还要注意，调用`one(e)`中的名称`one`是依赖于模板参数的，这仅仅是因为它的显式调用实参是依赖型名称。然而，如果我们是把这种“依赖于模板参数的类型”隐式地用作缺省实参，那么就不符合上述情况，因为编译器要到决定查找的时候，才会确认缺省实参是否是依赖型的，这同样是

一个先有鸡还是先有蛋的问题。为了避免细微的差池，我们更趋向于在允许使用`this->`前缀的地方都使用`this->`前缀，这同样适用于非模板代码。

如果你觉着反复的限定会影响代码美观，你可以在派生类中只引入依赖型基类中的名称一次：

```
// Variation 3:
template<typename T>
class DD3 : public Base<T> {
  public:
    using Base<T>::basefield;                // #1 dependent name now in scope
        void f() { basefield = 0; }                 // #2 fine
};
```

在`#2`处的查找是成功的，它会找到`#1`处的声明。然而，`using`声明直到实例化时才被确定，这也达成了我们的目的。这种机制也有些约束。例如，如果是多重继承，程序员必须严格地选择包含期望的成员的那一个基类。

在当前实例中查找限定名称时，C++标准规定了首先要在当前实例中查找，然后才是所有的非依赖型基类，这与非限定名称的查找类似。如果找到了某个名称，限定名称就会指代当前实例的某个成员，因而也就不是一个依赖型名称。如果找不到这样的名称，并且类还有其他的依赖型基类，那么限定名称就会指代一个未知的特化实例的某个成员。例如：

```
class NonDep {
  public:
    using Type = int;
};

template<typename T>
class Dep {
  public:
    using OtherType = T;
};

template<typename T>
class DepBase : public NonDep, public Dep<T> {
  public:
    void f() {
          typename DepBase<T>::Type t;        // finds NonDep::Type;
                                                                    // typename keyword is o
          typename DepBase<T>::OtherType* ot;        // finds nothing; DepBase<T>::OtherType
                                                                    // is a member of an unk
        }
};
```

13.5 后记

首个解析模板定义的编译器是由Taligent公司在20世纪90年代中期开发的。在这之前（即使在这之后的一段时间），大多数编译器都把模板看成是一系列要在（解析过程后面的）实例化时刻才被处理的标记。因此，除了处理诸如查找模板定义结束位置等少许操作以外，都不会进行其他的解析。在撰写本书的此刻，微软的Visual C++编译器仍然以这种方式工作。Edison Design Group's(EDG's)编译器前端使用了一种混合技术——在内部模板被视为一串注释的token，但是会执行“通用解析”来校验语法（EDG's的产品模仿大多数其他编译器；特别的，它相当程度地模仿了微软编译器的行为）。

Bill Gibbons是Taligent公司在C++委员会的代表，他极力主张让模板可以无二义性地进行解析。然而，直到惠普公司完成第一个完整的编译器之后，Taligent公司的努力才真正产品化，也才有了一个真正编译模板的C++编译器。和其他具有竞争性优点的产品一样，这个C++编译器很快就由于高质量的诊断信息而得到业界的认可。模板的诊断信息不会总是延迟到实例化时刻的事实也要归功于这个编译器。

在模板的早期开发过程中，Tom Pennello（Metaware公司的一位著名解析专家）就意识到了尖括号所带来的一些问题。Stroustrup也对这个话题进行了讨论[StroustrupDnE]，而且认为人们更喜欢阅读尖括号，而不是圆括号。然而，除了尖括号和圆括号，还存在其他的一些可能性：Pennello在1991年的C++标准大会（在达拉斯举办）上特别地提议使用大括号，例如（`List{::X}`）。然而，在那时，问题的扩展程度是非常有限的，因为嵌入在其他模板内部的模板（也称为成员模板）还是不合法的，因此也就不会涉及到P230节13.3.3的问题。最后，委员会拒绝了这个取代尖括号的提议。

在P237节13.4.2中描述的非依赖型名称和依赖型基类的名称查找规则是在1993年C++标准中引入的。早在1994年，Bjarne Stroustrup的[StroustrupDnE]首次公开描述了这一规则。然而直到1997年惠普才把这一规则引入其C++编译器，自那以后出现了大量的派生自依赖型基类的类模板代码。事实上，当惠普工程师开始测试该实现时，他们发现大部分以特殊方式使用模板的代码都无法再通过编译了。特别地，STL的所有实现都在成百上千个地方打破了这一规则。考虑到客户的转换成本，对于那些“假定非依赖型名称可以在依赖型基类中进行查找的”代码，惠普软化了相关的诊断信息。例如，对于位于类模板作用域的非依赖型名称，如果利用标准原则不能找到该名称，C++就会在依赖型基类中进行查找。如果仍然找不到，才会给出一个错误而编译失败。然而，如果在依赖型基类中找到了该名称，那么就会给出一个警告，对该名称进行标记并且看成是依赖型名称，然后在实例化的时候试图再次查找。

在查找过程中，“非依赖型基类中的名称会隐藏相同名称的模板参数（P236节13.4.1）”这一规则显然是一个疏忽，但是修改这一规则的建议还没有被C++标准委员会所认可。最好的办法就是避免使用非依赖型基类中的名称作为模板参数名称。命名转换对这一类问题都是一个好的解决方式。

友元注入一度被认为是有害的，因为它会使得程序的合法性与实例出现的顺序紧密相关。Bill Gibbons（此时他还在Taligent公司开发编译器）就是解决这一问题的最大支持者，因为消除实例顺序依赖性激活了一个新的、有趣的C++开发环境（传闻Taligent正在做）。然而，Barton-Nackman trick(P497节21.2.1)需要一种友元注入的形式，正是这种特殊的技术使它以基于ADL的当前（弱化）形式保留在语言中。

Andrew Koenig首次为操作符函数提出了ADL查找（这就是为什么有时候ADL也被称为Koenig查找），动机主要是考虑美观性：“用外围命名空间显式地限定操作符名称”看起来很拖沓（例如，对于a+b，我们需要这样编写：`N::operator+(a,b)`），而为每个操作符都书写using声明又会让代码看起来非常笨重。因此，才决定操作符可以在参数关联的命名空间中查找。ADL随后被扩展到普通函数名称的查找，得以容纳有限种类的友元名称注入，并为模板及其实例支持两阶段查找模型（第14章）。泛化的ADL规则也被称作扩展的Koenig查找。

尖括号hack的规格说明由David Vandevoorde通过其文献N1757在C++11中引入。他还通过解决核心议题1104的方式增添了有向图hack，以解决美国对C++ 11标准草案的审核要求。

第14章 实例化

模板实例化就是从泛型模板定义中生成类型、函数和变量的过程。C++模板实例化的概念非常基础，但有时又错综复杂。这一复杂性的其中一个底层原因在于：模板生成的实体定义不再局限于源代码单一的位置。模板本身的位置、模板使用的位置以及模板实参定义的位置均在实体的含义中扮演着重要角色。

本章我们会讲解如何组织源代码来正确使用模板。此外，我们调查了最流行的C++编译器处理模板实例所使用的各种各样的方法。尽管所有的方法都应该语义等价，但理解编译器实例化策略的基本原则是大有裨益的。在构建实际软件时，每种机制都带有一些小怪癖，相反地，每种机制都影响了标准C++的最终规范。

14.1 On-Demand实例化

当C++编译器遇到模板特化的使用时，它会用需要的实参来替换模板参数来生成特化体。这一过程是自动完成的，不需要客户端代码来引导（或者不需要模板定义来引导）。这一”on-demand“实例化特性使得C++与其他早期的编译型语言的类似功能大相径庭（如Ada或Eiffel，其中的一些语言需要显式地实例化引导，另外一些使用运行时分发机制来避免编译期实例化过程）。有时这也被称作”隐式(implicit)实例化“或者”自动(automatic)实例化“。

On-demand实例化意味着编译器常常需要访问模板完整的定义（换句话说，不只是声明）以及某些成员。考虑下面这一段精简的源码文件：

```
template<typename T> class C;        // #1 declaration only
C<int>* p = 0;                                          // #2 fine: definition of C<int> not needed

template<typename T>
class C{
  public:
    void f();                                         // #3 member declaration
};                                                                  // #4 class template definition

void g(C<int>& c)                                  // #5 use class template declaration only
{
  c.f();                                                  // #6 use class template definition;
}                                                                   // will need definition of C::f()
                                                                   // in this translation unit


template<typename T>
void C<T>::f()                                          // required definition due to #6
{
}
```

在源码的`#1`处，仅仅只有模板的声明，并没有定义（这种声明有时也被称作前置声明）。与普通类的情况一样，如果你声明的是一个指向某种类型的指针或引用（`#2`处的声明），那么在声明的作用域中，你并不需要看到该类模板的定义。例如，声明函数`g`的参数类型并不需要模板`C`的完整定义。然而，一旦某个组件需要知道模板特化体的大小或是访问了该特化体的成员，那么就需要看到完整的类模板定义。这就解释了为什么`#6`处必须看到类模板的定义。若非如此，编译器无法确认该成员是否存在、是否可访问（非private或protected）。更进一步，成员函数定义也是需要的，因为`#6`处的调用需要确认`C<int>::f()`是否存在。

另一个需要类模板实例化的表达式如下所示，这里需要`C<void>`实例化是因为它需要该类型的尺寸：

```
C<void>* p = new C<void>;
```

本例中，需要实例化来保证编译器可以确定`C<void>`的尺寸，该new表达式需要去确认要分配多少存储空间。你可能会发现，对这一模板来说，替换模板参数`T`的实参`X`的类型无论是什么，都不会影响模板的尺寸，毕竟`C<X>`是一个空类（没有成员变量或虚函数）。然而，编译器并不会通过分析模板定义来避免实例化（所有编译器实际上都会进行实例化）。此外，对于上例来

说，为了确定`C<void>`是否有可访问的默认构造器并确保`C<void>`没有成员`operator new`或`operator delete`操作符函数，实例化也同样是必要的。

在源代码中是否需要访问类模板的成员并不总是那么直观。例如，C++重载决议规则要求：如果候选函数的参数是类类型，那么该类类型就必须是可见的：

```
template<typename T>
class C {
  public:
    C(int);                 // a constructor that can be called with a single parameter
};                                  // may be used for implicit conversions

void candidate(C<double>);          // #1
void candidate(int) { }                  // #2

int main()
{
  candidate(42);                // both previous function declarations can be called
}
```

调用`candidate(42)`会采用`#2`处的声明。然而，在`#1`处的声明也会被实例化来检查对于这个调用来说它是否是可用的候选者（这个例子中，由于模板的单实参构造器可以把42隐式转换成一个类型为`C<double>`的右值）。请注意，如果模板不经实例化也可以找到调用函数（合适的候选），编译器还是被允许（但不强制）执行该实例化（上例的情景中，由于有精准匹配的候选者，隐式转换的那个不会被选择）。另外，令我们的惊讶的是：`C<double>`的实例化可能还会触发一个错误。

14.2 延迟实例化

到目前为止所展示的这些例子，和使用非模板类相比并没有本质上的区别。譬如，非模板类的许多用法会要求类类型的完整性（参考P154节10.3.1）。而对模板来说，编译器会用类模板定义来生成完整的定义。

现在有一个相关问题：模板实例化的程度如何？可以给出这样的模糊答案：会实例化到它实际需要的程度。换句话说，编译器在实例化模板时应该是“懒惰”的。让我们来细究“懒惰”在这里的真正意义。

14.2.1 部分实例化和完整实例化

如我们之前所见，编译器有时不需要替换类或函数模板的完整定义。例如：

```
template<typename T> T f(T p) { return 2*p; }
decltype(f(2)) x = 2;
```

本例中，`decltype(f(2))`所指示的类型并不需要函数模板`f()`的完整实例化。编译器因此只被允许替换`f()`的声明，而不是替换整个“身体”。这有时被称为部分实例化(partial instantiation)。

同样，如果引用类模板的实例而不需要将该实例作为完整类型，则编译器不应对该类模板实例执行完整的实例化。考虑下面的例子：

```
template<typename T> class Q {
  using Type = typename T::Type;
};

Q<int>* p = 0;                // OK: the body of Q<int> is not substituted
```

在这里，`Q<int>`完整的实例化会触发一个错误，因为在`T`是`int`类型时，`T::Type`并没有意义。但是因为本例并不需要完整的`Q<int>`，所以不会执行完整实例化，代码也是OK的(尽管可疑)。

变量模板也有“完整”和“部分”实例化的区别。下面的例子用以阐释：

```
template<typename T> T v = T::default_value();
decltype(v<int>) s;               // OK: initializer of v<int> not instantiated
```

`v<int>`的完整实例化会引起错误，但是如果只是需要变量模板实例的类型的话，是不需要进行完整实例化的。

有意思的是，别名模板没有这一区别：不存在两种方法来替换它们。

在C++中，当谈到“模板实例化”而没有说特定的完整或部分实例化时，往往意味着前者。也就是说，默认情况我们指的都是完整实例化。

14.2.2 实例化组件

当类模板隐式（完整）实例化时，其所有成员的声明也都会进行实例化，但是对应的定义却并不会实例化（即，成员是部分实例化的）。对此有一些特殊情况：首先，如果类模板包含一个匿名的联合体(union)，该联合体的成员的定义也会实例化；另一个特殊的情况出现在虚成员函数场景中，它们的定义作为模板实例化的结果，可能会也可能不会进行实例化。实际上，许多实现都会实例化该定义，因为“实现虚函数调用机制的内部结构”需要虚函数有一个链接实体存在。

实例化模板时，默认函数调用实参被单独考虑。具体来说，除非调用该函数（或成员函数）时确实使用了默认实参，否则它们不会被实例化。反之，如果调用该函数时显式地指定了实参去覆盖这一默认实参，那么默认实参就不会被实例化。

类似的，除非有必要，异常规范和默认成员初始化器也不会被实例化。

让我们用一些例子来阐释这些原则：

details/lazy1.hpp

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

template<int N>
class Danger {
  int arr[N];                // OK here, although would fail for N<=0
};

template<typename T, int N>
class Tricky {
  public:
    void noBodyHere(Safe<T> = 3);        // OK until usage of default value results in an error
        void inclass() {
          Danger<N> noBoomYet;                        // OK until inclass() is used with N<=0
        }
        struct Nested {
          Danger<N> pfew;                               // OK until Nested is used with N<=0
        };
        union {                                                          // due anonymous union:
          Danger<N> anonymous;                        // OK until Tricky is instantiated with N<=0
          int aligh;
        };
        void unsafe(T (*p)[N]);                         // OK until Tricky is instantiated with N<=
        void error(){
          Danger<-1> boom;                              // always ERROR (which not all compilers
        }
};
```

标准C ++编译器将审查这些模板定义以检查语法和常规语义约束。这样做时，当检查涉及模板参数的约束时，它将“假设最佳”。举个例子，`Danger::arr`的成员参数`N`可能是零或负数（非法的），但是编译器会假定不会出现这种情况。`inclass()`，`struct Nested`，匿名联合体的定义因而都没有问题。

出于同样的原因，只要`N`还是一个未被替换的模板参数时，成员`unsafe(T (*p)[N])`的声明也不是问题。

`noBodyHere()`的默认实参规格声明(`=3`)看起来很诡异，因为模板`Safe<>`并不能以一个整型数来初始化，但是编译器会假定：对于`Safe<T>`泛型定义来说，它实际上并不需要默认实参；或者

是`Safe<T>`的特化体会引入使用一个整型数来初始化的能力（见第16章）。然而，成员函数`error()`的定义必定会引起一个错误，即使模板尚未实例化，这是因为`Danger<-1>`的使用需要一个完整的类`Danger<-1>`的定义，也就会生成该类并尝试去定义一个负数尺寸的数组。有趣的是，虽然标准明确指出此段代码无效，但它还是允许编译器在未实际使用模板实例时不去诊断这个错误。也就是说，只要`Tricky<T,N>::error()`对任何具体的`T`和`N`类型都未被使用，那么编译器就不用抛出这个错误。例如，GCC和Visual C++在撰写此书时都不会抛出这一错误。

让我们来分析一下，在增加下面的一行定义语句时，会发生什么：

```
Tricky<int, -1> inst;
```

这将引起编译器（完整）实例化`Tricky<int, -1>`，在模板`Tricky<>`定义中替换`T`为`int`，`N`为`-1`。并非所有的成员定义都是必要的，但是默认构造器和析构器（本例中都是隐式声明的）一定会被调用到，因此它们的定义必须是可用的（在我们的例子中，它们都会隐式生成）。如上所述，`Tricky<int, -1>`的成员会部分实例化（即，它们的声明会被替换）：这一过程可能会引起错误。例如，`unsafe(T (*p)[N])`的声明创建了一个负数尺寸的数组类型，这就是一个错误。类似的，`anonymous`成员现在也会抛出一个错误，因为并不能生成`Danger<-1>`类型。另一方面，成员`inclass()`和`struct Nested`的定义现在还不会被实例化，因此对完整类型`Danger<-1>`的需求并不会产生错误（它们都包含了一个无效的数组定义）。

如上所述，当实例化一个模板时，对于虚函数实际上是需要提供定义的。否则，就会遇到链接错误。例如：
details/lazy2.cpp

```
template<typename T>
class VirtualClass {
  public:
    virtual ~VirtualClass() {}
        virtual T vmem();               // Likely ERROR if instantiated without definition
};

int main()
{
  VirtualClass<int> inst;
}
```

最后，`operator->`值得留意。考虑：

```
template<typename T>
class C{
  public:
    T operator-> ();
};
```

通常来说，`operator->`必须返回一个指针类型或是另一个应用了`operator->`的类类型。`C<int>`的完全体会触发一个错误，因为它声明了一个`int`返回类型的`operator->`。然而，因为某些常见的类模板定义实现了这种（返回类型为`T`或者`T*`）定义，所以语言规则更加灵活。于是，只有在重载决议规则确实选择了用户自定义的`operator->`时，才要求该自定义`operator->`只能返回一个应用了其他（例如，内建的）`operator->`的类型。这甚至对模板之外的代码也同样生效（尽管这种无约束行为(relaxed behavior)在那些上下文中用处不大）。因此，这里的声明不会触发错误，尽管`int`会替代该返回类型。

14.3 C++实例化模型

模板实例化就是从对应的模板实体通过合适地模板参数替换来得到一个常规的类型、函数或是变量的过程。这可能听起来直截了当，但实际上需要遵循非常多的细节。

14.3.1 两阶段查找

在第13章中，我们曾看到依赖型名称无法在解析模板时被找到。取而代之的是，它们会在实例化的时刻再次进行查找。非依赖型名称则会在更早的阶段被查找，因此当模板第一次看到它的时候，就可以诊断出许多错误。这就引出了“两阶段查找”的概念。第一阶段查找发生在解析模板的时候，而第二阶段查找发生在模板实例化的时候：

1. 在第一阶段，当解析模板时，非依赖型名称会并用一般查找规则和ADL规则（如果可行的话）。非限定依赖型名称（诸如函数调用中的函数名称，它们之所以是依赖型名称，是因为它们具有依赖型实参）会使用普通查找规则，但是这一查找结果并不会作为最终结果，而是要等到第二阶段的另一个查找过程完成（也就是模板实例化的时候）。
2. 在第二阶段，此时的模板实例化被称作POI(point of instantiation)，依赖型限定名称会在此时被查找（对选定的实例用模板实参替换模板参数），而且还会对非限定依赖型名称进行额外的ADL查找（它们曾在第一阶段进行过普通查找）。

对非限定依赖型名称，首次的普通查找（并不是终态）被用来判断该名称是否是一个模板。考虑

下面的例子：

```
namespace N {
  template<typename> void g() {}
  enum E { e };
}

template<typename> void f() {}

template<typename T> void h(T p) {
  f<int>(p);                // #1
  g<int>(p);                // #2 ERROR
}

int main() {
  h(N::e);                        // calls template h with T = N::E
}
```

在`#1`行，当看到跟着一个`<`的名称`f`时，编译器就需要判断`<`到底是一个尖括号还是一个小于号。这取决于`f`是否是一个已知的模板名称。在本例中，普通查找会找到`f`的声明，它确实是一个模板，因此这里会以尖括号来成功解析。

而在`#2`行，这里会产生一个错误，这是因为普通查找并不能找到模板`g`，因此，`<`就被认为是一个小于号操作符，对于我们的例子来说这就是个语法错误。如果想让该解析通过，那么在用`T` = `N::E`实例化`h`的时候最终得用ADL找到一个模板`N::g`（尽管`N`是与`E`关联的命名空间），但是只有先成功解析`h`的泛型定义，这才能行得通。

> 译者注：示例中的g和f都漏写了一个模板参数作为函数参数。不过无所谓，C++20已经允许这么写了。

14.3.2 POI

如上所述，C++编译器会在模板客户端代码的某些位置访问模板实体的声明或者定义。当某些代码结构引用了模板特化，而且为了生成该特化需要实例化相应的模板定义时，就会在源代码中产生一个POI。POI是源代码中的一个点，在这里会插入已被替换的模板。例如：

```
class MyInt {
  public:
    MyInt(int i);
};

MyInt operator - (MyInt const&);

bool operator > (MyInt const&, MyInt const&);

using Int = MyInt;

template<typename T>
void f(T i)
{
  if(i > 0) {
    g(-i);
  }
}

// #1
void g(Int)
{
  // #2
  f<Int>(42);        // point of call
  // #3
}
// #4
```

C++编译器看到`f<Int>(42)`时，它知道模板`f`需要用`MyInt`替换`T`来实例化：这就产生了一个POI。`#2`和`#3`与该调用点紧邻，但是它们都不适合做POI，因为C++不允许我们在这里插入`::f<Int>(Int)`的定义。此外，`#1`和`#4`两处的本质区别在于，在`#4`处，函数`g(Int)`是可见的，因此模板依赖的调用`g(-i)`可以在`#4`处被解析。然而，如果我们假定`#1`是POI的话，那么调用`g(-i)`将不能被解析，因为`g(Int)`在`#1`处是不可见的。幸运的是，对于函数模板特化的引用，C++把它的POI定义，置于紧跟在“包含这个引用的定义或声明所在的最近的命名空间作用域”之后。在我们的例子中，这个位置就是`#4`。

你可能好奇为什么这个例子引入了类型`MyInt`而不是用`int`基础类型。这是因为，在POI执行的第二次查找（指`g(-i)`）仅仅使用了ADL，而基础类型`int`并没有关联的命名空间，因此，如果

使用`int`类型，就不会发生ADL查找，也就不能找到函数`g`。所以，如果你用下面的类型别名声明语句：

```
using Int = int;
```

代码将无法通过编译。下面的例子有着类似的问题：

```
template<typename T>
void f1(T x)
{
  g1(x);        // #1
}

void g1(int)
{
}

int main()
{
  f1(7);        // ERROR: g1 not found!
}
// #2 POI for f1<int>(int)
```

`f1(7)`调用对`f1<int>(int)`产生了一个POI紧随其后（在位置`#2`）。在这一实例中，关键点在于函数`g1`的查找。当首次遇到模板定义`f1`时，它会注意到非限定名称`g1`是一个依赖型名称，因为它作为一个函数名称，有着依赖型实参（实参`x`的类型取决于模板参数`T`）。因此，`g1`会在`#1`处使用一般查找规则，然而，在`#1`处找不到任何的`g1`。在`#2`处，即POI处，函数名称被再一次查找（在关联的命名空间和类中），但是唯一的实参类型是一个`int`型，它根本没有关联的命名空间和类。因此，`g1`永远都无法被找到，尽管在这里（POI处）其实用一般查找就可以找到`g1`。

变量模板POI的处理与函数模板相似。而对于类模板特化来说，情况则不太一样，如下例所示：

```
template<typename T>
class S {
  public:
    T m;
};

// #1
unsigned long h()
{
  // #2
  return (unsigned long)sizeof(S<int>);
  // #3
}
// #4
```

老规矩，`#2`和`#3`都不能作为POI，这两个位置不能进行命名空间作用域类`S<int>`的定义（模板是不能出现在函数作用域内部的）。假如我们可以遵循函数模板实例的规则，POI将会出现在位置`#4`处，然而，这样一来，表达式`sizeof(S<int>)`是无效的，这是因为`S<int>`的尺寸直到`#4`之后才能被确定。因此，生成的类模板实例的引用被紧邻地定义在包含该引用的声明或定义的命名空间作用域之前。在我们的例子中，这个位置就是`#1`。

模板在实例化时，可能还需要进行额外的实例化。请看下方这一简例：

```
template<typename T>
class S {
  public:
    using I = int;
};

