C++ 模板（第二版）

作者：David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor

翻译：五车书馆

https://github.com/Walton1128/CPP-Templates-2nd--

关于本书

本书第一版大约出版于 15 年前。起初我们的目的是编写一本对 C++工程师有帮助的 C++模板权威指南。目前该项目从以下几个方面来看是成功的：它的作用得到了不少读者的认可，也多次被推荐为参考书目，并屡获好评。

第一版已经很老了，虽然其中不少内容对 modern C++工程师依然很有帮助，但是鉴于 C++近年来的不断发展，比如 modern C++中从 C++11 到 C++14，再到 C++17 标准的制定，对第一版中部分内容的修订势在必行。

对于第二版，我们的宗旨依然没有变：提供 C++模板的权威指南，它既应该是一本内容全面的参考书，也应是一本容易理解的教程。只是这一次我们针对的是 modern C++，它要远远复杂于本书第一版出版时的那个 C++。

目前的 C++编程环境要好于本书第一版发布之时。比如这期间出现了一些深入探讨模板应用的书籍。更重要的是，我们可以从互联网上获取更多的 C++模板知识，以及基于模板的编程技术和应用实例。因此在这一版中，我们将重点关注那些可以被广泛应用的技术。

第一版中的部分内容目前来看已经过时了，因为 modern C++提供了可以完成相同功能，但又颇为简单的方法。因此这一部分内容会被从第二版中删除，不过不用担心，modern C++中对应的更为先进的内容会被加入进来。

尽管 C++模板的概念已经出现了 20 多年了，目前 C++开发者社区中依然会不断发现其在软件开发中新的应用场景。本书的目标之一是和读者分享这些内容，当然也希望能够启发读者产生新的理解和发现。

阅读本书前应该具备哪些知识？

为了更充分的利用本书，你需要已经比较熟悉 C++。因为我们会介绍该语言的某些细节，而不是它的一些基础知识。你应该了解类和继承的概念，并且能够使用标准库的输入输出流（IOstreams）和容器（container）进行编程。也应该已经了解 Modern C++的一些内容，比如 auto，decltype，移动语义和 lambda 表达式。在必要的时候，我们也会回顾下部分知识点，即使它们可能和模板没有直接关系。这能够确保无论是专家级程序员还是普通程序员都能很好的使用本书。

我们将主要介绍 C++的 2011，2014 和 2017 标准。但是由于在编写本书的时候，2017 标准才刚刚杀青，因此我们不会假设大部分读者都对它比较熟悉。以上每一次标准的修订都对模板的表现和使用方法有重要影响，我们会对其中和我们主旨相关的部分做简要介绍。不过，

我们的目的既不是介绍 modern C++的新标准，也不是详细介绍自 C++98 和 C++03 标准以来的所有变化。我们会从 modern C++新标准（C++11，C++14 和 C++17）出发来介绍模板，偶尔也会介绍那些只有 modern C++才有的新技术，或者新标准鼓励我们使用的新技术。

如何阅读本书

如果你是一个想去学习或者复习模板概念的 C++程序员，请仔细阅读第一部分。即使你已经非常熟悉模板，也请快速浏览下这一部分，这能够让你熟悉我们的写作风格以及我们常用的术语。该部分也涵盖了如何组织模板相关代码的内容。

根据你自己的学习方法，可以自行决定是先仔细学习第二部分的模板知识，还是直接阅读第三部分的实际编程技巧（必要时可以回头参考第二部分中的内容）。如果你购买本书的目的就是为了解开你心中的某些困惑的话，后一种方法可能更为实用。

正文所引用的附录中也包含有很多有用的信息。我们会在保证正确的情况下将它们展现地尽可能有趣一些。

根据我们的经验，从示例代码开始学习是一个很好的方法。因此在本书中你会看到很多的示例代码。其中一些只是为了展示某一抽象概念，因此可能只有几行，而另一些可能就是介绍了某种具体应用场景的完整程序了。对于后一种情况，代码所在文件会在相应的 C++注释中注明，你可以在如下链接中找到它们：http://www.tmplbook.com。

C++11，14 和 17 标准

第一版 C++标准发布于 1998 年，随后于 2003 年做了一次技术修订。因此“旧的 C++标准”指的就是 C++98 或者 C++03.

C++11 是在 ISO C++标准委员会主导下的第一版主要修订， 它引入了非常多的新特性。本书会讨论其中和模板有关的一部分新特性，包含：

- 变参模板（Variadic templates）
- 模板别名（Alias templates）
- 移动语义，右值引用和完美转发（Move semantics, rvalue references, and perfect forwarding）
- 标准类型萃取（Standard type traits）
-

C++14 和 C++17 紧随其后，也引入了一些新的语言特性，虽然不像 C++11 那么多。本书涉及到的和模板有关的新特性包含但不限于：

- 变量模板（Variable templates， C++14）
- 泛型 lambdas（Generic Lambdas， C++14）

- 类模板参数推断（Class template argument deduction， C++17）
- 编译期 if （Compile-time if， C++17）
- 折叠表达式（Fold expression， C++17）

我们甚至介绍了“Concept（模板接口）”这一确定将在 C++20 中包含的概念。

在编写本书的时候，C++11 和 C++14 已经被大多数主流编译器支持，C++17 的大部分特性也已被支持。不同编译器对新标准不同特性的实现仍然会有很大地不同。其中一些编译器可以编译本书中的大部分代码，少数编译器可能无法编译本书中的部分代码。不过我们认为这一问题会很快得到解决，主要是因为几乎所有的于程序员都会要求他们的供应商尽快去支持新标准。

虽然如此，C++这门语言依然会随时时间继续发展。C++委员会的专家们（不管他们是否会加入 C++标准委员会）也一直都在讨论着很多可以进一步优化这一语言的方法， 而且目前已经有一些备选方案会影响到模板，第 17 章将会介绍这一方面的发展趋势。

目录

目录

第一部分

基础知识

本部分将会介绍 C++模板的一些基础概念和语言特性。将会通过函数模板和类模板的例子来讨论模板的目的和概念。然后会继续介绍一些其他的模板特性，比如非类型模板参数（nontype template parameters），变参模板（variadic templates），typename 关键字和成员模板（member templates）。也会讨论如何处理移动语义（move semantics），如何声明模板参数，以及如何使用泛型代码实现可以在编译阶段执行的程序（compile-time programming）。在结尾处我们会针对一些术语和模板在实际中的应用，给应用开发工程师和泛型库的开发者们提供一些建议。

为什么要使用模板？

C++要求我们要用特定的类型来声明变量，函数以及其他一些内容。这样很多代码可能就只是处理的变量类型有所不同。比如对不同的数据类型，quicksort 的算法实现在结构上可能完全一样，不管是对整形的 array，还是字符串类型的 vector，只要他们所包含的内容之间可以相互比较。

如果你所使用的语言不支持这一泛型特性，你将可能只有如下糟糕的选择：

1. 你可以对不同的类型一遍又一遍的实现相同的算法。
2. 你可以在某一个公共基类（common base type，比如 Object 和 void*）里面实现通用的算法代码。
3. 你也可以使用特殊的预处理方法。

如果你是从其它语言转向 C++，你可能已经使用过以上几种或全部的方法了。然而他们都各有各的缺点：

1. 如果你一遍又一遍地实现相同算法，你就是在重复地制造轮子！你会犯相同的错误，而且为了避免犯更多的错误，你也不会倾向于使用复杂但是很高效的算法。
2. 如果在公共基类里实现统一的代码，就等于放弃了类型检查的好处。而且，有时候某些类必须要从某些特殊的基类派生出来，这会进一步增加维护代码的复杂度。
3. 如果采用预处理的方式，你需要实现一些“愚蠢的文本替换机制”，这将很难兼顾作用域和类型检查，因此也就更容易引发奇怪的语义错误。

而模板这一方案就不会有这些问题。模板是为了一种或者多种未明确定义的类型而定义的函数或者类。在使用模板时，需要显式地或者隐式地指定模板参数。由于模板是 C++的语言特性，类型和作用域检查将依然得到支持。

目前模板正在被广泛使用。比如在 C++标准库中，几乎所有的代码都用到了模板。标准库提

供了一些排序算法来排序某种特定类型的值或者对象，也提供类一些数据结构（亦称容器）来维护某种特定类型的元素，对于字符串而言，这一“特定类型”指的就是“字符”。当然这只是最基础的功能。模板还允许我们参数化函数或者类的行为，优化代码以及参数化其他信息。这些高级特性会在后面某些章节介绍，我们接下来将先从一些简单模板开始介绍。

第 1 章 函数模板（Function Templates）

本章将介绍函数模板。函数模板是被参数化的函数，因此他们代表的是一组具有相似行为的函数。

1.1 函数模板初探

函数模板提供了适用于不同数据类型的函数行为。也就是说，函数模板代表的是一组函数。除了某些信息未被明确指定之外，他们看起来很像普通函数。这些未被指定的信息就是被参数化的信息。我们将通过下面一个简单的例子来说明这一问题。

1.1.1 定义模板

以下就是一个函数模板，它返回两个数之中的最大值：

```
template<typename T>
T max (T a, T b)
{
    // 如果 b < a，返回 a，否则返回 b
    return b < a ? a : b;
}
```

这个模板定义了一组函数，它们都返回函数的两个参数中值较大的那一个。这两个参数的类型并没有被明确指定，而是被表示为模板参数 T。如你所见，模板参数必须按照如下语法声明：

```
template< 由逗号分割的模板参数>
```

在我们的例子中，模板参数是 *typename T*。请留意<和>的使用，它们在这里被称为尖括号。关键字 *typename* 标识了一个*类型参数*。这是到目前为止 C++中模板参数最典型的用法，当然也有其他参数（非类型模板参数），我们将在第 3 章介绍。

在这里 T 是类型参数。你可以用任意标识作为类型参数名，但是习惯上是用 T。类型参数可以代表任意类型，它在模板被调用的时候决定。但是该类型（可以是基础类型，类或者其它类型）应该支持模板中用到的运算符。在本例中，类型 T 必须支持小于运算符，因为 a 和 b 在做比较时用到了它。例子中不太容易看出的一点是，为了支持返回值，T 还应该是可拷贝的。

由于历史原因，除了 typename 之外你还可以使用 class 来定义类型参数。关键字 typename 在 C++98 标准发展过程中引入的较晚。在那之前，关键字 class 是唯一可以用来定义类型参

数的方法，而且目前这一方法依然有效。因此模板 max()也可以被定义成如下等效的方式：

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

从语义上来讲，这样写不会有任何不同。因此，在这里你依然可以使用任意类型作为类型参数。只是用 class 的话可能会引起一些歧义（T 并不是只能是 class 类型），你应该优先使用 typename。但是与定义 class 的情况不同，在声明模板类型参数的时候，不可以用关键字 struct 取代 typename。

1.1.2 使用模板

下面的程序展示了使用模板的方法：

```
#include "max1.hpp"
#include <iostream>
#include <string>
int main()
{
    int i = 42;
    std::cout << "max(7,i): " << ::max(7,i) << ' \n' ;
    double f1 = 3.4;
    double f2 = -6.7;
    std::cout << "max(f1,f2): " << ::max(f1,f2) << ' \n' ;
    std::string s1 = "mathematics";
    std::string s2 = "math";
    std::cout << "max(s1,s2): " << ::max(s1,s2) << ' \n' ;
}
```

在这段代码中，max()被调用了三次：一次是比较两个 int，一次是比较两个 double，还有一次是比较两个 std::string。每一次都会算出最大值。下面是输出结果：

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

注意在调用 max()模板的时候使用了作用域限制符::。这样程序将会在全局作用域中查找 max()模板。否则的话，在某些情况下标准库中的 std::max()模板将会被调用，或者有时候不太容易确定具体哪一个模板会被调用。

在编译阶段，模板并不是被编译成一个可以支持多种类型的实体。而是对每一个用于该模板的类型都会产生一个独立的实体。因此在本例中，max()会被编译出三个实体，因为它被用于三种类型。比如第一次调用时：

```
int i = 42;
…
max(7,i)
…
```

函数模板的类型参数是 int。因此语义上等效于调用了如下函数：

```
int max (int a, int b)
{
    return b < a ? a : b;
}
```

以上用具体类型取代模板类型参数的过程叫做“实例化”。它会产生模板的一个实例。

值得注意的是，模板的实例化不需要程序员做额外的请求，只是简单的使用函数模板就会触发这一实例化过程。

同样的，另外两次调用也会分别为 double 和 std::string 各实例化出一个实例，就像是分别定义了下面两个函数一样：

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

另外，只要结果是有意义的，void 作为模板参数也是有效的。比如:

```
template<typename T>
T foo(T*)
{ }
void* vp = nullptr;
foo(vp); // OK: 模板参数被推断为 void
foo(void*)
```

1.1.3 两阶段编译检查（Two-Phase Translation）

在实例化模板的时候，如果模板参数类型不支持所有模板中用到的操作符，将会遇到编译期错误。比如：

```
std::complex<float> c1, c2; // std::complex<>没有提供小于运算符
…
::max(c1,c2); // 编译期 ERROR
```

但是在定义的地方并没有遇到错误提示。这是因为模板是被分两步编译的：

1. 在模板定义阶段，模板的检查并不包含类型参数的检查。只包含下面几个方面：
   - 语法检查。比如少了分号。
   - 使用了未定义的不依赖于模板参数的名称（类型名，函数名，……）。
   - 未使用模板参数的 static assertions。

2. 在模板实例化阶段，为确保所有代码都是有效的，模板会再次被检查，尤其是那些依赖

于类型参数的部分。

比如：

```
template<typename T>
void foo(T t)
{
    undeclared(); // 如果 undeclared()未定义，第一阶段就会报错，因为与模板参数无关
    undeclared(t); //如果 undeclared(t)未定义，第二阶段会报错，因为与模板参数有关
    static_assert(sizeof(int) > 10,"int too small");  // 与模板参数无关，总是报错
    static_assert(sizeof(T) > 10, "T too small"); //与模板参数有关，只会在第二阶段报错
}
```

名称被检查两次这一现象被称为“两阶段查找”，在 14.3.1 节中会进行更细致的讨论。

需要注意的是，有些编译器并不会执行第一阶段中的所有检查。因此如果模板没有被至少实例化一次的话，你可能一直都不会发现代码中的常规错误。

1.1.4 编译和链接

两阶段的编译检查给模板的处理带来了一个问题：当实例化一个模板的时候，编译器需要（一定程度上）看到模板的完整定义。这不同于函数编译和链接分离的思想，函数在编译阶段只需要声明就够了。第 9 章将讨论如何应对这一问题。我们将暂时采取最简单的方法：将模板的实现写在头文件里。

1.2 模板参数推断

当我们调用形如 max()的函数模板来处理某些变量时，模板参数将由被传递的调用参数决定。如果我们传递两个 int 类型的参数给模板函数，C++编译器会将模板参数 T 推断为 int。

不过 T 可能只是实际传递的函数参数类型的一部分。比如我们定义了如下接受常量引用作为函数参数的模板：

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

此时如果我们传递 int 类型的调用参数，由于调用参数和 int const &匹配，类型参数 T 将被

推断为 int。

类型推断中的类型转换

在类型推断的时候自动的类型转换是受限制的：

- 如果调用参数是按引用传递的，任何类型转换都不被允许。通过模板类型参数 T 定义的两个参数，它们实参的类型必须完全一样。

- 如果调用参数是按值传递的，那么只有退化（decay）这一类简单转换是被允许的：const 和 volatile 限制符会被忽略，引用被转换成被引用的类型，raw array 和函数被转换为相应的指针类型。通过模板类型参数 T 定义的两个参数，它们实参的类型在退化（decay）后必须一样。

例如：

```
template<typename T>
T max (T a, T b);
…
int const c = 42;
Int i = 1;   //原书缺少 i 的定义
max(i, c); // OK: T 被推断为 int，c 中的 const 被 decay 掉
max(c, c); // OK:  T 被推断为 int
int& ir = i;
max(i, ir); // OK: T 被推断为 int， ir 中的引用被 decay 掉
int arr[4];
foo(&i, arr); // OK: T 被推断为 int*
```

但是像下面这样是错误的：

```
max(4, 7.2); // ERROR: 不确定 T 该被推断为 int 还是 double
std::string s;
foo("hello", s); //ERROR: 不确定 T 该被推断为 const[6] 还是 std::string
```

有两种办法解决以上错误：

1. 对参数做类型转换

   ```
   max(static_cast<double>(4), 7.2); // OK
   ```

2. 显式地指出类型参数 T 的类型，这样编译器就不再会去做类型推导。

   ```
   max<double>(4, 7.2); // OK
   ```

3. 指明调用参数可能有不同的类型（多个模板参数）。

1.3 节会进一步讨论这些内容。7.2 节和第 15 章会更详细的介绍基于模板类型推断的类型转换规则。

对默认调用参数的类型推断

需要注意的是，类型推断并不适用于默认调用参数。例如：

```
template<typename T>
void f(T = "");
...
f(1); // OK: T 被推断为 int，调用 f<int> (1)
f(); // ERROR: 无法推断 T 的类型
```

为应对这一情况，你需要给模板类型参数也声明一个默认参数，1.4 节会介绍这一内容：

```
template<typename T = std::string>
void f(T = "");
…
f(); // OK
```

1.3 多个模板参数

目前我们看到了与函数模板相关的两组参数：

1. 模板参数，定义在函数模板前面的尖括号里：

   ```
   template<typename T> // T 是模板参数
   ```

2. 调用参数，定义在函数模板名称后面的圆括号里：

   ```
   T max (T a, T b) // a 和 b 是调用参数
   ```

模板参数可以是一个或者多个。比如，你可以定义这样一个 max()模板，它可能接受两个不同类型的调用参数：

```
template<typename T1, typename T2>
T1 max (T1 a, T2 b)
{
    return b < a ? a : b;
}
…
auto m = ::max(4, 7.2); // OK，但是返回类型是第一个模板参数 T1 的类型
```

看上去如你所愿，它可以接受两个不同类型的调用参数。但是如示例代码所示，这也导致了一个问题。如果你使用其中一个类型参数的类型作为返回类型，不管是不是和调用者预期地一样，当应该返回另一个类型的值的时候，返回值会被做类型转换。这将导致返回值的具体类型和参数的传递顺序有关。如果传递 66.66 和 42 给这个函数模板，返回值是 double 类型的 66.66，但是如果传递 42 和 66.66，返回值却是 int 类型的 66。

C++提供了多种应对这一问题的方法：

1. 引入第三个模板参数作为返回类型。
2. 让编译器找出返回类型。

3. 将返回类型定义为两个参数类型的“公共类型”

下面将逐一进行讨论。

1.3.1 作为返回类型的模板参数

按照之前的讨论，模板类型推断允许我们像调用普通函数一样调用函数模板：我们可以不去显式的指出模板参数的类型。

但是也提到，我们也可以显式的指出模板参数的类型：

```
template<typename T>
T max (T a, T b);
…
::max<double>(4, 7.2); // max()被针对 double 实例化
```

当模板参数和调用参数之间没有必然的联系，且模板参数不能确定的时候，就要显式的指明模板参数。比如你可以引入第三个模板来指定函数模板的返回类型：

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

但是模板类型推断不会考虑返回类型，而 RT 又没有被用作调用参数的类型。因此 RT 不会被推断。这样就必须显式的指明模板参数的类型。比如：

```
template<typename T1, typename T2, typename RT>
RT max (T1 a, T2 b);
…
::max<int,double,double>(4, 7.2); // OK, 但是太繁琐
```

到目前为止，我们看到的情况是，要么所有模板参数都被显式指定，要么一个都不指定。另一种办法是只指定第一个模板参数的类型，其余参数的类型通过推断获得。通常而言，我们必须显式指定所有模板参数的类型，直到某一个模板参数的类型可以被推断出来为止。因此，如果你改变了上面例子中的模板参数顺序，调用时只需要指定返回值的类型就可以了：

```
template<typename RT, typename T1, typename T2>
RT max (T1 a, T2 b);
…
::max<double>(4, 7.2) //OK: 返回类型是 double，T1 和 T2 根据调用参数推断
```

在本例中，调用 max<double>时，显式的指明了 RT 的类型是 double，T1 和 T2 则基于传入调用参数的类型被推断为 int 和 double。

然而改进版的 max()并没有带来显著的变化。使用单模板参数的版本，即使传入的两个调用参数的类型不同，你依然可以显式的指定模板参数类型（也作为返回类型）。因此为了简洁，我们最好还是使用单模板参数的版本。（在接下来讨论其它模板问题的时候，我们也会基于单模板参数的版本）

对于模板参数推断的详细介绍，请参见第 15 章。

1.3.2 返回类型推断

如果返回类型是由模板参数决定的，那么推断返回类型最简单也是最好的办法就是让编译器来做这件事。从 C++14 开始，这成为可能，而且不需要把返回类型声明为任何模板参数类型（不过你需要声明返回类型为 auto）：

```
template<typename T1, typename T2>
auto max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

事实上，在不使用*尾置返回类型（trailing return type）的情况下*将 auto 用于返回类型，要求返回类型必须能够通过函数体中的返回语句推断出来。当然，这首先要求返回类型能够从函数体中推断出来。因此，必须要有这样可以用来推断返回类型的返回语句，而且多个返回语句之间的推断结果必须一致。

在 C++14 之前，要想让编译器推断出返回类型，就必须让或多或少的函数实现成为函数声明的一部分。在 C++11 中，尾置返回类型（trailing return type）允许我们使用函数的调用参数。也就是说，我们可以基于运算符?:的结果声明返回类型：

```
template<typename T1, typename T2>
auto max (T1 a, T2 b) -> decltype(b<a?a:b)
{
    return b < a ? a : b;
}
```

在这里，返回类型是由运算符?:的结果决定的，这虽然复杂但是可以得到想要的结果。

需要注意的是

```
template<typename T1, typename T2>
auto max (T1 a, T2 b) -> decltype(b<a?a:b);
```

是一个声明，编译器在编译阶段会根据运算符?:的返回结果来决定实际的返回类型。不过具体的实现可以有所不同，事实上用 true 作为运算符?:的条件就足够了：

```
template<typename T1, typename T2>
auto max (T1 a, T2 b) -> decltype(true?a:b);
```

但是在某些情况下会有一个严重的问题：由于 T 可能是引用类型，返回类型就也可能被推断为引用类型。因此你应该返回的是 decay 后的 T，像下面这样：

```
#include <type_traits>
template<typename T1, typename T2>
```

```
auto max (T1 a, T2 b) -> typename std::decay<decltype(true? a:b)>::type
{
    return b < a ? a : b;
}
```

在这里我们用到了类型萃取（type trait）std::decay<>，它返回其 type 成员作为目标类型，定义在标准库<type_trait>中（参见 D.5）。由于其 type 成员是一个类型，为了获取其结果，需要用关键字 typename 修饰这个表达式。

在这里请注意，在初始化 auto 变量的时候其类型总是退化之后了的类型。当返回类型是 auto 的时候也是这样。用 auto 作为返回结果的效果就像下面这样，a 的类型将被推断为 i 退化后的类型，也就是 int：

```
int i = 42;
int const& ir = i; // ir 是 i 的引用
auto a = ir; // a 的类型是 it decay 之后的类型，也就是 int
```

1.3.3 将返回类型声明为公共类型（Common Type）

从 C++11 开始，标准库提供了一种指定“更一般类型”的方式。std::common_type<>::type 产生的类型是他的两个模板参数的公共类型。比如：

```
#include <type_traits>
template<typename T1, typename T2>
std::common_type_t<T1,T2> max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

同样的，std::common_type 也是一个类型萃取（type trait），定义在<type_traits>中，它返回一个结构体，结构体的 type 成员被用作目标类型。因此其主要应用场景如下：

```
typename std::common_type<T1,T2>::type //since C++11
```

不过从 C++14 开始，你可以简化“萃取”的用法，只要在后面加个_t，就可以省掉 typename 和::type（参见 2.8 节），简化后的版本变成：

```
std::common_type_t<T1,T2>    // equivalent since C++14
```

std::common_type<>的实现用到了一些比较取巧的模板编程手法，具体请参见 25.5.2 节。它根据运算符?:的语法规则或者对某些类型的特化来决定目标类型。因此::max(4, 7.2)和::max(7.2, 4)都返回 double 类型的 7.2. 需要注意的是，std::common_type<>的结果也是退化的，具体参见 D.5。

1.4 默认模板参数

你也可以给模板参数指定默认值。这些默认值被称为默认模板参数并且可以用于任意类型的模板。它们甚至可以根据其前面的模板参数来决定自己的类型。

比如如果你想将前述定义返回类型的方法和多模板参数一起使用，你可以为返回类型引入一个模板参数 RT，并将其默认类型声明为其它两个模板参数的公共类型。同样地，我们也有多种实现方法：

1. 我们可以直接使用运算符?:。不过由于我们必须在调用参数 a 和 b 被声明之前使用运算符?:，我们只能像下面这样：

```
#include <type_traits>
template<typename T1, typename T2, typename RT =
std::decay_t<decltype(true ? T1() : T2())>>
RT max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

请注意在这里我们用到了 std::decay_t<>来确保返回的值不是引用类型。

同样值得注意的是，这一实现方式要求我们能够调用两个模板参数的默认构造参数。还有另一种方法，使用 std::declval，不过这将使得声明部分变得更加复杂。作为例子可以参见 11.2.3 节。

2. 我们也可以利用类型萃取 std::common_type<>作为返回类型的默认值：

```
#include <type_traits>
template<typename T1, typename T2, typename RT =
std::common_type_t<T1,T2>>
RT max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

在这里 std::common_type<>也是会做类型退化的，因此返回类型不会是引用。

在以上两种情况下，作为调用者，你即可以使用 RT 的默认值作为返回类型：

```
auto a = ::max(4, 7.2);
```

也可以显式的指出所有的模板参数的类型：

```
auto b = ::max<double,int,long double>(7.2, 4);
```

但是，我们再次遇到这样一个问题：为了显式指出返回类型，我们必须显式的指出全部三个模板参数的类型。因此我们希望能够将返回类型作为第一个模板参数，并且依然能够从其它

两个模板参数推断出它的类型。

原则上这是可行的，即使后面的模板参数没有默认值，我们依然可以让第一个模板参数有默认值：

```
template<typename RT = long, typename T1, typename T2>
RT max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

基于这个定义，你可以这样调用：

```
int i; long l;
...
max(i, l); // 返回值类型是 long (RT 的默认值)
max<int>(4, 42); //返回 int，因为其被显式指定
```

但是只有当模板参数具有一个“天生的”默认值时，这才有意义。我们真正想要的是从前面的模板参数推导出想要的默认值。原则是这也是可行的，就如 26.5.1 节讨论的那样，但是他是基于类型萃取的，并且会使问变得更加复杂。

基于以上原因，最好也是最简单的办法就是像 1.3.2 节讨论的那样让编译器来推断出返回类型。

1.5 函数模板的重载

像普通函数一样，模板也是可以重载的。也就是说，你可以定义多个有相同函数名的函数，当实际调用的时候，由 C++编译器负责决定具体该调用哪一个函数。即使在不考虑模板的时候，这一决策过程也可能异常复杂。本节将讨论包含模板的重载。如果你还不熟悉没有模板时的重载规则，请先看一下附录 C， 那里比较详细的总结了模板的解析过程。

下面几行程序展示了函数模板的重载：

```
// maximum of two int values:
int max (int a, int b)
{
    return b < a ? a : b;
}
// maximum of two values of any type:
template<typename T>
T max (T a, T b)
{
    return b < a ? a : b;
}
```