// #1
template<typename T>
void f()
{
  S<char>::I var1 = 41;
  typename S<T>::I var2 = 42;
}

int main()
{
  f<double>();
}
// #2: #2a, #2b
```

根据我们之前的讨论，`f<double>()`的POI位于`#2`处。函数模板`f()`还引用了类模板特化`S<char>`，它的POI位于`#1`处。与此同时它还引用了`S<T>`，但是因为这仍然是一个依赖型名称，我们此时此刻无法真正完成实例化。然而，如果我们在`#2`处实例化`f<double>()`，我们会注意到同时也需要实例化`S<double>`的定义。这种副(secondary)POI（或这叫过渡的POI）的定义位置会有些差异。对于函数模板，副POI与主(primary)POI严格一致；而对于类模板，副POI会（在最近的命名空间作用域中）先于主POI。在我们的例子中，这意味着`f<double>()`会被放在`#2b`处，而前面紧邻的`#2a`处会是`S<double>`的副POI。请注意`S<char>`与`S<double>`POI的差别。

编译单元通常会包含相同实例的多个POI。对类模板实例，在每个编译单元中，只有首个POI会被保留，后续的那些都会被忽略（它们不会被真正视为POI）。对函数模板实例和变量模板实例，所有的POI都会被保留。无论是哪一种情形，ODR原则都会要求：对保留的任何一个POI处所出现的同种实例化体，都必须是等价的；但是C++编译器既不需要保证这一原则，也不需要诊断是否违反这一原则。这就允许C++编译器随便选择一个POI来完成真正的实例化，而不必担心其他POI会产生不同的实例化结果。

事实上，大多数编译器会对大部分函数模板的实例化，直到编译单元末尾处再延迟进行。某些实例化不能被拖延，这其中包括：判定某个推导的返回类型所需的实例化时（参考P296节15.10.1

和P303节15.10.4）、函数是`constexpr`且必须产生一个常量结果时。有些编译器在首次使用内联函数时会进行实例化，以便立即内联调用。这实际上将对应模板特化的POI转移到了翻译单元的末尾，这是C++标准所允许的替代POI的方式。

14.3.3 包含式模型

当遇到POI时，对应模板的定义必须是可访问的。对类特化来说，这意味着类模板定义必须在编译单元中被更早地看见。而对函数模板和变量模板（以及类模板的成员函数和静态数据成员）的POI来说，也同样需要。典型的模板定义被简单的通过`#include`语句引入到编译单元，尽管是非类型模板也一样。这种模板定义的源码模型被称为包含式模型，它目前是当下C++标准所支持的模板的唯一自动源码模型。

尽管包含式模型鼓励程序员将所有模板定义都放在头文件中，以便它们可以满足可能出现的任何POI，但显式地使用“显式实例化声明(explicit instantiation declarations)”和“显式实例化定义(explicit instantiation definitions)”（P260节14.5）来管理实例化也是可行的。从逻辑上讲，这样做并不是一件容易的事，大多数时候程序员会更喜欢依靠自动的实例化机制。用自动方案实现的一个挑战是要解决跨不同编译单元为函数模板或变量模板（或类模板实例的相同成员函数或静态数据成员）的特化体实现完全相同的POI。我们随后会讨论这个问题的解法。

14.4 几种实现方案

本节我们来回顾一下支持包含式模型的几种C++实现。所有的这些实现都依赖于两个基础组件：编译器和链接器。编译器将源代码编译成目标文件，它们包含机器码和符号注释（跨引用其他目标文件和库）。链接器通过组合这些目标文件解决它们包含的跨引用符号来创建可执行程序或库文件。在下面的内容中，即使完全有可能（但不流行）以其他方式实现C ++（例如，你可以假想出一个C++解释器），我们也将采用这种模型。

当类模板特化在多个编译单元中被使用时，编译器会为每个编译单元重复实例化过程。这几乎不会造成什么问题，因为类定义并没有直接产出低层级代码。它们仅仅由C++实现体在内部使用，用来审查并解释各种其他表达式和声明。在这方面，类定义的多个实例化体与类定义的多个包含（在不同编译单元中通常通过头文件包含）没有实质性区别。

然而，如果你实例化一个（非内联）函数模板，情况就有些不同了。如果你想提供某个普通的非内联函数的多个定义，那么就会违反ODR原则。例如，假设你编译和链接下面这两个文件：

```
// ==== a.cpp:
int main()
{
}

// ==== b.cpp:
int main()
{
}
```

C++编译器会对每个模块进行单独编译，此时没有什么问题，因为在每个编译单元内它们都合法。然而，如果你想把它们链接在一起，你的链接器很可能会抗议：不允许出现重复的定义。

反之，我们考虑模板的场合：

```
// ==== t.hpp:
// common header (inclusion model)
template<typename T>
class S {
  public:
    void f();
};

template<typename T>
void S::f()                // member definition
{
}

void helper(S<int>*);
// ==== a.cpp:
#include "t.hpp"
void helper(S<int>* s)
{
  s->f();        // #1 first point of instantiation of S::f
}

// ==== b.cpp:
#include "t.hpp"
int main()
{
  S<int> s;
  helper(&s);
  s.f();        // #2 second point of instantiation of S::f
}
```

如果链接器处理类模板实例化的成员函数与处理普通函数或成员函数的方式一致，那么编译器就需要保证它只会生成一份代码，要么在`#1`处生成，要么在`#2`处生成（两处POI的位置）。为了达成这一目标，编译器需要在每个编译单元中都携带其他的编译单元的信息，而这对于C++编译器来说在引入模板之前是从未有过的要求。接下来，我们讨论C++实现中已投入使用的三种类型解决方案。

请注意，模板实例化产生的所有的链接实体都有同样的问题：实例化的函数模板和成员函数模板，以及实例化的静态数据成员和实例化的变量模板。

14.4.1 贪婪实例化

首个实现贪婪实例化的C++编译器是由Borland公司开发的。现如今，这一技术已经在各种C++系统上被广泛使用了。

贪婪实例化假定链接器会意识到特定的实体（尤其是可链接的模板实例化体），它们大多在多个目标文件和库中重复出现。编译器会以一种特殊的方式标记这些实体。当链接器发现了多个实例时，它会保留单个并丢弃掉所有其他的。这就是贪婪实例化的处理方法。

理论上，贪婪实例化有一些严重的缺陷：

- 编译器会在生成和优化N个实例化体时浪费时间，它只需要保持一个即可。
- 链接器一般不会检查两个实例化体是否相同，因为一个模板特化的多个实例生成的代码可能有些合法的无关紧要的差别。这些微小的差异不应该导致链接器失败（编译器在实例化的时刻可能因状态不同而产生细微的差异）。然而，这常常会导致链接器无法注意到更多的充足的差异，比如某一个实例化是使用严格的浮点数运算法则，而另一个确是松弛的、高性能的浮点数运算法则。
- 所有的目标文件加起来可能大小远远超过理应生成的替换体总和，这是因为相同的代码会被复制多次。

实践当中，这些缺陷看起来并没有引起重大问题。也许这是因为贪婪实例化在一个重要方面与竞品相比非常有优势：源对象之间的原始依赖被保留了下来。尤其是，每个编译单元只产生一个目标文件，并且在相应的源文件（它包含了实例化后的定义）中，每个目标文件都包含针对所有可链接定义的代码，而且这些代码是已经经过编译的代码。另一个重要的收益在于所有的函数模板实例都是内联的候选对象而无需求助于昂贵的“链接时”优化机制（实际上，函数模板实例常常是短小的函数而从内联中得益）。其他的实例化机制则需要专门对函数模板进行内联（判定）处理，以确保它们是否可以内联展开。然而，贪婪实例化甚至允许非内联函数模板也进行内联展开。

最后值得一提的是，允许可链接实体重复定义的链接器机制，通常还被用于处理重复的“内联函数溢出”(spilled inlined functions)和“虚函数调度表“(virtual function dispatch tables)。如果这一机制不可用，那么替代方法通常是以内部链接来发出这些项，但这会增大代码的体积。内联函数必须具有单一地址的要求使得以符合标准的方式去实现这一替代方法变得相当困难。

14.4.2 查询实例化

上世纪90年代中期，一家名为Sun Microsystems的公司发行了它们的C++编译器的新版实现（版本4.0），这一版本以一种新的有趣的方式解决了实例化问题，我们称之为查询实例化

(queried instantiation)。查询实例化在概念上明显更简单、优雅，而且按照时间顺序，它也是我们在此回顾的实例化方案中最新的一种。在这一方案中，程序中参与的所有编译单元会汇集一个共享的数据库。该数据库可以追溯哪些特化体被实例化了，并且可以找到其所依赖的源代码。生成的特化体本身会把信息存储在数据库中。当可链接实体遇到一个POI时，会进入下面的处理流程：

1. 尚无可用的特化体：这种情况会进行实例化，特化的结果会保存到数据库中。
2. 特化体虽可用但超期了，因为自它生成以来源代码发生了变化。这种情况同样会进行实例化，新的特化结果会覆盖数据库中旧的那一个。
3. 数据库中有最新可用的特化体。这种情况什么都不用做。

尽管从概念上来讲非常简单，但这一设计还是要面临一些实现上的挑战：

- 正确的维护数据库内容相对于源代码的依赖性并不是一件简单的事情。尽管将第三种情况误认为是第二种也不会导致错误，但是这样做会增加编译器完成的工作量（并因此增加了总体构建时间）。
- 并行编译多个源文件非常常见，因此，工业级实现需要支持适当数量的并发控制。

尽管存在这些挑战，这一方案还是可以非常有效地实施。此外，没有明显的病态场景会导致该方案的伸缩性变差。例如，与贪婪实例化相比，贪婪实例化可能会导致许多浪费的工作。

不幸的是，数据库的使用可能对程序员来说也存在一些问题。这些问题中的大部分的源头都在于传统的继承自C编译器的编译模型将不再可用：单一的编译单元不再会产生单独的目标文件。例如，假设你希望链接最终的程序，链接操作不仅需要各个编译单元所关联的目标文件的内容，还需要数据库中存储的目标文件。类似地，如果你创建了一个二进制库文件，你需要确保创建该库的工具（一般是一个链接器或是一个打包器）也能意识到数据库中的内容。这些问题大都可以通过不将实例化体存储在数据库，而是在目标文件中第一个引起实例化的地方放置目标代码的方式来缓解。

库文件还面临另一个挑战。许多生成的特化体可以打包在同一个库中。当库被另一个项目所添加时，该项目的数据库也需要意识到该库的数据库中已经可用的那些实例化体。否则，一旦项目创建了存在于库中的某个实例化的POI，就会遇到重复的实例化。一种可以解决该问题的策略是效仿贪婪实例化的链接器技术；让链接器意识到生成的特化体，并把它们淘汰掉（尽管如此，它的发生频率要比贪婪实例化要少得多）。源文件、目标文件以及库文件的各种复杂组织形式通常也会带来一些很难解决的问题，诸如找不到实例化体，因为包含该实例化体的目标代码可能并没有被链接入最终的可执行程序中。

总而言之，查询实例化最终没能在市场中存活，甚至Sun的编译器目前也在使用贪婪实例化。

14.4.3 迭代实例化

第一个支持C++模板的编译器是Cfront 3.0，它是语言之父Bjarne Stroustrup开发C++语言时所写的编译器的后浪。Cfront有一个不予变通的限制：它必须有良好的跨平台移植性。这就意味着：（1）在多个目标平台中，它都是使用C语言作为共同的目标表示；（2）它使用了局部的目标链接器，即链接器无法察觉到模板的存在。实际上，Cfront以普通C函数的形式来分发模板实例化体，因此它也必须避免重复的实例化体。虽然Cfront的源模型与标准的包含式模型有所差异，但它的实例化策略可以通过一些修改而适应包含式模型。于是，它也值得被公认为是迭代实例化的第一个实现。

Cfront的迭代过程如下所述：

1. 编译源代码，此时不要实例化任何需要链接的特化体
2. 使用预链接器(prelinker)链接目标文件
3. 预链接器调用链接器，解析错误信息，判断是否缺少某个实例化体。如果缺少的话，预链接器会调用编译器，来编译包含所需模板定义的源代码，然后（可选地）生成该缺少的实例化体。
4. 重复第3步，直到不再生成新的定义。

第3步中，这种迭代的要求基于这样的事实：在实例化一个可链接实体过程中，可能会要求”另一个仍未实例化“的实体进行实例化；最后，所有的迭代都已经完成，链接器才会成功创建一个完整的程序。

原始Cfront方案的缺陷相当严重：

- 要完成一次完整的链接，所需要的时间不仅包含预链接器的时间开销，还包括每次询问重新编译和重新链接的时间。某些使用Cfront系统的用户会抱怨说：”链接时间往往需要几天，而同样的工作，如果采用前面介绍的其他候选解决方案，则一个小时就足够了。”
- 诊断信息（错误和警告）延迟到了链接期，当链接大型程序时，这个缺点才是最严重的。譬如，对于模板定义中的某个书写错误，开发者可能需要等待漫长的几个小时才能检查出来。
- 需要进行特别地处理，来记住包含特殊定义的源代码的位置，Cfront（在一些情况下）会使用一个中心库，他不得不克服查询实例化方案中所面临的中心数据库的一些挑战。另外，原始Cfront实现并不支持并行编译。

迭代原则后来被Edison Design Group(EDG)和惠普的C++编译器实现精炼了一番，消除了原始Cfront实现的一些缺陷。实际上，这些实现体表现相当好，尽管从头开始构建比其他的替代方案

更耗时，但后续的构建时间却相当有可比性。不过，相对而言，很少有C ++编译器使用迭代实例化。

14.5 显式实例化

为模板特化显式地生成POI是可行的，我们把获得这种特化的结构称为显式实例化引导(explicit instantiation directive)。从语法上来说，它由关键字`template`和紧随其后的待实例化的特化声明组成。例如：

```
template<typename T>
void f(T)
{
}

// four valid explicit instantiations:
template void f<int>(int);
template void f<>(float);
template void f(long);
template void f(char);
```

注意上面的每一个实例化引导都是有效的。模板实参可以被推导（见第15章）。

类模板的成员也可以通过这种方式显式实例化：

```
template<typename T>
class S {
  public:
    void f() {
        }
};

template void S<int>::f();

template class S<void>;
```

此外，通过显式实例化该类模板特化本身，其所有的成员也都可以被显式实例化。因为这些显式实例化引导确保了具有名称的模板特化的定义被创造了出来，上面的显式实例化引导更准确地来

说，指的是显式实例化定义(explicit instantiation definitions)。显式实例化的模板特化不应该被显示地特化，反之亦然，因为这样会产生两个不同的定义（也就违反了ODR原则）。

14.5.1 手动实例化

许多C++程序员都观察到了自动模板实例化在编译期有一个值得一提的负面影响。这对于实现了贪婪实例化的编译器来说确实如此（P256节14.4.1），因为相同的模板特化可以在许多不同的编译单元中实例化。

有一种缩短构建时间的技术：在单一位置手动实例化程序所需的那些模板特化，并禁止其在所有其他编译单元中实例化。一种确保这种禁止行为的可行方法是：除非在编译单元中，有显示地实例化，否则不提供其模板定义。例如：

```
// ===== translation unit 1:
template<typename T> void f();        // no definition: prevents instantiation
                                                                  // in this translation unit
void g()
{
  f<int>();
}

// ===== translation unit 2:
template<typename T> void f()
{
  // implementation
}

template void f<int>();                          // manual instantiation

void g();

int main()
{
  g();
}
```

在第一个编译单元中，编译器看不到函数模板`f`的定义，因此它不会实例化`f<int>`。第二个编译单元借由显式实例化定义提供了`f<int>`的定义，如果没有该定义的话，程序链接会失败。

手动实例化有一个明显的缺陷：我们必须小心地追溯哪些实体会被实例化。对于大型项目来说，这很快就变成一个负担，因此我们并不推荐使用。我们已经在好几个项目中使用了这种做法，这些项目最初低估了这种负担，然而随着代码的成熟，我们对一开始的决定感到遗憾。

然而，手动实例化也有一些优势，因为实例化转变成了程序的需求。显然，它避免了大型头文件的开销，也避免了在多个编译单元中重复实例化具有相同参数的相同模板的开销。此外，模板定义的源代码可以隐藏起来，只不过客户端程序此后就再也无法创建额外的实例化体了。

手动实例化的一些负担可以通过将模板定义摆放至第三方源文件中来减轻，按照惯例，以`.tpp`作为扩展。对我们的函数`f`来说，就会变成：

```
// ===== f.hpp
template<typename T> void f();        // no definition: prevents instantiation

// ===== t.hpp
#include "f.hpp"
template<typename T> void f()        // definition
{
        // implementation
}

// ===== f.cpp
#include "f.tpp"

template void f<int>();               // manual instantiation
```

这种结构提供了某种灵活性。你可以仅仅引用`f.hpp`来获取`f`的声明，此时不会有自动实例化。显式实例化体可以被手动地添加到`f.cpp`中（如果需要的话）。或者，如果手动实例化太费劲，你也可以包含`f.tpp`来启用自动实例化。

14.5.2 显式实例化声明

消除冗余自动实例化的一种更有针对性的方法是使用显式实例化声明，该声明是一个以关键字`extern`为前缀的显式实例化引导。显式实例化声明通常会抑制命名模板特化的自动实例化，因为它声明命名模板特化将在程序中的某个位置定义（通过显式实例化定义）。之所以说是通常来说，是因为有一些特例存在：

- 内联函数仍可以实例化，以展开成内联样式（但不会生成单独的目标代码）。

- 具有`auto`或`decltype(auto)`推导的类型和具有返回类型推导的函数仍然可以被实例化，以判断它们的类型。
- 其值可用作常量表达式的变量仍可以被实例化，以便对其值进行求值。
- 引用类型的变量仍然可以被实例化，因此可以解析它们引用的实体。
- 类模板和别名模板仍然可以被实例化，以检查其返回类型。

通过使用显式实例化声明，我们可以在头文件(t.hpp)中为`f`提供模板定义，然后通过使用特化来抑制自动实例化，如下：

```
// ===== t.hpp
template<typename T> void f()
{
}

extern template void f<int>();                          // declared but not defined
extern template void f<float>();                  // declared but not defined

// ===== t.cpp
template void f<int>();                                          // definition
template void f<float)();                                  // definition
```

每个显式实例化声明必须与一个相应的显式实例化定义配对，该定义必须遵循该显式实例化声明。忽略定义将导致链接器错误。

当在许多不同的编译单元中使用某些特定的特化时，可以使用显式实例化声明来改善编译或链接时间。与手动实例化（每次需要新的特化时，都需要手动更新显式实例化定义的列表）不同的是，在任何时候都可以引入显式实例化声明作为优化项。然而，与手动实例化相比，编译器的受益可能没有那么显著，这是因为可能会发生一些冗余的自动实例化，以及模板定义作为头文件的一部分，仍然会被解析。

14.6 编译期if语句

正如在P134节8.5中介绍的，C++17增加了一种新的语句——编译器if，在书写模板时非常有用，同时也对实例化过程产生了一种新的影响。

下面的例子展示了这一基本操作：

```
template<typename T> bool f(T p) {
  if constexpr (sizeof(T) <= sizeof(long long)) {
    return p > 0;
  } else {
    return p.compare(0) > 0;
  }
}

bool g(int n) {
  return f(n);        // OK
}
```

编译器if是一个if语句，其中关键字`if`后面紧跟着一个`constexpr`关键字（如本例所示）。跟随在后面的是一个小括号条件语句，该语句必须是一个常量布尔值（也可以是隐式转换为`bool`值的情形）。编译器因而就会知道该选择哪一个分支，而另一个未被选中的分支则被称作“丢弃的分支”。特别有趣的是，在模板（包括泛型lambda）的实例化过程中，被丢弃的分支不会进行实例化。对于这一示例代码的合法性来说，该机制很有必要：我们用`T=int`来实例化`f(T)`，会使得else分支被丢弃。如果该分支未被丢弃的话，它就会进行实例化，此时表达式`p.compare(0)`会引起一个错误（当`p`是简单的`int`型时，这段代码是不合法的）。

在C++17的constexpr if语句出现之前，规避这类错误需要进行显式模板特化或重载（见第16章）才能起到相似的效果。

上面的例子，在C++14中，可能会按如下方法来实现：

```
template<bool b> struct Dispatch {          // only to be instantiated when b is false
  static bool f(T p) {                                  // (due to next specialization for true)
    return p.compare(0) > 0;
  }
};

template<> struct Dispatch<true> {
  static bool f(T p) {
    return p > 0;
  }
};

template<typename T> bool f(T p) {
  return Dispatch<sizeof(T) <= sizeof(long long)>::f(p);
}

bool g(int n) {
  return f(n);                // OK
}
```

显然，constexpr if这一替代方案的引入使得我们的意图简明扼要、一目了然。然而，它需要（编译器）的实现去提炼实例化单元：此前的函数定义始终都是作为整体来实例化，现在它必须禁用其中的一部分。

另一个非常好用的constexpr if的场景是处理函数模板包的递归表达式。为了泛化这一例子，我们引用P134节8.5中出现的例子：

```
template<typename Head, typename... Remainder>
void f(Head&& h, Remainder&&... r) {
  doSomething(std::forward<Head>(h));
  if constexpr (sizeof...(r) != 0) {
    // handle the remainder recursively (perfectly forwarding the arguments):
        f(std::forward<Remainder>(r)...);
  }
}
```

如果没有constexpr if语句，我们需要对`f()`模板实现一个额外的重载来保证递归的终结。

甚至在非模板上下文中，constexpr if语句有时也能起到独特的效果：

```
void h();
void g() {
  if constexpr (sizeof(int) == 1) {
    h();
  }
}
```

大部分平台，`g()`中的条件都是`false`，对`h()`的调用也就会被丢弃掉。因此，`h()`甚至完全不需要被定义（当然，除非它在别的地方被使用到了）。如果在此示例中省略了关键字`constexpr`，则在链接期会触发“缺少`h()`的定义”的错误。

14.7 标准库中的显式实例化

C++标准库包含了若干数量的模板，这些模板通常仅仅与一些基础类型一起使用。例如，和`std::basic_string`类模板一起最常用的类型就是`char`或`wchar_t`，尽管使用其他的类字符类型也可以完成实例化。因此，对标准库的实现来说，通常会为这些常见的情景引入显式实例化声明。例如：

```
namespace std {
  template<typename charT, typename traits = char_traits<charT>,
                           typename Allocator = allocator<charT>>
  class basic_string {
    ...
  };

  extern template class basic_string<char>;
  extern template class basic_string<wchar_t>;
}
```

实现了标准库的源文件会包含相应的显式实例化定义，因此这些常见的实现体可以在所有使用标准库的编译单元中共享。类似的显示实例化还出现在各种“流(stream)”类类型中，诸如`basic_iostream`，`basic_istream`等等。

14.8 后记

本章处理了两个有一定联系但并不相同的议题：C++模板编译模型和各种C++模板实例化机制。

编译模型在程序编译的各个阶段确定模板的含义。特别地，它确定了实例化模板中各种结构的含义。名称查找是编译模型的重要组成部分。

标准C++仅仅支持单个编译模型，即包含式模型。然而，在1998和2003标准中还支持一个叫分离式模型的模板编译模型。分离式模型允许模板定义可以在其实例化体所在的不同的编译单元中书写。这种导出式模板(exported templates)仅曾经由Edison Design Group(EDG)实现过一次。EDG在实现中付出的努力确定了以下两点：（1）实现C++模板的分离式模型相当的困难，而且完成这一任务的耗时远超预期；（2）分离式模型的假定好处（例如优化编译时间）由于模型的复杂性而无法实现。随着2011标准的制定工作逐渐结束，很明显其他实现者将不会支持这一功能，于是，C++标准委员会根据投票结果最终从语言中删除了导出式模板。如果你对分离式模型的细节感兴趣，可以看看本书的第一版，里面描述了导出式模板的行为。

实例化机制是一种外部机制，用以允许C++实现者去正确地创建实例化体。这些机制可能会受限于链接器和其他软件构建工具的需求。尽管每一种实例化机制都各不相同且各有利弊，但它们对日常C++编程来说并没有显著的影响。

就在C++11标准完成之后，Walter Bright, Herb Sutter和Andrei Alexandrescu提议了一种“static if”特性，它与“constexpr if”不同（文献N3329）。这是一种更为宽泛的特性，它甚至可以出现在函数定义外部（Walter Bright是D语言的设计者和实现者，它有一个相似的特性）。例如：

```
template<unsigned long N>
struct Fact {
  static if (N <= 1) {
    constexpr unsigned long value = 1;
  } else {
    constexpr unsigned long value = N*Fact<N-1>::value;
  }
};
```

请注意看在上例中，类作用域声明是如何条件化的。然而，这种强大的能力是有争议的，有些委员会成员担心它可能会被滥用，而另一些委员会成员则不喜欢该提案的某些技术方面（诸如花括号未引入作用域，以及完全不分析丢弃的分支）。

几年之后，Ville Voutilainen又提出了一个提案(P0128)，该提案的大部分内容在日后摇身一变促成了constexpr if语句的诞生。它经历了几轮小版本的设计迭代（涉及临时关键字`static_if`和`constexpr_if`），并且在Jens Maurer的帮助下，Ville最终将该提议编入了该语言中（由文献P0292r2）。

第15章 模板实参推导

如果每个函数模板都要显式地指定模板实参，那么代码一下子就变得笨重起来（型如：`concat<std::string, int>(s, 3)`）。幸运的是，C++编译器常常可以自动判断模板实参类型，这是通过一个十分高效的过程——模板实参推导——来完成的。

本章中我们将详述模板实参推导这一过程的细节。C++世界的诸多大道产生的结果向来直观，模板实参推导也不例外。深入理解本章还可以使我们日后避免遇到出人意料的情景。

模板实参推导起初是为了简化函数模板的调用而被发明出来，但随着发展，它已被扩展到各种其他用途，其中包括：根据初始化器确定变量的类型。

15.1 推导过程

基本的推导过程会去比较“函数调用的实参类型”与“函数模板对应位置的参数化类型”，然后针对要被推导的一到多个参数，分别尝试去推断一个正确的替换项。每个“实参-参数对”都会独立分析，并且如果最终得出的结论有矛盾，那么推导过程就以失败告终。

考虑下面的例子：

```
template<typename T>
T max(T a, T b)
{
  return b < a ? a : b;
}

auto g = max(1, 1.0);
```

这里第一个调用实参的类型是`int`，因此我们原生的`max()`模板的参数`T`会被姑且推导成`int`。然而，第二个调用实参是`double`类型，基于此，`T`会被推导为`double`：这就与前一个推导产生

了矛盾。注意：我们称之为“推导过程失败”，而不是“程序非法”。毕竟，可能存在另一个名为`max`（函数模板可以像普通函数那样被重载；参考P15节1.5和第16章）的模板，它的推导可以成功。

即使所有被推导的模板实参都可以一致地确定（即不产生矛盾），推导过程仍然可能会失败。这种情况发生于：在函数声明中，进行替换的模板实参可能会导致无效的结构。请看下例：

```
template<typename T>
typename T::ElementT at(T a, int i)
{
  return a[i];
}

void f(int* p)
{
  int x = at(p, 7);
}
```

这里`T`被推导为`int*`(`T`出现的地方只有一种参数类型，因此显然不会有矛盾)。然而，将`T`替换为`int*`在C++中对于返回类型`T::ElementT`来说显然是非法的，因此推导还是失败了。

我们仍然需要挖掘实参-参数的匹配是如何进行的。我们会使用下面的术语来进行描述：匹配类型A（调用实参的类型）和参数化类型P（调用参数的声明）。如果调用参数被声明为引用，那么P就是引用背后的类型，A是实参的类型。如果调用参数并非引用，那么P就是参数类型，而A类型则会经历数组和函数类型到指针类型的退化、以及忽略顶层`const`和`volatile`限定符，最终获取。例如：