```
int main()
{
    ::max(7, 42); // calls the nontemplate for two ints
    ::max(7.0, 42.0); // calls max<double> (by argument deduction)
    ::max('a', 'b'); //calls max<char> (by argument deduction)
    ::max<>(7, 42); // calls max<int> (by argumentdeduction)
    ::max<double>(7, 42); // calls max<double> (no argumentdeduction)
    ::max('a', 42.7); //calls the nontemplate for two ints
}
```

如你所见，***一个非模板函数可以和一个与其同名的函数模板共存，并且这个同名的函数模板可以被实例化为与非模板函数具有相同类型的调用参数。在所有其它因素都相同的情况下，模板解析过程将优先选择非模板函数，而不是从模板实例化出来的函数。***第一个调用就属于这种情况：

```
::max(7, 42); // both int values match the nontemplate function perfectly
```

如果模板可以实例化出一个更匹配的函数，那么就会选择这个模板。正如第二和第三次调用 max()时那样：

```
::max(7.0, 42.0); // calls the max<double> (by argument deduction)
::max('a', 'b'); //calls the max<char> (by argument deduction)
```

在这里模板更匹配一些，因为它不需要从 double 和 char 到 int 的转换。（参见 C.2 中的模板解析过程）

也可以显式指定一个空的模板列表。这表明它会被解析成一个模板调用，其所有的模板参数会被通过调用参数推断出来：

```
::max<>(7, 42); // calls max<int> (by argument deduction)
```

由于在模板参数推断时不允许自动类型转换，而常规函数是允许的，因此最后一个调用会选择非模板参函数（'a'和 42.7 都被转换成 int）：

```
::max('a', 42.7); //only the nontemplate function allows nontrivial
conversions
```

一个有趣的例子是我们可以专门为 max()实现一个可以显式指定返回值类型的模板：

```
template<typename T1, typename T2>
auto max (T1 a, T2 b)
{
    return b < a ? a : b;
}
template<typename RT, typename T1, typename T2>
RT max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

现在我们可以像下面这样调用 max():

```
auto a = ::max(4, 7.2); // uses first template
auto b = ::max<long double>(7.2, 4); // uses second template
```

但是像下面这样调用的话：

```
auto c = ::max<int>(4, 7.2); // ERROR: both function templates match
```

两个模板都是匹配的，这会导致模板解析过程不知道该调用哪一个模板，从而导致未知错误。因此当重载函数模板的时候，你要保证对任意一个调用，都只会有一个模板匹配。

一个比较有用的例子是为指针和 C 字符串重载 max()模板：

```
#include <cstring>
#include <string>
// maximum of two values of any type:
template<typename T>
T max (T a, T b)
{
    return b < a ? a : b;
}
 // maximum of two pointers:
template<typename T>
T* max (T* a, T* b)
{
    return *b < *a ? a : b;
}
// maximum of two C-strings:
char const* max (char const* a, char const* b)
{
    return std::strcmp(b,a) < 0 ? a : b;
}

int main ()
{
    int a = 7;
    int b = 42;
    auto m1 = ::max(a,b); // max() for two values of type int
    std::string s1 = "hey"; "
    std::string s2 = "you"; "
    auto m2 = ::max(s1,s2); // max() for two values of type std::string
    int* p1 = &b;
    int* p2 = &a;
    auto m3 = ::max(p1,p2); // max() for two pointers
    char const* x = hello"; "
```

```
    char const* y = "world"; "
    auto m4 = ::max(x,y);  // max() for two C-strings
}
```

注意上面所有 max()的重载模板中，调用参数都是按值传递的。通常而言，在重载模板的时候，要尽可能少地做改动。你应该只是改变模板参数的个数或者显式的指定某些模板参数。否则，可能会遇到意想不到的问题。比如，如果你实现了一个按引用传递的 max()模板，然后又重载了一个按值传递两个 C 字符串作为参数的模板，你不能用接受三个参数的模板来计算三个 C 字符串的最大值：

```
#include <cstring>
// maximum of two values of any type (call-by-reference)
template<typenameT> T const& max (T const& a, T const& b)
{
    return b < a ? a : b;
}
// maximum of two C-strings (call-by-value)
char const* max (char const* a, char const* b)
{
    return std::strcmp(b,a) < 0 ? a : b;
}
 // maximum of three values of any type (call-by-reference)
template<typename T>
T const& max (T const& a, T const& b, T const& c)
{
    return max (max(a,b), c); // error if max(a,b) uses call-by-value
}

int main ()
{
    auto m1 = ::max(7, 42, 68); // OK
    char const* s1 = "frederic";
    char const* s2 = "anica";
    char const* s3 = "lucas";
    auto m2 = ::max(s1, s2, s3); //run-time ERROR
}
```

问题在于当用三个 C 字符串作为参数调用 max()的时候，

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

会遇到 run-time error, 这是因为对 C 字符串，max(max(a, b), c)会创建一个用于返回的临时局部变量，而在返回语句接受后，这个临时变量会被销毁，导致 man()使用了一个悬空的引用。不幸的是，这个错误几乎在所有情况下都不太容易被发现。

作为对比，在求三个 int 最大值的 max()调用中，则不会遇到这个问题。这里虽然也会创建

三个临时变量，但是这三个临时变量是在 main()里面创建的，而且会一直持续到语句结束。

这只是模板解析规则和期望结果不一致的一个例子。

再者，需要确保函数模板在被调用时，其已经在前方某处定义。这是由于在我们调用某个模板时，其相关定义不一定是可见的。比如我们定义了一个三参数的 max(), 由于它看不到适用于两个 int 的 max()，因此它最终会调用两个参数的模板函数：

```
#include <iostream>
// maximum of two values of any type:
template<typename T>
T max (T a, T b)
{
    std::cout << "max<T>() \n";
    return b < a ? a : b;
}
// maximum of three values of any type:
template<typename T>
T max (T a, T b, T c)
{
    return max (max(a,b), c); // uses the template version even for ints
} //because the following declaration comes
// too late:
// maximum of two int values:
int max (int a, int b)
{
    std::cout << "max(int,int) \n";
    return b < a ? a : b;
}
int main()
{
    ::max(47,11,33); // OOPS: uses max<T>() instead of max(int,int)
}
```

第 13.2 节会对这背后的原因做详细讨论。

1.6 难道，我们不应该...?

或许，即使是这些简单的函数模板，也会导致比较多的问题。在这里有三个很常见的问题值得我们讨论。

1.6.1 按值传递还是按引用传递？

你可能会比较困惑，为什么我们声明的函数通常都是按值传递，而不是按引用传递。通常而言，建议将按引用传递用于除简单类型（比如基础类型和 std::string_view）以外的类型，这样可以免除不必要的拷贝成本。

不过出于以下原因，按值传递通常更好一些：

- 语法简单。
- 编译器能够更好地进行优化。
- 移动语义通常使拷贝成本比较低。
- 某些情况下可能没有拷贝或者移动。

再有就是，对于模板，还有一些特有情况：

- 模板既可以用于简单类型，也可以用于复杂类型，因此如果默认选择适合于复杂类型可能方式，可能会对简单类型产生不利影响。
- 作为调用者，你通常可以使用 std::ref()和 std::cref()（参见 7.3 节）来按引用传递参数。
- 虽然按值传递 string literal 和 raw array 经常会遇到问题，但是按照引用传递它们通常只会遇到更大的问题。第 7 章会对此做进一步讨论。在本书中，除了某些不得不用按引用传递的地方，我们会尽量使用按值传递。

1.6.2 为什么不适用 inline？

通常而言，函数模板不需要被声明成 inline。不同于非 inline 函数，我们可以把非 inline 的函数模板定义在头文件里，然后在多个编译单元里 include 这个文件。

唯一一个例外是模板对某些类型的全特化，这时候最终的 code 不在是“泛型”的（所有的模板参数都已被指定）。详情请参见 9.2 节。

严格地从语言角度来看，inline 只意味着在程序中函数的定义可以出现很多次。不过它也给了编译器一个暗示，在调用该函数的地方函数应该被展开成 inline 的：这样做在某些情况下可以提高效率，但是在另一些情况下也可能降低效率。现代编译器在没有关键字 inline 暗示的情况下，通常也可以很好的决定是否将函数展开成 inline 的。当然，编译器在做决定的时候依然会将关键字 inline 纳入考虑因素。

1.6.3 为什么不用 constexpr？

从 C++11 开始，你可以通过使用关键字 constexpr 来在编译阶段进行某些计算。对于很多模板，这是有意义的。

比如为了可以在编译阶段使用求最大值的函数，你必须将其定义成下面这样：

```
template<typename T1, typename T2>
constexpr auto max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

如此你就可以在编译阶段的上下文中，实时地使用这个求最大值的函数模板：

```
int a[::max(sizeof(char),1000u)];
```

或者指定 std::array<>的大小：

```
std::array<std::string, ::max(sizeof(char),1000u)> arr;
```

在这里我们传递的 1000 是 unsigned int 类型，这样可以避免直接比较一个有符号数值和一个无符号数值时产生的警报。

8.2 节还会讨论其它一些使用 constexpr 的例子。但是，为了更专注于模板的基本原理，我们接下来在讨论模板特性的时候会跳过 constexpr。

1.7 总结

- 函数模板定义了一组适用于不同类型的函数。
- 当向模板函数传递变量时，函数模板会自行推断模板参数的类型，来决定去实例化出那种类型的函数。
- 你也可以显式的指出模板参数的类型。
- 你可以定义模板参数的默认值。这个默认值可以使用该模板参数前面的模板参数的类型，而且其后面的模板参数可以没有默认值。
- 函数模板可以被重载。
- 当定义新的函数模板来重载已有的函数模板时，必须要确保在任何调用情况下都只有一个模板是最匹配的。
- 当你重载函数模板的时候，最好只是显式地指出了模板参数得了类型。
- 确保在调用某个函数模板之前，编译器已经看到了相对应的模板定义。

第 2 章 类模板（Class Templates）

和函数类似，类也可以被一个或多个类型参数化。容器类（Container classes）就是典型的一个例子，它可以被用来处理某一指定类型的元素。通过使用类模板，你也可以实现适用于多种类型的容器类。在本章中，我们将以一个栈（stack）的例子来展示类模板的使用。

2.1 Stack 类模板的实现

和函数模板一样，我们把类模板 Stack<>的声明和定义都放在头文件里：

```
#include <vector>
#include <cassert>
template<typename T>
class Stack {
    private:
        std::vector<T> elems; // elements
    public:
        void push(T const& elem); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
};
template<typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
template<typename T>
void Stack<T>::pop ()
{
    assert(!elems.empty());
    elems.pop_back(); // remove last element
}
template<typename T>
T const& Stack<T>::top () const
{
    assert(!elems.empty());
    return elems.back(); // return copy of last element
}
```

如上所示，这个类模板是通过使用一个 C++标准库的类模板 vector<>实现的。这样我们就不需要自己来实现内存管理，拷贝构造函数和赋值构造函数了，从而可以把更多的精力放在这个类模板的接口实现上。

2.1.1 声明一个类模板

声明类模板和声明函数模板类似：在开始定义具体内容之前，需要先声明一个或者多个作为模板的类型参数的标识符。同样地，这一标识符通常用 T 表示：

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

在这里，同样可以用关键字 class 取代 typename：

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

在类模板内部，T 可以像普通类型一样被用来声明成员变量和成员函数。在这个例子中，T 被用于声明 vector 中元素的类型，用于声明成员函数 push()的参数类型，也被用于成员函数 top 的返回类型：

```
template<typename T>
class Stack {
    private:
        std::vector<T> elems; // elements
    public:
        void push(T const& elem); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
};
```

这个类的类型是 Stack<T>， 其中 T 是模板参数。在将这个 Stack<T>类型用于声明的时候，除非可以推断出模板参数的类型，否则就必须使用 Stack<T>（Stack 后面必须跟着<T>）。不过，如果在类模板内部使用 Stack 而不是 Stack<T>，表明这个内部类的模板参数类型和模板类的参数类型相同（细节请参见 13.2.3 节）。

比如，如果需要定义自己的复制构造函数和赋值构造函数，通常应该定义成这样：

```
template<typename T>
```

```
class Stack {
    ...
    Stack (Stack const&); // copy constructor
    Stack& operator= (Stack const&); // assignment operator
    ...
};
```

它和下面的定义是等效的：

```
template<typename T>
class Stack {
    ...
    Stack (Stack<T> const&); // copy constructor
    Stack<T>& operator= (Stack<T> const&); // assignment operator
    ...
};
```

一般<T>暗示要对某些模板参数做特殊处理，所以最好还是使用第一种方式。

但是如果在类模板的外面，就需要这样定义：

```
template<typename T>
bool operator== (Stack<T> const& lhs, Stack<T> const& rhs);
```

注意在只需要类的名字而不是类型的地方，可以只用 Stack。这和声明构造函数和析构函数的情况相同。

另外，不同于非模板类，不可以在函数内部或者块作用域内（{...}）声明和定义模板。通常模板只能定义在 global/namespace 作用域，或者是其它类的声明里面（相关细节请参见 12.1 节）。

2.1.2 成员函数的实现

定义类模板的成员函数时，必须指出它是一个模板，也必须使用该类模板的所有类型限制。因此，要像下面这样定义 Stack<T>的成员函数 push():

```
template<typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
```

这里调用了其 vector 成员的 push_back()方法，它向 vector 的尾部追加一个元素。

注意 vector 的 pop_back()方法只是删除掉尾部的元素，并不会返回这一元素。这主要是为了异常安全（exception safety）。实现一个异常安全并且能够返回被删除元素的 pop()方法是不可能的（Tom Cargill 首次在 [CargillExceptionSafety] 中对这一问题进行了讨论，

[SutterExceptional]的条款 10 也对这进行了讨论。）。不过如果忽略掉这一风险，我们依然可以实现一个返回被删除元素的 pop()。为了达到这一目的，我们只需要用 T 定义一个和 vector 元素有相同类型的局部变量就可以了：

```
template<typename T>
T Stack<T>::pop ()
{
    assert(!elems.empty());
    T elem = elems.back(); // save copy of last element
    elems.pop_back(); // remove last element
    return elem; // return copy of saved element
}
```

由于 vector 的 back()（返回其最后一个元素）和 pop_back()（删除最后一个元素）方法在 vector 为空的时候行为未定义，因此需要对 vector 是否为空进行测试。在程序中我们断言（assert）vector 不能为空，这样可以确保不会对空的 Stack 调用 pop()方法。在 top()中也是这样，它返回（但不删除）首元素：

```
template<typename T>
T const& Stack<T>::top () const
{
    assert(!elems.empty());
    return elems.back(); // return copy of last element
}
```

当然，就如同其它成员函数一样，你也可以把类模板的成员函数以内联函数的形式实现在类模板的内部。比如：

```
template<typename T>
class Stack {
    ...
    void push (T const& elem) {
        elems.push_back(elem); // append copy of passed elem
    }
    ...
};
```

2.2 Stack 类模板的使用

直到 C++17，在使用类模板的对象的时候都需要显式的指明模板参数。下面的例子展示了该如何使 用 Stack<>类模板：

```
#include "stack1.hpp"
#include <iostream>
#include <string>
int main()
{
```

```
    Stack< int> intStack; // stack of ints
    Stack<std::string> stringStack; // stack of strings
    // manipulate int stack
    intStack.push(7);
    std::cout << intStack.top() << ' \n' ;
    // manipulate string stack
    stringStack.push("hello");
    std::cout << stringStack.top() << ' \n' ;
    stringStack.pop(); The default constructor, push(), and top() are
                       instantiated for both int and strings.
}
                       However, pop()is instantiated only for strings.
```

通过声明 Stack<int>类型，在类模板内部 int 会被用作类型 T。被创建的 instStack 会使用一个存储 int 的 vector 作为其 elems 成员，而且所有被用到的成员函数都会被用 int 实例化。同样的，对于用 Stack<std::string>定义的对象，它会使用一个存储 std::string 的 vector 作为其 elems 成员，所有被用到的成员函数也都会用 std::string 实例化。

注意，模板函数和模板成员函数只有在被调用的时候才会实例化。这样一方面会节省时间和空间，同样也允许只是部分的使用类模板，我们会在 2.3 节对此进行讨论。

在这个例子中，对 int 和 std::string，默认构造函数，push()以及 top()函数都会被实例化。而 pop()只会针对 std::string 实例化。如果一个类模板有 static 成员，对每一个用到这个类模板的类型，相应的静态成员也只会被实例化一次。

被实例化之后的类模板类型（Stack<int>之类）可以像其它常规类型一样使用。可以用 const 以及 volatile 修饰它，或者用它来创建数组和引用。可以通过 typedef 和 using 将它用于类型定义的一部分(关于类型定义，请参见 2.8 节)，也可以用它来实例化其它的模板类型。比如：

```
void foo(Stack <int> const& s) // parameter s is int stack
{
    using IntStack = Stack <int>; // IntStack is another name for
Stack<int>
    Stack< int> istack[10]; // istack is array of 10 int stacks
    IntStack istack2[10]; // istack2 is also an array of10 int stacks
(same type)
    ...
}
```

模板参数可以是任意类型，比如指向 float 的指针，甚至是存储 int 的 stack:

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

模板参数唯一的要求是：它要支持模板中被用到的各种操作（运算符）。

在 C++11 之前，在两个相邻的模板尖括号之间必须要有空格：

```
Stack<Stack< int> > intStackStack; // OK with all C++ versions
```

如果你不这样做，>>会被解析成调用>>运算符，这会导致语法错误：

```
Stack<Stack< int>> intStackStack; // ERROR before C++11
```

这样要求的原因是，它可以帮助编译器在第一次 pass 源代码的时候，不依赖于语义就能对源代码进行正确的标记。但是由于漏掉空格是一个典型错误，而且需要相应的错误信息来进行处理，因此代码的语义被越来越多的考虑进来。从 C++11 开始，通过"angle bracket hack"技术（参考 13.3.1 节），在两个相邻的模板尖括号之间不再要求必须使用空格。

2.3 部分地使用类模板

一个类模板通常会对用来实例化它的类型进行多种操作（包含构造函数和析构函数）。这可能会让你以为，要为模板参数提供所有被模板成员函数用到的操作。但是事实不是这样：模板参数只需要提供那些会被用到的操作（而不是可能会被用到的操作）。

比如 Stack<>类可能会提供一个成员函数 printOn()来打印整个 stack 的内容，它会调用 operator <<来依次打印每一个元素：

```
template<typename T>
class Stack {
    ...
    void printOn() (std::ostream& strm) const {
        for (T const& elem : elems) {
            strm << elem << ' '; // call << for each element
        }
    }
};
```

对于一个类内的函数,如何判断它是成员函数or非成员函数:
① 函数名如果含有别的类名限定,
如, B::f(int a), 是其它类的成员函数, 这是显而易见的情况.
比较有隐藏性的是, 单个函数名的情况, 如这里的 printOn()
② 形参中有本类型对象的形参, 则为非成员函数,
因为需要引入对象才能打印对象的数据.
如 2.4节的 operator<< (std::ostream& strm, Stack<T> const& s)
{ s.printOn(strm)}
③ 形参中没有本类型对象, 在函数体中可以直接使用对象的数据
如, 这里的 printOn函数, 形参中没有Stack<>对象,
对象的数据elems可以直接拿来用

视角① T == std::pair<int, int>
视角② T1 == int, T2 == int
视角 ③ T1 == std::pair<>, T2 ==int, T3 == int

这个类依然可以用于那些没有提供 operator <<运算符的元素：

```
Stack<std::pair< int, int>> ps; // note: std::pair<> has no operator<<
defined
ps.push({4, 5}); // OK
ps.push({6, 7}); // OK
std::cout << ps.top().first << '\n'; // OK
std::cout << ps.top().second << '\n'; // OK
```

只有在调用 printOn()的时候，才会导致错误，因为它无法为这一类型实例化出对 operator<<的调用：

```
ps.printOn(std::cout); // ERROR: operator<< not supported for element
type
```

2.3.1 Concept（最好不要汉化这一概念）

这样就有一个问题：我们如何才能知道为了实例化一个模板需要哪些操作？名词 concept 通常被用来表示一组反复被模板库要求的限制条件。例如 C++标准库是基于这样一些 concepts 的：可随机进入的迭代器（random access iterator）和可默认构造的（default constructible）。

目前（比如在 C++17 中），concepts 还只是或多或少的出现在文档当中（比如代码注释）。这会导致严重的问题，因为不遵守这些限制会导致让人难以理解的错误信息（参考 9.4 节）。

近年来有一些方法和尝试，试图在语言特性层面支持对 concepts 的定义和检查。但是直到 C++17，还没有哪一种方法得以被标准化。

从 C++11 开始，你至少可以通过关键字 static_assert 和其它一些预定义的类型萃取（type traits）来做一些简单的检查。比如：

```
template<typename T>
class C
{
    static_assert(std::is_default_constructible<T>::value,
    "Class C requires default-constructible elements");
    …
};
```

即使没有这个 static_assert，如果需要 T 的默认构造函数的话，依然会遇到编译错误。只不过这个错误信息可能会包含整个模板实例化过程中所有的历史信息，从实例化被触发的地方直到模板定义中引发错误的地方（参见 9.4 节）。

然而还有更复杂的情况需要检查，比如模板类型 T 的实例需要提供一个特殊的成员函数，或者需要能够通过 operator <进行比较。这一类情况的详细例子请参见 19.6.3 节。

关于 C++ concept 的详细讨论，请参见附录 E。

2.4 友元

相比于通过 printOn()来打印 stack 的内容，更好的办法是去重载 stack 的 operator <<运算符。而且和非模板类的情况一样，operator<<应该被实现为非成员函数，在其实现中可以调用 printOn()：

```
template<typename T>
class Stack {
    …        成员函数, 形参中无需引入对象, 对象的数据可以直接访问.
    void printOn() (std::ostream& strm) const {
        …
    }
```

```
    friend std::ostream& operator<< (std::ostream& strm, Stack<T>
const& s) {                    非成员函数, 所以形参中需要引入对象 才能 访问对象的数据
        s.printOn(strm);
        return strm;
    }
};
```

注意在这里 Stack<>的 operator<<并不是一个函数模板（对于在模板类内定义这一情况），而是在需要的时候，随类模板实例化出来的一个常规函数。

然而如果你试着先声明一个友元函数，然后再去定义它，情况会变的很复杂。事实上我们有两种选择：

1. 可以隐式的声明一个新的函数模板，但是必须使用一个不同于类模板的模板参数，比如用 U：

```
template<typename T>
class Stack {
    …
    template<typename U>
    friend std::ostream& operator<< (std::ostream&, Stack<U> const&);
};
```

无论是继续使用 T 还是省略掉模板参数声明，都不可以（要么是里面的 T 隐藏了外面的 T，要么是在命名空间作用域内声明了一个非模板函数）。

2. 也可以先将 Stack<T>的 operator<<声明为一个模板，这要求先对 Stack<T>进行声明：

```
template<typename T>
class Stack;
template<typename T>
std::ostream& operator<< (std::ostream&, Stack<T> const&);
```

接着就可以将这一模板声明为 Stack<T>的友元：

```
template<typename T>
class Stack {
    …
    friend std::ostream& operator<< <T> (std::ostream&, Stack<T>
const&);
}
```

注意这里在 operator<<后面用了<T>，这相当于声明了一个特例化之后的非成员函数模板作为友元。如果没有<T>的话，则相当于定义了一个新的非模板函数。具体细节参见 12.5.2 节。

无论如何，你依然可以将 Stack<T>用于没有定义 operator <<的元素，只是当你调用 operator<<的时候会遇到一个错误：

```
Stack<std::pair< int, int>> ps; // std::pair<> has no operator<< defined
ps.push({4, 5}); // OK
ps.push({6, 7}); // OK
std::cout << ps.top().first << ' \n' ; // OK
std::cout << ps.top().second << ' \n' ; // OK
std::cout <<ps << '\n'; // ERROR: operator<< not supported // for element
type
```

2.5 模板类的特例化

可以对类模板的某一个模板参数进行特化。和函数模板的重载（参见 1.5 节）类似，类模板的特化允许我们对某一特定类型做优化，或者去修正类模板针对某一特定类型实例化之后的行为。不过如果对类模板进行了特化，那么也需要去特化所有的成员函数。虽然允许只特例化模板类的一个成员函数，不过一旦你这样做，你就无法再去特化那些未被特化的部分了。

为了特化一个类模板，在类模板声明的前面需要有一个 template<>，并且需要指明所希望特化的类型。这些用于特化类模板的类型被用作模板参数，并且需要紧跟在类名的后面：

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

对于被特化的模板，所有成员函数的定义都应该被定义成“常规”成员函数，也就是说所有出现 T 的地方，都应该被替换成用于特化类模板的类型：

```
void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
```

下面是一个用 std::string 实例化 Stack<>类模板的完整例子：

```
#include "stack1.hpp"
#include <deque>
#include <string>
#include <cassert>

template<>
class Stack<std::string> {
    private:
        std::deque<std::string> elems; // elements
    public:
        void push(std::string const&); // push element
        void pop(); // pop element
        std::string const& top() const; // return top element
```

```
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
};
void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
void Stack<std::string>::pop ()
{
    assert(!elems.empty());
    elems.pop_back(); // remove last element
}
std::string const& Stack<std::string>::top () const
{
    assert(!elems.empty());
    return elems.back(); // return copy of last element
}
```

在这个例子中，特例化之后的类在向 push()传递参数的时候使用了引用语义，对当前 std::string 类型这是有意义的，这可以提高性能（如果使用 forwarding reference【Effective Modern C++ 解释了和万能引用(Universal Reference 的异同)】传递参数的话会更好一些，6.1 节会介绍这一内容）。

另一个不同是使用了一个 deque 而不再是 vector 来存储 stack 里面的元素。虽然这样做可能不会有什么好处，不过这能够说明，模板类特例化之后的实现可能和模板类的原始实现有很大不同。

2.6 部分特例化

类模板可以只被部分的特例化。这样就可以为某些特殊情况提供特殊的实现，不过使用者还是要定义一部分模板参数。比如，可以特殊化一个 Stack<>来专门处理指针：

```
#include "stack1.hpp"
// partial specialization of class Stack<> for pointers:
template<typename T>
class Stack<T*> {
    Private:
        std::vector<T*> elems; // elements
    public:
        void push(T*); // push element
        T* pop(); // pop element
        T* top() const; // return top element
        bool empty() const { // return whether the stack is empty
```

```
            return elems.empty();
        }
};
template<typename T>
void Stack<T*>::push (T* elem)
{
    elems.push_back(elem); // append copy of passed elem
}
template<typename T>
T* Stack<T*>::pop ()
{
    assert(!elems.empty());
    T* p = elems.back();
    elems.pop_back(); // remove last element
    return p; // and return it (unlike in the general case)
}
template<typename T>
T* Stack<T*>::top () const
{
    assert(!elems.empty());
    return elems.back(); // return copy of last element
}
```

通过

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

定义了一个依然是被类型 T 参数化，但是被特化用来处理指针的类模板（Stack<T*>）。

同样的，特例化之后的函数接口可能不同。比如对 pop()，他在这里返回的是一个指针，因此如果这个指针是通过 new 创建的话，可以对这个被删除的值调用 delete：

```
Stack< int*> ptrStack; // stack of pointers (specialimplementation)
ptrStack.push(new int{42});
std::cout << *ptrStack.top() << ' \n' ;
delete ptrStack.pop();
```

多模板参数的部分特例化

类模板也可以特例化多个模板参数之间的关系。比如对下面这个类模板：

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

进行如下这些特例化都是可以的：

```
// partial specialization: both template parameters have same type
template<typename T>
class MyClass<T,T> {
    …
};
// partial specialization: second type is int
template<typename T>
class MyClass<T,int> {
    …
};
// partial specialization: both template parameters are pointer types
template<typename T1, typename T2>
class MyClass<T1*,T2*> {
    …
};
```

下面的例子展示了以上各种类模板被使用的情况：

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

如果有不止一个特例化的版本可以以相同的情形匹配某一个调用，说明定义是有歧义的：

```
MyClass< int, int> m; // ERROR: matches MyClass<T,T> // and
MyClass<T,int>
MyClass< int*, int*> m; // ERROR: matches MyClass<T,T> // and
  MyClass<T1*,T2*>
```

为了消除第二种歧义，你可以提供一个单独的特例化版本来处理相同类型的指针：

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

更多关于部分特例化的信息，请参见 16.4 节。

2.7 默认类模板参数

和函数模板一样，也可以给类模板的模板参数指定默认值。比如对 Stack<>，你可以将其用来容纳元素的容器声明为第二个模板参数，并指定其默认值是 std::vector<>:

```
#include <vector>
```

```
#include <cassert>
template<typename T, typename Cont = std::vector<T>>
class Stack {
    private:
        Cont elems; // elements
    public:
        void push(T const& elem); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is
            emptyreturn elems.empty();
        }
};
template<typename T, typename Cont>
void Stack<T,Cont>::push (T const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
template<typename T, typename Cont>
void Stack<T,Cont>::pop ()
{
    assert(!elems.empty());
    elems.pop_back(); // remove last element
}
template<typename T, typename Cont>
T const& Stack<T,Cont>::top () const
{
    assert(!elems.empty());
    return elems.back(); // return copy of last element
}
```

由于现在有两个模板参数，因此每个成员函数的定义也应该包含两个模板参数：

```
template<typename T, typename Cont>
void Stack<T,Cont>::push (T const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
```

这个 Stack<>模板可以像之前一样使用。如果只提供第一个模板参数作为元素类型，那么 vector 将被用来处理 Stack 中的元素：

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

```
        ...
};
```

而且在程序中，也可以为 Stack 指定一个容器类型：

```
#include "stack3.hpp"
#include <iostream>
#include <deque>
int main()
{
    // stack of ints:
    Stack< int> intStack;
    // stack of doubles using a std::deque<> to manage the elements
    Stack< double,std::deque< double>> dblStack;
    // manipulate int stack
    intStack.push(7);
    std::cout << intStack.top() << ' \n' ;
    intStack.pop();
    // manipulate double stack
    dblStack.push(42.42);
    std::cout << dblStack.top() << ' \n' ;
    dblStack.pop();
}
```

通过

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

定义了一个处理 double 型元素的 Stack，其使用的容器是 std::deque<>。

2.8 类型别名（Type Aliases）

通过给类模板定义一个新的名字，可以使类模板的使用变得更方便。

Typedefs 和 Alias 声明

为了简化给类模板定义新名字的过程，有两种方法可用：

1. 使用关键字 typedef:

```
typedef Stack<int> IntStack; // typedef
void foo (IntStack const& s); // s is stack of ints
IntStack istack[10]; // istack is array of 10 stacks of ints
```

我们称这种声明方式为 typedef， 被定义的名字叫做 typedef-name.

2. 使用关键字 using （从 C++11 开始）

```
using IntStack = Stack <int>; // alias declaration
void foo (IntStack const& s); // s is stack of ints
IntStack istack[10]; // istack is array of 10 stacks of ints
```

按照[DosReisMarcusAliasTemplates] 的说法，这一过程叫做 alias declaration。在这两种情况下我们都只是为一个已经存在的类型定义了一个别名，并没有定义新的类型。因此在：

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

或者：

```
using IntStack = Stack <int>;
```

之后，IntStack 和 Stack<int>将是两个等效的符号。

以上两种给一个已经存在的类型定义新名字的方式，被称为 type alias declaration。新的名字被称为 type alias。

由于使用 alias declaration（使用 using 的情况，新的名字总是在=的左边）可读性更好，在本书中接下来的内容中，我们将优先使用这一方法。

Alias Templates（别名模板）

using比typedef更强大

不同于 typedef， alias declaration 也可以被模板化，这样就可以给一组类型取一个方便的名字。这一特性从 C++11 开始生效，被称作 alias templates。

下面的 DequeStack 别名模板是被元素类型 T 参数化的，代表将其元素存储在 std::deque 中的一组 Stack：

模板别名, 一组: 模板 DequeStack<T> == 模板 Stack<T, std::deque<T>>
类型别名, 一个: using IntStack = Stack<int>; // typedef Stack<int> IntStack;

```
template<typename T>
using DequeStack = Stack<T, std::deque<T>>;
```

因此，类模板和 alias templates 都是可以被参数化的类型。同样地，这里 alias template 只是一个已经存在的类型的新名字，原来的名字依然可用。DequeStack<int> 和 Stack<int, std::deque<int>>代表的是同一种类型。

同样的，通常模板（包含 Alias Templates）只可以被声明和定义在 global/namespace 作用域，或者在一个类的声明中。

Alias Templates for Member Types（class 成员的别名模板）

别名模板 在 定义类模板成员的类型简称 时 十分有用

使用 alias templates 可以很方便的给类模板的成员类型定义一个快捷方式，在：

```
struct C {
    typedef … iterator;
```

```
    …
};
```

或者

```
struct MyType {
    using iterator = …;
    …
};
```

之后，下面这样的定义：

```
template<typename T>
using MyTypeIterator = typename MyType<T>::iterator;
```

允许我们使用：

```
MyTypeIterator< int> pos;
```

取代：

```
typename MyType<T>::iterator pos;
```

Type Traits Suffix_t （Suffix_t 类型萃取）

从 C++14 开始，标准库使用上面的技术，给标准库中所有返回一个类型的 type trait 定义了快捷方式。比如为了能够使用：

```
std::add_const_t<T> // since C++14
```

而不是：

```
typename std::add_const<T>::type // since C++11
```

标准库做了如下定义：

```
namespace std {
    template<typename T>
    using add_const_t = typename add_const<T>::type;
}
```

2.9 类模板的类型推导

不是不再次要求，而是如果constructor能够deduce的话，才不需要。

直到 C++17，使用类模板时都必须显式指出所有的模板参数的类型（除非它们有默认值）。从 C++17 开始，这一要求不在那么严格了。如果构造函数能够推断出所有模板参数的类型（对那些没有默认值的模板参数），就不再需要显式的指明模板参数的类型。

比如在之前所有的例子中，不指定模板类型就可以调用 copy constructor:

```
Stack< int> intStack1; // stack of strings
Stack< int> intStack2 = intStack1; // OK in all versions
Stack intStack3 = intStack1; // OK since C++17
```

通过提供一个接受初始化参数的构造函数，就可以推断出 Stack 的元素类型。比如可以定义下面这样一个 Stack，它可以被一个元素初始化：

```
template<typename T>
class Stack {
    private:
        std::vector<T> elems; // elements
    public:
        Stack () = default;
        Stack (T const& elem) // initialize stack with one element
        : elems({elem}) {
        }
        ...
};
```

然后就可以像这样声明一个 Stack:

```
Stack intStack = 0; // Stack<int> deduced since C++17
```

通过用 0 初始化这个 stack 时，模板参数 T 被推断为 int，这样就会实例化出一个 Stack<int>。

但是请注意下面这些细节：

- 由于定义了接受 int 作为参数的构造函数，要记得向编译器要求生成默认构造函数及其全部默认行为，这是因为默认构造函数只有在没有定义其它构造函数的情况下才会默认生成，方法如下：

  ```
  Stack() = default;
  ```

- 在初始化 Stack 的 vector 成员 elems 时，参数 elem 被用{}括了起来，这相当于用只有一个元素 elem 的初始化列表初始化了 elems:

  ```
  : elems({elem})
  ```

  这是因为 vector 没有可以直接接受一个参数的构造函数。

和函数模板不同，类模板可能无法部分的推断模板类型参数（比如在显式的指定了一部分类模板参数的情况下）。具体细节请参见 15.12 节。

类模板对字符串常量参数的类型推断（Class Template Arguments Deduction with String Literals ）

原则上，可以通过字符串常量来初始化 Stack：

```
Stack stringStack = "bottom"; // Stack<char const[7]> deduced since
C++17
```

不过这样会带来一堆问题：当参数是按照 T 的引用传递的时候（上面例子中接受一个参数的构造函数，是按照引用传递的），参数类型不会被 decay，也就是说一个裸的数组类型不会被转换成裸指针。这样我们就等于初始化了一个这样的 Stack:

```
Stack< char const[7]>
```

类模板中的 T 都会被实例化成 char const[7]。这样就不能继续向 Stack 追加一个不同维度的字符串常量了，因为它的类型不是 char const[7]。详细的讨论请参见 7.4 节。

不过如果参数是按值传递的，参数类型就会被 decay，也就是说会将裸数组退化成裸指针。这样构造函数的参数类型 T 会被推断为 char const *，实例化后的类模板类型会被推断为 Stack<char const *>。

基于以上原因，可能有必要将构造函数声明成按值传递参数的形式：

```
template<typename T>
class Stack {
    private:
        std::vector<T> elems; // elements
    public:
        Stack (T elem) // initialize stack with one element by value
        : elems({elem}) { // to decay on class tmpl arg deduction
        }
    ...
};
```

这样下面的初始化方式就可以正常工作：

```
Stack stringStack = "bottom"; // Stack<char const*> deduced since C++17
```

在这个例子中，最好将临时变量 elem move 到 stack 中，这样可以免除不必要的拷贝:

```
template<typename T>
class Stack {
    private:
        std::vector<T> elems; // elements
    public:
        Stack (T elem) // initialize stack with one element by value
        : elems({std::move(elem)}) {
        }
  ...
};
```

推断指引（Deduction Guides）

针对以上问题，除了将构造函数声明成按值传递的，还有一个解决方案：由于在容器中处理裸指针容易导致很多问题，对于容器一类的类，不应该将类型推断为字符的裸指针（char const *）。

可以通过提供“推断指引”来提供额外的模板参数推断规则，或者修正已有的模板参数推断规则。比如你可以定义，当传递一个字符串常量或者 C 类型的字符串时，应该用 std::string 实例化 Stack 模板类：

```
Stack( char const*) -> Stack<std::string>;
```

这个指引语句必须出现在和模板类的定义相同的作用域或者命名空间内。通常它紧跟着模板类的定义。->后面的类型被称为推断指引的”guided type”。

现在，根据这个定义：

```
Stack stringStack{"bottom"}; // OK: Stack<std::string> deduced since
C++17
```

Stack 将被推断为 Stack<std::string>。但是下面这个定义依然不可以：

```
Stack stringStack = "bottom"; // Stack<std::string> deduced, but still
not valid
```

此时模板参数类型被推断为 std::string，也会实例化出 Stack<std::string>:

```
class Stack {
    private:
        std::vector<std::string> elems; // elements
    public:
        Stack (std::string const& elem) // initialize stack with one
element
            : elems({elem}) {
        }
    ...
};
```

但是根据语言规则，不能通过将字符串字面量传递给一个期望接受 std::string 的构造函数来拷贝初始化（使用=初始化）一个对象，因此必须要像下面这样来初始化这个 Stack:

```
Stack stringStack{"bottom"}; // Stack<std::string> deduced and valid
```

如果还不是很确信的话，这里可以明确告诉你，模板参数推断的结果是可以拷贝的。在将 stringStack 声明为 Stack<std::string>之后，下面的初始化语句声明的也将是 Stack<std::string>类型的变量（通过拷贝构造函数），而不是用 Stack<std::string>类型的元素去初始化一个 stack（也就是说，Stack 存储的元素类型是 std::string，而不是 Stack<std::string>）：

```
Stack stack2{stringStack}; // Stack<std::string> deduced
Stack stack3(stringStack); // Stack<std::string> deduced
```

```
Stack stack4 = {stringStack}; // Stack<std::string> deduced
```

更多关于类模板的参数类型推导的内容，请参见 15.12 节。

2.10 聚合类的模板化（Templatized Aggregates）

聚合类（这样一类 class 或者 struct：没有用户定义的显式的，或者继承而来的构造函数，没有 private 或者 protected 的非静态成员，没有虚函数，没有 virtual，private 或者 protected 的基类）也可以是模板。比如：

```
template<typename T>
struct ValueWithComment {
    T value;
    std::string comment;
};
```

定义了一个成员 val 的类型被参数化了的聚合类。可以像定义其它类模板的对象一样定义一个聚合类的对象：

```
ValueWithComment< int> vc;
vc.value = 42;
vc.comment = "initial value";
```

从 C++17 开始，对于聚合类的类模板甚至可以使用“类型推断指引”：

```
ValueWithComment(
char const*, char const*) -> ValueWithComment<std::string>;
ValueWithComment vc2 = {"hello", "initial value"};
```

没有“推断指引”的话，就不能使用上述初始化方法，因为 ValueWithComment 没有相应的构造函数来完成相关类型推断。

标准库的 std::array<>类也是一个聚合类，其元素类型和尺寸都是被参数化的。C++17 也给它定义了“推断指引”，在 4.4.4 节会做进一步讨论。

2.11 总结

- 类模板是一个被实现为有一个或多个类型参数待定的类。
- 使用类模板时，需要显式或者隐式地传递相应的待定类型参数作为模板参数。之后类模板会被按照传入的模板参数实例化（并且被编译）。
- 对于类模板，只有其被用到的成员函数才会被实例化。
- 可以针对某些特定类型对类模板进行特化。
- 也可以针对某些特定类型对类模板进行部分特化。
- 从 C++17 开始，可以（不是一定可以）通过类模板的构造函数来推断模板参数的类型。

- 可以定义聚合类的类模板。
- 调用参数如果是按值传递的，那么相应的模板类型会 decay。
- 模板只能被声明以及定义在 global 或者 namespace 作用域，或者是定义在其它类的定义里面。

第 3 章 非类型模板参数

对于之前介绍的函数模板和类模板，其模板参数不一定非得是某种具体的类型，也可以是常规数值。和类模板使用类型作为参数类似，可以使代码的另一些细节留到被使用时再确定，只是对非类型模板参数，待定的不再是类型，而是某个数值。在使用这种模板时需要显式的指出待定数值的具体值，之后代码会被实例化。本章会通过一个新版的 Stack 类模板来展示这一特性。顺便也会介绍一下函数模板的非类型参数，并讨论这一技术的一些限制。

3.1 类模板的非类型参数

作为和之前章节中 Stack 实现方式的对比，可以定义一个使用固定尺寸的 array 作为容器的 Stack。这种方式的优点是可以避免由开发者或者标准库容器负责的内存管理开销。不过对不同应用，这一固定尺寸的具体大小也很难确定。如果指定的值过小，那么 Stack 就会很容易满。如果指定的值过大，则可能造成内存浪费。因此最好是让 Stack 的用户根据自身情况指定 Stack 的大小。

为此，可以将 Stack 的大小定义成模板的参数：

```
#include <array>
#include <cassert>
template<typename T, std::size_t Maxsize>
class Stack {
    private:
        std::array<T, Maxsize> elems; // elements
        std::size_t numElems; // current number of elements
    public:
        Stack(); // constructor
        void push(T const& elem); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { //return whether the stack is empty
            return numElems == 0;
        }
        std::size_t size() const { //return current number of elements
            return numElems;
        }
};
template<typename T, std::size_t Maxsize>
Stack<T,Maxsize>::Stack ()
: numElems(0) //start with no elements
{
```

```
// nothing else to do
}
template<typename T, std::size_t Maxsize>
void Stack<T,Maxsize>::push (T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem; // append element
    ++numElems; // increment number of elements
}
template<typename T, std::size_t Maxsize>
void Stack<T,Maxsize>::pop ()
{
    assert(!elems.empty());
    --numElems; // decrement number of elements
}
template<typename T, std::size_t Maxsize>
T const& Stack<T,Maxsize>::top () const
{
    assert(!elems.empty());
    return elems[numElems-1]; // return last element
}
```

第二个新的模板参数 Maxsize 是 int 类型的。通过它指定了 Stack 中 array 的大小：

```
template<typename T, std::size_t Maxsize>
class Stack {
    private:
        std::array<T,Maxsize> elems; // elements
        ...
};
```

成员函数 push()也用它来检测 Stack 是否已满：

```
template<typename T, std::size_t Maxsize>
void Stack<T,Maxsize>::push (T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem; // append element
    ++numElems; // increment number of elements
}
```

为了使用这个类模板，需要同时指出 Stack 中元素的类型和 Stack 的最大容量：

```
#include "stacknontype.hpp"
#include <iostream>
#include <string>
int main()
```

```
{
    Stack<int,20> int20Stack; // stack of up to 20 ints
    Stack<int,40> int40Stack; // stack of up to 40 ints
    Stack<std::string,40> stringStack; // stack of up to 40 strings
    // manipulate stack of up to 20 ints
    int20Stack.push(7);
    std::cout << int20Stack.top() << ' \n' ;
    int20Stack.pop();
    // manipulate stack of up to 40 strings
    stringStack.push("hello");
    std::cout << stringStack.top() << ' \n' ;
    stringStack.pop();
}
```

上面每一次模板的使用都会实例化出一个新的类型。因此 int20Stack 和 int40Stack 是两种不同的类型，而且由于它们之间没有定义隐式或者显式的类型转换规则。也就不能使用其中一个取代另一个，或者将其中一个赋值给另一个。

对非类型模板参数，也可以指定默认值：

```
template<typename T = int, std::size_t Maxsize = 100>
class Stack {
    ...
};
```

但是从程序设计的角度来看，这可能不是一个好的设计方案。默认值应该是直观上正确的。不过对于一个普通的 Stack，无论是默认的 int 类型还是 Stack 的最大尺寸 100，看上去都不够直观。因此最好是让程序员同时显式地指定两个模板参数，这样在声明的时候这两个模板参数通常都会被文档化。

3.2 函数模板的非类型参数

同样也可以给函数模板定义非类型模板参数。比如下面的这个函数模板，定义了一组可以返回传入参数和某个值之和的函数：

```
template<int Val, typename T>
T addValue (T x)
{
    return x + Val;
}
```

当该类函数或操作是被用作其它函数的参数时，可能会很有用。比如当使用 C++标准库给一个集合中的所有元素增加某个值的时候，可以将这个函数模板的一个实例化版本用作第 4 个参数：

```
std::transform (source.begin(), source.end(), //start and end of source
```

```
    dest.begin(),  //start of destination
    addValue<5,int>);  // operation
```

第 4 个参数是从 addValue<>()实例化出一个可以给传入的 int 型参数加 5 的函数实例。这一实例会被用来处理集合 source 中的所有元素，并将结果保存到目标集合 dest 中。

注意在这里必须将 addValue<>()的模板参数 T 指定为 int 类型。因为类型推断只会对立即发生的调用起作用，而 std::transform()又需要一个完整的类型来推断其第四个参数的类型。目前还不支持先部分地替换或者推断模板参数的类型，然后再基于具体情况去推断其余的模板参数。

同样也可以基于前面的模板参数推断出当前模板参数的类型。比如可以通过传入的非类型模板参数推断出返回类型：

```
template<auto Val, typename T = decltype(Val)>
T foo();
```

或者可以通过如下方式确保传入的非类型模板参数的类型和类型参数的类型一致：

```
template<typename T, T Val = T{}>
T bar();
```

3.3 非类型模板参数的限制

使用非类型模板参数是有限制的。通常它们只能是整形常量（包含枚举），指向 objects/functions/members 的指针，objects 或者 functions 的左值引用，或者是 std::nullptr_t（类型是 nullptr）。

浮点型数值或者 class 类型的对象都不能作为非类型模板参数使用：

```
template<double VAT>  // ERROR: floating-point values are not
double process (double v)  // allowed as template parameters
{
    return v * VAT;
}
template<std::string name> // ERROR: class-type objects are not
class MyClass { // allowed as template parameters
    ...
};
```

当传递对象的指针或者引用作为模板参数时，对象不能是字符串常量，临时变量或者数据成员以及其它子对象。由于在 C++17 之前，C++版本的每次更新都会放宽以上限制，因此还有一些针对不同版本的限制：

- 在 C++11 中，对象必须要有外部链接。
- 在 C++14 中，对象必须是外部链接或者内部链接。

因此下面的写法是不对的：

```
template<char const* name>
class MyClass {
    …
};
MyClass<"hello"> x; //ERROR: string literal "hello" not allowed
```

不过有如下变通方法（视 C++版本而定）：

```
extern char const s03[] = "hi"; // external linkage
char const s11[] = "hi"; // internal linkage
int main()
{
    MyClass<s03> m03; // OK (all versions)
    MyClass<s11> m11; // OK since C++11
    static char const s17[] = "hi"; // no linkage
    MyClass<s17> m17; // OK since C++17
}
```

上面三种情况下，都是用”hello”初始化了一个字符串常量数组，然后将这个字符串常量数组对象用于类模板中被声明为 char const *的模板参数。如果这个对象有外部链接（s03），那么对所有版本的 C++都是有效的，如果对象有内部链接（s11），那么对 C++11 和 C++14 也是有效的，而对 C++17，即使对象没有链接属性也是有效的。

12.3.3 节对这一问题进行了更详细的讨论，17.2 节则对这一问题未来可能的变化进行了讨论。

避免无效表达式

非类型模板参数可以是任何编译器表达式。比如：

```
template<int I, bool B>
class C;
…
C<sizeof(int) + 4, sizeof(int)==4> c;
```

不过如果在表达式中使用了 operator >，就必须将相应表达式放在括号里面，否则>会被作为模板参数列表末尾的>，从而截断了参数列表：

```
C<42, sizeof(int) > 4> c; // ERROR: first > ends the template argument
list
C<42, (sizeof(int) > 4)> c; // OK
```

3.4 用 auto 作为非模板类型参数的类型

从 C++17 开始，可以不指定非类型模板参数的具体类型（代之以 auto），从而使其可以用于任意有效的非类型模板参数的类型。通过这一特性，可以定义如下更为泛化的大小固定的 Stack 类：

```
#include <array>
#include <cassert>
template<typename T, auto Maxsize>
class Stack {
    public:
        using size_type = decltype(Maxsize);
    private:
        std::array<T,Maxsize> elems; // elements
        size_type numElems; // current number of elements
    public:
        Stack(); // constructor
        void push(T const& elem); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { //return whether the stack isempty
            return numElems == 0;
        }
        size_type size() const { //return current number of elements
            return numElems;
        }
};
// constructor
template<typename T, auto Maxsize>
Stack<T,Maxsize>::Stack ()
: numElems(0) //start with no elements
{
// nothing else to do
}
 template<typename T, auto Maxsize>
void Stack<T,Maxsize>::push (T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem; // append element
    ++numElems; // increment number of elements
}
 template<typename T, auto Maxsize>
void Stack<T,Maxsize>::pop ()
{
    assert(!elems.empty());
```

```
    --numElems; // decrement number of elements
}
template<typename T, auto Maxsize>
T const& Stack<T,Maxsize>::top () const
{
    assert(!elems.empty());
    return elems[numElems-1]; // return last element
}
```

通过使用 auto 的如下定义：

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

定义了类型待定的 Maxsize。它的类型可以是任意非类型参数所允许的类型。

在模板内部，既可以使用它的值：

```
std::array<T,Maxsize> elems; // elements
```

也可以使用它的类型：

```
using size_type = decltype(Maxsize);
```

然后可以将它用于成员函数 size()的返回类型：

```
size_type size() const { //return current number of elements
    return numElems;
}
```

从 C++14 开始，也可以通过使用 auto，让编译器推断出具体的返回类型：

```
auto size() const { //return current number of elements
    return numElems;
}
```

根据这个类的声明，Stack 中 numElems 成员的类型是由非类型模板参数的类型决定的，当像下面这样使用它的时候：

```
#include <iostream>
#include <string>
#include "stackauto.hpp"
int main()
{
    Stack<int,20u> int20Stack; // stack of up to 20 ints
    Stack<std::string,40> stringStack; // stack of up to 40 strings
    // manipulate stack of up to 20 ints
    int20Stack.push(7);
```

```
    std::cout << int20Stack.top() << ' \n' ;auto size1 =
int20Stack.size();
    // manipulate stack of up to 40 strings
    stringStack.push("hello");
    std::cout << stringStack.top() << ' \n' ;
    auto size2 = stringStack.size();
    if (!std::is_same_v<decltype(size1), decltype(size2)>) {
        std::cout << "size types differ" << ' \n' ;
    }
}
```

对于

```
Stack<int,20u> int20Stack; // stack of up to 20 ints
```

由于传递的非类型参数是 20u，因此内部的 size_type 是 unsigned int 类型的。

对于

```
Stack<std::string,40> stringStack; // stack of up to 40 strings
```

由于传递的非类型参数是 int，因此内部的 size_type 是 int 类型的。

因为这两个 Stack 中成员函数 size()的返回类型是不一样的，所以

```
auto size1 = int20Stack.size();
…
auto size2 = stringStack.size();
```

中 size1 和 size2 的类型也不一样。这可以通过标准类型萃取 std::is_same（详见 D3.3）和 decltype 来验证：

```
if (!std::is_same<decltype(size1), decltype(size2)>::value) {
    std::cout << "size types differ" << ' \n' ;
}
```

输出结果将是：

```
size types differ
```

从 C++17 开始，对于返回类型的类型萃取，可以通过使用下标_v 省略掉::value（参见 5.6 节）：

```
if (!std::is_same_v<decltype(size1), decltype(size2)>) {
    std::cout << "size types differ" << ' \n' ;
}
```

注意关于非类型模板参数的限制依然存在。尤其是那些在 3.3 节讨论的限制。比如：

```
Stack<int,3.14> sd; // ERROR: Floating-point nontype argument
```

由于可以将字符串作为常量数组用于非类型模板参数（从 C++17 开始甚至可以是静态的局部变量，参见 3.3 节），下面的用法也是可以的：

```
#include <iostream>
template<auto T> // take value of any possible nontype
parameter (since C++17)
class Message {
    public:
        void print() {
            std::cout << T << ' \n' ;
        }
};
int main()
{
    Message<42> msg1;
    msg1.print(); // initialize with int 42 and print that value
    static char const s[] = "hello";
    Message<s> msg2; // initialize with char const[6] "hello"
    msg2.print(); // and print that value
}
```

也可以使用 template<decltype(auto)>，这样可以将 N 实例化成引用类型：

```
template<decltype(auto) N>
class C {
    ...
};
int i;
C<(i)> x; // N is int&
```

更多细节请参见 15.10.1 节。

3.4 总结

- 模板的参数不只可以是类型，也可以是数值。
- 不可以将浮点型或者 class 类型的对象用于非类型模板参数。使用指向字符串常量，临时变量和子对象的指针或引用也有一些限制。
- 通过使用关键字 auto，可以使非类型模板参数的类型更为泛化。

第 4 章 变参模板

从 C++11 开始，模板可以接受一组数量可变的参数。这样就可以在参数数量和参数类型都不确定的情况下使用模板。一个典型应用是通过 class 或者 framework 向模板传递一组数量和类型都不确定的参数。另一个应用是提供泛型代码处理一组数量任意且类型也任意的参数。

4.1 变参模板

可以将模板参数定义成能够接受任意多个模板参数的情况。这一类模板被称为变参模板（variadic template）。

4.1.1 变参模板实列

比如，可以通过调用下面代码中的 print()函数来打印一组数量和类型都不确定的参数：

```
#include <iostream>
void print ()
{}
template<typename T, typename… Types>
void print (T firstArg, Types… args)
{
    std::cout << firstArg << ' \n' ; //print first argument
    print(args…); // call print() for remaining arguments
}
```

如果传入的参数是一个或者多个，就会调用这个函数模板，这里通过将第一个参数单独声明，就可以先打印第一个参数，然后再递归的调用 print()来打印剩余的参数。这些被称为 args 的剩余参数，是一个函数参数包（function parameter pack）：

```
void print (T firstArg, Types… args)
```

这里使用了通过模板参数包（template parameter pack）定义的类型“Types”：

```
template<typename T, typename… Types>
```

为了结束递归，重载了不接受参数的非模板函数 print()，它会在参数包为空的时候被调用。

比如，这样一个调用：