```
template<typename T> void f(T);                 // parameterized type P is T
template<typename T> void g(T&);          // parameterized type P is also T

double arr[20];
int const seven = 7;

f(arr);                 // nonreference parameter:          T is double*
g(arr);                 // reference parameter:                 T is double[20]
f(seven);          // nonreference parameter:          T is int
g(seven);          // reference parameter:                 T is int const
f(7);                  // nonreference parameter:         T is int
g(7);                  // reference parameter:                 T is int => ERROR: can't pass 7 to int
```

对调用`f(arr)`来说，`arr`数组类型会退化为类型`double*`，也就是被推导出来的`T`的类型。在`f(seven)`中`const`限定符被忽略了，因此T被推导为`int`。`g(arr)`的推导则恰恰相反，`T`被推导为类型`double[20]`(没有发生退化)。类似地，`g(seven)`有一个类型为`int const`的左值实参，并且因为在匹配引用参数时，`const`和`volatile`限定符不会被去除，`T`会被推导成`int const`。然而，`g(7)`想要推导`T`为`int`（非类的右值表达式永远不会有cv限定），这一推导最终会失败，这是因为实参`7`无法作为一个`int&`类型的参数被传递（译者注：右值不能传参给左值引用）。

引用型参数不会退化这一事实，对于参数为字符串字面量的场合来说可能会令人诧异。再来看看使用引用型参数的`max()`模板声明：

```
template<typename T>
T const& max(T const& a, T const& b);
```

对于表达式`max("Apple", "Pie")`来说，我们合理的期望`T`能被推导为`char const*`。然而事与愿违，`Apple`的类型是`char const[6]`、`Pie`的类型是`char const[4]`。由于推导涉及了引用型参数，这里并不会进行数组到指针的退化，因此若想要推导成功，`T`必须既得是`char[6]`又得是`char[4]`。显然，这绝无可能。可以参考P115节7.4中对于如何处理这一场景的一个探讨。

15.2 推导上下文

比仅是一个`T`要复杂得多的参数类型也可以匹配给定的实参类型。这里有一些相当基础的例子：

```
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*));

class S {
  public:
    void f(double*);
};

void g(int*** ppp)
{
  bool b[42];
  f1(ppp);                // deduces T to be int**
  f2(b);                // deduces E to be bool and N to be 42
  f3(&S::f);        // deduces T1 = void, T2 = S, and T3 = double
}
```

复杂的类型声明都是用比它更简单的结构（例如指针、引用、数组、函数声明；成员指针声明；模板ID等）来组成的，匹配过程从最顶层结构开始处理，向下递归到各种组成元素。可以说基于这一方法，大部分类型声明结构都可以进行匹配，而这些结构也被称为“推导上下文“。然而，有一些结构不能作为推导上下文。诸如：

- 限定类型的名称。例如，形如`Q<T>::X`的类型名称永远不会用来推导模板参数`T`。
- 不仅仅是非类型参数的非类型表达式。例如，形如`S<I+1>`的类型名称永远不会用于推导`I`。再比如，`T`也不会通过匹配形如`int(&)[sizeof(S<T>)]`类型的参数来推导。

这些限制合乎常理，因为通常来说，推导并不是唯一的（甚至不一定是有限的），尽管有时候会很容易忽略这些限定类型的名称。此外，不能推导的上下文并不直接意味着：对应的程序有错误、甚至是前面分析过的参数不能再次进行类型推导。为了阐释这一事实，考虑下面这个更为错综复杂的例子：

```
template<int N>
class X {
  public:
    using I = int;
        void f(int) {}
};

template<int N>
void fppm(void (X<N>::*p)(typename X<N>::I));

int main()
{
  fppm(&X<33>::f);                // fine: N deduced to be 33
}
```

在函数模板`fppm()`中，子结构`X<N>::I`是一个不可推导上下文。然而，具有成员指针类型（即`X<N>::*p`）的成员类型部分`X<N>`是一个可推导上下文。于是，可以根据这个可推导上下文获得参数`N`，然后把`N`放入不可推导上下文`X<N>::I`，就能获得与实参`&X<33>::f`相配的类型。因此基于这个实参-参数对的推导就是成功的。

反之，对于完全依赖推导上下文的参数类型来说，有可能会产生推导矛盾。例如，假设我们已恰当地声明过类模板`X`和`Y`：

```
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: deduction fails
}
```

第二个调用的问题在于两个实参对于参数`T`的推导存在矛盾（对此二例，函数调用实参都是临时的对象，这一对象借由调用类模板`X`的默认构造器而获得）。

15.3 特殊的推导情景

还有一些特殊的情景：用于推导的实参-参数对（A, P）并非来源于函数调用的实参和函数模板的参数。第一种情景出现在取函数模板地址的时候。此时，P是函数模板声明的参数化类型（即下面`f`的类型），而A是被赋值（或者初始化）的指针（即下面的`pf`）所代表的函数类型。例如：

```
template<typename T>
void f(T, T);

void (*pf)(char,char) = &f;
```

在本例中，P是`void(T, T)`，而A是`void(char, char)`。推导随着`T`被`char`替换而成功，而`pf`用特化体`f<char>`的地址进行初始化。

类似地，函数类型在一些其他特殊情况下也被P和A所使用：

- 确定重载函数模板之间的偏序
- 将某个显式特化体与某个函数模板匹配
- 将某个显式实例化体与某个模板匹配
- 将某个友元函数模板特化体与某个模板匹配
- 将占位(replacement)`operator delete`或是`operator delete[]`与对应的占位`operator new`或`operator new[]`模板匹配。

这些话题中的部分内容，以及类模板偏特化中模板实参推导的使用，会在第16章中进行展开。

另一种特殊情况和类型转换运算符模板一起出现。例如：

```
class S {
  public:
    template<typename T> operator T&();
};
```

在这种情况下，对于实参-参数对(P, A)，它的获取过程就好像涉及到了我们试图转换的类型的实参和转换运算符的返回类型的参数一样。下面的代码阐释了这一情景：

```
void f(int (&)[20]);

void g(S s)
{
  f(s);
}
```

这里，我们试图把`S`转换为类型`int(&)[20]`，因此，类型A就是`int[20]`，而类型P为`T`。`T`用`int[20]`替换，推导得以成功。

最后，对于`auto`占位类型来说，也需要一些特殊的处理。这会在P303节15.10.4中进行讨论。

15.4 初始化列表(initializer list)

当函数调用的实参是一个初始化列表时，该实参是没有特定类型的，因此通常来说，对于给定实参-参数对(A, P)，不会进行任何推导，因为这里并不存在A。例如：

```
#include <initializer_list>

template<typename T> void f(T p);

int main() {
  f({1, 2, 3});               // ERROR: cannot deduce T from a braced list
}
```

然而，如果在移除引用、顶层const和volatile限定后，参数类型`P`与某个具有可推导模式的类型`P'`的`std::initializer_list<P'>`等价，则推导过程会将初始化列表的每个元素类型与`P'`进行比较，仅当所有元素具有相同类型时，推导才会成功。

deduce/initlist.cpp

```
#include <initializer_list>

template<typename T> void f(std::initializer_list<T>);

int main()
{
        f({2,3,5,7,9});                                                 // OK: T is deduced to int
        f({'a', 'e', 'i', 'o', 'u', 42});        // ERROR: T deduced to both char and int
}
```

类似地，如果参数类型`P`是对具有元素类型`P'`的数组类型的引用，其中`P'`是具有可推导模式的某个类型，那么推导过程也会将初始化列表的每个元素的类型与`P'`进行比较，当且仅当所有元素具有相同的类型时，推导才会成功。此外，如果（数组）边界有一个可推导模式（即，使用一个非类型模板参数），那么该边界会被推导为初始化列表中元素的数量。

15.5 参数包

推导过程会逐一匹配每个实参到每个参数来确定模板实参的值。然而在对可变模板进行模板实参推导时，参数和实参之间1比1的关系就被打破了，这是因为一个参数包可以匹配多个实参。在本例中，同一个参数包(P)被匹配到了多个实参(A)，并且每次匹配都会为P中的任何模板参数包产生附加值：

```
template<typename First, typename... Rest>
void f(First first, Rest... rest);

void g(int i, double j, int* k)
{
  f(i, j, k);        // deduces First to int, Rest to {double, int*}
}
```

此处对首个函数参数的推导很简单，毕竟它并没有卷入任何参数包。第二个函数参数，`rest`，是一个函数参数包。它的类型是一个包展开(`Rest...`)，其模式为类型`Rest`：该模式用作P，与第二和第三调用参数的类型A进行比较。当匹配第一个A时（类型`double`），模板参数包`Rest`的第一个值被推导为`double`。类似地，与第二个A进行匹配时，模板参数包`Rest`的第二个值被推导为`int*`。因此，推导确定了参数包`Rest`的值序列为`{double, int*}`。替换以上推导结果就可以得到函数类型`void(int, double, int*)`，它与函数调用的每个实参类型相匹配。

由于对函数参数包进行推导使用了扩展的模式进行比较，所以该模式可以是任意复杂的，并且多个模板参数和参数包的值可以从每个实参类型中确定。考虑下面的函数`h1()`和`h2()`的推导行为：

```
template<typename T, typename U> class pair { };

template<typename T, typename... Rest>
void h1(pair<T, Rest> const&...);
template<typename... Ts, typename... Rest>
void h2(pair<Ts, Rest> const&...);

void foo(pair<int, float> pif, pair<int, double> pid, pair<double, double> pdd)
{
  h1(pif, pid);         // OK: deduces T to int, Rest to {float, double}
  h2(pif, pid);         // OK: deduces Ts to {int, int}, Rest to {float, double}
  h1(pif, pdd);         // ERROR: T deduced to int from the 1st arg, but to double from the 2nd
  h2(pif, pdd);         // OK: deduces Ts to {int, double}, Rest to {float, double}
}
```

对`h1()`和`h2()`来说，P都是引用类型，它们分别与非限定版本的引用相匹配，再次用于推导每个参数类型（分别为`pair<T, Rest>`和`pair<Ts, Rest>`的引用）。由于所有的参数和实参都是类模板`pair`的特化，因此进行了模板实参的比较。对`h1()`来说，第一个模板实参`T`不是参数包，因此它的值是独立地对每个实参进行推导的。如果推导的结果出现矛盾（正如对`h1`的第二次调用那样），推导就会失败。对于`h1()`和`h2()`中的第二个`pair`模板实参`Rest`、以及`h2()`中的第一个`pair`模板实参`Ts`，推导会根据A的每个实参类型来确定一连串的参数包的值。

参数包的推导不仅限于“实参-参数对”来自调用参数的函数参数包。实际上，在函数参数列表或模板参数列表末尾的包展开处推导都会被使用。例如，考虑一个简单的`Tuple`类型上的两个相似操作：

```
template<typename... Types> class Tuple { };

template<typename... Types>
bool f1(Tuple<Types...>, Tuple<Types...>);

template<typename... Types1, typename... Types2>
bool f2(Tuple<Types1...>, Tuple<Types2...>);

void bar(Tuple<short, int, long> sv, Tuple<unsigned short, unsigned, unsigned long> uv)
{
  f1(sv, sv);        // OK: Types is deduced to {short int, long}
  f2(sv, sv);        // OK: Types1 is deduced to {short, int, long},
                     //     Types2 is deduced to {short, int, long}
  f1(sv, uv);        // ERROR: Types is deduced to {short, int, long} from the 1st arg,
                     //        but to {unsigned short, unsigned, unsigned long} from
  f2(sv, uv);        // OK: Types1 is deduced to {short, int, long},
                     //     Types2 is deduced to {unsigned short, unsigned, unsigned
}
```

在`f1()`和`f2()`中，模板参数包都是将`Tuple`类型内嵌的包展开模式与调用实参所提供的`Tuple`类型进行比较，为一致的模板参数包推导出正确的值。函数`f1()`对两个函数参数使用相同的模板参数包`Types`，确保只有当两个函数调用实参有相同的`Tuple`特化体类型时，才能推导成功。而`f2()`则为每个函数参数各使用了一个参数包，因此两个调用参数可以不同——也就可以使用`Tuple`的两种特化体类型。

15.5.1 字面量操作符模板

字面量操作符模板的实参通过一种独特的方式来确定。下面的例子进行了阐释：

```
template<char...> int operator "" _B7();        // #1
...
int a = 121_B7;                 // #2
```

这里，#2处的初始化器包含了一个用户定义的字面量（它会转换成对字面操作符模板的调用，使用的模板实参列表为`<'1','2','1'>`）。因此，字面量操作符的实现体可能如下：

```
template <char... cs>
int operator"" _B7()
{
  std::array<char,sizeof...(cs)> chars{cs...};        // initialize array of passed chars
  for(char c : chars) {                                                    // and use it (prin
    std::cout << "'" << c << "'";
  }
  std::cout << '\n';
  return ...;
}
```

它会为121.5_B7输出`'1' '2' '1' '.' '5'`。

请注意，仅在没有后缀的情况下仍然有效的数值字面量才支持此技术。例如：

```
auto b = 01.3_B7;         // OK: deduces <'0','1','.','3'>
auto c = 0xFF00_B7;         // OK: deduces <'0','x','F','F','0','0'>
auto d = 0815_B7;         // ERROR: 8 is no valid octal literal
auto e = hello_B7;         // ERROR: identifier hello_B7 is not defined
auto f = "hello"_B7;         // ERROR: literal operator _B7 does not match
```

参考P599节25.6对这一特性的应用：编译期计算整型字面量。

15.6 右值引用

C++11引入的右值引用促生了许多新技术，包括移动语义和完美转发。本节会描述右值引用与推导之间的交互。

15.6.1 引用折叠法则

开发者不允许直接声明“引用的引用”：

```
int const& r = 42;
int const& & ref2ref = i;        // ERROR: reference to reference is invalid
```

然而，当通过模板参数替换、类型别名或是`decltype`结构构造类型时，“引用的引用”将被允许。

例如：

```
using RI = int&;
int i = 42;
RI r = i;
RI const& rr = r;        // OK: rr has type int&
```

判定像是这种组织结构的类型结果的规则，就是众所周知的引用折叠法则。首先，任何应用于内部引用顶层的`const`或`volatile`限定都会被舍弃（也就是说，只有内层引用的底层限定才会被保留）。此后，这两种引用会根据表15.1推导出单一引用，这种推导方式可以总结为一句话："如果某个引用是左值引用，那么结果也一定是左值引用，否则就是右值引用"。

内层引用	外层引用	结果引用
&	&	&
&	&&	&
&&	&	&
&&	&&	&&

表15.1 引用折叠法则

展示这一规则的更多示例：

```
using RCI = int const &;
RCI volatile&& r = 42;        // OK: r has type int const &;
using RRI = int&&;
RRI const&& rr = 42;        // OK: rr has type int&&
```

这里`volatile`被应用在`RCI`这一引用类型（`int const&`的别名）的顶层，因此会被丢弃掉。这一类型的顶层又放置了一个右值引用，但是由于底层类型是一个左值引用（左值引用在引用折叠规则中"更优先"），所以最终的类型保留为`int const&`（或者`RCI`类型、一个等价的别名）。类似地，`RRI`的顶层const会被丢弃，在右值引用类型上应用一个右值引用，最后的结果依然是一个右值引用类型（可以绑定到像42这样的右值上）。

15.6.2 转发引用

如同P91节6.1所介绍的那样，当函数参数是一个转发引用（函数模板参数中的右值引用）时，模板实参推导会呈现另一种表现形式。此时，模板实参推导不仅会考虑函数调用实参的类型，同时也会考虑该实参是左值还是右值。如果实参是一个左值，那么模板实参推导所确定的类型就是该实参类型的左值引用类型，引用折叠规则会确保所替换的参数可以成为一个左值引用。如果实参不是左值，那么模板参数所推导的类型就是实参类型，而替代的参数是该类型的右值引用。例如：

```
template<typename T> void f(T&& p);        // p is a forwarding reference

void g()
{
  int i;
  int const j = 0;
  f(i);                // argument is an lvalue; deduces T to int& and
                          // parameter p has type int&
  f(j);                // argument is an lvalue; deduces T to int const&
                          // parameter p has type int const&
  f(2);                // argument is an rvalue; deduces T to int
                          // parameter p has type int&&
}
```

在调用`f(i)`中，模板参数`T`被推导为`int&`，因为表达式`i`是一个类型为`int`的左值。`T`替换`int&`到参数类型`T&&`中需要引用折叠，这里我们使用规则`&` + `&&` -> `&`来得出结论：参数类型为`int&`，如此就可以完美的接受`int`类型的左值。相对的，在调用`f(2)`中，实参`2`是一个右值，模板参数因此直接被推导为右值的类型（即`int`）。这里不需要进行引用折叠，其结果直接就是`int&&`（同样地，对实参来说这是一个合适的参数类型）。

当`T`被推导为一个引用类型时，对于模板的实例化来说有些有趣的效果。例如，使用类型`T`声明的局部变量，在用左值实例化后，会有一个引用类型，而此时它就需要一个初始化器：

```
template<typename T> void f(T&&)        // p is a forwarding reference
{
  T x;        // for passed lvalues, x is a reference
  ...
}
```

这就意味着函数`f()`的定义需要很小心地使用类型`T`，或者函数模板本身根本不为左值参数生效。为了解决这一困境，`std::remove_reference`类型萃取常常被用来确保`x`不是一个引用：

```
template<typename T> void f(T&&)        // p is a forwarding reference
{
  std::remove_reference_t<T> x;                // x is never a reference
  ...
}
```

15.6.3 完美转发

右值引用特殊的推导规则和引用折叠法则组合在一起使得编写一个接受任何实参的函数模板来捕捉其表征属性（它的类型、是左值还是右值）成为了可能。函数模板此后可以“转发”这一实参给另一个函数，恰如此例：

```
class C {
  ...
};

void g(C&);
void g(C const&);
void g(C&&);

template<typename T>
void forwardToG(T&& x)
{
  g(static_cast<T&&>(x));        // forward x to g()
}

void foo()
{
  C v;
  C const c;
  forwardToG(v);        // eventually calls g(C&)
  forwardToG(c);        // eventually calls g(C const&)
  forwardToG(C());        // eventually calls g(C&&)
  forwardToG(std::move(v));        // eventually calls g(C&&)
}
```

上例所展示的技术被称为完美转发(perfect forwarding)，因为通过`forwardToG()`间接调用`g()`的效果与直接调用`g()`相同：没有额外的拷贝，选择的重载函数`g()`也一模一样。

`static_cast`的使用需要一些额外的解释。在每个`forwardToG()`的实例化体中，参数`x`要么是一个左值引用，要么是一个右值引用。而无论如何，表达式`x`本身一定是一个（其引用类型的）左值。`static_cast`会将`x`转换为其原始类型（不管左值还是右值）。类型`T&&`要么折叠成一个左值引用（如果原本的实参是一个左值，那么`T`就是一个左值引用），要么是一个右值引用（原本的实参就是一个右值），因此`static_cast`的结果就有了一致的类型，不论原本的实参是左值也好、右值也罢，如此，就实现了完美转发。

如P91节6.1所介绍的那样，C++标准库提供了一个函数模板`std::forward<>()`（在头文件`<utility>`中），它被用来取代`static_cast`进行完美转发。相比晦涩难懂的`static_cast`结构来说，使用这一模板对开发者来说更加表意，同时也防止了诸如少写了一个`&`所导致的错误。那么，上面的例子可以更为简明地写成这个样子：

```
#include <utility>

template<typename T> void forwardToG(T&& x)
{
  g(std::forward<T>(x));        // forward x to g()
}
```

可变模板的完美转发

完美转发与可变模板搭配在一起，可以让函数模板接受任意数量的函数调用实参并将它们逐一转发到另一个函数：

```
template<typename... Ts> void forwardToG(Ts&&... xs)
{
  g(std::forward<Ts>(xs)...);        // forward all xs to g()
}
```

`forwardToG()`的实参会为参数包`Ts`分别被推导出合适的值（见P275节15.5），因此类型以及每个参数的左值性或右值性都会被捕获。包展开（见P201节12.4.1）在调用`g()`时会将每个实参都应用上述的完美转发技术进行转发。

尽管它拥有一个“完美转发”的名字，但实际上，从它不能捕获表达式所有感兴趣属性的意义上来

说，完美转发实际上并不“完美”。例如，它无法区分左值是不是一个位域(bit-field)左值，也无法捕获表达式是否有特定的常量值。后者尤其在我们处理空指针常量时常常导致问题（它是一个整型类型、常量零值）。由于表达式常量值不会被完美转发所捕获，下例中的重载决议对直接调用`g()`和转发调用`g()`来说，表现上会有所区别：

```
void g(int*);
void g(...);

template<typename T> void forwardToG(T&& x)
{
  g(std::forward<T>(x));        // forward x to g()
}

void foo()
{
  g(0);                                 // calls g(int*)
  forwardToG(0);        // eventually calls g(...)
}
```

这也是为什么使用`nullptr`(C++11所引入)取代空指针常量的一个原因：

```
g(nullptr);                // calls g(int*)
forwardToG(nullptr);        // eventually calls g(int*)
```

我们所有完美转发的例子都聚焦于传递的函数实参要如何保留其精准的类型以及它是一个左值或是右值。当转发函数调用的返回值需要传递给另一个函数时，也面临着同样的问题（类型和值的分类，对左值和右值的概括在附录B中进行了讨论）。可以借助C++11引入的`decltype`语法（在P298节15.10.2中描述），使用这样一个有些繁琐的惯用法来解决：

```
template<typename... Ts>
auto forwardToG(Ts&&... xs) -> decltype(g(std::forward<Ts>(xs))
{
  return g(std::forward<Ts>(xs)...);        // forward all xs to g()
}
```

请注意，`return`语句的表达式被拷贝到了`decltype`类型里，因此返回表达式的准确类型会被计算出来。尾随返回类型被使用（即，函数名称前的`auto`占位符和指示返回类型的`->`），使得函

数参数包`xs`也在`decltype`类型的作用域。该转发函数会“完美地”转发所有实参给`g()`，然后再“完美地”转发其返回值给调用者。

C++14引入了额外的特性来简化这一情景：

```
template<typename... Ts>
decltype(auto) forwardToG(Ts&&... xs)
{
  return g(std::forward<Ts>(xs)...);        // forward all xs to g()
}
```

使用`decltype(auto)`做返回类型会指示编译器通过函数定义来推导返回类型。参见P296节15.10.1和P301节15.10.3。

15.6.4 意外的推导

对完美转发来说，右值引用的特殊推导规则非常有用。然而，有时候它们可能会令人惊讶，这是因为函数模板通常会泛化函数签名中的类型，不会影响它所允许的参数是何种类型（左值或右值）。考虑下例：

```
void int_lvalues(int&);                 // accepts lvalues of type int
templte<typename T> void lvalues(T&);         // accepts lvalues of any type

void int_rvalues(int&&);         // accepts rvalues of type int
template<typename T> void anything(T&&);         // SURPRISE: accepts lvalues and
                                                                                // rvalues
```

抽象出一个像`int_lvalues`那样的函数的开发者，可能会对函数模板`anything`可以接受左值而感到诧异。幸运的是，只有当函数参数写成特定的模板参数`&&`的形式时（作为函数模板的一部分且命名的模板参数是由该函数模板所声明），才会应用这一推导行为。因此，下面这些例子的情形都不会应用推导规则：

```
template<typename T>
class X
{
  public:
    X(X&&);               // X is not a template parameter
        X(T&&);               // this constructor is not a function template

        template<typename Other> X(X<U>&&);        // X<U> is not a template parameter
        template<typename U> X(U, T&&);               // T is a template parameter from
                                                                                    // an outer templa
};
```

尽管模板推导规则有着这些令人惊讶的行为，在实践中，这种行为导致问题的情况并不经常出现。当出现问题时，你可以组合使用SFINAE（参考P129节8.4和P284节15.7）和诸如`std::enable_if`的类型萃取来约束模板只能接受右值：

```
template<typename T>
typename std::enable_if<!std::is_lvalue_reference<T>::value>::type
rvalues(T&&);        // accepts rvalue of any type
```

15.7 SFINAE(Substitution Failure Is Not An Error)

SFINAE(替换失败并非错误)原则在P129节8.4中介绍过，它是模板实参推导中在重载决议期间防止不相干的函数模板产生错误的关键先生。

例如，考虑这样一对函数模板，它们从给定的容器或数组榨取起始的迭代器：

```
template<typename T, unsigned N>
T* begin(T (&array)[N])
{
  return array;
}

template<typename Container>
typename Container::iterator begin(Container& c)
{
  return c.begin();
}

int main()
{
  std::vector<int> v;
  int a[10];

  ::begin(v);        // OK: only container begin() matches, because the first deduction fails
  ::begin(a);        // OK: only array begin() matches, because the second substitution fails
}
```

第一个`begin()`调用的实参是`std::vector<int>`，它试图为两个`begin()`函数模板做模板实参推导：

- 对数组`begin()`的模板实参推导失败了，因为`std::vector`不是一个数组，所以被忽略。
- 模板实参推导对容器`begin`成功了，`Container`被推导成`std::vector<int>`，因此函数模板可以被实例化，也可以被调用。

第二个`begin()`调用的实参是一个数组，也会部分失败：

- 对数组`begin()`推导成功，`T`被推导为`int`，`N`被推导为`10`。
- 对容器`begin()`来说，推导需要将`Container`替换为`int[10]`，这本身没有问题，但是如此产生的返回类型`Container::iterator`却是无效的（因为数组类型并没有嵌套的名为`iterator`的类型）。在其他上下文中，试图访问一个本不存在的嵌套类型会立即导致一个编译期错误。而在模板实参的替换中，SFINAE会将这种错误转换成推导失败，并且不再将这一函数模板纳入考虑。因此，第二个`begin()`候选会被忽略，第一个`begin()`函数模板的特化体会被调用。

15.7.1 立即上下文

SFINAE阻止了那些无效类型或表达式的生成，包括因歧义或非法访问控制所产生的错误，它们发生在函数模板替换的立即上下文中。比起定义“函数模板替换的立即上下文”，对“不在该上下文中”进行定义可能更为容易。具体来说，在函数模板替换过程中，为了推导而发生的下面这些实例化期间的事，都不在函数模板替换的立即上下文中：

- 类模板的定义（即，类模板本身以及其基类列表）
- 函数模板的定义（即，函数模板本身，对构造函数来说，是其构造初始化器）
- 变量模板初始化
- 默认实参
- 默认成员初始化
- 异常规范(exception specification)

此外，任何由替换过程所触发的特殊成员函数的隐式定义也不属于替换的立即上下文。除这些以外，其余部分都被算在立即上下文中。

因此，如果在替换函数模板声明的模板参数时需要类模板实例化（因为该类被引用了），则实例化过程产生的错误并不在函数模板替换的即时上下文中，因此它会产生一个真正的错误（即使另一个函数模板可以无错误地匹配上）。例如：

```
template<typename T>
class Array {
  public:
    using iterator = T*;
};

template<typename T>
void f(Array<T>::iterator first, Array<T>::iterator last);

template<typename T>
void f(T*, T*);

int main()
{
  f<int&>(0, 0);        // ERROR: substituting int& for T in the first function template
                                          // instantiates Array<int&>, which then fails
}
```

本例与前例最主要的差别在于失败发生的位置。前例中，失败发生在形成一个类型为`typename Container::iterator`之时，它在`begin()`函数模板替换的立即上下文中。而本例中，失败发生在`Array<int&>`的实例化体中，尽管它是由函数模板上下文所触发，但实际上是发生在类模板`Array`的上下文中。因此，SFINAE原则并不适用，编译器会产生一个错误。

这里有一个C++14的例子——基于推导返回类型（P296节15.10.1）——在函数模板定义的实例化时导致错误：

```
template<typename T> auto f(T p) {
  return p->m;
}

int f(...);

template<typename T> auto g(T p) -> decltype(f(p));

int main()
{
  g(42);
}
```

调用`g(42)`会推导`T`为`int`。这使得`g()`声明的替换需要我们去确定`f(p)`的类型（`p`现在已知为类型`int`），然后再确定`f()`的返回类型。`f()`有两个候选者。非模板候选者是匹配的，但它不是一个良选，这是因为它匹配的是一个省略型参数。不幸的是，模板候选者有一个推导的返回类型，因而我们必须实例化它的定义来确定该返回类型。该实例化会因为`p->m`无效而失败（因为`p`是`int`），并且该错误发生在替换上下文之外（因为它在随后的函数定义实例化体中），这就导致本次失败会产生一个错误。为此，我们推荐在可以容易地显式化指定返回类型时，避免使用推导返回类型。

SFINAE设计之初，是旨在消除由函数模板重载所带来的因非意图匹配而产生的奇怪错误，正如容器`begin`这一例子。然而，探测无效表达式或类型的能力可以实现卓越的编译期技巧，以允许我们判断某个特定的语法是否是合法的。这些技巧将在P416节19.4中进行讨论。

在P424节19.4.4中，有一个特别的例子：让类型萃取SFINAE-friendly来避免立即上下文所产生的问题。

15.8 推导的限制

模板实参推导是一个强大的特性，对于大部分函数模板调用来说它消除了显式地指定模板实参的必要性，并且还使能了函数模板重载（见P15节1.5）和类模板偏特化（见P347节16.4）。然而，开发者可能会在使用模板时遇到一些使用上的限制，这些限制会在本节中进行讨论。

15.8.1 合法的实参转换

通常来说，模板推导会尝试去找到一个函数模板参数的替换，使得参数化类型P与类型A等同。然而，当无法达成这一条件，而P在推导上下文中又包含了一个模板参数时，一些差别也可以容忍：

- 如果原始的参数使用了引用声明，被替换的P类型相比A类型可以有进一步的`const/volatile`限定
- 如果A类型是一个指针或是类成员指针类型，它可以通过限定转换（换句话说，就是一种增加`const`或/和`volatile`限定符的转换）来转换成一个替换的P类型。
- 除非推导发生于类型转换操作符模板，替代的P类型可以是A类型的基类或是指向其基类的指针。举个例子：

```
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);         // deduction succeeds with T substituted with long
}
```

如果P在推导上下文中不包含模板参数，那么所有的隐式转换都是合法的。例如：

```
template<typename T> int f(T, typename T::X);

struct V {
  V();
  struct X {
    X(double);
  };
}v;
int r = f(v, 7.0);        // OK: T is deduced to V through the first parameter,
                                        // which causes the second parameter to have type V::X
                                        // which can be constructed from a double value
```

仅当严格匹配不可行时才会考虑宽松的匹配要求。即便附加了这些转换，推导也仅仅在可以找到满足A类型到P类型的合适替换时才会成功。

请注意，这些规则的适用范围相当狭隘，例如它不考虑为使调用成功而可行的函数实参的各种转换。比如，对下面`max()`函数模板的调用（该模板在P269节15.1介绍）：

```
std::string maxWithHello(std::string s)
{
  return ::max(s, "hello");
}
```

这里，模板实参推导根据第一个实参会把`T`推导为`std::string`，而第二个实参会把`T`推导为`char[6]`，所以模板实参推导会失败，这是因为两个参数使用的是同一个模板实参。这种失败可能有些令人诧异，因为字符串字面量`"hello"`可以被隐式转换成`std::string`，并且调用`::max<std::string>(s, "helloa")`是可行的。

或许还有更令人惊讶的：当两个实参有着从公共基类继承下来的不同的类类型时，推导并不会将公共基类作为推导类型的候选者进行考虑。可参考P7节1.2关于这一议题的讨论以及可行的解决方案。

15.8.2 类模板实参

C++17之前，模板实参推导仅仅应用于函数和成员函数模板。特别地，类模板的实参不会根据其中某一个构造器的实参来进行推导。例如：

```
template<typename T>
class S {
  public:
    S(T b) : a(b) {}
  private:
    T a;
};
S x(12);        // ERROR before C++17: the class template parameter T was not deduced from
                        // the constructor call argument 12
```

这一限制在C++17中被解除——参考P313节15.12。

15.8.3 默认调用实参

函数调用的默认实参可以在函数模板中指定，正如普通函数：

```
template<typename T>
void init(T* loc, T const& val = T())
{
  *loc = val;
}
```

事实上，如上例所示，函数调用的默认实参可以依赖于模板参数。这种依赖型默认实参仅在没有提供显式的实参时才会被实例化。这一原则保证了下方示例的合法性：

```
class S {
  public:
    S(int, int);
};

S s(0, 0);

int main()
{
  init(&s, S(7,42));        // T() is invalid for T = S, but the default
                                                    // call argument T() needs no instantiation
                                                  // because  an explicit argument is given
}
```

即使默认实参不具有依赖性，它也依然无法被用于推导模板实参。这意味着在C++中，下面的写法是非法的：

```
template<typename T>
void f(T x = 42)
{
}

int main()
{
  f<int>();         // OK: T = int
  f();                 // ERROR: cannot deduce T from default call argument
}
```

15.8.4 异常规范

与默认实参一样，异常规范也仅仅在它们被需要时才会实例化。这意味着他们不会参与模板实参推导。例如：

```
template<typename T>
void f(T, int) noexcept(nonexistent(T()));        // #1

template<typename T>
void f(T, ...);        // #2 (C-style vararg function)

void test(int i)
{
  f(i, i);        // ERROR: chooses #1, but the expression nonexistent(T()) is ill-formed
}
```

函数标记#1处的`noexcept`规范尝试调用一个`nonexistent`函数。通常来说，函数模板声明中这样的错误会直接触发模板实参推导失败（SFINAE），然后再通过选择标记#2处的函数使用省略型参数匹配是重载决议中最差的匹配，参考附录C）来匹配调用`f(i, i)`。然而，由于异常规范并没有参与到模板实参推导，重载决议还是会选择标记#1，这就导致当`noexcept`规范在随后实例化时，程序出现问题。

相同的规则适用于列出潜在异常类型的异常规范：

```
template<typename T>
void g(T, int) throw(typename T::Nonexistent);        // #1

template<typename T>
void g(T, ...);                                                    // #2

void test(int i)
{
  g(i, i);        // ERROR: chooses #1, but the type T::Nonexistent is ill-formed
}
```

然而，这些“动态的”异常规范自C++11起就不再推荐使用(deprecated)，它们在C++17中被移除。

15.9 显式的函数模板实参

当函数模板实参无法被推导时，通过尾随在函数模板名后显式地指定亦然可行。例如：

```
template<typename T> T default_value()
{
  return T{};
}

int main()
{
  return default_value<int>();
}
```

对可推导的模板参数来说这也是可行的：

```
template<typename T> void compute(T p)
{
  ...
}

int main()
{
  compute<double>(2);
}
```

一旦一个模板实参被显式指定了，其对应的参数就不再被推导。同时，函数调用的参数也被允许进行类型转换（对推导调用来说是不行的）。上例中，实参 `2` 在 `compute<double>(2)` 调用中会被隐式转换成 `double`。

也可以显式指定模板实参的其中一部分。然而，被显式指定的部分必须始终按模板参数从左到右排好顺序。因此，那些不能被推导的（或者最可能被显式指定的）参数应该放在最前面。例如：

```
template<typename Out, typename In>
Out convert(In p)
{
  ...
}

int main()
{
  auto x = convert<double>(42);        // the type of parameter ps is deduced,
                                                                    // but the return type is explic
}
```

有时候，通过指定一个空模板实参列表对于确保所选的函数是一个模板实例也很有用，此时模板实参还是会进行推导：

```
int f(int);                // #1
template<typename T> T f(T);        // #2

int main() {
  auto x = f(42);               // calls #1
  auto y = f<>(42);               // calls #2
}
```

这里`f(42)`会选择非模板函数，因为对于重载决议来说，相比函数模板，它更倾向于选择普通的函数（如果两者是等价的）。然而，对于`f<>(42)`来说，模板实参列表的存在打破了这一规则，非模板函数不再可选（即使没有指定实际的模板实参）。

在友元函数声明的上下文中，显式模板实参列表的存在会产生一个有趣的效用。考虑下面的例子：

```
void f();
template<typename> void f();
namespace N {
  class C {
    friend int f();                         // OK
        friend int f<>();                  // ERROR: return type conflict
  };
}
```

当使用普通的标识符命名一个友元函数时，该函数仅仅会在最近一层的封闭作用域内进行查找，如果没有找到的话，就会在该作用域内声明一个新的实体（但它会保留“不可见性”，除非通过ADL查找；参考P220节13.2.2）。这就是我们的第一个友元声明：在`N`作用域内没有找到`f`的声明，所以会声明一个不可见的`N::f()`。

然而，当使用标识符尾随模板实参列表来命名友元函数时，模板必须在那一刻对一般查找是可见的，一般查找会向上搜索任意层作用域（根据其所需要）。因此，我们第二个声明会找到全局的函数模板`f()`，但是编译器会提出一个错误：返回类型不匹配（由于没有执行ADL，故前一个友元函数的声明会被忽略）。

显式指定的模板实参使用SFINAE法则来替换：如果在某个函数模板替换的立即上下文中出现了错误，那么它就会被丢弃，但是其他模板依然可能会成功。例如：

```
template<typename T> typename T::EType f();        // #1
template<typename T> T f();                                    // #2

int main() {
  auto x = f<int*>();
}
```

这里，#1处候选者在`int*`替换`T`时会失败，但在#2处却会成功，因此也就会选择#2这一候选。事实上，如果在替换之后仅余一个候选者，那么带有显式模板实参的函数模板名称看起来非常像一个普通的函数名称，包括在许多情况下退化为函数指针类型。也就是说，替换上面的`main()`为：

```
int main() {
  auto x = f<int*>;         // OK: x is a pointer to function
}
```

这会产生合法的编译单元。然而，像是下面的例子：

```
template<typename T> void f(T);
template<typename T> void f(T, T);

int main() {
  auto x = f<int*>;                 // ERROR: there are two possible f<int*> here
}
```

这种用法就是非法的，因为`f<int*>`并没有标识着某一个单一的函数。

可变函数模板也可以使用显式模板实参：

```
template<typename ...Ts> void f(Ts... ps);

int main() {
  f<double, double, int>(1, 2, 3);         // OK: 1 and 2 are converted to double
}
```

有趣的是，包可以被部分显式指定、部分显式推导：

```
template<typename ...Ts> void f(Ts... ps);

int main() {
  f<double, int>(1, 2, 3);         // OK: the template arguments are <double, int, int>
}
```

15.10 初始化器和表达式推导

C++11引入了声明这样一种变量的能力：其类型可以根据初始化器推导。C++11也提供了一种机制来表示某个命名实体（变量或函数）或是表达式的类型。这些机制十分易用，C++14和C++17对这一主题又进行了补充。

15.10.1 auto类型指示符

`auto`类型指示符在很多地方有着用武之地（主要是命名空间作用域和局部作用域），它会根据变量的初始化器推导变量类型。此时，`auto`被称作为一个占位符类型（另一个占位符类型是`decltype(auto)`），我们会在P298节15.10.2中对它进行描述。例如：

```
template<typename Container>
void useContainer(Container const& container)
{
  auto pos = container.begin();
  while(pos != container.end()) {
    auto& element = *pos++;
        ... // operate on the element
  }
}
```

上例中的两个`auto`，避免了去书写两个又臭又长的类型名称：容器的迭代器类型和迭代器的值类型：

```
typename Container::const_iterator pos = container.begin();
...
typename std::iterator_traits<typename Container::iterator>::reference  element = *pos++;
```

`auto`的推导机制与模板实参推导机制相同。类型指示符`auto`取代模板类型参数`T`，然后推导可以继续进行，这就好像变量是一个函数参数，而其初始化器是相应的函数实参。对例子中第一个`auto`来说，对应的情景如下：

```
template<typename T> void deducePos(T pos);
deducePos(container.begin());
```

`T`是`auto`要推导的类型。这样做的直接后果之一是，类型为`auto`的变量永远不会是引用类型。第二个`auto`使用了`auto&`来展示了如何产生一个推导类型的引用。它的推导与下面的函数模板和调用等价：

```
template<typename T> deduceElement(T& element);
deduceElement(*pos++);
```

这里，`element`永远是引用类型，它的初始化器无法产生一个临时对象。

组合`auto`与右值引用亦是可行的，但是这样做就让它看起来像是一个转发引用，因为`auto&& r = ...;`的推导模型基于这样一个函数模板：

```
template<typename T> void f(T&& fr);        // auto replaced by template parameter T
```

这就解释了下面的例子：

```
int x;
auto&& rr = 42;         // OK: rvalue reference binds to an rvalue (auto = int)
auto&& lr = x;         // Also OK: auto = int& and reference collapsing makes
                                //                          lr an lvalue reference
```

在泛型代码中，这一技巧经常被用来绑定那些未知的函数或操作符调用结果的值类别（左值或是右值），而无需拷贝它们的结果。例如，常常推荐用这样的方式在循环中声明迭代值：

```
template<typename Container> void g(Container c) {
  for(auto&& x : c) {
    ...
  }
}
```

这里我们不知道容器迭代器接口的签名，但是使用`auto&&`可以让我们确信在迭代时不会引入额外的值拷贝。如果需要完美转发边界值，那么`std::forward<T>()`可以像往常那样对变量使用。这成全了一种“延迟的”完美转发，对此可以参考P167节11.3的示例。

除了引用，我们还可以通过组合使用`auto`，定制出`const`变量、指针或是成员指针等等，但是`auto`必须是其声明的“主”类型。它不能嵌套在模板实参或类型指示符后面的声明符中作为一部分而存在。下面的示例予以了解释：

```
template<typename T> struct X { T const m; };
auto const N = 400u;          // OK: constant of type unsigned int
auto* gp = (void*)nullptr;        // OK: gp has type void*
auto const S::*pm = &X<int>::m;                // OK: pm has type int const X<int>::*
X<auto> xa = X<int>();                        // ERROR: auto in template argument
int const auto::*pm2 = &X<int>::m;        // ERROR: auto is part of the "declarator"
```

至于为什么C++不支持上例中所有的情景，并没有什么技术上的原因，只不过是，C++委员会认为它所带来的额外实现成本以及潜在的滥用性超出了它的收益。

为了避免同时搞晕开发者和编译器，在C++11中古式的`auto`用法（作为一个存储类型指示符而存在）不再被允许（今后也一样）：

```
int g() {
  auto int r = 24;                // valid in C++03 but invalid in C++11
  return r;
}
```

`auto`的古式用法（继承自C语言）一直是冗余的。大多数编译器通常可以将该用途与占位符区别开来（其实大可不必），以提供从旧C++代码到新C++代码的过渡。只不过，`auto`的古式用法在实践中非常罕见。

返回类型的推导

C++14增设了另一个推导`auto`占位符的情景，它出现在函数返回类型。例如：

```
auto f() { return 42; }
```

定义了一个返回类型为`int`的函数（`42`的类型）。它也可以使用尾缀返回类型的语法来表示：

```
auto f() -> auto { return 42; }
```

此时，第一个`auto`宣布了尾缀返回类型，第二个`auto`是一个推导的占位符类型。只不过，没有什么理由去支持这种更啰嗦的语法。

对lambda来说有着相同的默认机制存在：如果没有显式地指定返回类型，lambda表达式返回的类型会按照`auto`来推导：

```
auto lm = [] (int x) { return f(x); };
                // same as: [] (int x) -> auto { return f(x); };
```

函数可以脱离定义而单独声明。对于返回类型需要推导的情景也是一样：

```
auto f();        // forward declaration
auto f() { return 42; }
```

但是，在这种情况下，前向声明的用法非常有限，因为在使用函数的任何位置，该定义都必须可见。也许令人惊讶的是，提供带有“已解决的”返回类型的前向声明是无效的。例如：

```
int known();
auto known() { return 42; }        // ERROR: incompatible return type
```

通常，由于风格上的偏爱，仅在将成员函数定义移到类定义外部时，前向声明推导的返回类型的函数才有实用价值：

```
struct S {
  auto f();                // the definition will follow the class definition
};
auto S::f() { return 42; }
```

可推导的非类型参数

在C++17之前，非类型参数只能通过指定的类型来声明。然而，这一类型可以是一个模板参数类型。例如：

```
template<typename T, T V> struct S;
S<int, 42>* ps;
```

在本例中，需要指定非类型模板实参的类型——即指定`int`和`42`，这可能很乏味。因此，C++17增加了声明非类型模板参数的能力，这些参数的实际类型是从相应的模板实参推导出来的。声明方式如下：

```
template<auto V> struct S;
```

此时就可以写成：`S<42>* ps;`。这里`S<42>`的类型`V`会被推导成`int`，这是因为`42`的类型是`int`。如果我们写作`S<42u>`，那么`V`的类型就会被推导成`unsigned int`(参考P294节15.10.1了解推导`auto`类型指示符的更多细节)。

请注意，对非类型模板参数类型的一般约束仍然有效。例如：

```
S<3.14>* pd;        // ERROR: floating-point nontype argument
```

具有这种可推导的非类型参数的模板定义通常还需要表示对应实参的实际类型。这可以通过`decltype`语法来完成（参考P298节15.10.2）。例如：

```
template<auto V> struct Value {
  using ArgType = decltype(V);
};
```

`auto`非类型模板参数在参数化类成员的模板时也很有用。例如：

```
template<typename> struct PMClassT;
template<typename C, typename M> struct PMClassT<M C::*> {
        using Type = C;
};
template<typename PM> using PMClass = typename PMClassT<PM>::Type;

template<auto PMD> struct CounterHandle {
  PMClass<decltype(PMD)>& c;
  CounterHandle(PMClass<decltype(PMD)>& c) : c(c) {
  }
  void incr() {
    ++(c.*PMD);
  }
};

struct S {
  int i;
};

int main() {
  S s{41};
  CounterHandle<&S::i> h(s);
  h.incr();        // increases s.i
}
```

这里我们使用了一个辅助类模板`PMClassT`的一个偏特化（参考P347节16.4）来借由成员指针类型追踪到它的“父”类类型。有了`auto`模板参数，我们只需要指定成员指针常量`&S::i`作为模板实参。在C++17之前，我们还需要指定一个成员指针类型，譬如`OldCounterHandle<int S::*, &S::i>`，看起来很笨重很冗余。

如你所愿，这一特性也可以为非类型参数包使用：

```
template<auto... VS> struct Values {
};
Values<1, 2, 3> beginning;
Values<1, 'x', nullptr> triplet;
```

`triplet`实例展示了每个非类型参数都可以被单独地推导。与多重可变声明场景（参考P303节

15.10.4）不同的是，这里不需要每个推导都是相同的。

如果我们想强制每个非类型模板参数都相同，也是可以实现的：

```
template<auto V1, decltype(V1)... VRest> struct
HomogeneousValues {
};
```

然而，此场景中模板实参列表不能为空。

可以参考P50节3.4中一个使用了`auto`作为模板参数类型的完整例子。

15.10.2 用`decltype`表示表达式的类型

尽管`auto`的使用可以避免书写变量类型，但若想要使用这一变量类型，就没有那么容易。`decltype`关键字解决了这一问题：它允许开发者对表达式或声明的类型做精确表达。只不过，开发者还是要谨慎对待`decltype`所产生的细微差别，而这取决于传递的参数是声明的实体还是一个表达式：

- 如果`e`是某个实体（诸如变量、函数、枚举或是数据成员）或类成员访问的名称，`decltype(e)`产生的是该实体或表示的类成员的声明类型。因此，`decltype`可以用来检查变量的类型。当你想要完全匹配现有的声明的类型时，这很有用。例如，考虑下面的两个变量`y1`和`y2`：

```
auto x = ...;
auto y1 = x + 1;
decltype(x) y2 = x + 1;
```

由于依赖于`x`的初始化器，`y1`的类型可能与`x`相同、也可能不同：它依赖于`+`的行为。如果`x`被推导为一个`int`，那么`y1`也会是`int`；如果`x`被推导为`char`，`y1`也会是一个`int`，因为`char`和`1`(定义为`int`类型)相加得到一个`int`。对`y2`类型使用的`decltype(x)`保证了`y2`始终与`x`具有相同的类型。

- 否则，如果e是任何其他表达式，则`decltype(e)`将生成一个可以反射出该表达式类型 + 值分类的类型，如下所示：
    - 如果`e`是类型`T`的左值(lvalue)，`decltype(e)`产生的是`T&`。
    - 如果`e`是类型`T`的将亡值(xvalue)，`decltype(e)`产生的是`T&&`。

- 如果`e`是类型`T`的纯右值(prvalue)，`decltype(e)`产生的是`T`。

可以参考附录B关于值分类的详细描述。这些差别可以通过下面的例子来演示：

```
void g(std::string&& s)
{
  // check the type of s:
  std::is_lvalue_reference<decltype(s)>::value;                // false
  std::is_rvalue_reference<decltype(s)>::value;                // true (s as declared)
  std::is_same<decltype(s), std::string&>::value;        // false
  std::is_same<decltype(s), std::string&&>::value;        // true

  // check the value category of s used as expression:
  std::is_lvalue_reference<decltype((s))>::value;        // true (s is an lvalue)
  std::is_rvalue_reference<decltype((s))>::value;        // false
  std::is_same<decltype((s)), std::string&>::value;        // true (T& signals an lvalue)
  std::is_same<decltype((s)), std::string&&>::value;        // false
}
```

前四个表达式中，`decltype`为变量`s`所使用：

```
decltype(s)                // declared type of entity a designated by s
```

这意味着`decltype`产生的是`s`声明的类型——`std::string&&`。后四个表达式中，`decltype`的操作数不是一个名称而是一个表达式`(s)`，名称在小括号中，此时，类型会反映出`(s)`的值类别：

```
decltype((s))        // check the value category of (s)
```

这是一个使用名称去引用变量的表达式，因此它是一个左值。根据上面的规则，这意味着`decltype((s))`是一个`std::string`的（左值）引用。这是C++中为数不多的几处，用括号括起来的表达式除了影响运算符的关联性之外，还可以改变程序的含义。

`decltype`会计算任意表达式`e`的类型这一事实在各个地方都可能有所帮助。具体而言，`decltype(e)`会充分保留表达式的信息，从而可以“完美地”描述返回表达式`e`本身的函数的返回类型：`decltype`会计算该表达式的类型，同时将表达式的值类别回传给函数的调用者。例如，考虑一个简单的转发函数`g()`，它返回被调用的`f()`的返回结果：

```
??? f();

decltype(f()) g()
{
  return f();
}
```

`g()`的返回类型依赖于`f()`的返回类型。如果`f()`返回的是一个`int&`，`g()`的返回类型的计算会首先判断表达式`f()`是否具有类型`int`。该表达式是一个左值，因为`f()`返回的是左值引用，因此`g()`声明的返回类型就会是`int&`。类似地，如果`f()`的返回类型是一个右值引用类型，`f()`的调用就是一个将亡值，而`decltype`会产生一个右值引用类型，这与`f()`返回的类型也完全匹配。本质上，这种形式的`decltype`拿到了任意表达式的主要特征（其类型和值分类），并以能够完美转发返回值的方式在类型系统中对其进行编码。

`decltype`在`auto`无法充分推导出值的场景中也十分有用。例如，假设我们有一个变量`pos`，它是某种未知的迭代器类型，我们希望创建一个变量`element`，该`element`可以通过`pos`解引用来获取。写作：

```
auto element = *pos;
```

然而，这里始终都会对元素进行一次拷贝。如果我们写成`auto& element = *pos;`，那我们拿到的始终是元素的引用，而当迭代器的`operator*`返回的是一个值时，程序就会出错。为了解决这一问题，我们可以用`decltype`去保留迭代器`operator*`所返回结果的值特性或是引用特性：

```
decltype(*pos) element = *pos;
```

当迭代器提供的是引用时，就会产生一个引用类型，否则，就会进行值拷贝。它的主要缺陷在于它需要将初始化表达式书写两次：第一次在`decltype`中（这里不会进行计算），第二次在实际的初始化器中。C++14引入了`decltype(auto)`语法来解决这一问题，我们马上就会讨论到。

15.10.3 decltype(auto)

C++14增加了一个组合使用`auto`和`decltype`的特性：`decltype(auto)`。正如`auto`这一类型指示符一样，它是一个类型占位符，并且变量的类型、返回类型或模板实参的类型由关联的表达式类型（初始化器、返回值或模板实参）确定。然而，与`auto`单单使用模板实参推导法则来确定

类型有所不同，实际的类型是通过对表达式直接应用`decltype`语法来确定的。举个例子来说明：