```
std::string s("world");
print (7.5, "hello", s);
```

会输出如下结果：

7.5
hello
World

因为这个调用首先会被扩展成：

```
print<double, char const*, std::string> (7.5, "hello", s);
```

其中：

- firstArg 的值是 7.5， 其类型 T 是 double。
- args 是一个可变模板参数，它包含类型是 char const*的“hello”和类型是 std::string 的“world”

在打印了 firstArg 对应的 7.5 之后，继续调用 print()打印剩余的参数，这时 print()被扩展为：

```
print<char const*, std::string> ("hello", s);
```

其中：

- firstArg 的值是“hello”，其类型 T 是 char const *。
- args 是一个可变模板参数，它包含的参数类型是 std::string。

在打印了 firstArg 对应的“hello”之后，继续调用 print()打印剩余的参数，这时 print()被扩展为：

```
print<std::string> (s);
```

其中：

- firstArg 的值是“world”，其类型 T 是 std::string。
- args 是一个空的可变模板参数，它没有任何值。

这样在打印了 firstArg 对应的“world”之后，就会调用被重载的不接受参数的非模板函数 print()，从而结束了递归。

4.1.2 变参和非变参模板的重载

上面的例子也可以这样实现：

```
#include <iostream>
template<typename T>
void print (T arg)
{
    std::cout << arg << ' \n' ; //print passed argument
}
template<typename T, typename… Types>
void print (T firstArg, Types… args)
{
```

```
    print(firstArg); // call print() for the first argument
    print(args…); // call print() for remainingarguments
}
```

也就是说，当两个函数模板的区别只在于尾部的参数包的时候，会优先选择没有尾部参数包的那一个函数模板。相关的、更详细的重载解析规则请参见 C3.1 节。

4.1.3 sizeof... 运算符

C++11 为变参模板引入了一种新的 sizeof 运算符：sizeof...。它会被扩展成参数包中所包含的参数数目。因此：

```
template<typename T, typename… Types>
void print (T firstArg, Types… args)
{
    std::cout << firstArg << ' \n' ; //print first argument
    std::cout << sizeof…(Types) << ' \n' ; //print number of remaining
types
    std::cout << sizeof…(args) << ' \n' ; //print number of remaining
args
    …
}
```

在将第一个参数打印之后，会将参数包中剩余的参数数目打印两次。如你所见，运算符 sizeof.. 既可以用于模板参数包，也可以用于函数参数包。

这样可能会让你觉得，可以不使用为了结束递归而重载的不接受参数的非模板函数 print()，只要在没有参数的时候不去调用任何函数就可以了：

```
template<typename T, typename… Types>
void print (T firstArg, Types… args)
{
    std::cout << firstArg << ' \n' ;
    if (sizeof…(args) > 0) { //error if sizeof…(args)==0
        print(args…); // and no print() for no arguments declared
    }
}
```

但是这一方式是错误的，因为通常函数模板中 if 语句的两个分支都会被实例化。是否使用被实例化出来的代码是在运行期间（run-time）决定的，而是否实例化代码是在编译期间（compile-time）决定的。因此如果在只有一个参数的时候调用 print()函数模板，虽然 args... 为空，if 语句中的 print(args...)也依然会被实例化，但此时没有定义不接受参数的 print()函数，因此会报错。

不过从 C++17 开始，可以使用编译阶段的 if 语句，这样通过一些稍微不同的语法，就可以

实现前面想要的功能。8.5 节会对这一部分内容进行讨论。

4.2 折叠表达式

从 C++17 开始，提供了一种可以用来计算参数包（可以有初始值）中所有参数运算结果的二元运算符。

比如，下面的函数会返回 s 中所有参数的和：

```
template<typename… T>
auto foldSum (T… s) {
    return (… + s);  // ((s1 + s2) + s3) …
}
```

如果参数包是空的，这个表达式将是不合规范的（不过此时对于运算符&&，结果会是 true，对运算符||，结果会是 false，对于逗号运算符，结果会是 void()）。

表 4.1 列举了可能的折叠表达式：

Fold Expression	Evaluation
(… op pack) (pack op …) (init op … op pack) (pack op … op init)	(((pack1 op pack2) op pack3) … op packN) (pack1 op (… (packN-1 op packN))) (((init op pack1) op pack2) … op packN) (pack1 op (… (packN op init)))

表 4.1 折叠表达式（从 C++17 开始）

几乎所有的二元运算符都可以用于折叠表达式（详情请参见 12.4.6 节）。比如可以使用折叠表达式和运算符->*遍历一条二叉树的路径：

```
// define binary tree structure and traverse helpers:
struct Node {
    int value;
    Node* left;
    Node* right;
    Node(int i=0) : value(i), left(nullptr), right(nullptr) {
    }
    …
};
auto left = &Node::left;
auto right = &Node::right;
// traverse tree, using fold expression:
template<typename T, typename… TP>
Node* traverse (T np, TP… paths) {
    return (np ->* … ->* paths); // np ->* paths1 ->* paths2 …
}
int main()
```

```
{
    // init binary tree structure:
    Node* root = new Node{0};
    root->left = new Node{1};
    root->left->right = new Node{2};
    …
    // traverse binary tree:
    Node* node = traverse(root, left, right);
    …
}
```

这里

```
(np ->* … ->* paths)
```

使用了折叠表达式从 np 开始遍历了 paths 中所有可变成员。

通过这样一个使用了初始化器的折叠表达式，似乎可以简化打印变参模板参数的过程，像上面那样：

```
template<typename… Types>
void print (Types const&… args)
{
    (std::cout << … << args) << ' \n' ;
}
```

不过这样在参数包各元素之间并不会打印空格。为了打印空格，还需要下面这样一个类模板，它可以在所有要打印的参数后面追加一个空格：

```
template<typename T>
class AddSpace
{
    private:
        T const& ref; // refer to argument passed in constructor
    public:
        AddSpace(T const& r): ref(r) {
        }
        friend std::ostream& operator<< (std::ostream& os, AddSpace<T>
s) {
            return os << s.ref <<' ' ; // output passed argument and a space
        }
};
template<typename… Args>
void print (Args… args) {
    ( std::cout << … << AddSpace<Args>(args) ) << ' \n' ;
}
```

注意在表达式 AddSpace(args)中使用了类模板的参数推导（见 2.9 节），相当于使用了 AddSpace<Args>(args)，它会给传进来的每一个参数创建一个引用了该参数的 AddSpace 对象，当将这个对象用于输出的时候，会在其后面加一个空格。

更多关于折叠表达式的内容请参见 12.4.6 节。

4.3 变参模板的使用

变参模板在泛型库的开发中有重要的作用，比如 C++标准库。

一个重要的作用是转发任意类型和数量的参数。比如在如下情况下会使用这一特性：

- 向一个由智能指针管理的，在堆中创建的对象的构造函数传递参数：

```
// create shared pointer to complex<float> initialized by 4.2 and 7.7:
auto sp = std::make_shared<std::complex<float>>(4.2, 7.7);
```

- 向一个由库启动的 thread 传递参数：

```
std::thread t (foo, 42, "hello"); //call foo(42,"hello") in a separate thread
```

- 向一个被 push 进 vector 中的对象的构造函数传递参数：

```
std::vector<Customer> v;
…
v.emplace("Tim", "Jovi", 1962); //insert a Customer initialized by three arguments
```

通常是使用移动语义对参数进行完美转发（perfectly forwarded）（参见 6.1 节），它们像下面这样进行声明：

```
namespace std {
    template<typename T, typename… Args> shared_ptr<T>
    make_shared(Args&&… args);

    class thread {
        public:
            template<typename F, typename… Args>
            explicit thread(F&& f, Args&&… args);
        …
    };

    template<typename T, typename Allocator = allocator<T>>
    class vector {
        public:
            template<typename… Args>
            reference emplace_back(Args&&… args);
```

```
            …
    };
}
```

注意，之前关于常规模板参数的规则同样适用于变参模板参数。比如，如果参数是按值传递的，那么其参数会被拷贝，类型也会退化（decay）。如果是按引用传递的，那么参数会是实参的引用，并且类型不会退化：

```
// args are copies with decayed types:
template<typename… Args> void foo (Args… args);
// args are nondecayed references to passed objects:
template<typename… Args> void bar (Args const&… args);
```

4.4 变参类模板和变参表达式

除了上面提到的例子，参数包还可以出现在其它一些地方，比如表达式，类模板，using 声明，甚至是推断指引中。完整的列表请参见 12.4.2 节。

4.4.1 变参表达式

除了转发所有参数之外，还可以做些别的事情。比如计算它们的值。

下面的例子先是将参数包中的所有的参数都翻倍，然后将结果传给 print()：

```
template<typename… T>
void printDoubled (T const&… args)
{
    print (args + args…);
}
```

如果这样调用它：

```
printDoubled(7.5, std::string("hello"), std::complex<float>(4,2));
```

效果上和下面的调用相同（除了构造函数方面的不同）：

```
print(7.5 + 7.5, std::string("hello") + std::string("hello"),
std::complex<float>(4,2) + std::complex<float>(4,2);
```

如果只是想向每个参数加 1，省略号…中的点不能紧跟在数值后面：

```
template<typename… T>
void addOne (T const&… args)
{
    print (args + 1…); // ERROR: 1… is a literal with too many decimal
points
```

```
    print (args + 1 …); // OK
    print ((args + 1)…); // OK
}
```

编译阶段的表达式同样可以像上面那样包含模板参数包。比如下面这个例子可以用来判断所有参数包中参数的类型是否相同：

```
template<typename T1, typename… TN>
constexpr bool isHomogeneous (T1, TN…)
{
    return (std::is_same<T1,TN>::value && …); // since C++17
}
```

这是折叠表达式的一种应用（参见 4.2 节）。对于：

```
isHomogeneous(43, -1, "hello")
```

会被扩展成：

```
std::is_same<int,int>::value && std::is_same<int,char const*>::value
```

结果自然是 false。而对：

```
isHomogeneous("hello", "", "world", "!")
```

结果则是 true，因为所有的参数类型都被推断为 char const *（这里因为是按值传递，所以发生了类型退还，否则类型将依次被推断为：char const[6], char const[1], char const[6]和 char const[2]）。

4.4.2 变参下标（Variadic Indices）

作为另外一个例子，下面的函数通过一组变参下标来访问第一个参数中相应的元素：

```
template<typename C, typename… Idx>
void printElems (C const& coll, Idx… idx)
{
    print (coll[idx]…);
}
```

当调用：

```
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printElems(coll,2,0,3);
```

时，相当于调用了：

```
print (coll[2], coll[0], coll[3]);
```

也可以将非类型模板参数声明成参数包。比如对：

```
template<std::size_t… Idx, typename C>
```

```
void printIdx (C const& coll)
{
    print(coll[Idx]…);
}
```

可以这样调用：

```
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printIdx<2,0,3>(coll);
```

效果上和前面的例子相同。

4.4.3 变参类模板

类模板也可以是变参的。一个重要的例子是，通过任意多个模板参数指定了 class 相应数据成员的类型：

```
template<typename… Elements>class Tuple;
Tuple<int, std::string, char> t; // t can hold integer, string, and character
```

这一部分内容会在第 25 章讨论。

另一个例子是指定对象可能包含的类型：

```
template<typename… Types>
class Variant;
Variant<int, std::string, char> v; // v can hold integer, string, or character
```

这一部分内容会在 26 章介绍。

也可以将 class 定义成代表了一组下表的类型：

```
// type for arbitrary number of indices:
template<std::size_t…>
struct Indices {
};
```

可以用它定义一个通过 print()打印 std::array 或者 std::tuple 中元素的函数，具体打印哪些元素由编译阶段的 get<>从给定的下标中获取：

```
template<typename T, std::size_t… Idx>
void printByIdx(T t, Indices<Idx…>)
{
    print(std::get<Idx>(t)…);
}
```

可以像下面这样使用这个模板：

```
std::array<std::string, 5> arr = {"Hello", "my", "new", "!", "World"};
printByIdx(arr, Indices<0, 4, 3>());
```

或者像下面这样：

```
auto t = std::make_tuple(12, "monkeys", 2.0);
printByIdx(t, Indices<0, 1, 2>());
```

这是迈向元编程（meta-programming）的第一步，在 8.1 节和第 23 章会有相应的介绍。

4.4.4 变参推断指引

推断指引（参见 2.9 节）也可以是变参的。比如在 C++标准库中，为 std::array 定义了如下推断指引：

```
namespace std {
    template<typename T, typename… U> array(T, U…)
    -> array<enable_if_t<(is_same_v<T, U> && …), T>, (1 + sizeof…(U))>;
}
```

针对这样的初始化:

```
std::array a{42,45,77};
```

会将指引中的 T 推断为 array（首）元素的类型，而 U...会被推断为剩余元素的类型。因此 array 中元素总数目是 1 + sizeof...(U)，等效于如下声明:

```
std::array<int, 3> a{42,45,77};
```

其中对 array 第一个参的操作 std::enable_if<>是一个折叠表达式（和 4.1 节中的 isHomogeneous()情况类似），可以展开成这样：

```
is_same_v<T, U1> && is_same_v<T, U2> && is_same_v<T, U3> …
```

如果结果是 false（也就是说 array 中元素不是同一种类型），推断指引会被弃用，总的类型推断失败。这样标准库就可以确保在推断指引成功的情况下，所有元素都是同一种类型。

4.4.5 变参基类及其使用

最后，考虑如下例子：

```
#include <string>
#include <unordered_set>
class Customer
{
    private:
```

```
        std::string name;
    public:
        Customer(std::string const& n) : name(n) { }
        std::string getName() const { return name; }
};
struct CustomerEq {
    bool operator() (Customer const& c1, Customer const& c2) const {
        return c1.getName() == c2.getName();
    }
};
struct CustomerHash {
    std::size_t operator() (Customer const& c) const {
        return std::hash<std::string>()(c.getName());
    }
};
// define class that combines operator() for variadic base classes:
template<typename… Bases>
struct Overloader : Bases…
{
    using Bases::operator()…; // OK since C++17
};
int main()
{
    // combine hasher and equality for customers in one type:
    using CustomerOP = Overloader<CustomerHash,CustomerEq>;
    std::unordered_set<Customer,CustomerHash,CustomerEq> coll1;
    std::unordered_set<Customer,CustomerOP,CustomerOP> coll2;
    …
}
```

这里首先定义了一个 Customer 类和一些用来比较 Customer 对象以及计算这些对象 hash 值的函数对象。通过

```
template<typename… Bases>
struct Overloader : Bases…
{
    using Bases::operator()…; // OK since C++17
};
```

从个数不定的基类派生出了一个新的类，并且从其每个基类中引入了 operator()的声明。比如通过：

```
using CustomerOP = Overloader<CustomerHash,CustomerEq>;
```

从 CustomerHash 和 CustomerEq 派生出了 CustomerOP，而且派生类中会包含两个基类中的 operator()的实现。

在 26.4 节介绍了另外一个使用了该技术的例子。

4.5 总结

- 通过使用参数包，模板可以有任意多个任意类型的参数。
- 为了处理这些参数，需要使用递归，而且需要一个非变参函数终结递归（如果使用编译期判断，则不需要非变参函数来终结递归）。
- 运算符 sizeof...用来计算参数包中模板参数的数目。
- 变参模板的一个典型应用是用来发送（forward）任意多个任意类型的模板参数。
- 通过使用折叠表达式，可以将某种运算应用于参数包中的所有参数。

第 5 章 基础技巧

本章将涉及一些和模板实际使用有关的晋级知识，包含：关键字 typename 的使用，定义为模板的成员函数以及嵌套类，模板参数模板（template template parameters），零初始化以及其它一些关于使用字符串常量作为模板参数的细节讨论。这些内容有时会比较复杂，但是作为一个 C++的日常使用者，应该至少已经听说过它们了。

5.1 typename 关键字

关键字 typename 在 C++标准化过程中被引入进来，用来澄清模板内部的一个标识符代表的是某种类型，而不是数据成员。考虑下面这个例子：

```
template<typename T>
class MyClass {
    public:
        ...
        void foo() {
            typename T::SubType* ptr;
        }
};
```

其中第二个 typename 被用来澄清 SubType 是定义在 class T 中的一个类型。因此在这里 ptr 是一个指向 T::SubType 类型的指针。

如果没有 typename 的话，SubType 会被假设成一个非类型成员（比如 static 成员或者一个枚举常量，亦或者是内部嵌套类或者 using 声明的 public 别名）。这样的话，表达式 T::SubType* ptr

会被理解成 class T 的 static 成员 SubType 与 ptr 的乘法运算，这不是一个错误，因为对 MyClass<>的某些实例化版本而言，这可能是有效的代码。

通常而言，当一个依赖于模板参数的名称代表的是某种类型的时候，就必须使用 typename。13.3.2 节会对这一内容做进一步的讨论。

使用 typename 的一种场景是用来声明泛型代码中标准容器的迭代器：

```
#include <iostream>
// print elements of an STL container
template<typename T>
void printcoll (T const& coll)
{
    typename T::const_iterator pos; // iterator to iterate over coll
    typename T::const_iterator end(coll.end()); // end position
```

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

在这个函数模板中，调用参数是一个类型为 T 的标准容器。为了遍历容器中的所有元素，使用了声明于每个标准容器中的迭代器类型：

```
class stlcontainer {
    public:
        using iterator = ···; // iterator for read/write access
        using const_iterator = ···; // iterator for read access
        ···
};
```

因此为了使用模板类型 T 的 cons_iterator，必须在其前面使用 typename:

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

关于在 C++17 之前 typename 使用的更多细节请参见 13.3.2 节。但是对于 C++20，在某些常规情况下可能不再需要 typename（参见 17.1 节）。

5.2 零初始化

对于基础类型，比如 int，double 以及指针类型，由于它们没有默认构造函数，因此它们不会被默认初始化成一个有意义的值。比如任何未被初始化的局部变量的值都是未定义的：

```
void foo()
{
    int x; // x has undefined value
    int* ptr; // ptr points to anywhere (instead of nowhere)
}
```

因此在定义模板时，如果想让一个模板类型的变量被初始化成一个默认值，那么只是简单的定义是不够的，因为对内置类型，它们不会被初始化：

```
template<typename T>
void foo()
{
    T x; // x has undefined value if T is built-in type
}
```

出于这个原因，对于内置类型，最好显式的调用其默认构造函数来将它们初始化成 0（对于 bool 类型，初始化为 false，对于指针类型，初始化成 nullptr）。通过下面你的写法就可以保证即使是内置类型也可以得到适当的初始化：

```
template<typename T>
```

```
void foo()
{
    T x{}; // x is zero (or false) if T is a built-in type
}
```

这种初始化的方法被称为“值初始化（value initialization）”，它要么调用一个对象已有的构造函数，要么就用零来初始化这个对象。即使它有显式的构造函数也是这样。

在 C++11 之前，确保一个对象得到显示初始化的方式是：

```
T x = T(); // x is zero (or false) if T is a built-in type
```

在 C++17 之前，只有在与拷贝初始化对应的构造函数没有被声明为 explicit 的时候，这一方式才有效（目前也依然被支持）。从 C++17 开始，由于强制拷贝省略（mandatory copy elision）的使用，这一限制被解除，因此在 C++17 之后以上两种方式都有效。不过对于用花括号初始化的情况，如果没有可用的默认构造函数，它还可以使用列表初始化构造函数（initializer-list constructor）。

为确保类模板中类型被参数化了的成员得到适当的初始化，可以定义一个默认的构造函数并在其中对相应成员做初始化：

```
template<typename T>
class MyClass {
    private:
        T x;
    public:
        MyClass() : x{} { // ensures that x is initialized even for
built-in types
        }
    ...
};
```

C++11 之前的语法：

```
MyClass() : x() { //ensures that x is initialized even forbuilt-in types
}
```

也依然有效。

从 C++11 开始也可以通过如下方式对非静态成员进行默认初始化：

```
template<typename T>
class MyClass {
    private:
        T x{}; // zero-initialize x unless otherwise specified
    ...
};
```

但是不可以对默认参数使用这一方式，比如：

```
template<typename T>
void foo(T p{}) { //ERROR
    ...
}
```

对这种情况必须像下面这样初始化：

```
template<typename T>
void foo(T p = T{}) { //OK (must use T() before C++11)
    ...
}
```

5.3 使用 this->

对于类模板，如果它的基类也是依赖于模板参数的，那么对它而言即使 x 是继承而来的，使用 this->x 和 x 也不一定是等效的。比如：

```
template<typename T>
class Base {
    public:
        void bar();
};
template<typename T>
class Derived : Base<T> {
    public:
        void foo() {
            bar(); // calls external bar() or error
        }
};
```

Derived 中的 bar()永远不会被解析成 Base 中的 bar()。因此这样做要么会遇到错误，要么就是调用了其它地方的 bar()（比如可能是定义在其它地方的 global 的 bar()）。

13.4.2 节对这一问题有更详细的讨论。目前作为经验法则，建议当使用定义于基类中的、依赖于模板参数的成员时，用 this->或者 Base<T>::来修饰它。

5.4 使用裸数组或者字符串常量的模板

当向模板传递裸数组或者字符串常量时，需要格外注意以下内容：
第一，如果参数是按引用传递的，那么参数类型不会退化（decay）。也就是说当传递”hello”作为参数时，模板类型会被推断为 char const[6]。这样当向模板传递长度不同的裸数组或者字符串常量时就可能遇到问题，因为它们对应的模板类型不一样。只有当按值传递参数时，模板类型才会退化（decay），这样字符串常量会被推断为 char const *。相关内容会在第 7

章进行讨论。

不过也可以像下面这样定义专门用来处理裸数组或者字符串常量的模板：

```
template<typename T, int N, int M>
bool less (T(&a)[N], T(&b)[M])
{
    for (int i = 0; i<N && i<M; ++i)
    {
        if (a[i]<b[i]) return true; if (b[i]<a[i]) return false;
    }
    return N < M;
}
```

当像下面这样使用该模板的时候：

```
int x[] = {1, 2, 3};
int y[] = {1, 2, 3, 4, 5};
std::cout << less(x,y) << ' \n' ;
```

less<>中的 T 会被实例化成 int，N 被实例化成 3，M 被实例化成 5。

也可以将该模板用于字符串常量：

```
std::cout << less("ab","abc") << ' \n' ;
```

这里 less<>中的 T 会被实例化成 char const，N 被实例化成 3，M 被实例化成 4。

如果想定义一个只是用来处理字符串常量的函数模板，可以像下面这样：

```
template<int N, int M>
bool less (char const(&a)[N], char const(&b)[M])
{
    for (int i = 0; i<N && i<M; ++i) {
        if (a[i]<b[i]) return true;
        if (b[i]<a[i]) return false;
    }
    return N < M;
}
```

请注意你可以、某些情况下可能也必须去为边界未知的数组做重载或者部分特化。下面的代码展示了对数组所做的所有可能的重载：

```
#include <iostream>
template<typename T>
struct MyClass;  // 主模板
template<typename T, std::size_t SZ>
struct MyClass<T[SZ]> // partial specialization for arrays of known
bounds
```

```
{
    static void print()
     {
        std::cout << "print() for T[" << SZ << "]\n";
    }
};
template<typename T, std::size_t SZ>
struct MyClass<T(&)[SZ]>  // partial spec. for references to arrays of
known bounds
{
    static void print() {
        std::cout << "print() for T(&)[" << SZ <<"]\n";
    }
};
template<typename T>
struct MyClass<T[]> // partial specialization for arrays of unknown
bounds
{
    static void print() {
    std::cout << "print() for T[]\n";
}
};
template<typename T>
struct MyClass<T(&)[]> // partial spec. for references to arrays of
unknown bounds
{
    static void print() {
    std::cout << "print() for T(&)[]\n";
}
};
template<typename T>
struct MyClass<T*> // partial specialization for pointers
{
    static void print() {
    std::cout << "print() for T*\n";
}
};
```

上面的代码针对以下类型对 MyClass<>做了特化：边界已知和未知的数组，边界已知和未知的数组的引用，以及指针。它们之间互不相同，在各种情况下的调用关系如下：

```
#include "arrays.hpp"
template<typename T1, typename T2, typename T3>
void foo(int a1[7], int a2[], // pointers by language rules
    int (&a3)[42], // reference to array of known bound
```

```
    int (&x0)[], // reference to array of unknown bound
    T1 x1, // passing by value decays
    T2& x2, T3&& x3) // passing by reference
{
    MyClass<decltype(a1)>::print(); // uses MyClass<T*>
    MyClass<decltype(a2)>::print(); // uses MyClass<T*> a1, a2 退化成
指针
    MyClass<decltype(a3)>::print(); // uses MyClass<T(&)[SZ]>
    MyClass<decltype(x0)>::print(); // uses MyClass<T(&)[]>
    MyClass<decltype(x1)>::print(); // uses MyClass<T*>
    MyClass<decltype(x2)>::print(); // uses MyClass<T(&)[]>
    MyClass<decltype(x3)>::print(); // uses MyClass<T(&)[]> // 万能引
用，引用折叠
}

int main()
{
    int a[42];
    MyClass<decltype(a)>::print(); // uses MyClass<T[SZ]>
    extern int x[]; // forward declare array
    MyClass<decltype(x)>::print(); // uses MyClass<T[]>
    foo(a, a, a, x, x, x, x);
}

int x[] = {0, 8, 15}; // define forward-declared array
```

注意，根据语言规则，如果调用参数被声明为数组的话，那么它的真实类型是指针类型。而且针对未知边界数组定义的模板，可以用于不完整类型，比如：

```
extern int i[];
```

当这一数组被按照引用传递时，它的类型是 int(&)[]，同样可以用于模板参数。

19.3.1 节会介绍另一个在泛型代码中使用了不同数组类型的例子。

5.5 成员模板

类的成员也可以是模板，对嵌套类和成员函数都是这样。这一功能的作用和优点同样可以通过 Stack<>类模板得到展现。通常只有当两个 stack 类型相同的时候才可以相互赋值（stack 的类型相同说明它们的元素类型也相同）。即使两个 stack 的元素类型之间可以隐式转换，也不能相互赋值：

```
Stack<int> intStack1, intStack2; // stacks for ints
Stack<float> floatStack; // stack for floats
...
```

```
intStack1 = intStack2; // OK: stacks have same type
floatStack = intStack1; // ERROR: stacks have different types
```

默认的赋值运算符要求等号两边的对象类型必须相同，因此如果两个 stack 之间的元素类型不同的话，这一条件将得不到满足。

但是，只要将赋值运算符定义成模板，就可以将两个元素类型可以做转换的 stack 相互赋值。新的 Stack<>定义如下：

```
template<typename T>
class Stack {
    private:
        std::deque<T> elems; // elements
    public:
        void push(T const&); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
        // assign stack of elements of type T2
        template<typename T2>
        Stack& operator= (Stack<T2> const&);
};
```

以上代码中有如下两点改动：

1. 赋值运算符的参数是一个元素类型为 T2 的 stack。
2. 新的模板使用 std::deque<>作为内部容器。这是为了方便新的赋值运算符的定义。

新的赋值运算符被定义成下面这样：

```
template<typename T>
template<typename T2>
Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
{
    Stack<T2> tmp(op2); // create a copy of the assigned stack
    elems.clear(); // remove existing elements
    while (!tmp.empty()) { // copy all elements
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}
```

下面先来看一下成员模板的定义语法。在模板类型为 T 的模板内部，定义了一个模板类型为 T2 的内部模板：

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

在模板函数内部，你可能希望简化 op2 中相关元素的访问。但是由于 op2 属于另一种类型（如果用来实例化类模板的参数类型不同，那么实例化出来的类的类型也不同），因此最好使用它们的公共接口。这样访问元素的唯一方法就是通过调用 top()。这就要求 op2 中所有元素相继出现在栈顶，为了不去改动 op2，就需要做一次 op2 的拷贝。由于 top()返回的是最后一个被添加进 stack 的元素，因此需要选用一个支持在另一端插入元素的容器，这就是为什么选用 std::deque<>的原因，因为它的 push_front()方法可以将元素添加到另一端。

为了访问 op2 的私有成员，可以将其它所有类型的 stack 模板的实例都定义成友元：

```
template<typename T>
class Stack {
    private:
        std::deque<T> elems; // elements
    public:
        Void push(T const&); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
        // assign stack of elements of type T2
        template<typename T2>
        Stack& operator= (Stack<T2> const&);
        // to get access to private members of Stack<T2> for any type
T2:
        template<typename> friend class Stack;
};
```

如你所见，由于模板参数的名字不会被用到，因此可以被省略掉：

```
template<typename> friend class Stack;
```

这样就就可以将赋值运算符定义成如下形式：

```
template<typename T>
template<typename T2>
Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
{
    elems.clear(); // remove existing elements
    elems.insert(elems.begin(), // insert at the beginning
    op2.elems.begin(), // all elements from op2
    op2.elems.end());
    return *this;
```

```
}
```

无论采用哪种实现方式，都可以通过这个成员模板将存储 int 的 stack 赋值给存储 float 的 stack：

```
Stack<int> intStack; // stack for ints
Stack<float> floatStack; // stack for floats
...
floatStack = intStack; // OK: stacks have different types,
// but int converts to float
```

当然，这样的赋值就不会改变 floatStack 的类型，也不会改变它的元素的类型。在赋值之后，floatStack 存储的元素依然是 float 类型，top()返回的值也依然是 float 类型。

看上去这个赋值运算符模板不会进行类型检查，这样就可以在存储任意类型的两个 stack 之间相互赋值，但是事实不是这样。必要的类型检查会在将源 stack（上文中的 op2 或者其备份 temp）中的元素插入到目标 stack 中的时候进行：

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

比如如果将存储 string 的 stack 赋值给存储 int 的 stack，那么在编译这一行代码的时候会遇到如下错误信息：不能将通过 tmp.top()返回的 string 用作 elems.push_front()的参数（不同编译器产生的错误信息可能会有所不同，但大体上都是这个意思）：

```
Stack<std::string> stringStack; // stack of strings
Stack<float> floatStack; // stack of floats
...
floatStack = stringStack; // ERROR: std::string doesn't convert to float
```

同样也可以将内部的容器类型参数化：

```
template<typename T, typename Cont = std::deque<T>>
class Stack {
    private:
        Cont elems; // elements
    public:
        void push(T const&); // push element
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
        // assign stack of elements of type T2
        template<typename T2, typename Cont2>
        Stack& operator= (Stack<T2,Cont2> const&);
        // to get access to private members of Stack<T2> for any type
T2:
        template<typename, typename> friend class Stack;
```

```
};
```

此时赋值运算符的实现会像下面这样：

```
template<typename T, typename Cont>
    template<typename T2, typename Cont2>
Stack<T,Cont>& Stack<T,Cont>::operator= (Stack<T2,Cont2> const& op2)
{
    elems.clear(); // remove existing elements
    elems.insert(elems.begin(), // insert at the beginning
    op2.elems.begin(), // all elements from op2
    op2.elems.end());
    return *this;
}
```

记住，对类模板而言，其成员函数只有在被用到的时候才会被实例化。因此对上面的例子，如果能够避免在不同元素类型的 stack 之间赋值的话，甚至可以使用 vector（没有 push_front 方法）作为内部容器：

```
// stack for ints using a vector as an internal container
Stack<int,std::vector<int>> vStack;
…
vStack.push(42); vStack.push(7);
std::cout << vStack.top() << ’ \n’ ;
```

由于没有用到赋值运算符模板，程序运行良好，不会报错说 vector 没有 push_front()方法。

关于最后一个例子的完整实现，请参见 basics 目录中所有以 stack7 作为名字开头的文件。

成员模板的特例化

成员函数模板也可以被全部或者部分地特例化。比如对下面这个例子：

```
class BoolString {
    private:
        std::string value;
    public:
        BoolString (std::string const& s)
        : value(s) {}
        template<typename T = std::string>
        T get() const { // get value (converted to T)
            return value;
        }
};
```

可以像下面这样对其成员函数模板 get()进行全特例化：

```
// full specialization for BoolString::getValue<>() for bool
```

```
template<>
inline bool BoolString::get<bool>() const {
    return value == "true" || value == "1" || value == "on";
}
```

注意我们不需要也不能够对特例化的版本进行声明；只能定义它们。由于这是一个定义于头文件中的全实例化版本，如果有多个编译单 include 了这个头文件，为避免重复定义的错误，必须将它定义成 inline 的。

可以像下面这样使用这个 class 以及它的全特例化版本：

```
std::cout << std::boolalpha;
BoolString s1("hello");
std::cout << s1.get() << ' \n' ; //prints hello
std::cout << s1.get<bool>() << ' \n' ; //prints false
BoolString s2("on");
std::cout << s2.get<bool>() << ' \n' ; //prints true
```

特殊成员函数的模板

如果能够通过特殊成员函数 copy 或者 move 对象，那么相应的特殊成员函数（copy 构造函数以及 move 构造函数）也将可以被模板化。和前面定义的赋值运算符类似，构造函数也可以是模板。但是需要注意的是，构造函数模板或者赋值运算符模板不会取代预定义的构造函数和赋值运算符。成员函数模板不会被算作用来 copy 或者 move 对象的特殊成员函数。在上面的例子中，如果在相同类型的 stack 之间相互赋值，调用的依然是默认赋值运算符。

这种行为既有好处也有坏处：

- 某些情况下，对于某些调用，构造函数模板或者赋值运算符模板可能比预定义的 copy/move 构造函数或者赋值运算符更匹配，虽然这些特殊成员函数模板可能原本只打算用于在不同类型的 stack 之间做初始化。详情请参见 6.2 节。
- 想要对 copy/move 构造函数进行模板化并不是一件容易的事情，比如该如何限制其存在的场景。详情请参见 6.4 节。

5.5.1 .template 的使用

某些情况下，*在调用成员模板的时候需要显式地指定其模板参数的类型*。这时候就需要使用关键字 template 来确保符号<会被理解为模板参数列表的开始，而不是一个比较运算符。考虑下面这个使用了标准库中的 bitset 的例子：

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

```
}
```

对于 bitset 类型的 bs，调用了其成员函数模板 to_string()，并且指定了 to_string()模板的所有模板参数。如果没有.template 的话，编译器会将 to_string()后面的<符号理解成小于运算符，而不是模板的参数列表的开始。*这一这种情况只有在点号前面的对象依赖于模板参数的时候才会发生。在我们的例子中，bs 依赖于模板参数 N。*

.template 标识符（标识符->template 和::template 也类似）只能被用于模板内部，并且它前面的对象应该依赖于模板参数。详情请参见 13.3.3 节。

5.5.2 泛型 lambdas 和成员模板

在 C++14 中引入的泛型 lambdas，是一种成员模板的简化。对于一个简单的计算两个任意类型参数之和的 lambda：

```
[] (auto x, auto y) {
    return x + y;
}
```

编译器会默认为它构造下面这样一个类：

```
class SomeCompilerSpecificName {
    public:
        SomeCompilerSpecificName(); // constructor only callable by
compiler
        template<typename T1, typename T2>
        auto operator() (T1 x, T2 y) const {
            return x + y;
        }
};
```

更多细节请参见 15.10.6 节

5.6 变量模板

从 C++14 开始，变量也可以被某种类型参数化。称为变量模板。

例如可以通过下面的代码定义 pi，但是参数化了其类型：

```
template<typename T>
constexpr T pi{3.1415926535897932385};
```

注意，和其它几种模板类似，这个定义最好不要出现在函数内部或者块作用域内部。

在使用变量模板的时候，必须指明它的类型。比如下面的代码在定义 pi<>的作用域内使用了两个不同的变量：

```
std::cout << pi<double> << ' \n' ;
std::cout << pi<float> << ' \n' ;
```

变量模板也可以用于不同编译单元：

```
template<typename T> T val{}; // zero initialized value//== translation
unit 1:
#include "header.hpp"
int main()
{
    val<long> = 42;
    print();
}
 //== translation unit 2:
#include "header.hpp"
void print()
{
    std::cout << val<long> << ' \n' ; // OK: prints 42
}
```

也可有默认模板类型：

```
template<typename T = long double>
constexpr T pi = T{3.14159265358979323 85};
```

可以像下面这样使用默认类型或者其它类型：

```
std::cout << pi<> << ' \n' ; //outputs a long double
std::cout << pi<float> << ' \n' ; //outputs a float
```

只是无论怎样都要使用尖括号<>，不可以只用 pi：

```
std::cout << pi << ' \n' ; //ERROR
```

同样可以用非类型参数对变量模板进行参数化，也可以将非类型参数用于参数器的初始化。比如：

```
#include <iostream>
#include <array>
template<int N>
std::array<int,N> arr{}; // array with N elements, zero-initialized
template<auto N>
constexpr decltype(N) dval = N; // type of dval depends on passed value

int main()
{
    std::cout << dval<' c' > << ' \n' ; // N has value ' c'  of type char
```

```
    arr<10>[0] = 42; // sets first element of global arr
    for (std::size_t i=0; i<arr<10>.size(); ++i) { // uses values set
in arr
        std::cout << arr<10>[i] << ' \n' ;
    }
}
```

注意在不同编译单元间初始化或者遍历 arr 的时候，使用的都是同一个全局作用域里的 std::array<int, 10> arr。

用于数据成员的变量模板

变量模板的一种应用场景是，用于定义代表类模板成员的变量模板。比如如果像下面这样定义一个类模板：

```
template<typename T>
class MyClass {
    public:
        static constexpr int max = 1000;
};
```

那么就可以为 MyClass<>的不同特例化版本定义不同的值：

```
template<typename T>
int myMax = MyClass<T>::max;
```

应用工程师就可以使用下面这样的代码：

```
auto i = myMax<std::string>;
```

而不是：

```
auto i = MyClass<std::string>::max;
```

这意味着对于一个标准库的类：

```
namespace std {
    template<typename T>
    class numeric_limits {
        public:
            …
        static constexpr bool is_signed = false;
            …
    };
}
```

可以定义：

```
template<typename T>
constexpr bool isSigned = std::numeric_limits<T>::is_signed;
```

这样就可以用：

```
isSigned<char>
```

代替：

```
std::numeric_limits<char>::is_signed
```

类型萃取 Suffix_v

从 C++17 开始，标准库用变量模板为其用来产生一个值（布尔型）的类型萃取定义了简化方式。比如为了能够使用：

```
std::is_const_v<T> // since C++17
```

而不是：

```
std::is_const<T>::value //since C++11
```

标准库做了如下定义：

```
namespace std {
    template<typename T>
        constexpr bool is_const_v = is_const<T>::value;
}
```

5.7 模板参数模板

如果允许模板参数也是一个类模板的话，会有不少好处。在这里依然使用 Stack 类模板作为例子。

对 5.5 节中的 stack 模板，如果不想使用默认的内部容器类型 std::deque，那么就需要两次指定 stack 元素的类型。也就是说为了指定内部容器的类型，必须同时指出容器的类型和元素的类型：

```
Stack<int, std::vector<int>> vStack; // integer stack that uses a vector
```

使用模板参数模板，在声明 Stack 类模板的时候就可以只指定容器的类型而不去指定容器中元素的类型：

```
Stack<int, std::vector> vStack; // integer stack that uses a vector
```

为此就需要在 Stack 的定义中将第二个模板参数声明为模板参数模板。可能像下面这样：

```
template<typename T,
template<typename Elem> class Cont = std::deque>
class Stack {
    private:
        Cont<T> elems; // elements
    public:
        void push(T const&); // push element
```

```
        void pop(); // pop element
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
    ...
};
```

区别在于第二个模板参数被定义为一个类模板：

```
template<typename Elem> class Cont
```

默认类型也从 std::deque<T>变成 std::deque。这个参数必须是一个类模板，它将被第一个模板参数实例化：

```
Cont<T> elems;
```

用第一个模板参数实例化第二个模板参数的情况是由 Stack 自身的情况决定的。实际上，可以在类模板内部用任意类型实例化一个模板参数模板。

和往常一样，声明模板参数时可以使用 class 代替 typename。在 C++11 之前，Cont 只能被某个类模板的名字取代。

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

从 C++11 开始，也可以用别名模板（alias template）取代 Cont，但是直到 C++17，在声明模板参数模板时才可以用 typename 代替 class：

```
template<typename T, template<typename Elem> typename Cont =
std::deque>
class Stack { //ERROR before C++17
    ...
};
```

这两个变化的目的都一样：用 class 代替 typename 不会妨碍我们使用别名模板（alias template）作为和 Cont 对应的模板参数。

由于模板参数模板中的模板参数没有被用到，作为惯例可以省略它（除非它对文档编写有帮助）：

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

成员函数也要做相应的更改。必须将第二个模板参数指定为模板参数模板。比如对于 push() 成员，其实现如下：

```
template<typename T, template<typename> class Cont>
void Stack<T,Cont>::push (T const& elem)
{
    elems.push_back(elem); // append copy of passed elem
}
```

注意，虽然模板参数模板是类或者别名类（alias templates）的占位符，但是并没有与其对应的函数模板或者变量模板的占位符。

模板参数模板的参数匹配

如果你尝试使用新版本的 Stack，可能会遇到错误说默认的 std::deque 和模板参数模板 Cont 不匹配。这是因为在 C++17 之前，template<typename Elem> typename Cont = std::deque 中的模板参数必须和实际参数（std::deque）的模板参数匹配（对变参模板有些例外，见 12.3.4 节）。而且实际参数（std::deque 有两个参数，第二个是默认参数 allocator）的默认参数也要被匹配，这样 template<typename Elem> typename Cont = std::dequ 就不满足以上要求（不过对 C++17 可以）。

作为变通，可以将类模板定义成下面这样：

```
template<typename T, template<typename Elem,
typename Alloc = std::allocator<Elem>> class Cont = std::deque>
class Stack {
   private:
      Cont<T> elems; // elements
   ...
};
```

其中的 Alloc 同样可以被省略掉。

因此最终的 Stack 模板会像下面这样（包含了赋值运算符模板）：

```
#include <deque>
#include <cassert>
#include <memory>
template<typename T,  template<typename Elem, typename =
std::allocator<Elem>> class Cont = std::deque>
class Stack {
   private:
      Cont<T> elems; // elements
   public:
      void push(T const&); // push element
      void pop(); // pop element
```

```
        T const& top() const; // return top element
        bool empty() const { // return whether the stack is empty
            return elems.empty();
        }
        // assign stack of elements of type T2
        template<typename T2, template<typename Elem2,
        typename = std::allocator<Elem2> >class Cont2>
        Stack<T,Cont>& operator= (Stack<T2,Cont2> const&);
        // to get access to private members of any Stack with elements
of type T2:
        template<typename, template<typename, typename>class>
        friend class Stack;
};
template<typename T, template<typename,typename> class Cont>
void Stack<T,Cont>::push (T const& elem)
{elems.push_back(elem); // append copy of passed elem
}
template<typename T, template<typename,typename> class Cont>
void Stack<T,Cont>::pop ()
{
    assert(!elems.empty());
    elems.pop_back(); // remove last element
}
template<typename T, template<typename,typename> class Cont>
T const& Stack<T,Cont>::top () const
{
    assert(!elems.empty());
    return elems.back(); // return copy of last element
}
template<typename T, template<typename,typename> class Cont>
template<typename T2, template<typename,typename> class Cont2>
Stack<T,Cont>&
Stack<T,Cont>::operator= (Stack<T2,Cont2> const& op2)
{
    elems.clear(); // remove existing elements
    elems.insert(elems.begin(), // insert at the beginning
    op2.elems.begin(), // all elements from op2
    op2.elems.end());
    return *this;
}
```

这里为了访问赋值运算符 op2 中的元素，将其它所有类型的 Stack 声明为 friend（省略模板参数的名称）：

```
template<typename, template<typename, typename>class>
```

```
friend class Stack;
```

同样，不是所有的标准库容器都可以用做 Cont 参数。比如 std::array 就不行，因为它有一个非类型的代表数组长度的模板参数，在上面的模板中没有与之对应的模板参数。

下面的例子用到了最终版 Stack 模板的各种特性：

```
#include "stack9.hpp"
#include <iostream>
#include <vector>
int main()
{
    Stack<int> iStack; // stack of ints
    Stack<float> fStack; // stack of floats
    // manipulate int stack
    iStack.push(1);
    iStack.push(2);
    std::cout << "iStack.top(): " << iStack.top() << '\n';
    // manipulate float stack:
    fStack.push(3.3);
    std::cout << "fStack.top(): " << fStack.top() << '\n';
    // assign stack of different type and manipulate again
    fStack = iStack;
    fStack.push(4.4);
    std::cout << "fStack.top(): " << fStack.top() << '\n';
    // stack for doubless using a vector as an internal container
    Stack<double, std::vector> vStack;
    vStack.push(5.5);
    vStack.push(6.6);
    std::cout << "vStack.top(): " << vStack.top() << '\n';
    vStack = fStack;
    std::cout << "vStack: ";
    while (! vStack.empty()) {
        std::cout << vStack.top() << ' ';
        vStack.pop();
    }
    std::cout << '\n';
}
```

程序输出如下：

```
iStack.top(): 2
fStack.top(): 3.3
fStack.top(): 4.4
vStack.top(): 6.6
vStack: 4.4 2 1
```

关于模板参数模板的进一步讨论，参见 12.2.3 节，12.3.4 节和 19.2.2 节。

5.8 总结

- 为了使用依赖于模板参数的类型名称，需要用 typename 修饰该名称。
- 为了访问依赖于模板参数的父类中的成员，需要用 this->或者类名修饰该成员。
- 嵌套类或者成员函数也可以是模板。一种应用场景是实现可以进行内部类型转换的泛型代码。
- 模板化的构造函数或者赋值运算符不会取代预定义的构造函数和赋值运算符。
- 使用花括号初始化或者显式地调用默认构造函数，可以保证变量或者成员模板即使被内置类型实例化，也可以被初始化成默认值。
- 可以为裸数组提供专门的特化模板，它也可以被用于字符串常量。
- 只有在裸数组和字符串常量不是被按引用传递的时候，参数类型推断才会退化。（裸数组退化成指针）
- 可以定义变量模板（从 C++14 开始）。
- 模板参数也可以是类模板，称为模板参数模板（template template parameters）。
- 模板参数模板的参数类型必须得到严格匹配。

第 6 章 移动语义和 enable_if<>

移动语义（move semantics）是 C++11 引入的一个重要特性。在 copy 或者赋值的时候，可以通过它将源对象中的内部资源 move（“steal”）到目标对象，而不是 copy 这些内容。当然这样做的前提是源对象不在需要这些内部资源或者状态（因为源对象将会被丢弃）。

移动语义对模板的设计有重要影响，在泛型代码中也引入了一些特殊的规则来支持移动语义。本章将会介绍移动语义这一特性。

6.1 完美转发（Perfect Forwarding）

（本节讲的不好，建议参考《effective modern c++》和《C++ Primer》）

假设希望实现的泛型代码可以将被传递参数的基本特性转发出去：

- 可变对象被转发之后依然可变。
- Const 对象被转发之后依然是 const 的。
- 可移动对象（可以从中窃取资源的对象）被转发之后依然是可移动的。

不使用模板的话，为达到这一目的就需要对以上三种情况分别编程。比如为了将调用 f()时传递的参数转发给函数 g():

```
#include <utility>
#include <iostream>
class X {
    ...
};
void g (X&) {
    std::cout << "g() for variable\n";
}
void g (X const&) {
    std::cout << "g() for constant\n";
}
void g (X&&) {
    std::cout << "g() for movable object\n";
}
 // let f() forward argument val to g():
void f (X& val) {
  g(val); // val is non-const lvalue => calls g(X&)
}
void f (X const& val) {
  g(val); // val is const lvalue => calls g(X const&)
```

```
}
void f (X&& val) {
  g(std::move(val)); // val is non-const lvalue => needs ::move() to
call g(X&&)
}
int main()
{
    X v; // create variable
    X const c; // create constant
    f(v); // f() for nonconstant object calls f(X&) => calls g(X&)
    f(c); // f() for constant object calls f(X const&) => calls g(X const&)
    f(X()); // f() for temporary calls f(X&&) => calls g(X&&)
    f(std::move(v)); // f() for movable variable calls f(X&&) => calls
g(X&&)
}
```

这里定义了三种不同的 f()，它们分别将其参数转发给 g():

```
void f (X& val) {
    g(val); // val is non-const lvalue => calls g(X&)
}
void f (X const& val) {
    g(val); // val is const lvalue => calls g(X const&)
}
void f (X&& val) {
    g(std::move(val)); // val is non-const lvalue => needs std::move()
to call g(X&&)
}
```

注意其中针对可移动对象（一个右值引用）的代码不同于其它两组代码；它需要用 std::move()来处理其参数，因为参数的移动语义不会被一起传递。虽然第三个 f()中的 val 被声明成右值引用，但是当其在 f()内部被使用时，它依然是一个非常量左值（参考附录 B），其行为也将和第一个 f()中的情况一样。因此如果不使用 std::move()的话，在第三个 f()中调用的将是 g(X&)而不是 g(X&&)。

如果试图在泛型代码中统一以上三种情况，会遇到这样一个问题：

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

这个模板只对前两种情况有效，对第三种用于可移动对象的情况无效。

基于这一原因，C++11 引入了特殊的规则对参数进行完美转发（perfect forwarding）。实现这一目的的惯用方法如下：

```
template<typename T>
void f (T&& val) {
    g(std::forward<T>(val)); // perfect forward val to g()
}
```

注意 std::move 没有模板参数，并且会无条件地移动其参数；而 std::forward<>会跟据被传递参数的具体情况决定是否“转发”其潜在的移动语义。

不要以为模板参数 T 的 T&&和具体类型 X 的 X&&是一样的。虽然语法上看上去类似，但是它们适用于不同的规则：

- 具体类型 X 的 X&&声明了一个右值引用参数。只能被绑定到一个可移动对象上（一个 prvalue，比如临时对象，一个 xvalue，比如通过 std::move()传递的参数，更多细节参见附录 B）。它的值总是可变的，而且总是可以被“窃取”。
- 模板参数 T 的 T&&声明了一个转发引用（亦称万能引用）。可以被绑定到可变、不可变（比如 const）或者可移动对象上。在函数内部这个参数也可以是可变、不可变或者指向一个可以被窃取内部数据的值。

注意 T 必须是模板参数的名字。只是依赖于模板参数是不可以的。对于模板参数 T，形如 typename T::iterator&&的声明只是声明了一个右值引用，不是一个转发引用。

因此，一个可以完美转发其参数的程序会像下面这样：

```
#include <utility>
#include <iostream>
class X {
    ...
};
void g (X&) {
    std::cout << "g() for variable\n";
}
void g (X const&) {
    std::cout << "g() for constant\n";
}
void g (X&&) {
    std::cout << "g() for movable object\n";
}
// let f() perfect forward argument val to g():
template<typename T>
void f (T&& val) {
    g(std::forward<T>(val)); // call the right g() for any passed
argument val
}

int main()
{
```

```
    X v; // create variable
    X const c; // create constant
    f(v); // f() for variable calls f(X&) => calls g(X&)
    f(c); // f() for constant calls f(X const&) => calls g(X const&)
    f(X()); // f() for temporary calls f(X&&) => calls g(X&&)
    f(std::move(v));  // f() for move-enabled variable calls f(X&&)=>
calls g(X&&)
}
```

完美转发同样可以被用于变参模板。更多关于完美转发的细节请参见 15.6.3 节。

6.2 特殊成员函数模板

特殊成员函数也可以是模板，比如构造函数，但是有时候这可能会带来令人意外的结果。

考虑下面这个例子：

```
#include <utility>
#include <string>
#include <iostream>
class Person
{
    private:
        std::string name;
    public:
    // constructor for passed initial name:
    explicit Person(std::string const& n) : name(n) {
        std::cout << "copying string-CONSTR for ' " << name << "' \n";
    }
    explicit Person(std::string&& n) : name(std::move(n)) {
        std::cout << "moving string-CONSTR for ' " << name << "' \n";
    }
    // copy and move constructor:
    Person (Person const& p) : name(p.name) {
        std::cout << "COPY-CONSTR Person ' " << name << "' \n";
    }
    Person (Person&& p) : name(std::move(p.name)) {
        std::cout << "MOVE-CONSTR Person ' " << name << "' \n";
    }
};
int main(){
    std::string s = "sname";
    Person p1(s); // init with string object => calls copying
string-CONSTR
```

```
    Person p2("tmp"); // init with string literal => calls moving
string-CONSTR
    Person p3(p1); // copy Person => calls COPY-CONSTR
    Person p4(std::move(p1)); // move Person => calls MOVE-CONST
}
```

例子中 Person 类有一个 string 类型的 name 成员和几个初始化构造函数。为了支持移动语义，重载了接受 std::string 作为参数的构造函数：

- 一个以 std::string 对象为参数，并用其副本来初始化 name 成员：

```
Person(std::string const& n) : name(n) {
    std::cout << "copying string-CONSTR for ' " << name << "' \n";
}
```

- 一个以可移动的 std::string 对象作为参数，并通过 std:move()从中窃取值来初始化 name：

```
Person(std::string&& n) : name(std::move(n)) {
    std::cout << "moving string-CONSTR for ' " << name << "' \n";
}
```

和预期的一样，当传递一个正在使用的值（左值）作为参数时，会调用第一个构造函数，而以可移动对象（右值）为参数时，则会调用第二个构造函数：

```
std::string s = "sname";
Person p1(s); // init with string object => calls copying string-CONSTR
Person p2("tmp"); // init with string literal => calls moving
string-CONSTR
```

除了这两个构造函数，例子中还提供了一个拷贝构造函数和一个移动构造函数，从中可以看出 Person 对象是如何被拷贝和移动的：

```
Person p3(p1); // copy Person => calls COPY-CONSTR
Person p4(std::move(p1)); // move Person => calls MOVE-CONSTR
```

现在将上面两个以 std::string 作为参数的构造函数替换为一个泛型的构造函数，它将传入的参数完美转发（perfect forward）给成员 name：

```
#include <utility>
#include <string>
#include <iostream>
class Person
{
    private:
        std::string name;
    public:
    // generic constructor for passed initial name:
        template<typename STR>
        explicit Person(STR&& n) : name(std::forward<STR>(n)) {
            std::cout << "TMPL-CONSTR for ' " << name << "' \n";
```

完美转发<什么类型的>(那个参数):

std::forward<STR>(n)
完美转发 类型为STR的参数n

```
        }
        // copy and move constructor:
        Person (Person const& p) : name(p.name) {
            std::cout << "COPY-CONSTR Person ' " << name << "' \n";
        }
        Person (Person&& p) : name(std::move(p.name)) {
            std::cout << "MOVE-CONSTR Person ' " << name << "' \n";
        }
};
```

这时如果传入参数是 std::string 的话，依然能够正常工作：

```
std::string s = "sname";
Person p1(s); // init with string object => calls TMPL-CONSTR
Person p2("tmp"); //init with string literal => calls TMPL-CONS
```

注意这里在构建 p2 的时候并不会创建一个临时的 std::string 对象：STR 的类型被推断为 char const[4]。但是将 std::forward<STR>用于指针参数没有太大意义。成员 name 将会被一个以 null 结尾的字符串构造。

但是，当试图调用拷贝构造函数的时候，会遇到错误：

```
Person p3(p1); // ERROR
```

而用一个可移动对象初始化 Person 的话却可以正常工作：

```
Person p4(std::move(p1)); // OK: move Person => calls MOVECONST
```

如果试图拷贝一个 Person 的 const 对象的话，也没有问题：