```
int i = 42;                 // i has type int
int const& ref = i;         // ref hastype int const& and refers to i

auto x = ref;         // x has type int and is a new independent object

decltype(auto) y = ref;         // y has type int const& and also refers to i
```

`y`的类型借由应用于初始化表达式的`decltype`获取，这里`ref`是一个`int const&`。相对地，`auto`类型推导法则产生的则是类型`int`。

另一个例子展示了在索引`std::vector`时的差别：

```
std::vector<int> v = { 42 };
auto x = v[0];                // x denotes a new object of type int
decltype(auto) y = v[0];        // y is a reference (type int&)
```

这就干净利落地解决了前面示例的问题：

```
decltype(*pos) element = *pos;
```

我们可以重写为：

```
decltype(auto) element = *pos;
```

对于返回类型来说它也常常十分便利。考虑下面的例子：

```
template<typename C> class Adapt
{
  C container;
  ...
  decltype(auto) operator[] (std::size_t idx) {
    return container[idx];
  }
};
```

如果`container[idx]`产生的是左值，我们希望传递左值给调用者（调用者应该希望拿到地址来修改它）：此时需要一个左值引用类型，`decltype(auto)`可以解析出来。如果产生的是一个纯右值，那么引用类型会导致引用悬挂，但是幸运的是，在这种情景下，`decltype(auto)`会产生一个对象类型（而非引用类型）。

与`auto`不一样的是，`decltype(auto)`不允许使用声明指示符去修改它的类型。例如：

```
decltype(auto)* p = (void*)nullptr;        // invalid
int const N = 100;
decltype(auto) const NN = N*N;               // invalid
```

同时也请注意初始化器中的小括号可能很关键（因为它们对`decltype`结构来说本身很关键，如P91节6.1所讨论）：

```
int x;
decltype(auto) z = x;        // object of type int
decltype(auto) r = (x);        // reference of type int&
```

这尤其意味着括号可能对return语句的有效性产生严重影响：

```
int g();
...
decltype(auto) f() {
  int r = g();
  return (r);                // run-time ERROR: returns reference to temporary
}
```

自C++17起，`decltype(auto)`还可以对可推导的非类型参数使用（见P296节15.10.1）。下面的

例子进行了演示：

```
template<decltype(auto) Val> class S
{
  ...
};
constexpr int c = 42;
extern int v = 42;
S<c> sc;        // #1 produces S<42>
S<(v)> sv;        // #2 produces S<(int&)v>
```

在#1处，`c`没有小括号包裹，推导出的类型就是`c`类型本身（即`int`）。因为`c`是`42`的常量表达式，它就等价于`S<42>`。在#2处，小括号的包裹导致`decltype(auto)`会推导出一个引用类型`int&`，它可以绑定到全局变量`v`（类型为`int`）。因此，这样声明的类模板会依赖于`v`的引用，`v`值的改变都会影响类`S`的行为（参考P167节11.4了解更多细节）。（`S<v>`如果没有小括号的话，会产生一个错误，因为`decltype(v)`是一个`int`，此时期望的是一个类型为`int`的常量实参值。然而，`v`并不是一个常量`int`值。）

请注意，两种情况的性质有所不同。因此，我们认为像这种非类型模板参数可能会引起意外，并且预计不会被广泛地使用。

最后，给出关于在函数模板中使用推导的非类型参数的注解：

```
template<auto N> struct S {};
template<auto N> int f(S<N> p);
S<42> x;
int r = f(x);
```

本例中，函数模板`f<>()`的参数`N`的类型由`S`的非类型参数类型推导。这是可行的，因为形如`X<...>`的名称(`X`是一个类模板)是一个可推导上下文。

然而，也有一些模式是无法被推导的：

```
template<auto V> int f(decltype(V) p);
int r1 = deduce<42>(42);        // OK
int r2 = deduce(42);                // ERROR: decltype(V) is a nondeduced context
```

本例中，`decltype(V)`是一个不可推导上下文：并没有匹配实参`42`的独一无二的`V`值（例如，`decltype(7)`与`decltype(42)`产生相同的类型）。因此，非类型模板参数必须被显式地指定，才能使函数调用变得可行。

15.10.4 auto推导的特殊情景

除却简单的`auto`推导规则，还存在着一些特殊的情景。其一发生于变量的初始化器是一个初始化列表的场景。对应的函数调用推导必定失败，这是因为我们无法通过初始化列表实参来推导出一个模板参数的类型：

```
template<typename T>
void deduceT(T);
...
deduceT({2,3,4});        // ERROR
deduceT({1});               // ERROR
```

然而，如果我们的函数有着如下更特定的参数：

```
template<typename T>
void deduceInitList<std::initializer_list<T>);
...
deduceInitList({2, 3, 5, 7});        // OK: T deduced as int
```

那么推导就会成功。使用初始化列表来拷贝初始化（即，使用=初始化）一个`auto`变量就定义而言，可以写成更加具体的参数：

```
auto primes = { 2, 3, 5, 7};        // primes is std::initializer_list<int>
deduceT(primes);                                  // T deduced as std::initialize_list<int>
```

在C++17之前，`auto`变量与之对应的直接初始化（即，不使用=）也可以像这样处理，但是在C++17中对此进行了调整，以更好地满足大部分开发者所期望的行为：

```
auto oops { 0, 8, 15 }; // ERROR in C++17
auto val {2};                        // OK: val has type int in C++17
```

在C++17之前，两种初始化都是合法的，`oops`和`val`都会由类型`initializer_list<int>`进行初

始化。

有趣的是，为具有推导占位符类型作为返回类型的函数，返回一个花括号初始化列表是不合法的：

```
auto subtleError() {
  return { 1, 2, 3 }; // ERROR
}
```

这是因为函数作用域中的初始化列表是一个对象，它指向更底层的数组对象（每个元素值在列表中指定），在函数返回时它就过期了。允许这一语法通行就相当于认可悬垂引用的有效性。

另一种特殊的场景发生在多个变量使用同一个`auto`进行声明的地方，如下所示：

```
auto first = container.begin(), last = container.end();
```

此处，推导会为每个声明独立进行。换句话说，这里会为`first`引入模板类型参数`T1`，为`last`引入另一个模板类型参数`T2`。当且仅当两个推导都成功，且`T1`和`T2`具有相同的推导类型时，这些声明才是合法的。这会滋生一些有趣的案例：

```
char c;
auto *cp = &c, d = c;         // OK
auto e = c, f = c+1;         // ERROR: deduction mismatch char vs int
```

这里，共享的`auto`声明了两对变量。`cp`和`d`推导出同样的类型`char`，因此代码有效。然而`f`和`e`的声明却因为计算`c+1`时`char`和`int`的型别提升，导致推导结果不一致而最终产生错误。

推导返回类型的占位符也可能会出现某种平行的特殊情况。考虑下面的例子：

```
auto f(bool b) {
  if (b) {
    return 42.0;         // deduces return type double
  } else {
    return 0;                // ERROR: deduction conflict
  }
}
```

本例中，每个返回语句都会独立进行推导，但是二者推导的结果却不一致，因此程序非法。若返回表达式对该函数进行递归调用，此时推导不会进行，除非前面的推导已经确定了返回类型，否则程序依然不合法。这就意味着下面的代码不合法：

```
auto f(int n)
{
  if (n > 1) {
    return n * f(n-1);        // ERROR: type of f(n-1) unknown
  } else {
    return 1;
  }
}
```

但是下面的这段等价代码却合法：

```
auto f(int n)
{
  if (n <= 1) {
    return 1;                       // return type is deduced to be int
  } else {
    return n*f(n-1);        // OK: type of f(n-1) is int and so is type of n*f(n-1)
  }
}
```

推导的返回类型还有另一种特殊的情景，即推导的变量类型或推导的非类型参数类型中没有对应项：

```
auto f1() { }                       // OK: return type is void
auto f2() { return; }        // OK: return type is void
```

但是 `f1()` 和 `f2()` 都是合法的，并且推导出一个 `void` 返回类型。然而，如果返回类型的样式不匹配 `void`，比如这样的情景就是非法的：

```
auto* f3() { }        // ERROR: auto* cannot deduce as void
```

如你所愿，使用了推导返回类型的任何函数模板都需要该模板的立即实例化以确定返回类型。然

而，出现SFINAE（参考P129节8.4和P284节15.7）时会产生一个令人惊讶的后果。考虑下面的例子：

deduce/resulttypetmpl.cpp

```
template<typename T, typename U>
auto addA(T t, U u) -> decltype(t+u)
{
  return t + u;
}

void addA(...);
template<typename T, typename U>
auto addB(T t, U u) -> decltype(auto)
{
  return t + u;
}

void addB(...);

struct X{
};

using AddResultA = decltype(addA(X(), X()));        // OK: AddResultA is void
using AddResultB = decltype(addB(X(), X()));        // ERROR: instantiation of addB<X>
                                                                                    //
```

这里相比`decltype(t+u)`，`addB()`所使用的`decltype(auto)`会在重载决议期间引起一个错误：`addB()`模板函数体必须被完全实例化以确定其返回类型。而调用`addB()`的实例化体并不在立即上下文中（参考P285节15.7.1），因此不会被SFINAE筛出，而是产生了一个错误。因此请千万牢记：推导返回类型绝不仅仅是一个复杂的显式返回类型的缩写，它们在使用上要非常小心（即，要理解它们不应该在依赖于SFINAE的其他函数模板签名中被调用）。

15.10.5 结构化绑定

C++17增加了一种新的特性，名为结构化绑定(structured bindings)。它常常使用一个小例子来介绍：

```
struct MaybeInt { bool valid; int value; }
MaybeInt g();
auto const&& [b, N] = g();        // binds b and N to the members of the result of g()
```

调用`g()`产生了一个值（在本例中，是一个简单的聚合类类型MaybeInt），它可以被分解成“元素”（即`MaybeInt`的数据成员）。该调用产生的值就好像有一个标识符中括号列表`[b, N]`被不同的变量名所替换。假设该名称为`e`，那么初始化就等同于：

```
auto const&& e = g();
```

然后中括号中的每个标识符会绑定到`e`的对应元素上。因此，你可以认为`[b, N]`就是`e`中标识符的每个名字（我们会在下面讨论绑定的细节）。

语法上，结构化绑定必须总是有一个`auto`类型，它可以使用`const`或`volatile`限定符以及`&`和`&&`声明符来扩展（但是不能用`*`指针声明符或是其他结构）。它的后面跟随着一个中括号列表，其中至少得有一个标识符（让人想起lambda表达式的捕获列表）。后面必须要有一个初始化器。

三种不同类别的实体可以对结构化绑定进行初始化：

1. 第一种是简单的类类型，其中所有的非静态数据成员都是public权限（如上例）。为了应用这一场景，所有的非静态数据成员都必须是public权限（要么全部直接属于类本身，要么全部属于相同的、明确的公共基类；不得涉及匿名联合体）。在这种情况下，带括号的标识符的数量必须等于成员的数量，并且在结构化绑定范围内使用这些标识符之一就等于使用由`e`表示的对象的相应成员（具有所有相关属性；例如，如果相应的成员是位字段，则无法获取其地址）。
2. 第二种是数组。考虑下例：

```
int main() {
  double pt[3];
  auto& [x, y, z] = pt;
  x = 3.0; y = 4.0; z = 0.0;
  plot(pt);
}
```

一点都不奇怪，中括号中的初始化器只是未命名数组变量的相应元素的简写形式。数组元素的数

量必须等于括号内的初始化器的数量。

还有另一个例子：

```
auto f() -> int(&)[2];         // f() returns reference to int array
auto [ x, y ] = f();          // #1
auto& [ r, s ] = f();          // #2
```

行#1是特别的：通常来说，上面描述的实体`e`应该按照下面的形式来推导：

```
auto e = f();
```

这种推导会将数组的引用退化为指向数组的指针，但是数组的结构化绑定却并不会退化。反之，`e`被推导为一个数组类型的变量，类型与初始化器一致。此后该数组从初始化器中逐个元素拷贝：对于内置数组来说这是个不太寻常的概念。最后，`x`和`y`分别成为了表达式`e[0]`和`e[1]`的别名。

而行#2处则没有引入数组拷贝，它也遵循`auto`的法则。因此假想的`e`按照如下方式声明：

```
auto& e = f();
```

它会得到一个数组引用，`x`和`y`再次分别成为表达式`e[0]`和`e[1]`的别名（调用`f()`所返回数组的成员左值引用）。

3. 最后，第三个选项是允许类似`std::tuple`的类拥有通过模板基础协议`get<>`分解元素的能力。这里我们把`E`视为表达式`(e)`的类型（`e`的概念同上）。由于`E`是表达式的类型，它永远不会是一个引用类型。如果表达式`std::tuple_size<E>::value`是一个合法的整型常量表达式，它必须与中括号标识符的数量相等（并且协议会乱入，优先于选项一，但不优先于数组的选项二）。让我们用n0，n1，n2等表示括号中的标识符。如果`e`具有名为`get`的任何成员，则行为就像将这些标识符按如下声明：

```
std::tuple_element<i, E>::type& ni = e.get<i>();
```

如果`e`被推导为拥有引用类型，或是：

```
std::tuple_element<i, E>::type&& ni = e.get<i>();
```

如果`e`没有成员`get`，则相应的声明会变成：

```
std::tuple_element<i, E>::type& ni = get<i>(e);
```

或是

```
std::tuple_element<i, E>::type&& ni = get<i>(e);
```

`get`只会在关联的类和命名空间中查找。（在所有情景中，`get`都被假设为一个模板，因此跟随的`<`是一个尖括号（而非小于号）。）`std::tuple`，`std::pair`和`std::array`模板都实现了这一协议，下面的代码因此而合法：

```
#include <tuple>

std::tuple<bool, int> bi{true, 42};
auto [b, i] = bi;
int r = i;        // initializes r to 42
```

然而，对于添加`std::tuple_size`，`std::tuple_element`的特化并不困难，函数模板或是成员函数模板`get<>()`会让这一机制对任何类或枚举类型都能正常工作。例如：

```
#include <utility>

enum M {};

template<> class std::tuple_size<M> {
  public:
    static unsigned const value = 2;        // map M to a pair of values
};

template<> class std::tuple_element<0, M> {
  public:
    using type = int;                        // the first value will have type int
};

template<> class std::tuple_element<1, M> {
  public:
    using type = double;                // the second value will have type double
};

template<int> auto get(M);
template<> auto get<0>(M) { return 42; }
template<> auto get<1>(M) { return 7.0; }

auto [i, d] = M();        // as if: int&& i = 42; double&& d = 7.0;
```

注意，你只需要包含`<utility>`头文件来使用两个类元组（tuple-like）的访问协助函数`std::tuple_size<>`和`std::tuple_element<>`。

此外，还要注意上述的第三种情况（使用类元组协议）会执行一个真实的中括号初始化并绑定到实际的引用变量上；它们不是另一个表达式的别名（与第一、二类的类类型和数组的情况有所不同）。这很有趣，因为该引用初始化可能出错；例如，它可能会抛出异常，而异常如今是不可避免的。然而，C++标准化委员会也曾就不要关联标识符与初始化的引用进行过讨论，但是最后还是对每个标识符使用了`get<>()`表达式。这就使得结构化绑定在使用时，“第一个”值必须在“第二个”值被访问前进行测试（例如，基于`std::optional`）。

15.10.6 泛型lambda

lambda一经问世，很快就成了C++11中最流行的特性，一部分原因在于它们显著地简化了C++

标准库和许多其他流行的C++库中仿函数结构(functional constructs)的使用，而这归功于lambda简洁的语法。然而，在模板中lambda变得非常繁琐，这是因为它需要拼出参数和返回类型。例如，考虑这样一个函数模板，它在一个序列中寻找第一个负数值：

```
template<typename Iter>
Iter findNegative(Iter first, Iter last)
{
  return std::find_if(first, last, [](typename std::iterator_traits<Iter>::value_type value) { ret
}
```

在这一函数模板中，lambda最复杂的一部分就是它的参数类型。C++14引入了泛型lambda的概念，使得一个或多个参数类型可以使用`auto`来推导，而不用具体的写出：

```
template<typename Iter>
Iter findNegative(Iter first, Iter last)
{
  return std::find_if(first, last, [] (auto value) { return value < 0; });
}
```

对lambda参数`auto`的处理与使用初始化器的变量类型的`auto`处理相似：它同样由一个引入的模板类型参数`T`来取缔。然而，与变量场景不同的是，推导不会立刻执行，这是因为在lambda被创建的时候实参还是未知的。反之，lambda本身是个泛型，引入的模板类型参数被添加到了它的模板参数列表中。因此，上面例子的lambda可以使用任何实参类型来调用，只要该实参类型支持`< 0`操作且其结果可以被转换为`bool`即可。举个例子，这一lambda可以被`int`或是`float`值来调用。

为了理解lambda泛型的意义，我们先考虑一个非泛型lambda的实现模型：

```
[] (int i) {
  return i < 0;
}
```

C++编译器将该表达式翻译成一个新发明的lambda特定类类型的实例。这一实例被称作闭包(closure)或闭包对象(closure object)，类类型被称作闭包类型(closure type)。闭包类型有一个函数调用操作符，因此该闭包就是一个函数对象。对于这一lambda来说，闭包类型可能类似下面的类定义（为了方便与简洁，我们省略了函数到函数指针值的转换）：

```
class SomeCompilerSpecificNameX
{
  public:
    SomeCompilerSpecificNameX();        // only callable by the compiler
        bool operator() (int i) const
        {
          return i < 0;
        }
}
```

如果你检查lambda的类型分类，`std::is_class<>`始终会返回`true`（参考P705节D.2.1）。

因此，lambda表达式生成的是该类（闭包类型）的对象。例如：

```
foo(..., [] (int i) { return i < 0; });
```

创建了一个编译器内部特定的类`SomeCompilerSpecificNameX`的闭包对象：

```
foo(..., SomeCompilerSpecificNameX{});        // pass an object of the closure type
```

如果lambda想要捕获局部变量：

```
int x, y;
...
[x, y](int i) {
  return i > x && i < y;
}
```

这些捕获将被设计成相关类类型的初始化成员：

```
class SomeCompilerSpecificNameY {
  private:
          int _x, _y;
  public:
    SomeCompilerSpecificNameY(int x, int y) // only callable by the compiler
        : _x(x), _y(y) {
        }

        bool operator() (int i) const {
          return i > _x && i < _y;
        }
};
```

对泛型lambda来说，函数调用操作符是一个成员函数模板，所以我们简单的泛型lambda:

```
[] (auto i) { return i < 0; }
```

会被转移成下面的类（同样地，忽略了函数转换，在泛型lambda场景中它是一个转换函数模板）：

```
class SomCompilerSecificNameZ {
  public:
    SomeCompilerSpecificNameZ();        // only callable by compiler
        template<typename T>
        auto operator() (T i) const {
          return i < 0;
        }
};
```

成员函数模板会在闭包被调用时进行实例化，而不是在lambda表达式出现的地方。例如：

```
#include <iostream>

template<typename F, typename... Ts> void invoke (F f, Ts... ps) {
  f(ps...);
}

int main()
{
  invoke([](auto x, auto y) {
    std::cout << x+y << '\n'
  }, 21, 21);
}
```

这里，lambda表达式出现于`main()`中，所以这里会创建一个关联的闭包。然而，闭包的调用操作符并没有在此处实例化。反之，`invoke()`函数模板使用了闭包类型作为第一个参数类型，`int`作为第二、第三个参数类型进行了实例化。`invoke`的实例化被称为闭包的拷贝（依然是一个与原始lambda关联的闭包），并且它实例化了`operator()`闭包模板来满足实例化调用`f(ps...)`。

15.11 别名模板

别名模板的推导是“透明的“。这意味着当别名模板与模板实参一起出现时，别名的定义（即=右侧的类型）就会被实参所替换，产生的结果正是为推导所用。例如，模板实参推导对下面的三个调用都会成功：

deduce/aliastemplate.cpp

```
template<typename T, typename Cont>
class Stack;

template<typename T>
using DequeStack = Stack<T, std::deque<T>>;

template<typename T, typename Cont>
void f1(Stack<T, Cont>);

template<typename T>
void f2(DequeStack<T>);

template<typename T>
void f3(Stack<T, std::deque<T>);        // equivalent to f2

void test(DequeStack<int> intStack)
{
  f1(intStack);         // OK: T deduced to int, Cont deduced to std::deque<int>
  f2(intStack);         // OK: T deduced to int
  f3(intStack);         // OK: T deduced to int
}
```

在第一个调用中(`f1()`)，`intStaack`对别名模板`DequeStack`的使用对推导没有作用：指定类型`DequeStack<int>`被视为类型`Stack<int, std::deque<int>>`。

第二和第三个调用推导行为是一致的，因为`f2()`的`DequeStack<T>`和`f3()`的`Stack<T, std::deque<T>>`是等价的。对模板实参推导的目标来说，模板别名是透明的：它们可以用来区分和简化代码，但是对于推导如何进行没有任何影响。

请注意，这是因为别名模板不能特化（参考章节16了解模板特化这一话题的更多细节）才行得通。假设下面的代码可行：

```
template<typename T> using A = T;
template<> using A<int> = void;                 // ERROR, but suppose it were possible...
```

此时，我们无法将`A<T>`与`void`类型匹配，并得出结论`T`必须为`void`，因为`A<int>`和`A<void>`都等价于`void`。不可能做到这一点的事实保证，别名的每次使用都可以根

据其定义进行一般性的扩展，从而使别名可以进行透明地推导。

15.12 类模板实参推导

C++17引入了一种新的推导：从变量声明的初始化器或函数类型转换的指定参数中推导类类型的模板参数。例如：

```
template<typename T1, typename T2, typename T3 = T2>
class C
{
  public:
  // constructor for 0, 1, 2, or 3 arguments:
  C(T1 x = T1{}, T2 y = T2{}, T3 z = T3{});
  ...
};
C c1(22, 44.3, "hi");        // OK in C++17: T1 is int, T2 is double, T3 is char const*
C c2(22, 44.3);                        // OK in C++17: T1 is int, T2 and T3 are double
C c3("hi", "guy");                // OK in C++17: T1, T2, and T3 are char const*
C c4;                                        // ERROR: T1 and T2 are undefined
C c5("hi");                                // ERROR: T2 is undefined
```

请注意，所有的参数都必须由推导过程或默认实参来确定。显式地指定一部分参数并推导剩下的参数是行不通的。例如：

```
C<string> c10("hi", "my", 42);        // ERROR: only T1 explicitly specified, T2 not deduced
C<> c11(22, 44.3, 42);                          // ERROR: neither T1 nor T2 explicitly specified
C<string, string> c12("hi", "my");        // OK: T1 and T2 are deduced, T3 has default
```

15.12.1 推导指引

考虑P288节15.8.2的一个示例，我们略施一点点变化：

```
template<typename T>
class S {
  private:
    T a;
  public:
    S(T b) : a(b) {
        }
};

template<typename T> S(T) -> S<T>;        // deduction guide

S x{12};                // OK since C++17, same as S<int> x{12};
S y(12);                // OK since C++17, same as S<int> y(12);
auto z = S{12};        // OK since C++17, same as: auto z = S<int>{12};
```

新增的这种模板风格的结构叫做推导指引。它看起来有点像函数模板，但是它与函数模板在语法上有很多不同：

- 看起来像尾缀返回类型的部分不能写成一个传统的返回类型。我们称这个指定的类型（本例中为`S<T>`）指引类型(guided type)。
- 没有前导`auto`关键字来指示尾缀返回类型。
- 推导指引的“名称”必须是同作用域内更早出现的类模板的非受限名称。
- 指引的指引类型必须是一个模板ID，它的模板名称与指引名称一致。
- 可以使用`explicit`说明符声明。

在`S x(12);`这一声明中，说明符`S`被称为占位类类型(placeholder class type)。当使用这样的占位符时，被声明的变量名称必须紧随其后，并且后面一定要有初始化器。下面的代码是非法的：

```
S *p = &x;                // ERROR: syntax not permitted
```

如上例所书写的指引，声明`S x(12);`通过将与类`S`的推导指引视为重载集合，并尝试使用初始化器针对该重载集合来进行重载决议，对变量的类型进行推导。在这一场景中，集合内仅仅有一个指引在其中，它会成功地推导`T`为`int`，指引的指引类型为`S<int>`。这一指引类型因此被选为声明的类型。

请注意，如果类模板名称后面的多个声明都需要推导，那么每个声明都需要产生相同的类型。例如，使用上面的声明：

```
S s1(1), S2(2.0);        // ERROR: deduces S both as S<int> and S<double>
```

这与C++11中`auto`占位符类型的限制相似。

在前面的例子中，我们声明的推导指引与类`S`中声明的构造函数`S(T b)`之间有一个隐式的联系。然而，这种联系并不是必要的，这意味着推导指引也可以为聚合类模板所使用：

```
template<typename T>
struct A {
  T val;
};

template<typename T> A(T) -> A<T>;        // deduction guide
```

如果没有推导指引，我们必须始终显式地指定模板实参（即使在C++17中也一样）：

```
A<int> a1{42};                // OK
A<int> a2(42);                // ERROR: not aggregate initialization
A<int> a3 = {42};         // OK
A a4 = 42;                          // ERROR: can't deduce type
```

但是如果有了上面的指引，就可以写成：

```
A a4 = {42};                // OK
```

这里有一个微妙之处在于，初始化器必须也是一个合法的聚合类初始化器，也就是说，它必须是一个花括号初始化列表。下面的一些替换是不被允许的：

```
A a5(42);                // ERROR: not aggregate initialization
A a6 = 42;                // ERROR: not aggregate initialization
```

15.12.2 隐式推导指引

通常，对于类模板中的每个构造函数都需要一个推导指引。这使得类模板实参推导的设计者为推导引入了一种隐形机制。为类的主模板的每个构造函数和构造函数模板都引入了一个等价的隐式推导指引，如下所述：

- 隐式指引的模板参数列表由类模板的模板参数、构造函数模板的模板参数（构造函数模板的场合）构成。构造函数模板的模板参数会保留任何默认实参。
- 指引的“类函数”参数会从构造函数或构造函数模板中拷贝。
- 指引的指引类型就是模板的名称，其参数是从类模板中获取的模板参数。

让我们应用到一个原始的类模板示例：

```
template<typename T>
class S {
  private:
    T a;
  public:
    S(T b) : a(b) {
        }
};
```

模板参数列表为`typename T`，类函数参数列表就是`(T b)`，指引类型也就是`S<T>`。因此，我们获得了一个指引，它与我们此前书写的那个用户声明的指引等价：即，为了达成我们想要的效果，该指引完全不必要！也就是说，仅书写原始的简单类模板（无需推导指引），我们还是可以有效地写成`S x(12);`，其中`x`的类型依然是期望的`S<int>`。

推导指引有一个不幸的歧义。考虑一下我们简单的类模板`S`和下面的实例化语句：

```
S x{12};        // x has type S<int>
S y{s1}; S z(s1);
```

我们已经看到了`x`有着类型`S<int>`，但是`x`和`y`应该是什么类型呢？这两种类型直觉上应该是`S<S<int>>`和`S<int>`。委员会在富有争议的情况下决定，这两种情况下都应为`S<int>`。为什么这是有争议的呢？考虑使用`vector`类型的一个相似的例子：

```
std::vector v{1, 2, 3};                 // vector<int>, not surprising
std::vector w2{v, v};                 // vector<vector<int>>
std::vector w1{v};                        // vector<int>!
```

换句话说，拥有单个元素的花括号初始化器的推导与拥有多个元素的花括号初始化器有所差别。通常来说，人们只希望要其中的某一个结果，但是两者确并不一致。然而在泛型代码中，很容易忽视这一细小的差别：

```
template<typename T, typename... Ts>
auto f(T p, Ts... ps) {
  std::vector v{p, ps...};                // type depends on pack length
  ...
}
```

这里当`T`被推导为`vector`类型时，`v`在`ps`参数包为空或非空的情景下，`v`的类型是不一样的。

隐式模板指引本身的添加并没有争议。反对将它们引入的主要观点是该功能会自动将接口添加到现有库中。为了理解这一说法，再次考虑我们前面的类模板`S`。它的定义自C++引入类模板时就是有效的。假设，`S`的作者扩展了库，让`S`以更缜密的方式定义：

```
template<typename T>
struct ValueArg {
  using Type = T;
};

template<typename T>
class S {
  private:
    T a;
  public:
    using ArgType = typename ValueArg<T>::Type;
        S(ArgType b) : a(b) {
        }
};
```

在C++17之前，这样的转变（不太常见）不会影响现有的代码。然而，在C++17中它们禁用了隐式推导指引。让我们书写一个与隐式推导指引相仿的推导指引：模板参数列表和指引类型无需改变，但是类函数参数现在需要写成`ArgType`的形式，也就是`typename ValueArg<T>::Type`：

```
template<typename> S(typename ValueArg<T>::Type) -> S<T>;
```

回想一下P271节15.2，类似`ValueArg<T>::`的名称限定符不是一个推导上下文。因此这种形式的推导指引是没有用的，它无法解析`S x(12);`这样的声明。换句话说，库的作者执行了这一转换可能会破坏其在C++17中的客户端代码。

这种情况下库的作者要怎么办呢？我们的建议就是小心地考虑每一个构造函数，在库剩余的生命期内是否希望它作为隐式推导指引的来源。如果不希望，就用诸如`typename ValueArg<X>::Type`来替换每一个可推导的类型为`X`的构造函数参数的实例。很不幸，没有更简单的方法去把隐式推导指引摘除。

15.12.3 其他细微之处

注入式类名称

考虑下例：

```
template<typename T> struct X {
  template<typename Iter> X(Iter b, Iter e);
  template<typename Iter> auto f(Iter b, Iter e) {
    return X(b, e);                // What is this?
  }
};
```

这段代码在C++14中是合法的：`X(b, e)`中的`X`是注入式类名称，在该上下文中等价于`X<T>`（参考P221节13.2.3）。然而，对类模板实参推导这一规则来说，`X`会自然而然地等价于`X<Iter>`。

为了保留向后兼容性，类模板实参推导在模板名称是注入式类名称的场合下会被禁用。

转发引用

思考另一个例子：

```
template<typename T> struct Y {
  Y(T const&);
  Y(T&&);
};
void g(std::string s) {
  Y y = s;
}
```

显然，这里的目的是通过拷贝构造函数所关联的隐式推导指引架构`T`推导为`std::string`。然而，将隐式推导指引显式地声明出来反而发生令人惊讶的事：

```
template<typename T> Y(T const&) -> Y<T>;        // #1
template<typename T> Y(T&&) -> Y<T>;                // #2
```

回想P277节15.6中模板实参推导的`T&&`的行为：作为一个转发引用，如果调用实参是一个左值类型，那么`T`也会被推导成引用类型。在上例中，推导过程中的实参就是表达式`s`，它是一个左值。隐式指引#1会把`T`推导为`std::string`，但是需要的实参会被调整成`std::string const`。而指引#2则会将`T`推导成一个引用类型`std::string&`并产生一个相同类型的参数（这是因为引用折叠法则），这是一个更好的匹配候选，因为无需对类型添油加醋，附上一个`const`属性。

这一结果可能会令人惊讶，也可能会造成实例化错误（当类模板参数在不允许引用类型的上下文中使用时），更有甚者，会静默地生成非预期的实例（比如，生成悬垂引用）。

C++标准委员会因此决定，对于隐式推导指引，如果`T`是一个类模板参数（与构造函数模板参数对应；为那些特殊的推导规则而保留），在执行`T&&`的推导时，特殊的推导规则会被禁用。因此上面的例子可以将`T`推导为`std::string`，如你所愿。

explicit关键字

推导指引可以使用关键字`explicit`修饰。此时它仅仅会考虑直接的初始化场景，而不会考虑拷贝初始化场景。例如：

```
template<typename T, typename U> struct Z {
  Z(T const&);
  Z(T&&);
};

template<typename T> Z(T const&) -> Z<T, T&>;                // #1
template<typename T> explicit Z(T&&) -> Z<T, T>;        // #2

Z z1 = 1;         // only considers #1; same as: Z<int, int&> z1 = 1;
Z z2{2};        // prefers #2; same as: Z<int, int> z2{2};
```

注意这里的`z1`初始化使用了拷贝初始化，因此声明了`explicit`的推导指引#2就不会被考虑。

拷贝构造和初始化列表

考虑下面的类模板：

```
template<typename ...Ts> struct Tuple {
  Tuple(Ts...);
  Tuple(Tuple<Ts...> const&);
};
```

为了理解隐式指引的效果，我们用显式地声明它们：

```
template<typename... Ts> Tuple(Ts...) -> Tuple<Ts...>;
template<typename... Ts> Tuple(Tuple<Ts...> const&) -> Tuple<Ts...>;
```

现在看看下面的例子：

```
auto x = Tuple{1,2};
```

这显然会选择第一个指引，因此第一个构造函数：`x`就是一个`Tuple<int, int>`。让我们继续看看下面的例子，它们使用了`x`拷贝的语法：

```
Tuple a = x;
Tuple b(x);
```

对`a`和`b`来说，两个指引都可以匹配。第一个指引会选择类型`Tuple<Tuple<int, int>>`，拷贝构造器关联的指引会生成`Tuple<int, int>`。幸运的是，第二个指引更加匹配，因此`a`和`b`都会从`x`拷贝构造出来。

现在。考虑使用花括号列表的例子：

```
Tuple c{x, x};
Tuple d{x};
```

例子中的第一个`x`仅仅可以匹配第一个指引，因此会产生`Tuple<Tuple<int,int>, Tuple<int, int>>`。这完全符合直觉，不足为奇。第二个示例则会将`d`推导为类型`Tuple<Tuple<int>>`。然而，它被视为一个拷贝构造（即，更倾向于第二个隐式指引）。这也会发生在functional-notation转换的场景：

```
auto e = Tuple{x};
```

这里，`e`被推导为一个`Tuple<int, int>`，而非`Tuple<Tuple<int>>`。

指引仅为推导所用

推导指引并非函数模板：它们仅仅用来推导模板参数，并不会被“调用”。这意味着不论是通过引用还是通过值来传递实参对指引声明并不重要。例如：

```
template<typename T> struct X {
  ...
};

template<typename T> struct Y {
  Y(X<T> const&);
  Y(X<T>&&);
};

template<typename T> Y(X<T>) -> Y<T>
```

注意看推导指引并没有完全与`Y`的两个构造函数保持一致。然而，这并没有什么关系，因为指引仅仅为推导所用。给定类型为`X<TT>`的`xtt`左值或是右值，它都会选择推导类型`Y<TT>`。然后，初始化会在`Y<TT>`的构造器上执行重载决议以判断需要调用哪一个（这取决于`xtt`是左值还是右值）。

15.13 后记

函数模板的模板实参推导本就是C++原始设计的一部分。实际上，显式模板实参的使用在很多年之后才成了C++的一部分。

SFINAE是一个术语，它在本书的第一版就介绍过了。这一术语很快就在C++开发者委员会中盛行。然而，在C++98中，SFINAE并没有那么强大：它仅仅适用于一个有限的类型操作符集合，并且没有覆盖任意表达式或访问控制。由于越来越多的技术开始依赖于SFINAE（参考P416节19.4），推广SFINAE显而易见。Steve Adamczyk和John Spicer开发了在C++11中实现的措辞（见论文N2634）。尽管标准中的措词更改相对较小，但事实证明某些编译器的实现工作量不成比例。

`auto`类型指示符以及`decltype`语法最早在C++03中新增，但最终是C++11才正式引入。它们率先由Bjarne Stroustrup和Jaakko Jarvi发明（详见他们的论文N1607和N2343，里面分别有`auto`类型指示符和`decltype`）。

Stroustrup在他的原始C++实现（Cfront）中就已经考虑过`auto`语法。这一特性在C++11中引入，`auto`作为一个存储指示符的原始意义（从C语言继承）被保留下来，所以需要一个没有歧义的规则来决定该关键字应该如何解析。在Edison Design Group的前端实现这一特性的过程中，David Vandevoorde发现对于C++11开发者来说这可能会产生很多意外（N2337）。在审查了这一议题后，标准委员会决定抛弃`auto`的传统使用方法（在C++03程序中使用`auto`关键字的任何地方，都可以忽略它），见论文N2546（David Vandevoorde和Jens Maurer撰写）。这是在不首先弃用该功能的情况下从该语言中删除该功能的不寻常先例，但此后事实证明这是英明的决定。

GNU的GCC编译器接受一个扩展的`typeof`语法，它与`decltype`特性并没有什么差异，开发者曾一度发现它在模板编程中非常有用。不幸的是，这是在C语言的上下文中开发的功能，并不完全适合C ++。因此，C ++委员会无法按原样合并它，但也不能对其进行修改，因为这将破坏依赖GCC行为的现有代码。这就是为什么`decltype`没有被拼写成`typeof`的缘由。Jason Merrill和其他人提出了有力的论据，认为最好有不同的运算符，而不是（依赖于）目前的`decltype(x)`和`decltype((x))`之间的细微差别，但他们并没有说服力来更改最终规范。

在C++17中使用`auto`声明非类型模板参数的能力主要由Mike Spertus发明，齐心协力的还有James Touton, David Vandevoorde和其他人。这一特性的规格更改记录在P0127R2中。有趣的是，尚不清楚是否有意使用`decltype(auto)`代替`auto`成为该语言的一部分（显然，委员会未对此进行讨论，但超出了规范）。

Mike Spertus也驱动了C++17中类模板实参推导的开发，Richard Smith和Faisal Vali 贡献了显著的技术理念（包括推导指引）。论文P0091R3中具有被选为下一个语言标准的工作文件的规格说明。

结构化绑定主要由Herb Sutter所驱动，他与Gabriel Dos Reis和Bjarne Stroustrup撰写了论文P0144R1以提出这一特性。在委员会讨论期间进行了许多调整，包括使用方括号来分隔可分解的标识符。 Jens Maurer将提案翻译成标准的最终规范（P0217R3）。

第16章 特化与重载

到目前为止，我们已经学习了C++模板如何使得一个泛型定义能够扩展为一系列相关联的类、函数或变量。尽管这是一种强大的功能机制，但在许多情况下，对于特定的模板参数替换，泛型的操作远非最佳选择。

C++与其他流行的编程语言相比，对于泛型编程来说有着一独到之处，这是因为它有着一个丰富

的特性集，能够让某一个更加特化的设施对泛型定义进行无形替代。在本章，我们将会学习两种C++语言机制：模板特化和函数模板重载，它们与纯粹的泛型相比可以有所差别。

16.1 当“泛型代码”不完全契合时

考虑下例：

```
template<typename T>
class Array {
  private:
    T* data;
        ...
  public:
    Array(Array<T> const&);
        Array<T>& operator=(Array<T> const&);

        void exchangeWith(Array<T>* b) {
          T* tmp = data;
          data = b->data;
          b->data = tmp;
        }

        T& operator[](std::size_t k) {
          return data[k];
        }
        ...
};

template<typename T> inline
void exchange(T* a, T* b)
{
  T tmp(*a);
  *a = *b;
  *b = tmp;
}
```

对简单类型来说，`exchange()`的泛型实现表现良好。然而，对于有着昂贵的拷贝操作符的类型来说，相比于为特定的给定结构体量身定制的实现来说，泛型实现体更为昂贵（从机器周期和内

存使用两方面来说）。在我们的例子中，泛型实现体需要调用一次`Array<T>`的拷贝构造器和两次`Array<T>`的拷贝操作符（译者注：作者这里应该是想用`Array<T>`代入`exchange`的模板参数`T`）。对于大尺寸的数据结构来说，这些拷贝动作通常会涉及复制相对大量的内存。然而，`exchange()`的功能可以通过仅仅交换内部的`data`指针来取而代之，就好像在其成员函数`exchangeWith()`中所作的那样。

16.1.1 透明的客制化

在前例中，成员函数`exchangeWith()`提供了一个对泛型`exchange()`函数的一个高效替换体，但是这样一来，就需要使用一个不同的函数，而这会在以下几个方面给我们带来不便：

1. `Array`类的使用者不得不记住这一额外接口，并且必须在可以使用时万分小心。
2. 泛型算法通常无法区分多种变体。例如：

```
template<typename T>
void genericAlgorithm(T* x, T* y)
{
  ...
  exchange(x, y);        // How do we select the right algorithm?
  ...
}
```

基于这些考虑，C++模板提供了透明地客制化函数模板和类模板的方法。对函数模板来说，可以通过重载机制来达成。例如，我们可以编写一个重载的`quickExchange()`函数模板集合，如下所示：

```
template<typename T>
void quickExchange(T* a, T* b)        // #1
{
  T tmp(*a);
  *a = *b;
  *b = tmp;
}

template<typename T>
void quickExchange(Array<T>* a, Array<T>* b) // #2
{
  a->exchangeWith(b);
}

void demo(Array<int>* p1, Array<int>* p2)
{
  int x = 42, y = -7;
  quickExchange(&x, &y);        // uses #1
  quickExchange(p1, p2);        // uses #2
}
```

第一处`quickExchange()`的调用有两个类型为`int*`的实参，因此只有第一个模板才能推导成功，`T`由`int`替换。因此对于哪个函数应该被调用，毫无疑问。第二处调用则恰恰相反，它可以同时匹配上面的两个模板：第一个模板使用`Array<int>`替换`T`，第二个模板使用`int`替换`T`。另一方面，在两个函数替换的结果中，参数类型都是严格匹配调用实参的。通常来说，这应该得出一个调用有歧义的结论，但是相对于第一个模板来说，C++语言认为第二个模板“更加特化”。在其他方面都等同的场合，重载决议会倾向于选择更加特化的模板，因此这里会选择#2。

16.1.2 语义透明性

上一节中重载的使用，对达成透明客制化的实例化过程来说非常有用，但是有一点需要铭记：该“透明性”非常非常依赖于实现体的细节。为了厘清这一点，来看看我们的`quickExchange()`解决方案。尽管泛型算法和为`Array<T>`类型客制化的算法最后都可以交换指针所指向的值，但是二者各自所带来的副作用却是截然不同的。下面的代码通过对比交换结构对象和交换`Array<T>`对象的值这两种行为，解释得生动形象：

```
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;
  quickExchange(&a1, &a2);        // *p == 1 after this(still)
  quickExchange(&s1, &s2);        // *q == 2 after this
}
```

如示例所展示，在调用`quick_exchange()`后，指向第1个`Array`的指针`p`变成了指向第2个`Array`的指针（即使值没有改变）；然而，指向非`Array`（即`struct S`）`s1`的指针在交换操作执行之后，仍然指向`s1`，只是指针所指向的值发生了交换。这种差别足够显著，可能会让模板实现的客户端感到困惑。前缀`quick_`将焦点聚焦到这一事实：为了实现所期待的操作，可以走捷径。然而，原始的泛型`exchange()`模板也可以对`Array<T>`进行一个有效的优化：

```
template<typename T>
void exchange(Array<T>* a, Array<T>* b)
{
  T* p = &(*a)[0];
  T* q = &(*b)[0];
  for (std::size_t k = a->size(); k-- != 0; ) {
    exchange(p++, q++);
  }
}
```

对泛型代码来说，这一版本的优势在于不再需要额外的大尺寸临时`Array<T>`对象。`exchange()`模板会被递归地调用，因此对于诸如`Array<Array<char>>`这样的类型来说，可以获得更好的性能。同时也注意到模板的更加特化的版本并没有声明`inline`，这是因为它本身会做很多的递归操作，相对而言，原始的泛型实现体声明了`inline`，因为它仅仅执行了少数的几个操作（每一个操作可能都很昂贵）。

16.2 函数模板重载

在前面的章节中我们已经看到了两个同名函数模板可以共存，尽管它们可能会实例化出相同的参数类型。这里还有一个简单的例子：
details/funcoverload1.hpp

```
template<typename T>
int f(T)
{
  return 1;
}

template<typename T>
int f(T*)
{
  return 2;
}
```

当第一个模板使用`int*`替换`T`、第二个模板使用`int`替换`T`时，二者就会得到一个参数类型（以及返回类型）完全相同的函数。不仅是这些模板可以共存，就连它们各自的实例化体也可以共存（即使它们有相同的参数和返回类型）。

下例展示了像这样生成的两个函数要如何使用显式模板实参语法来调用：
details/funcoverload1.cpp

```
#include <iostream>
#include "funcoverload1.hpp"

int main()
{
  std::cout << f<int*>((int*)nullptr);        // calls f<T>(T)
  std::cout << f<int>((int*)nullptr);        // calls f<T>(T*)
}
```

该程序输出如下：

```
12
```

为了解释这一结果，我们来详细分析一下`f<int*>((int*)nullptr)`的调用。`f<int*>()`表示我们想要用`int*`来替换`f()`模板的第一个参数，此时无需依赖模板实参推导。本例中有多个模板`f()`，因此得以创造一个包含两个函数的重载集合，这两个函数通过模板`f<int*>(int*)`（由第一个模板生成）和`f<int*>(int**)`（由第二个模板生成）生成。调用实参`(int*)nullptr`的类型为`int*`。这仅仅与第一个模板生成的函数匹配，因此最终调用的就是该函数。

相对而言，第二个调用所创造的重载集合中包含了`f<int>(int)`（由第一个模板生成）和`f<int>(int*)`（由第二个模板生成），其中第二个模板是匹配的。

16.2.1 签名

两个函数如果拥有不同的签名，那么就可以在一个程序中共存。函数签名被定义为以下信息：

1. 函数的非限定名称（或者生成该函数的函数模板名称）。
2. 函数名称所属的类或命名空间作用域，并且如果函数名称拥有内部链接，还包括该名称声明所在的编译单元。
3. 函数的`const`、`volatile`或`const volatile`限定（前提是具有这样一个限定符的成员函数）
4. 函数的`&`或`&&`限定（前提是具有这样一个限定符的成员函数）
5. 函数参数的类型（如果函数是从函数模板中生成的，那么指的是替换前的模板参数）
6. 如果函数是从函数模板中生成，则包括它的函数返回类型
7. 如果函数是从函数模板中生成，则包括模板参数和模板实参

这意味着下面的模板和它们的实例化体可以在同一个程序中共存：

```
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，当它们定义在相同的作用域中时，它们并不能总被使用，这是因为实例化会产生重载歧义。例如，调用`f2(42)`对于上面声明的模板来说显然会产生歧义。另一个例子在下面演示：

```
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
  std::cout << "f1(T1, T2)\n";
}

template<typename T1, typename T2>
void f1(T2, T1)
{
  std::cout << "f1(T2, T1)\n";
}
// fine so far

int main()
{
  f1<char, char>('a', 'b');               // ERROR: ambiguous
}
```

这里，函数`f1<T1 = char, T2 = char>(T1, T2)`可以与函数`f1<T1 = char, T2 = char>(T2, T1)`共存，但是重载决议永远无法抉择出哪一个更合适。如果模板在不同的编译单元中出现，这两个实例化体实际上可以在同一个程序中共存（并且，链接器不应该抱怨重复的定义，这是因为实例化体的签名是有所区别的）：

```
// translation unit 1:
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
  std::cout << "f1(T1, T2)\n";
}

void g()
{
  f1<char, char>('a', 'b');
}

// translation unit 2:
#include <iostream>

template<typename T1, typename T2>
void f1(T2, T1)
{
  std::cout << "f1(T2, T1)\n";
}
extern void g();                // defined in translation unit 1

int main()
{
  f1<char, char>('a', 'b');
  g();
}
```

该程序是有效的，它的输出如下：

```
f1(T2, T1)
f1(T1, T2)
```

16.2.2 重载的函数模板的偏序

再次考虑一下先前的例子：我们发现在替换了给定的模板实参列表后(`<int*>`和`<int>`)，重载决

议最终会选择最合适的函数并进行调用：

```
std::cout << f<int*>((int*)nullptr);                // calls f<T>(T)
std::cout << f<int>((int*)nullptr);                        // calls f<T>(T*)
```

然而，即使显式模板实参没有提供，函数依然会做出这样的选择。本例中，模板实参推导发挥了作用。让我们稍微修改一下`main()`函数来讨论这一机制：

details/funcoverload2.cpp

```
#include <iostream>

template<typename T>
int f(T)
{
  return 1;
}

template<typename T>
int f(T*)
{
  return 2;
}

int main()
{
  std::cout << f(0);                                  // calls f<T>(T)
  std::cout << f(nullptr);                           // calls f<T>(T)
  std::cout << f((int*)nullptr);         // calls f<T>(T*)
}
```

考虑第一处调用`f(0)`：实参的类型是`int`，如果我们用`int`替换`T`，那么它与第一个模板的参数类型匹配。然而，第二个模板的参数类型始终都是一个指针，因此，在推导之后，对于该调用来说只会从第一个模板生成一个唯一的实例作为候选。对这一情景来说，重载决议是多余的。

对于第二处调用`f(nullptr)`来说也类似：实参类型是`std::nullptr_t`，同样地，它也仅与第一个模板匹配。

第三处调用`f((int*)nullptr)`比较有意思：实参推导对于两个模板来说都会成功，产生函

数 `f<int*>(int*)` 和 `f<int>(int*)`。从传统的重载决议视角来看，这两个使用 `int*` 实参的函数同等优秀，如此理应指出调用存在歧义（参考附录C）。然而，在这一案例中，额外的重载决议发挥了作用：更加特化的模板所生成的函数会被选择。在这里，第二个模板被认为是更加特化的，因此该代码示例的输出就是 `112`。

16.2.3 正规的排序规则

在上例中，我们可以很直观地看出第二个模板比第一个模板更加特化，这是因为第一个模板可以适配各种类型的实参，而第二个则只能容纳指针类型。然而，其他的例子可能没那么直观。下面我们来描述如何确定一个函数模板是否比另一个重载模板更特化的确切过程。请注意如下的偏序规则：有可能在给定两个模板时。它们俩都无法被认定比对方更特别。如果重载决议必须从这样的两个模板中选择一个，那么将无法做出决定，程序会产生有歧义错误。

假设我们正在比较两个名称相同的函数模板，这些模板对于给定的函数调用似乎可行。重载决议如下判定：

- 函数调用参数中没有被使用的默认实参和省略号参数在后续将不被纳入考虑。
- 然后，通过以下方式替换每一个模板实参，为两个模板合成各自的实参类型列表（对类型转换函数模板来说，还包括了返回类型）：
    i. 使用唯一的虚构类型替换每一个模板类型参数。
    ii. 使用唯一的虚构类模板替换每一个模板模板参数。
    iii. 使用适当类型的唯一虚构值替换每一个非类型模板参数。（虚构出的类型、模板和值在这一上下文中都与任何其他的类型、模板或值不同，这些其他的类型、模板或值要么是开发者使用的，要么是编译器在其他上下文中合成的。）
- 如果第二个模板对于第一份合成出来的实参类型列表可以进行成功的实参推导（能够进行精确的匹配），而反过来却不行（即第一个模板对第二份实参类型列表无法推导成功），那么我们就认为第一个模板要比第二个模板更加特化。相反地，如果第一个模板对于第二份实参类型列表可以精确匹配而推导成功，反过来则不行，那么我们就认为第二个模板比第一个模板更加特化。否则（要么无法推导成功，要么两个都成功），两个模板之间就没有顺序可言。让我们将此应用于前例的两个模板之上，使得这一概念更加具体。我们从这两个模板构造出两个实参类型列表，按此前描述的那样替换其模板参数：(`A1`)和(`A2*`)(`A1` 和 `A2` 是不同的构造出的类型)。显然，第一个模板对于第二个实参类型列表可以成功推导（将 `A2*` 替换 `T`）。然而，对于第二个模板来说，没有办法让 `T*` 匹配第一个实参类型列表中的非指针类型 `A1`。因此，我们得出第二个模板比第一个模板更加特化。

让我们来看一个更加错综复杂的例子，它涉及了多个函数参数：

```
template<typename T>
void t(T*, T const* = nullptr, ...);

template<typename T>
void t(T const*, T*, T* = nullptr);

void example(int* p)
{
  t(p, p);
}
```

首先，由于实际调用没有使用第一个模板的省略号参数和第二个模板的最后一个参数（由默认实参填充），故这些参数会在排序时被忽略。此外，注意到第一个模板的默认实参没有被用到，因此参与到排序中的是其对应的参数（即与之匹配的调用实参）。