```
Person const p2c("ctmp"); //init constant object with string literal
Person p3c(p2c); // OK: copy constant Person => calls COPY-CONSTR
```

问题出在这里：根据 C++重载解析规则（参见 16.2.5 节），对于一个非 const 左值的 Person p，成员模板

```
template<typename STR>
Person(STR&& n)
```

通常比预定义的拷贝构造函数更匹配：

```
Person (Person const& p)
```

这里 STR 可以直接被替换成 Person&，但是对拷贝构造函数还要做一步 const 转换。

额外提供一个非 const 的拷贝构造函数看上去是个不错的方法：

```
Person (Person& p)
```

不过这只是一个部分解决问题的方法，更好的办法依然是使用模板。我们真正想做的是当参数是一个 Person 对象或者一个可以转换成 Person 对象的表达式时，不要启用模板。这可以

通过 std::enable_if<>实现，它也正是下一节要讲的内容。

6.3 通过 std::enable_if<>禁用模板

从 C++11 开始，通过 C++标准库提供的辅助模板 std::enable_if<>，可以在某些编译期条件下忽略掉函数模板。

比如，如果函数模板 foo<>的定义如下：

```
template<typename T>
typename std::enable_if<(sizeof(T) > 4)>::type
foo() {
}
```

这一模板定义会在 sizeof(T) > 4 不成立的时候被忽略掉。如果 sizeof<T> > 4 成立，函数模板会展开成：

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

也就是说 std::enable_if<>是一种类型萃取（type trait），它会根据一个作为其（第一个）模板参数的编译期表达式决定其行为：

- 如果这个表达式结果为 true，它的 type 成员会返回一个类型：
  - 如果没有第二个模板参数，返回类型是 void。
  - 否则，返回类型是其第二个参数的类型。
- 如果表达式结果 false，则其成员类型是未定义的。根据模板的一个叫做 SFINAE（substitute failure is not an error，替换失败不是错误，将在 8.4 节进行介绍）的规则，这会导致包含 std::enable_if<>表达式的函数模板被忽略掉。

由于从 C++14 开始所有的模板萃取（type traits）都返回一个类型，因此可以使用一个与之对应的别名模板 std::enable_if_t<>，这样就可以省略掉 template 和::type 了。如下：

```
template<typename T>
std::enable_if_t<(sizeof(T) > 4)>
foo() {
}
```

如果给 std::enable_if<>或者 std::enable_if_t<>传递第二个模板参数：

```
template<typename T>
std::enable_if_t<(sizeof(T) > 4), T>
foo() {
    return T();
}
```

那么在 sizeof(T) > 4 时，enable_if 会被扩展成其第二个模板参数。因此如果与 T 对应的模板

参数被推断为 MyType，而且其 size 大于 4，那么其等效于：

```
MyType foo();
```

但是由于将 enable_if 表达式放在声明的中间不是一个明智的做法，因此使用 std::enable_if<> 的更常见的方法是使用一个额外的、有默认值的模板参数：

```
template<typename T, typename = std::enable_if_t<(sizeof(T) > 4)>>
void foo() {
}
```

如果 sizeof(T) > 4，它会被展开成：

```
template<typename T, typename = void>
void foo() {
}
```

如果你认为这依然不够明智，并且希望模板的约束更加明显，那么你可以用别名模板（alias template）给它定义一个别名：

```
template<typename T>
using EnableIfSizeGreater4 = std::enable_if_t<(sizeof(T) > 4)>;

template<typename T, typename = EnableIfSizeGreater4<T>>
void foo() {
}
```

关于 std::enable_if 的实现方法，请参见 20.3 节。

6.4 使用 enable_if<>

通过使用 enable_if<>可以解决 6.2 节中关于构造函数模板的问题。

我们要解决的问题是：当传递的模板参数的类型不正确的时候（比如不是 std::string 或者可以转换成 std::string 的类型），禁用如下构造函数模板：

```
template<typename STR>
Person(STR&& n);
```

为了这一目的，需要使用另一个标准库的类型萃取，std::is_convertiable<FROM, TO>。在 C++17 中，相应的构造函数模板的定义如下：

```
template<typename STR, typename =
std::enable_if_t<std::is_convertible_v<STR, std::string>>>
Person(STR&& n);
```

如果 STR 可以转换成 std::string，这个定义会扩展成：

```
template<typename STR, typename = void>
Person(STR&& n);
```

否则这个函数模板会被忽略。

这里同样可以使用别名模板给限制条件定义一个别名：

```
template<typename T>
using EnableIfString = std::enable_if_t<std::is_convertible_v<T,
std::string>>;
…
template<typename STR, typename = EnableIfString<STR>>
Person(STR&& n);
```

现在完整 Person 类如下：

```
#include <utility>
#include <string>
#include <iostream>
#include <type_traits>
template<typename T>
using EnableIfString =
std::enable_if_t<std::is_convertible_v<T,std::string>>;

class Person
{
    private:
        std::string name;
    public:
        // generic constructor for passed initial name:
        template<typename STR, typename = EnableIfString<STR>>
        explicit Person(STR&& n)
        : name(std::forward<STR>(n)) {
            std::cout << "TMPL-CONSTR for ’ " << name << "’ \n";
        }
        // copy and move constructor:
        Person (Person const& p) : name(p.name) {
            std::cout << "COPY-CONSTR Person ’ " << name << "’ \n";
        }
        Person (Person&& p) : name(std::move(p.name)) {
            std::cout << "MOVE-CONSTR Person ’ " << name << "’ \n";
        }
};
```

所有的调用也都会表现正常：

```
#include "specialmemtmpl3.hpp"
int main()
{
    std::string s = "sname";
```

```
    Person p1(s); // init with string object => calls TMPL-CONSTR
    Person p2("tmp"); // init with string literal => calls TMPL-CONSTR
    Person p3(p1); // OK => calls COPY-CONSTR
    Person p4(std::move(p1)); // OK => calls MOVE-CONST
}
```

注意在 C++14 中，由于没有给产生一个值的类型萃取定义带_v 的别名，必须使用如下定义：

```
template<typename T>
using                          EnableIfString                          =
std::enable_if_t<std::is_convertible<T,std::string>::value>;
```

而在 C++11 中，由于没有给产生一个类型的类型萃取定义带_t 的别名，必须使用如下定义：

```
template<typename T>
using EnableIfString
=           typename           std::enable_if<std::is_convertible<T,
std::string>::value >::type;
```

但是通过定义 EnableIfString，这些复杂的语法都被隐藏了。

除了使用要求类型之间可以隐式转换的 std::is_convertible<>之外，还可以使用 std::is_constructible<>，它要求可以用显式转换来做初始化。但是需要注意的是，它的参数顺序和 std::is_convertible<>相反：

```
template<typename T>
using                          EnableIfString                          =
std::enable_if_t<std::is_constructible_v<std::string, T>>;
```

D3.2 节讨论了 std::is_constructible<>的使用细节，D3.3 节讨论了 std::is_convertible<>的使用细节。关于 enable_if<>在变参模板中的使用，请参见 D.6 节。

禁用某些成员函数

注意我们不能通过使用 enable_if<>来禁用 copy/move 构造函数以及赋值构造函数。这是因为成员函数模板不会被算作特殊成员函数（依然会生成默认构造函数），而且在需要使用 copy 构造函数的地方，相应的成员函数模板会被忽略掉。因此即使像下面这样定义类模板：

```
class C {
    public:
        template<typename T>
        C (T const&) {
            std::cout << "tmpl copy constructor\n";}
    ...
};
```

在需要 copy 构造函数的地方依然会使用预定义的 copy 构造函数：

```
C x;
C y{x}; // still uses the predefined copy constructor (not the member
template)
```

删掉 copy 构造函数也不行，因为这样在需要 copy 构造函数的地方会报错说该函数被删除了。

但是也有一个办法：可以定义一个接受 const volatile 的 copy 构造函数并将其标示为 delete。这样做就不会再隐式声明一个接受 const 参数的 copy 构造函数。在此基础上，可以定义一个构造函数模板，对于 nonvolatile 的类型，它会优选被选择（相较于已删除的 copy 构造函数）：

```
class C
{
    public:
        …
        // user-define the predefined copy constructor as deleted
        // (with conversion to volatile to enable better matches)
        C(C const volatile&) = delete;
        // implement copy constructor template with better match:
        template<typename T>
        C (T const&) {
            std::cout << "tmpl copy constructor\n";
        }
    …
};
```

这样即使对常规 copy，也会调用模板构造函数：

```
C x;
C y{x}; // uses the member template
```

于是就可以给这个模板构造函数添加 enable_if<>限制。比如可以禁止对通过 int 类型参数实例化出来的 C<>模板实例进行 copy：

```
template<typename T>
class C
{
    public:
        …
        // user-define the predefined copy constructor as deleted
        // (with conversion to volatile to enable better matches)
        C(C const volatile&) = delete;
        // if T is no integral type, provide copy constructor template
with better match:
        template<typename U,
        typename = std::enable_if_t<!std::is_integral<U>::value>>
        C (C<U> const&) {
```

```
            ...
        }
    ...
};
```

6.5 使用 concept 简化 enable_if<>表达式

即使使用了模板别名，enable_if 的语法依然显得很蠢，因为它使用了一个变通方法：为了达到目的，使用了一个额外的模板参数，并且通过“滥用”这个参数对模板的使用做了限制。这样的代码不容易读懂，也使模板中剩余的代码不易理解。

原则上我们所需要的只是一个能够对函数施加限制的语言特性，当这一限制不被满足的时候，函数会被忽略掉。

这个语言特性就是人们期盼已久的 concept，可以通过其简单的语法对函数模板施加限制条件。不幸的是，虽然已经讨论了很久，但是 concept 依然没有被纳入 C++17 标准。一些编译器目前对 concept 提供了试验性的支持，不过其很有可能在 C++17 之后的标准中得到支持（目前确定将在 C++20 中得到支持）。通过使用 concept 可以写出下面这样的代码：

```
template<typename STR>
requires std::is_convertible_v<STR,std::string>
Person(STR&& n) : name(std::forward<STR>(n)) {
    ...
}
```

甚至可以将其中模板的使用条件定义成通用的 concept：

```
template<typename T>
concept ConvertibleToString = std::is_convertible_v<T,std::string>;
```

然后将这个 concept 用作模板条件：

```
template<typename STR>
requires ConvertibleToString<STR>
Person(STR&& n) : name(std::forward<STR>(n)) {
    ...
}
```

也可以写成下面这样：

```
template<ConvertibleToString STR>
Person(STR&& n) : name(std::forward<STR>(n)) {
    ...
}
```

更多关于 C++ concept 的细节请参见附录 E。

6.6 总结

- 在模板中，可以通过使用“转发引用”（亦称“万能引用”，声明方式为模板参数 T 加&&）和 std::forward<>将模板调用参完美地数转发出去。
- 将完美转发用于成员函数模板时，在 copy 或者 move 对象的时候它们可能比预定义的特殊成员函数更匹配。
- 可以通过使用 std::enable_if<>并在其条件为 false 的时候禁用模板。
- 通过使用 std::enable_if<>，可以避免一些由于构造函数模板或者赋值构造函数模板比隐式产生的特殊构造函数更加匹配而带来的问题。
- 可以通过删除对 const volatile 类型参数预定义的特殊成员函数，并结合使用 std::enable_if<>，将特殊成员函数模板化。
- 通过 concept 可以使用更直观的语法对函数模板施加限制。

第 7 章 按值传递还是按引用传递？

从一开始，C++就提供了按值传递（call-by-value）和按引用传递（call-by-reference）两种参数传递方式，但是具体该怎么选择，有时并不容易确定：通常对复杂类型用按引用传递的成本更低，但是也更复杂。C++11 又引入了移动语义（move semantics），也就是说又多了一种按引用传递的方式：

1. X const &（const 左值引用）
   参数引用了被传递的对象，并且参数不能被更改。
2. X &（非 const 左值引用）
   参数引用了被传递的对象，但是参数可以被更改。
3. X &&（右值引用）
   参数通过移动语义引用了被传递的对象，并且参数值可以被更改或者被“窃取”。

仅仅对已知的具体类型，决定参数的方式就已经很复杂了。在参数类型未知的模板中，就更难选择合适的传递方式了。

不过在 1.6.1 节中，我们曾经建议在函数模板中应该优先使用按值传递，除非遇到以下情况：

- 对象不允许被 copy。
- 参数被用于返回数据。
- 参数以及其所有属性需要被模板转发到别的地方。
- 可以获得明显的性能提升。

本章将讨论模板中传递参数的几种方式，并将证明为何应该优先使用按值传递，也列举了不该使用按值传递的情况。同时讨论了在处理字符串常量和裸指针时遇到的问题。

在阅读本章的过程中，最好先够熟悉下附录 B 中和数值分类有关的一些术语（lvalue，rvalue，prvalue，xvalue）。

7.1 按值传递

当按值传递参数时，原则上所有的参数都会被拷贝。因此每一个参数都会是被传递实参的一份拷贝。对于 class 的对象，参数会通过 class 的拷贝构造函数来做初始化。

调用拷贝构造函数的成本可能很高。但是有多种方法可以避免按值传递的高昂成本：事实上编译器可以通过移动语义（move semantics）来优化掉对象的拷贝，这样即使是对复杂类型的拷贝，其成本也不会很高。

比如下面这个简单的按值传递参数的函数模板：

```
template<typename T>
void printV (T arg) {
```

```
    ...
}
```

当将该函数模板用于 int 类型参数时，实例化后的代码是：

```
void printV (int arg) {
    ...
}
```

参数 arg 变成任意实参的一份拷贝，不管实参是一个对象，一个常量还是一个函数的返回值。

如果定义一个 std::string 对象并将其用于上面的函数模板：

```
std::string s = "hi";
printV(s);
```

模板参数 T 被实例化为 std::string，实例化后的代码是：

```
void printV (std::string arg)
{
    ...
}
```

在传递字符串时，arg 变成 s 的一份拷贝。此时这一拷贝是通过 std::string 的拷贝构造函数创建的，这可能会是一个成本很高的操作，因为这个拷贝操作会对源对象做一次深拷贝，它需要开辟足够的内存来存储字符串的值。

但是并不是所有的情况都会调用拷贝构造函数。考虑如下情况：

```
std::string returnString();
std::string s = "hi";
printV(s); //copy constructor
printV(std::string("hi")); //copying usually optimized away (if not,
move constructor)
printV(returnString()); // copying usually optimized away (if not, move
constructor)
printV(std::move(s)); // move constructor
```

在第一次调用中，被传递的参数是左值（lvalue），因此拷贝构造函数会被调用。但是在第二和第三次调用中，被传递的参数是纯右值（prvalue，pure right value，临时对象或者某个函数的返回值，参见附录 B），此时编译器会优化参数传递，使得拷贝构造函数不会被调用。从 C++17 开始，C++标准要求这一优化方案必须被实现。在 C++17 之前，如果编译器没有优化掉这一类拷贝，它至少应该先尝试使用移动语义，这通常也会使拷贝成本变得比较低廉。在最后一次调用中，被传递参数是 xvalue（一个使用了 std::move()的已经存在的非 const 对象），这会通过告知编译器我们不在需要 s 的值来强制调用移动构造函数（move constructor）。

综上所述，在调用 printV()（参数是按值传递的）的时候，只有在被传递的参数是 lvalue（对象在函数调用之前创建，并且通常在之后还会被用到，而且没有对其使用 std::move()）时，

调用成本才会比较高。不幸的是，这唯一的情况也是最常见的情况，因为我们几乎总是先创建一个对象，然后在将其传递给其它函数。

按值传递会导致类型退化（decay）

关于按值传递，还有一个必须被讲到的特性：当按值传递参数时，参数类型会退化（decay）。也就是说，裸数组会退化成指针，const 和 volatile 等限制符会被删除（就像用一个值去初始化一个用 auto 声明的对象那样）：

```
template<typename T>
void printV (T arg) {
    ...
}

std::string const c = "hi";
printV(c); // c decays so that arg has type std::string
printV("hi"); //decays to pointer so that arg has type char const*
int arr[4];
printV(arr); // decays to pointer so that arg has type int *
```

当传递字符串常量“hi”的时候，其类型 char const[3]退化成 char const *，这也就是模板参数 T 被推断出来的类型。此时模板会被实例化成：

```
void printV (char const* arg)
{
    ...
}
```

这一行为继承自 C 语言，既有优点也有缺点。通常它会简化对被传递字符串常量的处理，但是缺点是在 printV()内部无法区分被传递的是一个对象的指针还是一个存储一组对象的数组。在 7.4 节将专门讨论如何应对字符串常量和裸数组的问题。

7.2 按引用传递

现在来讨论按引用传递。按引用传递不会拷贝对象（因为形参将引用被传递的实参）。而且，按引用传递时参数类型也不会退化（decay）。不过，并不是在所有情况下都能使用按引用传递，即使在能使用的地方，有时候被推断出来的模板参数类型也会带来不少问题。

7.2.1 按 const 引用传递

为了避免（不必要的）拷贝，在传递非临时对象作为参数时，可以使用 const 引用传递。比

如：

```
template<typename T>
void printR (T const& arg) {
    ...
}
```

这个模板永远不会拷贝被传递对象（不管拷贝成本是高还是低）：

```
std::string returnString();
std::string s = "hi";
printR(s); // no copy
printR(std::string("hi")); // no copy
printR(returnString()); // no copy
printR(std::move(s)); // no copy
```

即使是按引用传递一个 int 类型的变量，虽然这样可能会事与愿违（不会提高性能，见下段中的解释），也依然不会拷贝。因此如下调用：

```
int i = 42;
printR(i); // passes reference instead of just copying i
```

会将 printR()实例化为：

```
void printR(int const& arg) {
    ...
}
```

这样做之所以不能提高性能，是因为在底层实现上，按引用传递还是通过传递参数的地址实现的。地址会被简单编码，这样可以提高从调用者向被调用者传递地址的效率。不过按地址传递可能会使编译器在编译调用者的代码时有一些困惑：被调用者会怎么处理这个地址？理论上被调用者可以随意更改该地址指向的内容。这样编译器就要假设在这次调用之后，所有缓存在寄存器中的值可能都会变为无效。而重新载入这些变量的值可能会很耗时（可能比拷贝对象的成本高很多）。你或许会问在按 const 引用传递参数时：为什么编译器不能推断出被调用者不会改变参数的值？不幸的是，确实不能，因为调用者可能会通过它自己的非 const 引用修改被引用对象的值（这个解释太好，另一种情况是被调用者可以通过 const_cast 移除参数中的 const）。

不过对可以 inline 的函数，情况可能会好一些：如果编译器可以展开 inline 函数，那么它就可以基于调用者和被调用者的信息，推断出被传递地址中的值是否会被更改。函数模板通常总是很短，因此很可能会被做 inline 展开。但是如果模板中有复杂的算法逻辑，那么它大概率就不会被做 inline 展开了。

按引用传递不会做类型退化（decay）

按引用传递参数时，其类型不会退化（decay）。也就是说不会把裸数组转换为指针，也不

会移除 const 和 volatile 等限制符。而且由于调用参数被声明为 T const &，被推断出来的模板参数 T 的类型将不包含 const。比如：

```
template<typename T>
void printR (T const& arg) {
    ...
}
std::string const c = "hi";
printR(c); // T deduced as std::string, arg is std::string const&
printR("hi"); // T deduced as char[3], arg is char const(&)[3]
int arr[4];
printR(arr); // T deduced as int[4], arg is int const(&)[4]
```

因此对于在 printR()中用 T 声明的变量，它们的类型中也不会包含 const。

7.2.2 按非 const 引用传递

如果想通过调用参数来返回变量值（比如修改被传递变量的值），就需要使用非 const 引用（要么就使用指针）。同样这时候也不会拷贝被传递的参数。被调用的函数模板可以直接访问被传递的参数。

考虑如下情况：

```
template<typename T>
void outR (T& arg) {
    ...
}
```

注意对于 outR()，通常不允许将临时变量（prvalue）或者通过 std::move()处理过的已存在的变量（xvalue）用作其参数：

```
std::string returnString();
std::string s = "hi";
outR(s); //OK: T deduced as std::string, arg is std::string&
outR(std::string("hi")); //ERROR: not allowed to pass a temporary
(prvalue)
outR(returnString()); // ERROR: not allowed to pass a temporary
(prvalue)
outR(std::move(s)); // ERROR: not allowed to pass an xvalue
```

同样可以传递非 const 类型的裸数组，其类型也不会 decay：

```
int arr[4];
outR(arr); // OK: T deduced as int[4], arg is int(&)[4]
```

这样就可以修改数组中元素的值，也可以处理数组的长度。比如：

```
template<typename T>
```

```
void outR (T& arg) {
    if (std::is_array<T>::value) {
        std::cout << "got array of " << std::extent<T>::value << "
elems\n";
    }
    ...
}
```

但是在这里情况有一些复杂。此时如果传递的参数是 const 的，arg 的类型就有可能被推断为 const 引用，也就是说这时可以传递一个右值（rvalue）作为参数，但是模板所期望的参数类型却是左值（lvalue）：

```
std::string const c = "hi";
outR(c); // OK: T deduced as std::string const
outR(returnConstString()); // OK: same if returnConstString() returns
const string
outR(std::move(c)); // OK: T deduced as std::string const6
outR("hi"); // OK: T deduced as char const[3]
```

在这种情况下，在函数模板内部，任何试图更改被传递参数的值的行为都是错误的。在调用表达式中也可以传递一个 const 对象，但是当函数被充分实例化之后（可能发生在接接下来的编译过程中），任何试图更改参数值的行为都会触发错误（但是这有可能发生在被调用模板的很深层次逻辑中，具体细节请参见 9.4 节）。

如果想禁止想非 const 应用传递 const 对象，有如下选择：

- 使用 static_assert 触发一个编译期错误：

```
template<typename T>
void outR (T& arg) {
static_assert(!std::is_const<T>::value, "out parameter of foo<T>(T&)
is const");
    ...
}
```

- 通过使用 std::enable_if<>（参见 6.3 节）禁用该情况下的模板：

```
template<typename T,
typename = std::enable_if_t<!std::is_const<T>::value>
void outR (T& arg) {
    ...
}
```

或者是在 concepts 被支持之后，通过 concepts 来禁用该模板（参见 6.5 节以及附录 E）：

```
template<typename T>
requires !std::is_const_v<T>
void outR (T& arg) {
    ...
```

```
}
```

7.2.3 按转发引用传递参数（Forwarding Reference）

使用引用调用（call-by-reference）的一个原因是可以对参数进行完美转发（perfect forward）（参见 6.1 节）。但是请记住在使用转发引用时（forwarding reference，被定义成一个模板参数的右值引用（rvalue reference）），有它自己特殊的规则。

考虑如下代码：

```
template<typename T>
void passR (T&& arg) { // arg declared as forwarding reference
    ...
}
```

可以将任意类型的参数传递给转发引用，而且和往常的按引用传递一样，都不会创建被传递参数的备份：

```
std::string s = "hi";
passR(s); // OK: T deduced as std::string& (also the type of arg)
passR(std::string("hi")); // OK: T deduced as std::string, arg is
std::string&&
passR(returnString()); // OK: T deduced as std::string, arg is
std::string&&
passR(std::move(s)); // OK: T deduced as std::string, arg is
std::string&&
passR(arr); // OK: T deduced as int(&)[4] (also the type of arg)
```

但是，这种情况下类型推断的特殊规则可能会导致意想不到的结果：

```
std::string const c = "hi";
passR(c); //OK: T deduced as std::string const&
passR("hi"); //OK: T deduced as char const(&)[3] (also the type of arg)
int arr[4];
passR(arr); //OK: T deduced as int (&)[4] (also the type of arg)
```

在以上三种情况中，都可以在 passR()内部从 arg 的类型得知被传递的参数是一个右值（rvalue）还是一个 const 或者非 const 的***左值***（lvalue）。这是唯一一种可以传递一个参数，并用它来区分以上三种情况的方法。

看上去将一个参数声明为转发引用总是完美的。但是，没有免费的午餐。

比如，由于转发引用是唯一一种可以将模板参数 T 隐式推断为引用的情况，此时如果在模板内部直接用 T 声明一个未初始化的局部变量，就会触发一个错误（引用对象在创建的时候必须被初始化）：

```
template<typename T>
void passR(T&& arg) { // arg is a forwarding reference
```

```
    T x; // for passed lvalues, x is a reference, which requires an initializer
        ...
    }
    foo(42); // OK: T deduced as int
    int i;
    foo(i); // ERROR: T deduced as int&, which makes the declaration of x
    in passR() invalid
```

关于处理这一情况的更多细节，请参见 15.6.2 节。

7.3 使用 std::ref()和 std::cref() （限于模板）

从 C++11 开始，可以让调用者自行决定向函数模板传递参数的方式。如果模板参数被声明成按值传递的，调用者可以使用定义在头文件<functional>中的 std::ref()和 std::cref()将参数按引用传递给函数模板。比如：

```
template<typename T>
void printT (T arg) {
    ...
}
std::string s = "hello";
printT(s); //pass s By value
printT(std::cref(s)); // pass s “as if by reference”
```

但是请注意，std::cref()并没有改变函数模板内部处理参数的方式。相反，在这里它使用了一个技巧：它用一个行为和引用类似的对象对参数进行了封装。事实上，它创建了一个 std::reference_wrapper<>的对象，该对象引用了原始参数，并被按值传递给了函数模板。Std::reference_wrapper<>可能只支持一个操作：向原始类型的隐式类型转换，该转换返回原始参数对象。因此当需要操作被传递对象时，都可以直接使用这个 std::reference_wrapper<>对象。比如：

```
#include <functional> // for std::cref()
#include <string>
#include <iostream>
void printString(std::string const& s)
{
    std::cout << s << '\n';
}
template<typename T>
void printT (T arg)
{
    printString(arg); // might convert arg back to std::string
}

int main()
```

按引用传递

输出字符串对象s

```
{
    std::string s = "hello";
    printT(s); // print s passed by value
    printT(std::cref(s)); // print s passed “as if by reference”
}
```

最后一个调用将一个 std::reference_wrapper<string const>对象按值传递给参数 arg，这样 std::reference_wrapper<string const>对象被传进函数模板并被转换为原始参数类型 std::string。

注意，编译器必须知道需要将 std::reference_wrapper<string const>对象转换为原始参数类型，才会进行隐式转换。因此 std::ref()和 std::cref()通常只有在通过泛型代码传递对象时才能正常工作。比如如果尝试直接输出传递进来的类型为 T 的对象，就会遇到错误，因为 std::reference_wrapper<string const>中并没有定义输出运算符：

```
template<typename T>
void printV (T arg) {
    std::cout << arg << ' \n' ;
}
…
std::string s = "hello";
printV(s); //OK
printV(std::cref(s)); // ERROR: no operator << for reference wrapper
defined
```

输出T类型(未必是字符串)对象s

同样下面的代码也会报错，因为不能将一个 std::reference_wrapper<string const>对象和一个 char const*或者 std::string 进行比较：

```
template<typename T1, typename T2>
bool isless(T1 arg1, T2 arg2)
{
    return arg1 < arg2;
}
…
std::string s = "hello";
if (isless(std::cref(s), "world")) … //ERROR
if (isless(std::cref(s), std::string("world"))) … //ERROR
```

此时即使让 arg1 和 arg2 使用相同的模板参数 T，也不会有帮助：