合成的两份实参列表分别是(`A1*, A1 const*`)和(`A2 const*, A2*`)。对于第二个模板来说，实参列表(`A1*, A1 const*`)可以成功推导（`A1 const`替换`T`），但是得到的结果并不能严格匹配，因为当用(`A1*, A1 const*`)类型的实参来调用`t<A1 const>(A1 const*, A1 const*, A1 const* = 0)`的时候，需要进行限定符的调整(即`const`)。类似地，第一个模板对于实参类型列表(`A2 const*, A2*`)也不能获得精确的匹配。因此，这两个模板之间并没有顺序关系，该调用存在歧义。

这种正规的排序规则通常都能产生符合直观的函数模板选择。然而，该原则偶尔也会产生不符合直觉选择的例子。因此，将来可能会修改某些规则，从而适用于所有例子。

16.2.4 模板和非模板

函数模板可以被非模板函数所重载。在选择实际调用的函数时，非模板函数将更为优先，除此之外没有什么其他区别。下面的例子说明了这一事实：

details/nontmpl1.cpp

```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) << '\n';        // prints: Nontemplate
}
```

程序会输出`Nontemplate`。

然而，当`const`和引用限定符不同时，重载决议的优先级会有所变更。例如：

details/nontmpl2.cpp

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

template<typename T>
std::string f(T&)
{
  return "Template";
}

std::string f(int const&)
{
  return "Nontemplate";
}

int main()
{
  int x = 7;
  std::cout << f(x) << '\n';        // prints: Template
  int const c = 7;
  std::cout << f(c) << '\n';        // prints: Nontemplate
}
```

程序会输出：

```
Template
Nontemplate
```

现在，当我们传递非常量`int`参数时，函数模板`f<>(T&)`是一个更合适的选择。原因在于对于`int`来说，`f<>(int&)`实例化体要比`f(int const&)`更合适。因此，这一差异不仅仅在于以下事实：一个函数是模板，而另一个函数不是模板。在这种情况下，实际应用到的是通用的重载决议规则（参考P682节C.2）。只有当使用`int const`调用`f()`时，两个函数的签名才会有相同的类型——`int const&`，而此时才会优先选择非模板函数。

出于这一原因，按下面的方式声明成员函数模板是个不错的主意：

```
template<typename T>
std::string f(T const&)
{
  return "Template";
}
```

只不过，当定义成员函数接受与拷贝或移动构造函数相同的实参时，这种效果很容易发生意外并引起出人意料的行为。例如：

details/tmplconstr.cpp

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

class C {
  public:
    C() = default;

        C(C const&) {
          std::cout << "copy constructor\n";
        }

        C(C&&) {
          std::cout << "move constructor\n";
        }

        template<typename T>
        C(T&&) {
          std::cout << "template constructor\n";
        }
};

int main()
{
  C x;
  C x2{x};                              // prints: template constructor
  C x3{std::move(x)};         // prints: move constructor
  C const c;
  C x4{c};                              // prints: copy constructor
  C x5{std::move(c)};         // prints: template constructor
}
```

程序输出如下：

```
template constructor
move constructor
copy constructor
template constructor
```

因此，成员函数模板要比`C`的拷贝构造函数更合适。而对于`std::move(c)`来说，它会产生`C const&&`类型（这是一种可行的类型，但是在语法上通常没有什么意义），成员函数模板此时也比移动构造函数更合适。

因此，通常当这些成员函数模板可能会屏蔽拷贝或移动构造函数时，必须部分地禁用它们。这在P99节6.4中解释过。

16.2.5 可变函数模板

可变函数模板（参考P200节12.4）在进行排序时需要被特殊对待，这是因为对参数包的推导（见P275节15.5）过程是将多个实参匹配到单一参数。这一行为对函数模板排序来说引入了各种有趣的场景，我们通过下例来展示：

details/variadicoverload.cpp

```
#inclue <iostream>

template<typename T>
int f(T*)
{
  return 1;
}

template<typename… Ts>
int f(Ts…)
{
  return 2;
}

template<typename… Ts>
int f(Ts*…)
{
  return 3;
}

int main()
{
  std::cout << f(0, 0.0);                                                    // calls f<>(Ts..
  std::cout << f((int*)nullptr, (double*)nullptr);        // calls f<>(Ts*...)
  std::cout << f((int*)nullptr);                                          // calls f<>(T*)
}
```

上例输出的结果是`231`，我们随后会进行讨论。

对第一个调用`f(0, 0.0)`来说，每个名称为`f`的函数模板都会被考虑：第一个函数模板`f(T*)`推导会失败，这一方面是因为模板参数`T`无法被成功推导，另一方面是因为实参的个数多于该非可变模板参数的个数；第二个函数模板`f(Ts...)`是可变模板，推导过程会针对两个实参的类型(分别是`int`和`double`)与函数参数包`(Ts)`的样式进行比较，将`Ts`推导为序列(`int`，`double`)；对于第三个函数模板——`f(Ts*...)`，推导过程会将每个实参类型与函数参数包`Ts*`的样式进行比较，该推导会失败（`Ts`无法被推导出来）。因此，最终只有第二个函数模板是可行的，也就不需要函数模板的顺序。

第二个调用——`f((int*)nullptr, (double*)nullptr)`更加有趣：对第一个函数模板的推导会失

败，因为实参个数多于模板参数个数；对第二个和第三个模板来说推导都会成功，我们显式地写出推导结果如下：

```
f<int*,double*>((int*)nullptr, (double*)nullptr)                // for second template

f<int,double>((int*)nullptr, (double*)nullptr)                          // for third template
```

排序规则会考虑第二个和第三个模板，它们都是这样的可变模板：当对可变模板应用P331节16.2.3中描述的正规的排序规则时，每个模板参数包都会由一个单一构造的类型、类模板或是值来替代。举例来说，第二个和第三个函数模板所合成的实参类型分别为`A1`和`A2*`，其中`A1`和`A2`都是唯一的构造出的类型。第二个模板对于第三个模板的实参类型列表可以推导成功（通过替换参数包`Ts`为单一元素序列(`A2*`)）。然而，无论如何构造`Ts*`的样式，第三个模板参数包始终无法匹配非指针类型`A1`，因此第三个函数模板(接受指针类型实参)要比第二个函数模板(接受任意实参)更加特化。

第三个调用——`f((int*)nullptr)`，又荡起了一层涟漪：三个函数模板的推导都是成功的，因此就需要给非可变参数模板和可变参数模板排排顺序。为了说明，我们比较第一个和第三个函数模板。这里，合成的实参类型分别是`A1*`和`A2*`，其中`A1*`和`A2*`都是唯一的构造出的类型。第一个模板对于第三个合成的实参列表可以推导成功（通过替换`T`为`A2`）。反过来，第三个模板对于第一个合成的实参列表也可以推导成功（通过替换参数包`Ts`为单一元素序列(`A1`)）。第一个和第三个模板之间的顺序可能会产生有歧义的结果。然而，还有这样一条特殊的规则：它禁止了那些源于函数参数包（例如，第三个模板参数包`Ts*...`）的实参去匹配一个非参数包（第一个模板参数`T*`）的参数。因此，第一个模板使用第三个合成的实参列表时推导会失败，于是我们可以认为第一个模板相比第三个模板更加特化。这一特殊的规则让非可变模板（拥有固定数量的参数）比可变模板（拥有可变数量的参数）更加特化。

前面描述的规则对发生在函数签名的类型中的包展开时有着同等用法。例如，在前面的示例中，我们可以将函数模板的每一个参数和实参包裹成一个可变类模板`Tuple`，来实现一个类似的示例而不用引入函数参数包：
details/tupleoverload.cpp

```cpp
#include <iostream>

template<typename... Ts> class Tuple
{
};

template<typename T>
int f(Tuple<T*>)
{
  return 1;
}

template<typename... Ts>
int f(Tuple<Ts...>)
{
  return 2;
}

template<typename... Ts>
int f(Tuple<Ts*...>)
{
  return 3;
}

int main()
{
  std::cout << f(Tuple<int, double>());               // calls f<>(Tuple<Ts...>)
  std::cout << f(Tuple<int*, double*>());        // calls f<>(Tuple<Ts*...>)
  std::cout << f(Tuple<int*>());                         // calls f<>(Tuple<T*>)
}
```

函数模板排序时，对模板实参到`Tuple`的包展开与我们前面示例中函数包展开有着相似的考量，运行结果输出：`231`。

16.3 显式特化

重载函数模板并根据偏序规则来选择“最”匹配的函数模板这一能力，使得我们可以透明地对泛型实现增加特化模板来调整代码以获得更高的效率。然而，类模板和变量模板无法被重载。取而代

之的是，类模板的透明客制化采用了另一种机制：显式特化。标准术语”显式特化“是指一种我们称之为“完整特化”的语言特性。它使用完全替代后的模板参数来提供一个模板实现体：没有保留任何模板参数。类模板、函数模板和变量模板都可以进行完整特化。

类模板的成员可以被定义在类定义体的外部（即，成员函数、嵌套类、静态数据成员和成员枚举类型）。

在后面的一节中，我们会描述“偏特化”。它与完整特化相似，只不过并没有完全替换模板参数而是在模板的替换中保留了一部分。完整特化和偏特化在我们的代码中都是同等“显式的”，这也是为什么我们在讨论中避开用术语”显式特化“的原因。全特化和偏特化都没有引入一个全新的模板或是模板实例。相反，这些结构为泛型模板中已经隐式声明的实例提供了替代的定义。这是一个相对重要的概念，它是与模板重载的主要区别。

16.3.1 类模板的完整特化

完整特化由连续的`template`，`<`和`>`语法块引导，且类名称的后面跟随着特化所声明的模板实参。下面的例子对此进行了说明：

```
template<typename T>
class S {
  public:
    void info() {
          std::cout << "generic (S<T>::info())\n";
        }
};

template<>
class S<void> {
  public:
    void msg() {
          std::cout << "fully specialized (S<void>::msg())\n";
        }
};
```

请注意看完整特化的实现，是如何无需以任何方式与泛型定义相关联的：这意味着我们可以使用不同名称的成员函数（`info`对`msg`）。二者的关联仅仅由类模板的名称所决定。

特化模板实参列表必须与模板参数列表一致。举例来说，为模板类型参数指定一个非类型值是不

合法的。然而，对于有着默认模板实参的模板参数来说，对应的模板实参也是可选的：

```
template<typename T>
class Types {
  public:
    using I = int;
};

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: 0 cannot substitute U

int main()
{
  S<int>*         pi;                // OK: uses #1, no definition needed
  S<int>         e1;                // ERROR: uses #1, but no definition available
  S<void>*        pv;                // OK: uses #2
  S<void, int> sv;         // OK: uses #2, definition available
  S<void, char> e2;         // ERROR: uses #1, but no definition available
  S<char, char> e3;         // ERROR: uses #3, but no definition available
}

template<>
class S<char, char> {         // definition for #3
};
```

如上例所展示，完整特化的声明可以无需定义体。然而，当声明了完整特化时，泛型定义就永远不会使用这一组既定的模板实参来实例化。因此，如果程序需要某个定义但是却找不到对应的实现体时就会出错。对类模板特化来说，有时“前置声明”类型会很有用，因为这样就可以构造相互依赖的类型。完整特化声明与普通的类声明在这一方面是等同的（记住它不是模板声明），唯一

的区别在于语法以及特化声明必须匹配前面的模板声明。因为这不是模板声明，完整特化类模板的成员可以使用普通的类外成员定义语法来定义（换句话说，不能指定模板前缀`template<>`）：

```
template<typename T>
class S;

template<> class S<char**> {
  public:
    void print() const;
};

// the following definition cannot be preceded by template<>
void S<char**>::print() const
{
  std::cout << "pointer to pointer to char\n";
}
```

一个更复杂的例子来加强理解这一概念：

```
template<typename T>
class Outside {
  public:
  template<typename U>
  class Inside{
  };
};

template<>
class Outside<void> {
  // there is no special connection between the following nested class
  // and the one defined in the generic template
  template<typename U>
  class Inside {
    private:
          static int count;
  };
};
// the following definition cannot be preceded by template<>
template<typename U>
int Outside<void>::Inside<U>::count = 1;
```

完整特化是泛型模板的特定实例化体的替代体，并且在同一个程序中无法同时存在显式完整特化体和模板生成的实例化体这两个版本。试图在同一个文件中使用两者通常会被编译器逮捕：

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

Invalid<double> x1;               // causes the instantiation of Invalid<double>

template<>
class Invalid<double>;        // ERROR: Invalid<double> already instantiated
```

不幸的是，如果在不同的编译单元中使用，问题可能不会被轻易捕获。下面的C++代码由两个文件组成，在多个平台上编译和链接这个例子都表示它是非法的，甚至是危险的：

```
// Translation unit 1:
template<typename T>
class Danger {
  public:
    enum { max = 10 };
};

char buffer[Danger<void> ::max];        // uses generic value

extern void clear(char*);

int main()
{
  clear(buffer);
}

// Translation unit 2:
template<typename T>
class Danger;

template<>
class Danger<void> {
  public:
    enum { max = 100 };
};

void clear(char* buf)
{
  // mismatch in array bound:
  for(int k = 0; k<Danger<void> ::max; ++k) {
    buf[k] = '\0';
  }
}
```

显然，为了保证简洁，我们对使该示例做了裁剪，但是它说明了：在使用特化时，必须非常小心地确认特化的声明对泛型模板的所有用户都是可见的。在实际应用中，这意味着：在模板声明所在的头文件中，特化的声明通常应该在模板的声明之后。然而，泛型实现也可能来自外部源码（诸如不能被修改的头文件），尽管现实中很少采用这种方式，但还是值得我们去创建一个包含泛型模板的头文件，并让特化声明位于泛型模板之后，以避免这种“难以排查”的错误。此外，通

常来说，最好避免从外部源码引入特化模板，除非明确表示设计的目的就是如此。

16.3.2 函数模板的完整特化

函数模板完整特化背后的语法和原则与类模板完整特化大体相同，只是加入了重载和实参推导。

如果可以借助实参推导（用实参类型来推导声明中给出的参数类型）和偏序来确定模板的特化版本，那么完整特化实现就可以忽略显式的模板实参。举个例子：

```
template<typename T>
int f(T)                // #1
{
  return 1;
}

template<typename T>
int f(T*)               // #2
{
  return 2;
}

template<> int f(int)        // OK: specialization of #1
{
  return 3;
}

template<> int f(int*)        // OK: specialization of #2
{
  return 4;
}
```

函数模板完整特化不能包含默认实参值。然而，对于被特化的模板所指定的任何默认实参，显式特化版本都可以使用这些默认实参值。例如：

```
template<typename T>
int f(T, T x = 42)
{
  return x;
}
template<> int f(int, int = 35) // ERROR
{
  return 0;
}
```

（这是因为完整特化提供的是一个替换的定义，而不是一个替换的声明。在调用函数模板的时点，该调用已经完全基于函数模板而完成解析了。）

完整特化的声明和普通声明（或者是一个普通的重声明）在很多方面都很类似。特别地，它不会声明一个模板，因此对于非内联完整特化函数模板特化来说，在程序中它只能有一个定义。然而，我们必须确保：函数模板的完整特化声明需跟随在模板定义之后，以避免试图使用一个由模板生成的函数。因此，模板`g()`的声明和完整特化声明应该被组织成两个文件，如下所示：

- 接口文件包含了主模板的定义和偏特化的定义，但是仅包含完整特化的声明：

```
#ifndef TEMPLATE_G_HPP
#define TEMPLATE_G_HPP

// template definition should appear in header file:
template<typename T>
int g(T, T x = 42)
{
  return x;
}

// specialization declaration inhibits instantiations of the template;
// definition should not appear here to avoid multiple definition errors

template<> int g(int, int y);
#endif // TEMPLATE_G_HPP
```

- 相应的，实现文件包含了完整特化的定义：

```
#include "template_g.hpp"
template<> int g(int, int y)
{
  return y/2;
}
```

或者完整特化也可以搞成内联，此时它的定义就可以放在同一个头文件中。

16.3.3 变量模板的完整特化

变量模板也可以被完整特化。如今，这一语法非常直观：

```
template<typename T> constexpr std::size_t SZ = sizeof(T);
template<> constexpr std::size_t SZ<void> = 0;
```

显然，该完整特化可以提供一个不同于模板所产生结果的初始化器。有趣的是，变量模板特化不需要与模板的类型匹配：

```
template<typename T> typename T::iterator null_iterator;
template<> BitIterator null_iterator<std::bitset<100>>;
                // BitIterator doesn't match T::iterator, an that is fine
```

16.3.4 成员的完整特化

类模板的成员模板、普通静态数据成员、普通成员函数都可以进行完整特化。每个类模板作用域都需要一个`template<>`前缀。如果要对一个成员模板进行特化，则必须加上另一个`template<>`前缀，来说明该声明表示的是一个特化。为了厘清上述含义，我们给出下列声明：

```
template<typename T>
class Outer {                                                    // #1
  public:
    template<typename U>
    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
    }
};
```

泛型模板`Outer`(#1)的普通成员`code`(#4)和`print()`(#5)具有单一的类模板作用域，因此完整特化时需要一个`template<>`前缀以及一组模板实参：

```
template<>
int Outer<void>::code = 12;

template<>
void Outer<void>::print() const
{
  std::cout << "Outer<void>";
}
```

这些定义将会用于替代类`Outer<void>`（在#4和#5处替代泛型定义），但是`Outer<void>`的其他成员仍然会通过#1处的模板来生成。请注意，在进行了这些声明之后，不能再次提供`Outer<void>`的显式特化。

正如函数模板完整特化那般，我们也需要一种方式来声明类模板普通成员的特化而不用去定义它（防止出现多个定义体）。尽管对于普通类的成员函数和静态数据成员而言，非定义的类外声明在C++中不被允许，但如果是针对类模板的特化成员，该声明是合法的。也就是说，前面的定义可以具有如下声明：

```
template<>
int Outer<void>::code;

template<>
void Outer<void>::print() const;
```

细心的读者可能会发现`Outer<void>::code`的完整特化非定义声明看上去就是一个使用默认构造器的初始化定义。实际上也确实如此，只不过这样的声明永远会被解析成非定义声明。因此，如果静态数据成员的类型是一个只能使用默认构造函数进行初始化的类型，我们就必须采用初始化列表语法。如下示例：

```
class DefaultInitOnly {
  public:
    DefaultInitOnly() = default;
    DefaultInitOnly(DefaultInitOnly const&) = delete;
};

template<typename T>
class Statics {
  private:
    static T sm;
};
```

下面的语句是一个声明：

```
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm;
```

如果想要一个定义并调用默认构造器：

```
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm{};
```

在C++11之前，这无法办到。对于这种特化也无法实现默认初始化。以前的经典办法是使用拷贝初始化：

```
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm =
DefaultInitOnly();
```

遗憾的是，对我们的例子来说这是行不通的，因为拷贝构造器被删除了。然而，C++17引入了强制复制省略(mandatory copy-elision)法则，这一法则使得该实现合法化，因为这里实际上不会真正调用拷贝构造器。

成员模板`Outer<T>::Inner`也可以使用特定的模板实参进行特化，对于该特化所在的外围`Outer<T>`而言，它不会影响`Outer<T>`相应实例化体的其他成员。同样的，由于存在一个外围模板，所以我们需要添加一个`template<>`前缀。代码应该写成下面这样：

```
template<>
template<typename X>
class Outer<wchar_t>::Inner {
  public:
    static long count; // member type changed
};

template<>
template<typename X>
long Outer<wchar_t>::Inner<X>::count;
```

模板`Outer<T>::Inner`也可以被完整特化，但只能针对某个给定的`Outer<T>`实例。我们现在需要两个`template<>`前缀：第一个是因为外围类的存在，第二个是因为我们完整特化了内层模板：

```
template<>
template<>
class Outer<char>::Inner<wchar_t> {
  public:
    enum { count = 1 };
};

// the following is not valid C++;
// template<> cannot follow a template parameter list
template<typename X>
template<> class Outer<X>::Inner<void>;        // ERROR
```

我们可以将此与`Outer<bool>`的成员模板特化进行比较。由于后者已经进行过完整特化了，也就没有外部模板了，此时我们只需要一个`template<>`前缀：

```
template<>
class Outer<bool>::Inner<wchar_t> {
  public:
    enum { count = 2 };
};
```

16.4 类模板的偏特化

模板的完整特化通常很有用，但有些时候我们更希望对类模板或变量模板的模板实参族进行特化，而不是针对某个具体实参列表进行完整特化。例如，假设下面是一个类模板实现的链表：

```
template<typename T>
class List {                                  // #1
  public:
    ...
        void append(T const&);
        inline std::size_t length() const;
        ...
};
```

使用该类模板的大型项目会为多种类型实例化出它的成员。对于非内联展开的成员函数来说（即`List<T>::append()`），这会导致对象代码的显著膨胀。然而，如果我们从一个底层视角来看，`List<int*>::append()`和`List<void*>::append()`是等同的。换句话说，我们可以指定所有的指针型`List`共享同一个实现体。尽管这无法在C++中直接表达，但我们可以指定所有的指针型`List`都从不同的模板定义中实例化，从而达成近似的目标：

```
template<typename T>
class List<T*> {                  // #2
  private:
    List<void*> impl;
        ...
  public:
    ...
        inline void append(T* p) {
          impl.append(p);
        }
        inline std::size_t length() const {
          return impl.length();
        }
        ...
};
```

在该上下文中，#1处的原始模板被称作主模板，后面的定义被称为偏特化（因为模板定义所使

用的模板实参只指定了一部分）。模板参数列表声明（`template<...>`），再加上显式指定的模板实参集合（在类模板名称后，本例中是`<T*>`），两者组合在一起就是偏特化语法的表征。

我们的代码中有一个问题，`List<void*>`会递归地包含相同的`List<void*>`类型。为了打破这一循环，我们可以在该偏特化之前先提供出一个完整特化：

```
template<>
class List<void*> {                // #3
  ...
  void append(void* p);
  inline std::size_t length() const;
  ...
}
```

而这之所以行得通，是因为完整特化的优先级要高于偏特化。因此，指针型`List`的所有的成员函数都会通过内联函数转发到`List<void*>`的实现体。这是一种对抗代码膨胀（C++模板经常会遇到）的有效方法。

偏特化声明的参数和实参列表存在着一些约束。下面是这些约束的部分内容：

1. 偏特化的实参必须与主模板对应的参数相匹配。
2. 偏特化的参数列表不能具有默认实参；作为替代，主类模板的默认实参会被使用。
3. 偏特化的非类型实参要么是一个非依赖型值，要么是一个普通的非类型模板参数。它们不能是更加复杂的表达式，诸如`2*N`（`N`是一个模板参数）。
4. 偏特化的模板实参列表不应该与主模板的参数列表完全相同（忽略重命名）。
5. 如果模板实参的某一个是包展开，那么它必须位于模板实参列表的最后。

用一个例子来解释这些约束：

```
template<typename T, int I = 3>
class S;                                  // primary template

template<typename T>
class S<int, T>;                 // ERROR: parameter kind mismatch

template<typename T = int>
class S<T, 10>;                           // ERROR: no default arguments

template<int I>
class S<int, I*2>;                 // ERROR: no nontype expressions

template<typename U, int K>
class S<U, K>;                           // ERROR: no significant difference from primary template

template<typename... Ts>
class Tuple;

template<typename Tail, typename... Ts>
class Tuple<Ts..., Tail>;                  // ERROR: pack expansion not at the end

template<typename Tail, typename... Ts>
class Tuple<Tuple<Ts...>, Tail>;         // OK: pack expansion is at the end of a nested template a
```

每个偏特化和完整特化一样，都和主模板相关联。模板被使用时，编译器总是会对主模板进行查找，但接下来还会匹配调用实参和相关联特化的实参（使用模板实参推导，如15章所描述），然后确定应该选择哪一个模板实现体。与函数模板实参推导一样，SFINAE原则会在这里应用：如果在试图匹配一个偏特化时产生了无效的结构，那么特化会被默默丢弃，然后继续对下一个候选进行试验（如果可行的话）。如果找不到匹配的特化，主模板就会被选择；如果能够找到多个匹配的特化，那么就会选择“最特殊”的特化（与重载函数模板所定义的规则一样），而这其中如果无法确定“最特殊”的那一个（即存在几个特殊程度相同的特化），那么程序就会抛出有歧义的错误。

最后，我们要指出：类模板偏特化的参数个数是可以和主模板不一样的，它既可以多于主模板，也可以少于主模板。让我们再次考虑泛型模板`List`（在#1处声明）。我们已经讨论了如何优化指针型`List`的情景，但我们希望可以针对特定的成员指针类型实现这种优化。下面的代码就是针对指向成员指针的指针，来实现这种优化：

```
// partial specialization for any pointer-to-void* member
template<typename C>
class List<void* C::*> {        // #4
  public:
    using ElementType = void* C::*:
        ...
        void append(ElementType pm);
        inline std::size_t length() const;
        ...
};

// partial specialization for any pointer-to-member-pointer type except
// pointer-to-void* member, which is handled earlier
// (note that this partial specialization has two template parameters,
// whereas the primary template only has one parameter)
// this specialization makes use of the prior one to achieve the
// desired optimization
template<typename T, typename C>
class List<T* C::*> {        // #5
  private:
    List<void* C::*> impl;
        ...
  public:
    using ElementType = T* C::*;
        ...
        inline void append(ElementType pm) {
          impl.append((void* C::*)pm);
        }
        inline std::size_t length() const {
          return impl.length();
        }
        ...
};
```

除了模板参数数量不同之外，我们看到在#4处定义的公共实现本身也是一个偏特化（对于简单的指针例子，这里应该是一个完整特化），而所有其他的偏特化(#5处的声明)都是把实现委托给这个公共实现。显然，在#4处的公共实现要比#5处的实现更加特化，因此也就不会造成歧义。

此外，显式书写的模板实参数量与主模板的模板参数数量甚至也可能不同。这会在拥有默认模板

实参以及拥有可变模板时发生：

```
template<typename... Elements>
class Tuple;        // primary template

template<typename T1>
class Tuple<T>;        // one-element tuple

template<typename T1, typename T2, typename... Rest>
class Tuple<T1, T2, Rest...>;        // tuple with two or more elements
```

16.5 变量模板的偏特化

变量模板在C++11标准的草稿中引入时，其许多方面的规范都被忽视了，其中的一些问题依然没有给出官方定论。然而，在现实中，各种编译器在实现时通常对这些问题的处理都有一致的表现。

这些问题中可能最令人诧异的是：标准会更倾向于偏特化变量模板，但是却并没有描述它们要如何声明或者它们意味着什么。因此，下面的内容基于实践中的C++实现（确实允许这种偏特化），而不是基于C++标准。

如你所愿，语法与变量模板的完整特化是类似的，除了`template<>`要被替换成实际的模板声明头，并且变量模板名称后跟随着模板实参列表必须依赖于模板参数。例如：

```
template<typename T> constexpr std::size_t SZ = sizeof(T);

template<typename T> constexpr std::size_t SZ<T&> = sizeof(void*);
```

与变量模板的完整特化一样，偏特化的类型也不需要匹配主模板的类型：

```
template<typename T> typename T::iterator null_iterator;

template<typename T, std::size_t N> T* null_iterator<T[N]> = null_ptr;
                // T* doesn't match T::iterator, and that is fine
```

变量模板偏特化可以指定的模板参数种类这一规则与类模板偏特化是相同的。类似地，为给定的

具体模板实参列表选择某一个特化的规则也是相同的。

16.6 后记

模板完整特化是C++模板机制中一开始就有的一部分。然而，函数模板重载和类模板偏特化则出现得晚一些。第一个实现了函数模板重载的是HP的C++编译器，而第一个实现了类模板偏特化的是EDG的C++ front end编译器。本章中描述的偏序规则最早由Steve Adamczyk和John Spicer发明（这两位都是EDG的成员）。

模板特化可以终止模板定义的无限递归（诸如P348节16.4中出现的`List<T*>`），这一项能力长久以来可谓广为人知。然而，Erwin Unruh可能是提出模板元编程（使用模板实例化机制在编译器执行非琐碎的计算。我们会在第23章中致力于这一话题）这一有趣概念的第一人。

你可能想知道为什么只有类模板和变量模板可以被偏特化。实际上大都是历史成因。为函数模板定义这一机制也本该是可行的（参考第17章）。在某些方面，函数模板的重载效果与之相似，但是也存在一些细微的差异。这些差异主要与以下事实有关：在用到的时候仅需要查找主模板，随后才考虑特化，以确定哪一个实现体会被使用。相反，在进行查找时，所有的重载函数模板都必须放入一个重载集合中，它们可能源于不同的命名空间或是类。这增加了模板名称被无意中重载的可能性。

相反地，我们也可以想象让类模板和变量模板以某种形式重载。举个例子：

```
// invalid overloading of class templates
template<typename T1, typename T2> class Pair;
template<int N1, int N2> class Pair;
```

然而，看起来对这一机制的需求并不迫切。

第17章 通往未来

C++模板几乎一直在不断发展，从1988年的初始设计，到1998年、2011年、2014年和2017年的各种标准化里程碑。可以说，在原初的98标准之后，模板至少与大部分语言新增的主要功能有关联。

本书的第一版罗列了一些我们在首个标准之后可能会看到的扩展能力，这其中的一部分已经得以

实现：

- 尖括号hack：C++11移除了需要在两个连续的尖括号之间插入一个空格的必要性
- 默认函数模板实参：C++11开始，函数模板参数可以具有默认实参
- Typedef模板：C++11引入了别名模板，具有类似的功能
- `typeof`操作符：C++11引入了`decltype`操作符，扮演了相同的角色（但是使用了一个不同的token来避免与已存在的扩展冲突，尽管该扩展并不满足C++开发者社区的需求）。
- 静态属性：第一版预测了编译器将直接支持某些type traits。事实上当前确实如此，尽管接口是使用标准库（然后使用若干traits的编译器扩展实现）来表达的。
- 个性实例化诊断：新的关键字`static_assert`实现了本书第一版所描述的`std::instantiation_error`的想法。
- 参数列表：在C++11中变成了参数包。
- 布局控制：C++11的`alignof`和`alignas`满足了本书第一版的需求。此外，C++17还新增了一个`std::variant`模板来支持union。
- 初始化器推导：C++17支持了类模板实参推导，算是同样的议题。
- 函数表达式：C++11的lambda表达式完整提供了这一功能（相比第一版的讨论使用了不一样的语法）。

第一版中其他方向的假设暂未收录到当前的语言规范，但其中的大部分目前仍在火热的讨论中，这里我们也对它们予以保留。与此同时，一些其他想法也在萌生，在此我们也会对其中的一部分想法进行表述。

17.1 宽容的`typename`法则

在本书的第一版中，在这一章节曾说过在未来可能会带来两种宽容的`typename`使用法则（228页节13.3.2）：允许在以前不允许的地方使用`typename`；当编译器可以相对轻松得推理出具有依赖型限定的限定名称指代的必定是某种类型时，可以省略`typename`。前者已经实现（C++11中的类型名可以在许多地方冗余使用），但后者还没有。

然而最近，有人再次呼吁在一些常见的上下文中将`typename`做成可选的，因为这些上下文对类型说明符的期望很明确：

- 在命名空间和类作用域中的函数和成员函数的返回类型与参数类型。在任何作用域中出现的函数、成员函数模板以及lambda表达式亦是如此。
- 声明的变量、变量模板以及静态数据成员的类型。对变量模板来说也一样。

- 在别名或别名模板的token `=` 之后的类型。
- 模板类型参数的默认实参。
- 跟随在`static_cast`，`const_cast`，`dynamic_cast`或是`reinterpret_cast`之后的尖括号内的类型。
- 在`new`表达式中命名的类型。

虽然这个列表相对来说是比较临时的，但事实证明，这种语言的改变将允许大多数使用`typename`的实例被省略，这将使代码更加紧凑和易读。

17.2 泛化的非类型模板参数

在非类型模板实参的限制中，最可能令模板初学者和老司机惊讶的是：没办法提供一个字符串字面值来作为模板实参。下面的例子看上去足够符合直觉：

```
template<char const* msg>
class Diagnoser {
  public:
    void print();
};

int main() {
  Diagnoser<"Suprise!">().print();
}
```

然而，这里有些潜在的隐患。在标准C++中，当且仅当`Diagnoser`的两个实例拥有相同的实参时，它们俩的类型才是一致的。在该示例中，实参是一个指针值（换句话说，是个地址）。然而，在不同位置的两个字面上相同的字符串字面值却并不一定有相同的地址。这个时候我们就会发现`Diagnoser<"X">`和`Diagnoser<"X">`实际上是两种截然不同的类型且彼此并不兼容！（注意，`"X"`的类型是`char const[2]`，但是当它作为模板实参传递时，退化成了`char const *`。）

基于这些考虑，C++标准禁止将字符串字面值作为模板的实参。然而，一些（厂商编译器）实现提供这一功能作为扩展。它们通过在模板实例的内部表示中使用实际的字符串字面值内容来实现这一点。尽管这显然是可行的，但一些C++语言评论员认为，一个可以由字符串字面值替换的非类型模板参数应该与可以由地址替换的参数声明方式不同。一个可能的方法是将字符串字面值捕捉在一个字符参数包中。举个例子：

```
template<char... msg>
class Diagnoser {
  public:
    void print();
};

int main() {
        // instantiates Diagnoser<'S','u','r','p','r','i','s','e','!'>
    Diagnoser<"Surprise!">().print();
}
```

我们还应该注意到这个问题的一个额外的技术细节。考虑以下模板声明，并假设语言已经扩展以接受字符串字面值作为模板参数的情况：

```
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;
}
```

在上述代码中，两个成员函数除了名字以外，其他都完全相同（这种情况不太寻常）。假设有一种实现采用了类似宏展开的方式对`Bracket<"X">`进行实例化：此时，如果两个成员函数被实例化到不同的编译单元，它们就会返回不同的值。有意思的是，在一些支持该扩展功能的C++编译器上进行测试后，发现它们有着这样的问题。

还有一个相关的议题，就是模板实参对浮点数字面值的支持（以及简单的常量浮点数表达式）。举个例子：

```
template<double Ratio>
class Converter {
  public:
    static double convert (double val) {
      return val*Ratio;
    }
};

using InchToMeter = Converter<0.0254>;
```

这个特性在某些C++实现中也予以了支持，同时也没什么技术上的挑战（与字符串字面值不同）。

C++11引入了字面值类类型的概念：一种可以在编译时接受常量值的类类型（包括通过`constexpr`函数进行的非平凡计算）。一旦这种类类型可用，马上就可以期待将它们用作非类型模板参数。然而，与上述描述的字符串字面值参数类似的问题出现了。特别地，两个类类型值的“相等性”并不是一个简单的问题，因为它通常是由操作符`==`的定义来确定的。这种相等性决定了两个实例是否相等，但实际上，链接器必须通过比较修饰后的名称来检查这种相等性。一个解决办法可能是在特定的字面值类中添加一个选项，标记它们具有平凡的相等性条件，即对类的标量成员进行两两比较。只有具有这种平凡相等性条件的类类型才被允许作为非类型模板参数类型。

17.3 函数模板的偏特化

在第16章中，我们讨论了类模板是如何做偏特化的，而函数模板仅支持简单的重载能力。这两种机制有些差异。

偏特化并没有引入一种新的模板：它是在既有模板（主模板）的基础上进行扩展。在查找类模板时，一开始只会考虑主模板。而在选择了主模板之后，如果发现有能够匹配模板实例的偏特化时，它的定义（也就是身体）就会被实例化出来以替代主模板的定义。（对完整特化来说也一样。）

相比之下，重载的函数模板是彼此完全独立的独立模板。在选择要实例化哪一个模板时，所有的重载模板都会被同时考虑，然后重载决议会尝试选出最适合的那一个。乍一看可能会觉着这种机

制完全可以作为替代品，但在实践中还是有着诸多限制：

- 特化类的成员模板而不去修改类的定义是可行的。然而，增加重载的成员需要对类的定义进行修改。在多数情况下，我们由于没有这一权限而无法这样操作。此外，C++标准当前也不允许我们向`std`命名空间新增模板，但是它允许我们做模板的特化。
- 对重载函数模板来说，它们的参数必须有所区别。考虑这样一个函数模板`R convert(T const &)`，其中`R`和`T`是模板参数。我们非常想用`R = void`来特化这一模板，但使用重载是办不到的。
- 对于有效的未重载的函数，当函数一旦被重载后，可能会变得永久失效。特别是，对给定的两个函数模板`f(T)`和`g(T)`（其中T是模板参数），表达式`g(&f<int>)`当且仅当`f`没有被重载时才有效（否则就无法决定`f`指代哪一个函数）。
- 友元声明指代一个特定的函数模板或是特定的函数模板的实例化。函数模板的重载版本可能不会自动地授权原始模板的使用权限。

上述总总共同组织成了一个对支持函数模板偏特化这一能力的有力论点。

函数模板偏特化的自然语法可以从类模板中提炼：

```
template<typename T>
T const& max(T const&, T const&);  // primary template

template<typename T>
T* const& max <T*>(T* const&, T *const&);  // partial specialization
```

一些语言设计者担心函数模板重载与这种偏特化实现之间的互动，举个例子：

```
template<typename T>
void add(T& x, int i);  // a primary template

template<typename T1, typename T2>
void add(T1 a, T2 b);  // another(overloaded) primary template

template<typename T>
void add<T*> (T*&, int);  // Which primary template does this specialize?
```

然而，我们预计此类情况将被视为错误，不会对该功能的实用性产生重大影响。

在C++11标准化期间曾简要讨论了这一扩展，但相对而言大家意兴阑珊。尽管如此，这个话题偶尔还会出现，因为它巧妙地解决了一些常见的编程问题。也许它将在未来的C++标准中再次被采用。

17.4 命名的模板实参

512页章节21.4描述了一种技术，它可以让我们为特定的参数提供一个非默认模板实参，而无需指定其他的具有默认值的模板实参。尽管这是一种有趣的技术，但很明显，为了达成这样一个简单的效果它做了太多的工作。因此，提供一种语言机制来命名模板实参是一个自然而然的想法。

我们应该注意到，在C++标准化过程中，Roland Hartinger曾提议了（详见【StroustrupDnE】之节6.5.1）一个相似的扩展（有时也被称作关键字实参(keyword arguments)）。虽然技术上是合理的，但还是由于种种原因，该提议最终没有被纳入语言标准。在这一点上，没什么理由去相信命名的模板实参会被纳入语言标准，但这个话题在委员会讨论中的确经常出现。

然而，为了完整起见，这里我们提及一个已经讨论过的句法想法：

```
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 {
  ...
};

void test(MatrixList ml)
{
        mySort(ml, Mutator<Matrix, .Swap = matrixSwap>);
}
```

在这里，实参名称的`.`用来表示我们是按名称来引用模板实参。该语法与C99标准所引入的“指定的初始化器”语法相似：

```
struct Rectangle { int top, left, width, height; };
struct Rectangle r = { .width = 10, .height = 10, .top = 0, .left = 0 };
```

当然，引入命名模板实参意味着模板的模板参数的名称现在是该模板公共接口的一部分，不能自由更改。可以通过一个更显式的选择语法来解决这一问题，如下所示：

```
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>);
}
```

17.5 重载的类模板

完全可以想象：类模板基于模板参数也可以进行重载。比如，我们可以创建一个`Array`模板家族，它们同时包括动态和静态尺寸的数组：

```
template<typename T>
class Array {
        // dynamically sized array
        ...
};

template<typename T, unsigned Size>
class Array {
        // fixed size array
        ...
};
```

重载无需受限于模板参数的数量变化，参数类型有所变化也行得通：

```
template<typename T1, typename T2>
class Pair {
        // pair of fields
        ...
};

template<int I1, int I2>
class Pair {
        // pair of constant integer values
        ...
};
```

尽管这一想法曾被一些语言设计者在非官方场合讨论过，但截止到目前，它还没有被正式地呈现给C++标准委员会。

17.6 非最终包展开的推导

包展开的模板实参推导当且仅当包展开位于实参列表的最后才可行。这就意味着，从一个列表中榨取出首个元素可以相当简单：

```
template<typename... Types>
struct Front;

template<typename FrontT, typename... Types>
struct Front<FrontT, Types...> {
        using Type = FrontT;
};
```

正如在347页节16.4中所描述的偏特化中的位置限制，我们没办法简单地榨取出列表的最后一个元素：

```
template<typename... Types>
struct Back;

template<typename BackT, typename... Types>
struct Back<Types..., BackT> {  // ERROR: pack expansion not at the end of
        using Type = BackT;         //        template argument list
};
```

可变函数模板的模板实参推导也有类似的限制。放宽模板实参推导和偏特化的规则，让包展开可以在模板实参列表中的任意位置出现，从而使得这种操作变得更简单，这一方法貌似可行。此外，虽然可能性较小，但推导也可以允许在同一参数列表中出现多个包展开：

```
template<typename... Types> class Tuple {
};

template<typename T, typename... Types>
struct Split;

template<typename T, typename... Before, typename... After>
struct Split<T, Before..., T, After...> {
        using before = Tuple<Before...>;
        using after = Tuple<After...>;
};
```

对多个包展开的支持引入了额外的复杂度。比方说，`Split`是在见到`T`出现的第一次、最后一次还是其中的某一次时进行分割呢？推导过程达到怎样的复杂度时才允许编译器放弃呢？

17.7 `void`的正则化

在编写模板时，规则性是一种美德：如果单一的结构能够覆盖所有情况，那么我们的模板就会变得更简单。我们的程序中有一个不太规则的方面：类型。例如，请看下例：

```
auto&& r = f();  // error if f() returns void
```

这行代码仅在`f()`返回一个`void`类型以外的类型时才能正常工作。当我们使用`decltype(auto)`时也会遇到同样的问题：

```
decltype(auto) r = f();  // error if f() returns void
```

`void`并非唯一的不规则类型：函数类型和引用类型也经常在一些情景中表现得有所例外。然而，然而，事实证明，`void`往往使我们的模板复杂化，它也没有深刻的理由变得不同寻常。比如，在162页节11.1.3中就有一个例子，它展示了`void`类型如何让完美的`std::invoke()` wrapper的实现变得复杂化。

我们可以宣布`void`是一种具有唯一值的正常值类型（如`std::nullptr_t`之于`nullptr`）。出于向后兼容性的目的，我们仍然必须为函数声明保留以下特殊情况：

```
void g(void);  // same as void g();
```

然而，在大多数其他方法中，`void`会成为一种完全的值类型。此时我们将可以用`void`来声明变量和引用：

```
void v = void{};
void&& rrv = f();
```

最重要的是，许多模板将不再需要为`void`情景进行特化处理。

17.8 模板的类型检查

模板编程的大部分复杂性源于编译器无法进行局部地检查模板定义是否正确。相反地，模板的大部分检查都发生在模板实例化期间，此时模板定义上下文和模板实例化上下文交织在一起。不同上下文的混合让我们难以追责：究竟是模板定义的问题（因其错误地使用了模板实参），还是模板使用者的问题（因其提供的模板实参未满足模板的需求）？这一问题可以用一个简单的例子来解释，我们用一个常规编译器所产生的诊断信息加以注解：

```
template<typename T>
T max(T a, T b)
{
        return b < a ? a : b;  // ERROR: "no match for operator <
                               //         (operator types are 'X' and 'X')"
}

struct X {
};
bool operator > (X, X);

int main()
{
        X a, b;
        X m = max(a, b);  // NOTE: "in instantiation of function template specialization
                          //       'max<X>' requested here"
}
```

可以看到实际的错误（缺少合适的`operator <`）是在函数模板`max()`的定义中检测出来的。也有可能真正的错误在于——`max()`应该使用`operator >`取而代之？然而，编译器在引起`max<X>`实例化的位置也给与了提示，这里或许才是真正的错误——`max()`被文档标注为需要一个`operator <`。无法回答这一问题往往会导致第143页9.4节中描述的"error novel"，在这种情况下，编译器会提供完整的模板实例化历史，从实例化的初始原因一直到检测到错误的实际模板定义。然后，程序员需要确定究竟是哪个模板定义（可能就是模板的最初使用）真正存在错误。

模板类型检查背后的思想是在模板内部描述模板的要求，以便编译器在编译失败时确定是模板定义还是模板使用上出了问题。解决这一问题的一种方法是在模板自身的签名中使用`concept`来描述模板的要求：

```
template<typename T> requires LessThanComparable<T>
T max(T a, T b)
{
        return b < a ? a : b;
}

struct X { };
bool operator> (X, X);

int main()
{
        X a, b;
        X m = max(a, b);  // ERROR: X does not meet the LessThanComparable requirement
}
```

通过对模板参数`T`的要求描述，编译器就可以确信函数模板`max()`仅对`T`使用了它所期望使用者提供的那些操作（在本例中，`LessThanComparable`是对`operator <`的需求）。此外，在使用模板时，编译器可以检查提供的模板实参是否提供了`max()`函数模板在工作时所需的所有行为。通过解耦这一类型检查问题，对编译器来说就可以提供出更精准的问题诊断信息。

在上例中，`LessThanComparable`被称作为一个`concept`：它表示编译器能够检查的某种类型上的限制（在更广泛的场合，是对一个类型集合上的限制）。Concept系统有着各种不同的方式来指定。

在C++11标准化周期中，曾为concepts设计并实现了一个复杂的系统，它足够强大可以用来检查模板POI和模板定义。前者意味着，在上例中，我们可以提前捕捉到`main()`中的错误，并诊断出`X`不满足`LessThanComparable`的限制。而后者意味着，在处理`max()`模板时，编译器会检查是否使用了`LessThanComparable`这一`concept`所不允许的操作（如果违反了此约束，则抛出诊断信息）。该C++11提议最终被移出了语言标准，主要是因为各种实践上的考虑（比如，仍有许多次要规范议题，其解决措施威胁着已经延后的标准）。

在C++11最终发布后，委员会成员提出并开发了一项新提案（最初称作"concepts lite"）。该系统并非旨在基于施加的限制来检查模板的正确性。相反地，它仅仅聚焦于POI。所以对于我们的`max()`示例，如果实现中使用了`>`操作符，并不会导致错误。然而，在`main()`中的错误依然存在，这是因为`X`并不满足`LessThanComparable`的要求。这一崭新的concepts提议得以实现，并被认定为"Concepts TS(TS代表Technical Specification)"，称作"C++ extensions for Concepts"。目前，该项TS的核心要素已经被整合到了下一个标准(即C++20)的草案中。附录E

涵盖了本书出版时该草案中规定的语言特性。

17.9 反射元编程

在编程上下文中，反射是指以程序化的方式来检查程序功能的能力（例如，回答诸如某个类型是否是一个整型数？或是某个class类型包含了哪些非静态成员变量？）。元编程这门技艺是指“编写可以编程的程序”，它通常被用来量产新的代码。反射元编程是一种自动合成代码的技艺，它能根据程序的现有特性（通常是类型）自适应地进行适配。

在本书的第三部分，我们会去探索模板是如何达成一些简单的反射制式和元编程（某种意义上，模板实例化是一种元编程制式，因为它合成了新的代码）。然而，C++17模板的能力在面对反射时有着诸多限制（比如，没有办法回答“某个class类型包含了哪些非静态成员变量”这一问题），并且元编程的选项在各种方法中也常常不太方便（尤其是语法笨重且性能拉胯）。

认识到在这一领域对新机制的需求，C++标准委员会创建了一个研究小组(SG7)来探索更加强大的反射选项。该小组的章程后来也扩展到了元编程。以下是正在考虑的选项之一的示例:

```
template<typename T> void report(T p) {
        constexpr {
                std::meta::info infoT = reflexpr(T);
                for (std::meta::info : std::meta::data_members(infoT)) {
                        -> {
                                std::cout << (: std::meta::name(info) :)
                                                << ": " << p.(.info.) << '\n';
                        }
                }
        }
        // code will be injected here
}
```

代码里展示了相当多的新事物。首先，`constexpr{...}`结构会强制这一语句在编译期进行计算，但是如果它在一个模板中出现，就仅会在模板实例化时才进行计算。其次，`relfexpr()`操作符对隐晦类型`std::meta::info`产出了一个表达式，用于找到其背后实参的反射信息（本例中就是类型`T`）。标准库的元函数允许去查询这一元信息，`std::meta::data_members`就是那些标准元函数的其中一个，它会生成一个`std::meta::info`对象序列，它们描述了该操作数背后的非静态成员变量。因此，该for循环真正的进行了对`p`的非静态数据成员的遍历。

该系统元编程能力的核心是在各种作用域内“注入”代码的能力。结构 `->{...}` 注入了语句和(或)声明，触发了 `constexpr` 的计算。在本例中，意味着是在 `constexpr{...}` 结构之后。注入的代码片段可以包含某些模式，通过值计算后重新替换。在本例中，`(:...:)` 会产生一个字符串字面值(`std::meta::name(info)` 会产生一个类字符串的对象，它表示成员变量实体的非限定名称，在本例中由 `info` 表示)。同样，表达式 `(.info.)` 生成了一个标识符，命名由 `info` 表示的实体。其他生成类型的模式，像模板实参列表等也都支持。

对号入座之后，对 `X` 类型：

```
struct X
{
        int x;
        std::string s;
};
```

实例化函数模板 `report()` 就会生成下面的代码：

```
template<> void report(X const& p) {
        std::cout << "x" << ": " << "p.x" << '\n';
        std::cout << "s" << ": " << "p.s" << '\n';
}
```

也就是说，该函数会自动生成一个输出class类型的非静态成员变量的函数。

这些类型的功能有很多应用。可能会有类似的能力最终被语言标准所采用，但只能说未来可期。在本书撰

17.10 包设施

参数包在C++11所引入，但对它们的处理往往需要使用递归的模板技术。回顾第263页14.6节中讨论的代码大纲：

```
template<typename Head, typename... Remainder>
void f(Head &&h, Remainder&&... r)
{
        doSomething(h);
        if constexpr (sizeof...(r) != 0) {
                // handle the remainder recusively (perfectly forwarding the arguments):
                f(r...);
        }
}
```

在使用了C++17的编译期`if`语句之后(第134页节8.5)，这一示例变得非常简单，但是它依然保留了在编译时可能会进行的昂贵的递归实例化技术。

几个委员会的提案尝试在某种程度上简化这种情况。一个例子是引入一种表示从包中选择特定元素的符号。具体而言，对于一个包`P`，已经有人建议使用符号`P.[N]`来表示该包中的第`N+1`个元素。同样，也有提案用于表示包的“切片”（例如，使用符号`P.[b, e]`）。

在审查这些提案时，已经清楚地看到它们与上面讨论的反射元编程的概念有些交互。目前尚不清楚是否会向语言中添加特定的包选择机制，还是将提供满足此需求的元编程工具。

17.11 模块

另一个即将到来的重大扩展模块，虽然与模板的关系只是间接的，但在这里提及它仍然是值得的，因为模板库是其中最大的受益者之一。

当前，库接口是通过指定头文件、用`#include`宏来引入到编译单元。这种方法有几个缺点，但最令人反感的两个缺点是（a）界面文本的含义可能会被之前包含的代码（例如，通过宏）意外修改，以及（b）每次重新处理该文本都会迅速主导构建时间。

模块是一种特性，它允许将编译为特定于编译器的格式，然后这些接口可以“导入”到翻译单元中，而不会受到宏展开或通过意外的额外声明修改代码含义的影响。而且，编译器可以只读取与客户端代码相关的编译模块文件的部分内容，从而大大加快编译过程。

这里给出模块定义的表现形式：

```
module MyLib;

void helper() {
   ...
}

export inline void libFunc() {
  ...
  helper()
  ...
}
```

该模块导出了函数`libFunc()`，他可以被client代码这样使用：

```
import MyLib;

int main() {
  libFunc();
}
```

`libFunc()`对client代码可见，但是`helper()`却是不可见的，尽管编译模块的文件很可能包含了有关于`helper()`的信息来支持内联。

C++模块的提案正在路上，标准委员会将在C++17之后进行集成。制定此类提案的担忧之一是如何从头文件世界过渡到模块世界。已经有一些设施可以在一定程度上实现这一点（例如，在不将其内容作为模块的一部分的情况下包含头文件的能力），以及仍在讨论的其他设施（例如，从模块导出宏的能力）。

模块对模板库来说非常有用，这是因为模板大部分都完全定义在头文件中。即使包含一个像是`<vector>`这样的基础头文件，也要处理上万行C++代码（即使该头文件中只有少量的声明会被引用）。其他的流行库还要再高一个数量级。避免对所有代码都进行编译从而降低成本，将是处理大型复杂代码库的C++程序员的一大兴趣。