```
template<typename T>
bool isless(T arg1, T arg2)
{
    return arg1 < arg2;
}
```

因为编译器在推断 arg1 和 arg2 的类型时会遇到类型冲突。

综上，std::reference_wrapper<>是为了让开发者能够像使用“第一类对象（first class object）”一样使用引用，可以对它进行拷贝并将其按值传递给函数模板。也可以将它用在 class 内部，比如让它持有一个指向容器中对象的引用。但是通常总是要将其转换会原始类型。

7.4 处理字符串常量和裸数组

到目前为止，我们看到了将字符串常量和裸数组用作模板参数时的不同效果：

- 按值传递时参数类型会 decay，参数类型会退化成指向其元素类型的指针。
- 按引用传递是参数类型不会 decay，参数类型是指向数组的引用。

两种情况各有其优缺点。将数组退化成指针，就不能区分它是指向对象的指针还是一个被传递进来的数组。另一方面，如果传递进来的是字符串常量，那么类型不退化的话就会带来问题，因为不同长度的字符串的类型是不同的。比如：

```
template<typename T>
void foo (T const& arg1, T const& arg2)
{
    ...
}
foo("hi", "guy"); //ERROR
```

这里 foo(“hi”, “guy”)不能通过编译，因为”hi”的类型是 char const [3]，而”guy”的类型是 char const [4]，但是函数模板要求两个参数的类型必须相同。这种 code 只有在两个字符串常量的长度相同时才能通过编译。因此，强烈建议在测试代码中使用长度不同的字符串。

如果将 foo()声明成按值传递的，这种调用可能可以正常运行：

```
template<typename T>
void foo (T arg1, T arg2)
{
    ...
}
foo("hi", "guy"); //compiles, but ···
```

但是这样并不能解决所有的问题。反而可能会更糟，编译期间的问题可能会变为运行期间的问题。考虑如下代码，它用==运算符比较两个传进来的参数：

```
template<typename T>
void foo (T arg1, T arg2)
{
    if (arg1 == arg2) { //OOPS: compares addresses of passed arrays
    ...
}
}
 foo("hi", "guy"); //compiles, but ···// 比较的是两者的指针地址
```

虽然可以编译，但更糟的是，编译期问题可能变成运行期问题，比如比较两个参数时，实际比较的是指针地址，而非指向的内容

如上，此时很容易就能知道需要将被传递进来的的字符指针理解成字符串。但是情况并不总是这么简单，因为模板还要处理类型可能已经退化过了的字符串常量参数（比如它们可能来自另一个按值传递的函数，或者对象是通过 auto 声明的）。

然而，退化在很多情况下是有帮助的，尤其是在需要验证两个对象（两个对象都是参数，或者一个对象是参数，并用它给另一个赋值）是否有相同的类型或者可以转换成相同的类型的时候。这种情况的一个典型应用就是用于完美转发（perfect forwarding）。但是使用完美转发需要将参数声明为转发引用。这时候就需要使用类型萃取 std::decay<>()显式的退化参数类型。可以参考 7.6 节 std::make_pair()这个例子。

注意，有些类型萃取本身可能就会对类型进行隐式退化，比如用来返回两个参数的公共类型的 std::common_type<>（请参见 1.3.3 节以及 D.5）。

7.4.1 关于字符串常量和裸数组的特殊实现

有时候可能必须要对数组参数和指针参数做不同的实现。此时当然不能退化数组的类型。

有时可能需要对数组参数和指针参数做不同的实现。当然，这要求传入的数组未发生退化。

为了区分这两种情况，必须要检测到被传递进来的参数是不是数组。通常有两种方法：

- 可以将模板定义成只能接受数组作为参数：

检查传入的是不是一个数组

```
template<typename T, std::size_t L1, std::size_t L2>
void foo(T (&arg1)[L1], T (&arg2)[L2])
{
    T* pa = arg1; // decay arg1
    T* pb = arg2; // decay arg2
    if (compareArrays(pa, L1, pb, L2)) {
        ...
    }
}
```

数组的引用，数组元素的类型为T

参数 arg1 和 arg2 必须是元素类型相同、长度可以不同的两个数组。但是为了支持多种不同类型的裸数组，可能需要更多实现方式（参见 5.4 节）。

- 可以使用类型萃取来检测参数是不是一个数组：

```
template<typename T, typename =
std::enable_if_t<std::is_array_v<T>>>
void foo (T&& arg1, T&& arg2)
{
    ...
}
```

由于这些特殊的处理方式过于复杂，最好还是使用一个不同的函数名来专门处理数组参数。或者更近一步，让模板调用者使用 std::vector 或者 std::array 作为参数。但是只要字符串还

是裸数组，就必须对它们进行单独考虑。

7.5 处理返回值

返回值也可以被按引用或者按值返回。但是按引用返回可能会带来一些麻烦，因为它所引用的对象不能被很好的控制。不过在日常编程中，也有一些情况更倾向于按引用返回：

- 返回容器或者字符串中的元素（比如通过[]运算符或者 front()方法访问元素）
- 允许修改类对象的成员
- 为链式调用返回一个对象（比如>>和<<运算符以及赋值运算符）

另外对成员的只读访问，通常也通过返回 const 引用实现。

但是如果使用不当，以上几种情况就可能导致一些问题。比如：

```
std::string* s = new std::string("whatever");
auto& c = (*s)[0];
delete s;
std::cout << c; //run-time ERROR
```

这里声明了一个指向字符串中元素的引用，但是在使用这个引用的地方，对应的字符串却不存在了（成了一个悬空引用），这将导致未定义的行为。这个例子看上去像是人为制造的（一个有经验的程序员应该可以意识到这个问题），但是情况也不都是这么明显。比如：

```
auto s = std::make_shared<std::string>("whatever");
auto& c = (*s)[0];
s.reset();
std::cout << c; //run-time ERROR
```

We should therefore ensure that function templates return their result by value.
因此，我们应该确保函数模板按值返回结果。

因此需要确保函数模板采用按值返回的方式。但是正如接下来要讨论的，使用函数模板 T 作为返回类型并不能保证返回值不会是引用，因为 T 在某些情况下会被隐式推断为引用类型：

```
template<typename T>
T retR(T&& p) // p is a forwarding reference
{
    return T{…}; // OOPS: returns by reference when called for lvalues
}
```

即使函数模板被声明为按值传递，也可以显式地将 T 指定为引用类型：

```
template<typename T>
T retV(T p) //Note: T might become a reference
{
    return T{…}; // OOPS: returns a reference if T is a reference
}
int x;
retV<int&>(x); // retT() instantiated for T as int&
```

安全起见，有两种选择：

- 用类型萃取 std::remove_reference<>（参见 D.4 节）将 T 转为非引用类型：

```
template<typename T>
typename std::remove_reference<T>::type retV(T p)
{
    return T{…}; // always returns by value
}
```

Std::decay<>（参见 D.4 节）之类的类型萃取可能也会有帮助，因为它们也会隐式的去掉类型的引用。

- 将返回类型声明为 auto，从而让编译器去推断返回类型，这是因为 auto 也会导致类型退化：

```
template<typename T>
auto retV(T p) // by-value return type deduced by compiler
{
    return T{…}; // always returns by value
}
```

7.6 关于模板参数声明的推荐方法

正如前几节介绍的那样，函数模板有多种传递参数的方式：

- 将参数声明成按值传递：

这一方法很简单，它会对字符串常量和裸数组的类型进行退化，但是对比较大的对象可能会受影响性能。在这种情况下，调用者仍然可以通过 std::cref()和 std::ref()按引用传递参数，但是要确保这一用法是有效的。

- 将参数声明成按引用传递：

对于比较大的对象这一方法能够提供比较好的性能。尤其是在下面几种情况下：

  - 将已经存在的对象（lvalue）按照左值引用传递，
  - 将临时对象（prvalue）或者被 std::move()转换为可移动的对象（xvalue）按右值引用传递，
  - 或者是将以上几种类型的对象按照转发引用传递。

由于这几种情况下参数类型都不会退化，因此在传递字符串常量和裸数组时要格外小心。对于转发引用，需要意识到模板参数可能会被隐式推断为引用类型（引用折叠）。

一般性建议

基于以上介绍，对于函数模板有如下建议：

1. 默认情况下，将参数声明为按值传递。这样做比较简单，即使对字符串常量也可以正常工作。对于比较小的对象、临时对象以及可移动对象，其性能也还不错。对于比较大的

对象，为了避免成本高昂的拷贝，可以使用 std::ref()和 std::cref()。

2. 如果有充分的理由，也可以不这么做：
   - 如果需要一个参数用于输出，或者即用于输入也用于输出，那么就将这个参数按非 const 引用传递。但是需要按照 7.2.2 节介绍的方法禁止其接受 const 对象。
   - 如果使用模板是为了转发它的参数，那么就使用完美转发（perfect forwarding）。也就是将参数声明为转发引用并在合适的地方使用 std::forward<>()。考虑使用 std::decay<>或者 std::common_type<>来处理不同的字符串常量类型以及裸数组类型的情况。
   - 如果重点考虑程序性能，而参数拷贝的成本又很高，那么就使用 const 引用。不过如果最终还是要对对象进行局部拷贝的话，这一条建议不适用。

3. 如果你更了解程序的情况，可以不遵循这些建议。但是请不要仅凭直觉对性能做评估。在这方面即使是程序专家也会犯错。真正可靠的是：测试结果。

不要过分泛型化

值得注意的是，在实际应用中，函数模板通常并不是为了所有可能的类型定义的。而是有一定的限制。比如你可能已经知道函数模板的参数只会是某些类型的 vector。这时候最好不要将该函数模板定义的过于泛型化，否则，可能会有一些令人意外的副作用。针对这种情况应该使用如下的方式定义模板：

```
template<typename T>
void printVector (std::vector<T> const& v)
{
    ...
}
```

这里通过的参数 v，可以确保 T 不会是引用类型，因为 vector 不能用引用作为其元素类型。而且将 vector 类型的参数声明为按值传递不会有什么好处，因为按值传递一个 vector 的成本明显会比较高昂（vector 的拷贝构造函数会拷贝 vector 中的所有元素）。此处如果直接将参数 v 的类型声明为 T，就不容易从函数模板的声明上看出该使用那种传递方式了。

以 std::make_pair<>为例

Std::make_pair<>()是一个很好的介绍参数传递机制相关陷阱的例子。使用它可以很方便的通过类型推断创建 std::pair<>对象。它的定义在各个版本的 C++中都不一样：

- 在第一版 C++标准 C++98 中，std::make_pair<>被定义在 std 命名空间中，并且使用按引用传递来避免不必要的拷贝：

```
template<typename T1, typename T2>
pair<T1,T2> make_pair (T1 const& a, T2 const& b)
{
    return pair<T1,T2>(a,b);
```

```
}
```

但是当使用 std::pair<>存储不同长度的字符串常量或者裸数组时，这样做会导致严重的问题。

- 因此在 C++03 中，该函数模板被定义成按值传递参数：

```
template<typename T1, typename T2>
pair<T1,T2> make_pair (T1 a, T2 b)
{
    return pair<T1,T2>(a,b);
}
```

正如你可以在 the rationale for the issue resolution 中读到的那样：看上去也这一方案对标准库的变化比其它两种建议都要小，而且其优点足以弥补它对性能造成的不利影响。

- 不过在 C++11 中，由于 make_pair<>()需要支持移动语义，就必须使用转发引用。因此，其定义大体上是这样：

```
template<typename T1, typename T2>
constexpr pair<typename decay<T1>::type, typename
decay<T2>::type>
make_pair (T1&& a, T2&& b)
{
    return pair<typename decay<T1>::type, typename
decay<T2>::type>(forward<T1>(a), forward<T2>(b));
}
```

完整的实现还要复杂的多：为了支持 std::ref() 和 std::cref()，该函数会将 std::reference_wrapper 展开成真正的引用。

目前 C++标准库在很多地方都使用了类似的方法对参数进行完美转发，而且通常都会结合 std::decay<>使用。

7.7 总结

- 最好使用不同长度的字符串常量对模板进行测试。
- 模板参数的类型在按值传递时会退化，按引用传递则不会。
- 可以使用 std::decay<>对按引用传递的模板参数的类型进行退化。
- 在某些情况下，对被声明成按值传递的函数模板，可以使用 std::cref()和 std::ref()将参数按引用进行传递。
- 按值传递模板参数的优点是简单，但是可能不会带来最好的性能。
- 除非有更好的理由，否则就将模板参数按值传递。
- 对于返回值，请确保按值返回（这也意味着某些情况下不能直接将模板参数直接用于返回类型）。

- 在比较关注性能时，做决定之前最好进行实际测试。不要相信直觉，它通常都不准确。

第 8 章 编译期编程

C++一直以来都包含一些可以被用来进行编译器计算的简单方法。模板则进一步增加了编译器计算的可能性，而且该语言进一步的发展通常也都是在这一工具箱里进行的。

比较简单的情况是，可以通过它来决定是否启用某个模板，或者在多个模板之间做选择。不过如果有足够多的信息，编译器甚至可以计算控制流的结果。

事实上，C++有很多可以支持编译期编程的特性：

- 从 C++98 开始，模板就有了编译期计算的能力，包括使用循环以及执行路径选择（然而有些人认为这是对模板特性的滥用，因为其语法不够直观）。
- 基于某些限制和要求，在编译期间，可以通过部分特例化在类模板的不同实现之间做选择。
- 通过 SFINAE（替换错误不算失败），可以基于不同的类型或者限制条件，在函数模板的不同实现方式之间做选择。
- 在 C++11 和 C++14 中，由于可以在 constexpr 中使用更直观的执行路径选择方法（从 C++14 开始，更多的语句得到支持，比如 for 循环，switch 语句等），编译期计算得到了更好的支持。
- C++17 则引入了编译期 if（compile-time if），通过它可以基于某些编译期的条件或限制弃用某些语句。它甚至可以用非模板函数。

本章将重点介绍这些特性在模板及其相关内容中的应用。

8.1 模板元编程

模板的实例化发生在编译期间（而动态语言的泛型是在程序运行期间决定的）。事实证明 C++模板的某些特性可以和实例化过程相结合，这样就产生了一种 C++自己内部的原始递归的“编程语言”。因此模板可以用来“计算一个程序的结果”。第 23 章会对这些特性进行全面介绍，这里通过一个简单的例子来展示它们的用处。

下面的代码在编译期间就能判断一个数是不是质数：

```
template<unsigned p, unsigned d> // p: number to check, d: current
divisor
struct DoIsPrime {
    static constexpr bool value = (p%d != 0) && DoIsPrime<p,d-1>::value;
};
template<unsigned p> // end recursion if divisor is 2
struct DoIsPrime<p,2> {
    static constexpr bool value = (p%2 != 0);
};
```

```
template<unsigned p> // primary template
struct IsPrime {
    // start recursion with divisor from p/2:
    static constexpr bool value = DoIsPrime<p,p/2>::value;
};
// special cases (to avoid endless recursion with template
instantiation):
template<>
struct IsPrime<0> { static constexpr bool value = false; };
template<>
struct IsPrime<1> { static constexpr bool value = false; };
template<>
struct IsPrime<2> { static constexpr bool value = true; };
template<>
struct IsPrime<3> { static constexpr bool value = true; };
```

IsPrime<>模板将结果存储在其成员 value 中。为了计算出模板参数是不是质数，它实例化了 DoIsPrime<>模板，这个模板会被递归展开，以计算 p 除以 p/2 和 2 之间的数之后是否会有余数。

比如，表达式：

```
IsPrime<9>::value
```

首先展开成：

```
DoIsPrime<9,4>::value
```

然后继续展开成：

```
9%4!=0 && DoIsPrime<9,3>::value
```

然后继续展开成：

```
9%4!=0 && 9%3!=0 && DoIsPrime<9,2>::value
```

然后继续展开成：

```
9%4!=0 && 9%3!=0 && 9%2!=0
```

由于 9%3 == 0，因此它将返回 false；

正如以上实例化过程展现的那样：

- 我们通过递归地展开 DoIsPrime<>来遍历所有介于 p/2 和 2 之间的数，以检查是否有某个数可以被 p 整除。
- 用 d 等于 2 偏特例化出来的 DoIsPrime<>被用于终止递归调用。

但是以上过程都是在编译期间进行的。也就是说：

```
IsPrime<9>::value
```

在编译期间就被扩展成 false 了。

上面展示的模板语法可以说是笨拙的，不过类似的代码从 C++98（以及更早的版本）开始就可以正常工作了，而且被证明对一些库的开发也有帮助。

更多细节请参见第 23 章。

8.2 通过 constexpr 进行计算

C++11 引入了一个叫做 constexpr 的新特性，它大大简化了各种类型的编译期计算。如果给定了合适的输入，constexpr 函数就可以在编译期间完成相应的计算。虽然 C++11 对 constexpr 函数的使用有诸多限制（比如 constexpt 函数的定义通常都只能包含一个 return 语句），但是在 C++14 中这些限制中的大部分都被移除了。当然，为了能够成功地进行 constexpr 函数中的计算，依然要求各个计算步骤都能在编译期进行：目前堆内存分配和异常抛出都不被支持。

在 C++11 中，判断一个数是不是质数的实现方式如下：

```
constexpr bool
doIsPrime (unsigned p, unsigned d) // p: number to check, d: current
divisor
{
    return d!=2 ? (p%d!=0) && doIsPrime(p,d-1) // check this and smaller
divisors
           : (p%2!=0); // end recursion if divisor is 2
}
constexpr bool isPrime (unsigned p)
{
    return p < 4 ? !(p<2) // handle special cases
           :   doIsPrime(p,p/2); // start recursion with divisor from
p/2
}
```

为了满足 C++11 中只能有一条语句的要求，此处只能使用条件运算符来进行条件选择。不过由于这个函数只用到了 C++的常规语法，因此它比第一版中，依赖于模板实例化的代码要容易理解的多。

在 C++14 中，constexpr 函数可以使用常规 C++代码中大部分的控制结构。因此为了判断一个数是不是质数，可以不再使用笨拙的模板方式（C++11 之前）以及略显神秘的单行代码方式（C++11），而直接使用一个简单的 for 循环：

```
constexpr bool isPrime (unsigned int p)
{
    for (unsigned int d=2; d<=p/2; ++d) {
```

```
        if (p % d == 0) {
            return false; // found divisor without remainder}
    }
    return p > 1; // no divisor without remainder found
}
```

在 C++11 和 C++14 中实现的 constexpr isPrime()，都可以通过直接调用：

```
isPrime(9)
```

来判断 9 是不是一个质数。但是上面所说的“可以”在编译期执行，并不是一定会在编译期执行。在需要编译期数值的上下文中（比如数组的长度和非类型模板参数），编译器会尝试在编译期对被调用的 constexpr 函数进行计算，此时如果无法在编译期进行计算，就会报错（因为此处必须要产生一个常量）。在其他上下文中，编译期可能会也可能不会尝试进行编译期计算，如果在编译期尝试了，但是现有条件不满足编译期计算的要求，那么也不会报错，相应的函数调用被推迟到运行期间执行。

比如：

```
constexpr bool b1 = isPrime(9); // evaluated at compile time
```

会在编译期进行计算（因为 b1 被 constexpr 修饰）。而对

```
const bool b2 = isPrime(9); // evaluated at compile time if in namespace
scope
```

如果 b2 被定义于全局作用域或者 namespace 作用域，也会在编译期进行计算。如果 b2 被定义于块作用域（{}内），那么将由编译器决定是否在编译期间进行计算。下面这个例子就属于这种情况：

```
bool fiftySevenIsPrime() {
    return isPrime(57); // evaluated at compile or running time
}
```

此时是否进行编译期计算将由编译期决定。

另一方面，在如下调用中：

```
int x;
…
std::cout << isPrime(x); // evaluated at run time
```

不管 x 是不是质数，调用都只会在运行期间执行。

8.3 通过部分特例化进行路径选择

诸如 isPrime()这种在编译期进行相关测试的功能，有一个有意思的应用场景：可以在编译期间通过部分特例化在不同的实现方案之间做选择。

比如，可以以一个非类型模板参数是不是质数为条件，在不同的模板之间做选择：

```
// primary helper template:
template<int SZ, bool = isPrime(SZ)>
struct Helper;
// implementation if SZ is not a prime number:
template<int SZ>
struct Helper<SZ, false>
{
    ...
};
// implementation if SZ is a prime number:
template<int SZ>
struct Helper<SZ, true>
{
    ...
};
template<typename T, std::size_t SZ>
long foo (std::array<T,SZ> const& coll)
{
    Helper<SZ> h; // implementation depends on whether array has prime
number as size
    ...
}
```

这里根据参数 std::array<>的 size 是不是一个质数，实现了两种 Helper<>模板。这一偏特例化的使用方法，被广泛用于基于模板参数属性，在不同模板实现方案之间做选择。

在上面的例子中，对两种可能的情况实现了两种偏特例化版本。但是也可以将主模板用于其中一种情况，然后再特例化一个版本代表另一种情况：

```
// primary helper template (used if no specialization fits):
template<int SZ, bool = isPrime(SZ)>
struct Helper
{
    ...
};
// special implementation if SZ is a prime number:
template<int SZ>
struct Helper<SZ, true>
{
    ...
};
```

由于函数模板不支持部分特例化，当基于一些限制在不同的函数实现之间做选择时，必须要

使用其它一些方法：

- 使用有 static 函数的类，
- 使用 6.3 节中介绍的 std::enable_if，
- 使用下一节将要介绍的 SFINAE 特性，
- 或者使用从 C++17 开始生效的编译期的 if 特性，这部分内容会在 8.5 节进行介绍。

第 20 章介绍了基于限制条件，在不同的函数实现之间做选择的相关技术。

8.4 SFINAE (Substitution Failure Is Not An Error，替换失败不是错误)

在 C++中，重载函数以支持不同类型的参数是很常规的操作。当编译器遇到一个重载函数的调用时，它必须分别考虑每一个重载版本，以选择其中类型最匹配的那一个（更多相关细节请参见附录 C）。

在一个函数调用的备选方案中包含函数模板时，编译器首先要决定应该将什么样的模板参数用于各种模板方案，然后用这些参数替换函数模板的参数列表以及返回类型，最后评估替换后的函数模板和这个调用的匹配情况（就像常规函数一样）。但是这一替换过程可能会遇到问题：替换产生的结果可能没有意义。不过这一类型的替换不会导致错误，C++语言规则要求忽略掉这一类型的替换结果。

这一原理被称为 SFINAE（发音类似 sfee-nay），代表的是“substitution failure is not an error”。

但是上面讲到的替换过程和实际的实例化过程不一样（参见 2.2 节）：即使对那些最终被证明不需要被实例化的模板也要进行替换（不然就无法知道到底需不需要实例化）。不过它只会替换直接出现在函数模板声明中的相关内容（不包含函数体）。

考虑如下的例子：

```
// number of elements in a raw array:
template<typename T, unsigned N>
std::size_t len (T(&)[N])
{
    return N;
}
// number of elements for a type having size_type:
template<typename T>
typename T::size_type len (T const& t)
{
    return t.size();
}
```

这里定义了两个接受一个泛型参数的函数模板 len()：

1. 第一个函数模板的参数类型是 T (&)[N]，也就是说它是一个包含了 N 个 T 型元素的数组。
2. 第二个函数模板的参数类型就是简单的 T，除了返回类型要是 T::size_type 之外没有别的限制，这要求被传递的参数类型必须有一个 size_type 成员。

当传递的参数是裸数组或者字符串常量时，只有那个为裸数组定义的函数模板能够匹配：

```
int a[10];
std::cout << len(a); // OK: only len() for array matches
std::cout << len("tmp"); //OK: only len() for array matches
```

如果只是从函数签名来看的话，对第二个函数模板也可以分别用 int[10]和 char const [4]替换类型参数 T，但是这种替换在处理返回类型 T::size_type 时会导致错误。因此对于这两个调用，第二个函数模板会被忽略掉。

如果传递 std::vector<>作为参数的话，则只有第二个模板参数能够匹配：

```
std::vector<int> v;
std::cout << len(v); // OK: only len() for a type with size_type matches
```

如果传递的是裸指针话，以上两个模板都不会被匹配上（但是不会因此而报错）。此时编译期会抱怨说没有发现合适的 len()函数：

```
int* p;
std::cout << len(p); // ERROR: no matching len() function found
```

但是这和传递一个有 size_type 成员但是没有 size()成员函数的情况不一样。比如如果传递的参数是 std::allocator<>:

```
std::allocator<int> x;
std::cout << len(x); // ERROR: len() function found, but can't size()
```

此时编译器会匹配到第二个函数模板。因此不会报错说没有发现合适的 len()函数，而是会报一个编译期错误说对 std::allocator<int>而言 size()是一个无效调用。此时第二个模板函数不会被忽略掉。

如果忽略掉那些在替换之后返回值类型为无效的备选项，那么编译器会选择另外一个参数类型匹配相差的备选项。比如：

```
// number of elements in a raw array:
template<typename T, unsigned N>
std::size_t len (T(&)[N])
{
    return N;
}
// number of elements for a type having size_type:
template<typename T>
typename T::size_type len (T const& t)
{
    return t.size();
```

```
  }
 // 对所有类型的应急选项：
std::size_t len (…)
{
    return 0;
}
```

此处额外提供了一个通用函数 len()，它总会匹配所有的调用，但是其匹配情况也总是所有重载选项中最差的（通过省略号...匹配）（参见 C.2）。

此时对于裸数组和 vector，都有两个函数可以匹配上，但是其中不是通过省略号（...）匹配的那一个是最佳匹配。对于指针，只有应急选项能够匹配上，此时编译器不会再报缺少适用于本次调用的 len()。不过对于 std::allocator<int>的调用，虽然第二个和第三个函数都能匹配上，但是第二个函数依然是最佳匹配项。因此编译器依然会报错说缺少 size()成员函数：

```
int a[10];
std::cout << len(a); // OK: len() for array is best match
std::cout << len("tmp"); //OK: len() for array is best match
std::vector<int> v;
std::cout << len(v); // OK: len() for a type with size_type is best match
int* p;
std::cout << len(p); // OK: only fallback len() matches
std::allocator<int> x;
std::cout << len(x); // ERROR: 2nd len() function matches best, but can't call size() for x
```

请参见 15.7 节中更多关于 SFINAE 的内容，以及 19.4 节中一些 SFINAE 的应用示例。

SFINAE and Overload Resolution

随着时间的推移，SFINAE 原理在模板开发者中变得越来越重要、越来越流行，以至于这个缩写常常被当作一个动词使用。当我们说“我们 SFINAE 掉了一个函数”时，意思是我们通过让模板在一些限制条件下产生无效代码，从而确保在这些条件下会忽略掉该模板。当你在 C++标准里读到“除非在某些情况下，该模板不应该参与重载解析过程”时，它的意思就是“在该情况下，使用 SFINAE 方法 SFINAE 掉了这个函数模板”。

比如 std::thread 类模板声明了如下构造函数：

```
namespace std {
    class thread {
        public:
            …
            template<typename F, typename… Args>
            explicit thread(F&& f, Args&&… args);
            …
```

```
    };
}
```

并做了如下备注：

备注：如果 decay_t<F>的类型和 std:thread 相同的话，该构造函数不应该参与重载解析过程。

它的意思是如果在调用该构造函数模板时，使用 std::thread 作为第一个也是唯一一个参数的话，那么这个构造函数模板就会被忽略掉。这是因为一个类似的成员函数模板在某些情况下可能比预定义的 copy 或者 move 构造函数更能匹配相关调用（相关细节请参见 6.2 节以及 16.2.4 节）。通过 SFINAE 掉将该构造函数模板用于 thread 的情况，就可以确保在用一个 thread 构造另一个 thread 的时候总是会调用预定义的 copy 或者 move 构造函数。

但是使用该技术逐项禁用相关模板是不明智的。幸运的是标准库提供了更简单的禁用模板的方法。其中最广为人知的一个就是在 6.3 节介绍的 std::enable_if<>。

因此典型的 std::thread 的实现如下：

```
namespace std {
    class thread {
        public:
            …
            template<typename F, typename… Args,
            typename =
            std::enable_if_t<!std::is_same_v<std::decay_t<F>,
thread>>>
            explicit thread(F&& f, Args&&… args);
            …
    };
}
```

关于 std:enable_if<>的实现请参见 20.3 节，它使用了部分特例化以及 SFINAE。

8.4.1 通过 decltype 进行 SFINAE（此处是动词）的表达式

对于有些限制条件，并不总是很容易地就能找到并设计出合适的表达式来 SFINAE 掉函数模板。

比如，对于有 size_type 成员但是没有 size()成员函数的参数类型，我们想要保证会忽略掉函数模板 len()。如果没有在函数声明中以某种方式要求 size()成员函数必须存在，这个函数模板就会被选择并在实例化过程中导致错误：

```
template<typename T>
typename T::size_type len (T const& t)
{
```

```
    return t.size();
}
std::allocator<int> x;
std::cout << len(x) << '\n'; //ERROR: len() selected, but x has no size()
```

处理这一情况有一种常用模式或者说习惯用法：

- 通过尾置返回类型语法（trailing return type syntax）来指定返回类型（在函数名前使用 auto，并在函数名后面的->后指定返回类型）。
- 通过 decltype 和逗号运算符定义返回类型。
- 将所有需要成立的表达式放在逗号运算符的前面（为了预防可能会发生的运算符被重载的情况，需要将这些表达式的类型转换为 void）。
- 在逗号运算符的末尾定义一个类型为返回类型的对象。

比如：

```
template<typename T>
auto len (T const& t) -> decltype( (void)(t.size()),
T::size_type() )
{
    return t.size();
}
```

这里返回类型被定义成：

```
decltype( (void)(t.size)(), T::size_type() )
```

类型指示符 decltype 的操作数是一组用逗号隔开的表达式，因此最后一个表达式 T::size_type() 会产生一个类型为返回类型的对象（decltype 会将其转换为返回类型）。而在最后一个逗号前面的所有表达式都必须成立，在这个例子中逗号前面只有 t.size()。之所以将其类型转换为 void，是为了避免因为用户重载了该表达式对应类型的逗号运算符而导致的不确定性。

注意 decltype 的操作数是不会被计算的，也就是说可以不调用构造函数而直接创建其“dummy”对象，相关内容将在 11.2.3 节讨论。

8.5 编译期 if

部分特例化，SFINAE 以及 std::enable_if 可以一起被用来禁用或者启用某个模板。而 C++17 又在此基础上引入了同样可以在编译期基于某些条件禁用或者启用相应模板的编译期 if 语句。通过使用 if constexpr(...)语法，编译器会使用编译期表达式来决定是使用 if 语句的 then 对应的部分还是 else 对应的部分。

作为第一个例子，考虑 4.1.1 节介绍的变参函数模板 print()。它用递归的方法打印其参数（可能是任意类型）。如果使用 constexp if，就可以在函数内部决定是否要继续递归下去，而不用再单独定义一个函数来终结递归：

```
template<typename T, typename… Types>
```

```
void print (T const& firstArg, Types const&… args)
{
    std::cout << firstArg << ' \n' ;
    if constexpr(sizeof…(args) > 0) {
        print(args…); //code only available if sizeof…(args)>0  (since
C++17)
    }
}
```

这里如果只给 print()传递一个参数，那么 args...就是一个空的参数包，此时 sizeof...(args)等于 0。这样 if 语句里面的语句就会被丢弃掉，也就是说这部分代码不会被实例化。因此也就不再需要一个单独的函数来终结递归。

事实上上面所说的不会被实例化，意思是对这部分代码只会进行第一阶段编译，此时只会做语法检查以及和模板参数无关的名称检查（参见 1.1.3 节）。比如：

```
template<typename T>
void foo(T t)
{
    if constexpr(std::is_integral_v<T>) {
        if (t > 0) {
            foo(t-1); // OK
        }
    }
    else {
        undeclared(t); // error if not declared and not
        discarded (i.e. T is not integral)
        undeclared(); // error if not declared (even if discarded)
        static_assert(false, "no integral"); // always asserts (even if
discarded)
        static_assert(!std::is_integral_v<T>, "no integral"); //OK
    }
}
```

此处 if constexpr 的使用并不仅限于模板函数，而是可以用于任意类型的函数。它所需要的只是一个可以返回布尔值的编译期表达式。比如：

```
int main()
{
    if constexpr(std::numeric_limits<char>::is_signed {
        foo(42); // OK
    }else {
        undeclared(42); // error if undeclared() not declared
        static_assert(false, "unsigned"); // always asserts (even if
discarded)
        static_assert(!std::numeric_limits<char>::is_signed, "char is
unsigned"); //OK
```

```
    }
}
```

利用这一特性，也可以让 8.2 节介绍的编译期函数 isPrime()在非类型参数不是质数的时候执行一些额外的代码：

```
template<typename T, std::size_t SZ>
void foo (std::array<T,SZ> const& coll)
{
    if constexpr(!isPrime(SZ)) {
        …
         //special additional handling if the passed array has no prime
number as size
    }
    …
}
```

更多细节请参见 14.6 节。

8.6 总结

- 模板提供了在编译器进行计算的能力（比如使用递归进行迭代以及使用部分特例化或者?:进行选择）。
- 通过使用 constexpr 函数，可以用在编译期上下文中能够被调用的“常规函数（要有 constexpr）”替代大部分的编译期计算工作。
- 通过使用部分特例化，可以基于某些编译期条件在不同的类模板实现之间做选择。
- 模板只有在被需要的时候才会被使用，对函数模板声明进行替换不会产生有效的代码。这一原理被称为 SFINAE。
- SFINAE 可以被用来专门为某些类型或者限制条件提供函数模板。
- 从 C++17 开始，可以通过使用编译期 if 基于某些编译期条件启用或者禁用某些语句。

第 9 章 在实践中使用模板

模板代码和常规代码有些不同。从某种程度上而言，模板介于宏和常规函数声明之间。虽然这样说可能过分简化了。它不仅会影响到我们用模板实现算法和数据结构的方法，也会影响到我们日常对包含模板的程序的分析和表达。

本章将解决这些实践性问题中的一部分，但是不会过多的讨论它们背后的细节。这些细节内容会在第 14 章进行探讨。为了让讨论不至过于复杂，假设我们的编译系统用到的都是很传统的编译器和链接器。

9.1 包含模式

有很多中组织模板源码的方式。本章讨论这其中最流行的一种方法：包含模式。

9.1.1 链接错误

大多数 C 和 C++程序员都会按照如下方式组织代码：

- 类和其它类型被放在头文件里。其文件扩展名为.hpp（或者.h, .H, .hh, .hxx）.
- 对于全局变量（非 inline）和函数（非 inline），只将其声明放在头文件里，定义则被放在一个被当作其自身编译单元的文件里。这一类文件的扩展名为.cpp（或者.C，.c，.cc，.cxx）。

这样做效果很好：既能够在整个程序中很容易的获得所需类型的定义，同时又避免了链接过程中的重复定义错误。

受这一惯例的影响，刚开始接触模板的程序员通常都会遇到下面这个程序中的错误。和处理“常规代码”的情况一样，在头文件中声明模板：

```
#ifndef MYFIRST_HPP
#define MYFIRST_HPP
// declaration of template
template<typename T>
void printTypeof (T const&);
#endif //MYFIRST_HPP
```

其中 printTypeof()是一个简单的辅助函数的声明，它会打印一些类型相关信息。而它的具体实现则被放在了一个 CPP 文件中：

```
#include <iostream>
#include <typeinfo>
```

```
#include "myfirst.hpp"
// implementation/definition of template
template<typename T>
void printTypeof (T const& x)
{
    std::cout << typeid(x).name() << ' \n' ;
}
```

这个函数用 typeid 运算符打印了一个用来描述被传递表达式的类型的字符串。该运算符返回一个左值静态类型 std::type_info，它的成员函数 name()可以返回某些表达式的类型。C++标准并没有要求 name()必须返回有意义的结果，但是在比较好的 C++实现中，它的返回结果应该能够很好的表述传递给 typeid 的参数的类型。

接着在另一个 CPP 文件中使用该模板，它会 include 该模板的头文件：

```
#include "myfirst.hpp"
// use of the template
int main()
{
    double ice = 3.0;
    printTypeof(ice); // call function template for type double
}
```

编译器很可能会正常编译这个程序，但是链接器则可能会报错说：找不到函数 printTypeof()的定义。

出现这一错误的原因是函数模板 printTypeof()的定义没有被实例化。为了实例化一个模板，编译器既需要知道需要实例化哪个函数，也需要知道应该用哪些模板参数来进行实例化。不幸的是，在上面这个例子中，这两组信息都是被放在别的文件里单独进行编译的。因此当编译器遇到对 printTypeof()的调用时，却找不到相对应的函数模板定义来针对 double 类型进行实例化，这样编译器只能假设这个函数被定义在别的地方，然后创建一个指向那个函数的引用（会在链接阶段由链接器进行解析）。另一方面，在编译器处理 myfirst.cpp 的时候，却没有任何指示让它用某种类型实例化模板。

9.1.2 头文件中的模板

解决以上问题的方法和处理宏以及 inline 函数的方法一样：将模板定义和模板声明都放在头文件里。

也就是说需要重写 myfirst.hpp，让它包含所有模板声明和模板定义，而不再提供 myfirst.cpp 文件：

```
#ifndef MYFIRST_HPP#define MYFIRST_HPP
#include <iostream>
#include <typeinfo>
// declaration of template
```

```
template<typename T>
void printTypeof (T const&);
// implementation/definition of template
template<typename T>
void printTypeof (T const& x)
{
    std::cout << typeid(x).name() << '\n';
}
#endif //MYFIRST_HPP
```

这种组织模板相关代码的方法被称为“包含模式”。使用这个方法，程序的编译，链接和执行都可以正常进行。

目前有几个问题需要指出。最值得注意的一个是，这一方法将大大增加 include 头文件 myfirst.hpp 的成本。在这个例子中，成本主要不是由模板自身定义导致的，而是由那些为了使用这个模板而必须包含的头文件导致的，比如<iostream>和<typeinfo>。由于诸如<iostream>的头文件还会包含一些它们自己的模板，因此这可能会带来额外的数万行的代码。

这是一个很实际的问题，因为对比较大的程序，它会大大的增加编译时间。后面的章节中会涉及到一些可能可以用来解决这一问题的方法，比如预编译头文件（9.2 节）和模板的显式实例化（14.5 节）。

尽管有编译时间的问题，但是除非有更好的方法，我们建议在可能的情况下还是尽量使用这一方式来组织模板代码。在写作本书的 2017 年，有一个正在准备阶段的机制：modules（C++20 已落实），我们会在 17.11 节中介绍相关内容。该机制让程序员能够更有逻辑的组织代码，可以让编译器分别编译所有的声明，然后在需要的地方高效地、有选择地导入处理之后的声明。

另一个不太明显的问题是，使用 include 方法时，非 inline 函数模板和 inline 函数以及宏之间有着明显的不同：非 inline 函数模板在被调用的地方不会被展开，而是会被实例化（产生一个函数的新的副本）。由于这是一个自动化过程，因此编译器可能会在两个不同的文件中实例化出两份函数的副本，某些链接器在遇到相同函数的两个定义时会报错。理论上我们不需要关心这一问题：这应该是 C++编译器处理的问题。在实践中也是这样，一切运转良好，我们不需要额外做些什么。但是对于比较大的、会创建自己的库的项目，可能会偶尔遇到问题。在第 14 章中关于实例化方案的讨论，以及对 C++编译系统的研读应该会对解决这一问题有帮助。

最后需要指出，以上例子中适用于常规函数模板的情况同样适用于类模板的成员函数和静态数据成员，甚至是成员函数模板。

9.2 模板和 inline

提高程序运行性能的一个常规手段是将函数声明为 inline 的。Inline 关键字的意思是给编译

器做一个暗示，要优先在函数调用处将函数体做 inline 替换展开，而不是按常规的调用机制执行。

但是编译器可能会忽略这一暗示。这样 inline 唯一可以保证的效果就是允许函数定义在程序中出现多次（因为其通常出现在被在多处调用的头文件中）。

和 inline 函数类似，函数模板也可以被定义在多个编译单元中。比如我们通常将模板定义放在头文件中，而这个头文件又被多个 CPP 文件包含。

但是这并不意味着函数模板在默认情况下就会使用 inline 替换。在模板调用处是否进行 inline 替换完全是由编译器决定的事情。编译器通常能够更好的评估 Inline 替换一个被调用函数是否能够提升程序性能。因此不同编译器之间对 inline 函数处理的精准原则也是不同的，这甚至会受编译选项的影响。

然而，通过使用合适的性能检测工具进行测试，程序员可能会比编译器更知道是否应该进行 inline 替换，因此也希望自己能够决定（而不是让编译器决定）是否需要进行 inline 替换。有时候这只能通过编译器的具体属性实现，比如 noinline 和 always_inline。

目前需要指出的一个问题是，就这一问题而言，函数模板在全特化之后和常规函数是一样的：除非其被定义成 inline 的，否则它只能被定义一次。更全面也更细致的介绍请参见附录 A。

9.3 预编译头文件

即使不适用模板，C++的头文件也会大到需要很长时间进行编译。而模板的引入则进一步加剧了这一问题，程序员对这一问题的抱怨促使编译器供应商提供了一种叫做预编译头文件（PCH: precomplied header）的方案来降低编译时间。这一方案不在 C++标准要求之中，因此其具体实现方式由编译器供应商自行决定。虽然我们没有过多的讨论创建以及使用预编译头文件的方式（该部分内容需要参考那些提供了该特性的 C++编译系统的文档），但是适当的了解其运作机制总是有帮助的。

当编译器编译一个文件的时候，它会从文件头开始编译并一直编译到结尾。当编译器处理文件中的符号（可能来自 include 文件）时，它会调整自己的状态，比如在符号表中添加新的条目，以方便随后的查找。在做这些事情的时候，编译器也可能会在目标文件中产生出一些代码。

预编译头文件方案的实现基于这样一个事实：在组织代码的时候，很多文件都以相同的几行代码作为开始。为了便于讨论，假设那些将要被编译文件的前 N 行内容都相同。这样就可以单独编译这 N 行代码，并将编译完成后的状态保存在一个预编译头文件中（precompiled header）。接着所有以这 N 行代码开始的文件，在编译时都会重新载入这个被保存的状态，然后从第 N+1 行开始编译。在这里需要指出，重新载入被保存的前 N 行代码的预编译状态可能会比再次编译这 N 行代码要快很多很多倍。但是保存这个状态可能要比单次编译这 N 行代码慢的多，编译时间可能延长 20%到 200%。

因此利用预编译头文件提高编译速度的关键点是：让尽可能多的文件，以尽可能多的相同的代码作为开始。也就是说在实践中，文件要以相同的#include 指令（它们可能占用大量的编译时间）开始。因此如果#include 头文件的顺序相同的话，就会对提高编译性能很有帮助。但是对下面的文件：

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

和

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

预编译头文件不会起作用，因为它们的起始状态并不一致（顺序不一致）。

一些程序员认为，即使可能会错过一个利用预编译头文件加速文件编译的机会，也应该多#include 一些可能用不到的头文件。这样做可以大大简化预编译头文件的使用方式。比如通常可以创建一个包含所有标准头文件的头文件，称之为 std.hpp：

宁可错杀一千，也不放过一个。宁愿多包含些多余的头文件，总比错过预编译加速好。
Some programmers decide that it is better to #include some extra unnecessary headers than to pass on an opportunity to accelerate the translation of a file using a precompiled header.

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

这个文件可以被预编译，其它所有用到标准库的文件都可以直接在文件开始处 include 这个头文件：

```
#include "std.hpp"
…
```

这个文件的编译会花费一些时间，但是如果内存足够的话，预编译方案的编译速度几乎要比在不使用预编译方案时编译其它任何一个标准库头文件都要快。标准头文件尤其适用于这一情况，因为它们很少发生变化，因此 std.hpp 的预编译头文件只会被编译一次。另外，预编译头文件也是项目依赖项配置的一部分（比如主流的 make 工具或者 IDE 工具在必要的时候会对它们进行更新）。

一个值得推荐的组织预编译头文件的方法是将它们按层级组织，从最常用以及最稳定的头文件（比如 std.hpp）到那些我们期望其一直都不会变化的（因此值得被预编译的）头文件。但是如果头文件正处于频繁的开发阶段，为它们创建预编译头文件可能会增加编译时间，而不是减少编译时间。总之记住一点，为稳定层创建的预编译头文件可以被重复使用，以提高那些不太稳定的头文件的编译速度。比如，除了上面已经预编译过的 std.hpp 文件，还有一个专为我们的项目准备的、尚未达到稳定状态的头文件 core.hpp：

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

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

由于这个文件（称之为 core.hpp）以#include “std.hpp”开始，编译器会去载入其对应的预编译头文件，然后继续编译之后的头文件，这期间不会对标准头文件进行再次编译。当整个文件处理完毕之后，就又产生了一个新的预编译头文件。由于编译器可以直接载入这个预编译的头文件，其它的应用就可以通过#include “core.hpp”头文件快速地使用其中的大量函数。

9.4 破译大篇幅的错误信息

常规函数的编译错误信息通常非常简单且直中要点。比如当编译器报错说”class X has no member ‘fun’”时，找到代码中相应的错误并不会很难。但是模板并不是这样。看下面这些例子。

简单的类型不匹配情况

考虑下面这个使用了 C++标准库的简单例子：

```
#include <string>
#include <map>
#include <algorithm>
int main()
{
    std::map<std::string,double> coll;
    …
    // find the first nonempty string in coll:
    auto pos = std::find_if (coll.begin(), coll.end(), [] (std::string
const& s){return s != ""; });
}
```

其中有一个相当小的错误：一个 lambda 函数被用来找到第一个匹配的字符串，它依次将 map 中的元素和一个字符串比较。但是，由于 map 中的元素是 key/value 对，因此传入 lambda 的元素也将是一个 std::pair<std::string const, double>，而它是不能直接和字符串进行比较的。

针对这个错误，主流的 GUN C++编译器会报如下错误：

```
1 In file included from /cygdrive/p/gcc/gcc61-
include/bits/stl_algobase.h:71:0,
2 from /cygdrive/p/gcc/gcc61-include/bits/char_traits.h:39,
3 from /cygdrive/p/gcc/gcc61-include/string:40,
4 from errornovel1.cpp:1:
```

```
5 /cygdrive/p/gcc/gcc61-
include/bits/predefined_ops.h: In instantiation of 'bool __gnu_cxx
::__ops::_Iter_pred<_Predicate>::operator()
(_Iterator) [with _Iterator = std::_Rb_tree_i
terator<std::pair<const std::__cxx11::basic_string<char>, doubl
::<lambda(const string&)>]':
6 /cygdrive/p/gcc/gcc61-
include/bits/stl_algo.h:104:42: required from '_InputIterator
std::__find_if(_InputIterator, _InputIterator, _Predicate, std:
[with _InputIterator = std::_Rb_tree_iterator<std::pair<const s
<char>, double> >; _Predicate = __gnu_cxx::__ops::_Iter_pred<
<lambda(const
string&)> >]'
7 /cygdrive/p/gcc/gcc61-
include/bits/stl_algo.h:161:23: required from '_Iterator std::__
find_if(_Iterator, _Iterator, _Predicate) [with _Iterator = std
pair<const std::__cxx11::basic_string<char>, double> >; _Predic
Iter_pred<main()::<lambda(const string&)> >]'
8 /cygdrive/p/gcc/gcc61-
include/bits/stl_algo.h:3824:28: required from '_IIter std::find
_if(_IIter, _IIter, _Predicate) [with _IIter = std::_Rb_tree_it
std::__cxx11::basic_string<char>, double> >; _Predicate = main
<lambda(const string&)
>]'
9 errornovel1.cpp:13:29: required from here
10 /cygdrive/p/gcc/gcc61-
include/bits/predefined_ops.h:234:11: error: no match for call to
'(main()::
<lambda(const string&)>) (std::pair<const std::__cxx11::basic_string<
double>&)'11 { return bool(_M_pred(*__it)); }
12 ^~~~~~~~~~~~~~~~~~~
13 /cygdrive/p/gcc/gcc61-
include/bits/predefined_ops.h:234:11: note: candidate: bool (*)
(
const string&) {aka bool (*)
(const std::__cxx11::basic_string<char>&)} <conversion>
14 /cygdrive/p/gcc/gcc61-
include/bits/predefined_ops.h:234:11: note: candidate expects 2
arguments, 2 provided
15 errornovel1.cpp:11:52: note: candidate: main()::
<lambda(const string&)>
16 [] (std::string const& s) {
17 ^
18 errornovel1.cpp:11:52: note: no known conversion for argument
```

std::__cxx11::basic_string<char>, double>' to 'const string& {a
basic_string<char>&}'

这个信息初看起来更像是小说而不是对 debug 有用的信息。如此长的信心很可能会让刚开始使用模板的程序员感到沮丧。但是当有了一定的经验之后，就能够应对这一类型的错误信息了，至少可以很容易的定位到出错的地方。

以上错误信息中第一部分的意思是，在一个函数模板的实例中遇到了错误，这个模板位于一个内部头文件 predefined_ops.h 中。在这一行以及后面的几行中，编译器报告了哪些模板被用哪些参数实例化了。（太过冗长，暂不翻译）In this case,
it all started with the statement ending on line 13 of errornovel1.cpp, which is:
Click here to view code image
auto pos = std::find_if (coll.begin(), coll.end(),
[] (std::string const& s) {
return s != "";
});
This caused the instantiation of a find_if template on line 115 of the
stl_algo.h header, where the code
Click here to view code image
_IIter std::find_if(_IIter, _IIter, _Predicate)
is instantiated with
Click here to view code image
_IIter = std::_Rb_tree_iterator<std::pair<const
std::__cxx11::basic_string<char>,
double> >_Predicate = main()::<lambda(const string&)>
The compiler reports all this in case we simply were not expecting all these
templates to be instantiated. It allows us to determine the chain of events that caused
the instantiations.
However, in our example, we’re willing to believe that all kinds of templates
needed to be instantiated, and we just wonder why it didn’t work. This information
comes in the last part of the message: The part that says “no match for call”
implies that a function call could not be resolved because the types of the arguments
and the parameter types didn’t match. It lists what is called
Click here to view code image
(main()::<lambda(const string&)>) (std::pair<const
std::__cxx11::basic_string<char>,
double>&)
and code that caused this call:
Click here to view code image
{ return bool(_M_pred(*__it)); }
Furthermore, just after this, the line containing “note: candidate:” explains
that there was a single candidate type expecting a const string& and that this
candidate is defined in line 11 of errornovel1.cpp as lambda []
(std::string const& s) combined with a reason why a possible candidate

didn’t fit:
Click here to view code image
no known conversion for argument 1
from ’ std::pair<const std::__cxx11::basic_string<char>, double>’
to ’ const string& {aka const std::__cxx11::basic_string<char>&}’
which describes the problem we have.
There is no doubt that the error message could be better. The actual problem could be emitted before the history of the instantiation, and instead of using fully expanded template instantiation names like std::__cxx11::basic_string<char>, using just std::string might be enough. However, it is also true that all the information in this diagnostic could be useful in some situations. It is therefore not surprising that other compilers provide similar information (although some use the structuring techniques mentioned).
For example, the Visual C++ compiler outputs something like:
Click here to view code image
1 c:\tools_root\cl\inc\algorithm(166): error C2664: 'bool main::<lambda_b863c1c7cd07048816
f454330789acb4>::operator ()
(const std::string &) const': cannot convert argument 1 from
'std::pair<const _Kty,_Ty>' to 'const std::string &'
2 with
3 [
4 _Kty=std::string,
5 _Ty=double
6]
7 c:\tools_root\cl\inc\algorithm(166): note: Reason: cannot convert
_Kty,_Ty>' to 'const std::string'
8 with
9 [
10 _Kty=std::string,
11 _Ty=double
12]
13 c:\tools_root\cl\inc\algorithm(166): note: No userdefined-conversion operator available
that can perform this conversion, or the operator cannot be called
14 c:\tools_root\cl\inc\algorithm(177): note: see reference to functi
ion '_InIt std::_Find_if_unchecked<std::_Tree_unchecked_iterator<_
(_InIt,_In
It,_Pr &)' being compiled
15 with
16 [
17 _InIt=std::_Tree_unchecked_iterator<std::_Tree_val<std:
<std::pair<const std::string,double>>>>,
18 _Mytree=std::_Tree_val<std::_Tree_simple_types<std::pai
double>>>,
19 _Pr=main::

<lambda_b863c1c7cd07048816f454330789acb4>
20]
21 main.cpp(13): note: see reference to function template instantiati
<std::_Tree_iterator<std::_Tree_val<std::_Tree_simple_types<std::p
,main::<lambda_b863c1c7cd07048816f454330789acb4>>
(_InIt,_InIt,_Pr)' being compiled
22 with
23 [
24 _InIt=std::_Tree_iterator<std::_Tree_val<std::_Tree_sim
const std::string,double>>>>,
25 _Kty=std::string,
26 _Ty=double,
27 _Pr=main::
<lambda_b863c1c7cd07048816f454330789acb4>
28]
Here, again, we provide the chain of instantiations with the information telling us what was instantiated by which arguments and where in the code, and we see twicethat we
Click here to view code image
cannot convert from ’std::pair<const _Kty,_Ty>’ to ’const std::string’
with
[
_Kty=std::string,
_Ty=double
]
Missing const on Some Compilers
Unfortunately, it sometimes happens that generic code is a problem only with some compilers. Consider the following example:
Click here to view code image
basics/errornovel2.cpp
#include <string>
#include <unordered_set>
class Customer
{
private:
std::string name;
public:
Customer (std::string const& n)
: name(n) {
} s
td::string getName() const {
return name;
}
};

```
int main()
{
// provide our own hash function:
struct MyCustomerHash {
// NOTE: missing const is only an error with g++ and clang:
std::size_t operator() (Customer const& c) {
return std::hash<std::string>()(c.getName());
}
};
// and use it for a hash table of Customers:std::unordered_set<Customer,MyCustomerHash>
coll; …
} W
```

ith Visual Studio 2013 or 2015, this code compiles as expected. However, with g++ or clang, the code causes significant error messages. On g++ 6.1, for example, the first error message is as follows:

Click here to view code image

```
1 In file included from /cygdrive/p/gcc/gcc61-
include/bits/hashtable.h:35:0,
2 from /cygdrive/p/gcc/gcc61-include/unordered_set:47,
3 from errornovel2.cpp:2:
4 /cygdrive/p/gcc/gcc61-include/bits/hashtable_policy.h: In
instantiation of 'struct std::
__detail::__is_noexcept_hash<Customer,
main()::MyCustomerHash>':
5 /cygdrive/p/gcc/gcc61-include/type_traits:143:12:
required from 'struct std::__and_<
std::__is_fast_hash<main()::MyCustomerHash>,
std::__detail::__is_noexcept_hash<Customer,
main()::MyCustomerHash> >'
6 /cygdrive/p/gcc/gcc61-include/type_traits:154:38:
required from 'struct std::__not_<
std::__and_<std::__is_fast_hash<main()::MyCustomerHash>,
std::__detail::__is_noexcept_
hash<Customer, main()::MyCustomerHash> > >'
7 /cygdrive/p/gcc/gcc61-
include/bits/unordered_set.h:95:63: required from 'class
std::
unordered_set<Customer, main()::MyCustomerHash>'
8 errornovel2.cpp:28:47: required from here
9 /cygdrive/p/gcc/gcc61-
include/bits/hashtable_policy.h:85:34: error: no match for
call to
'(const main()::MyCustomerHash) (const Customer&)'
10 noexcept(declval<const _Hash&>()(declval<const _Key&>
```

()))>
11 ~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~
12 errornovel2.cpp:22:17: note: candidate: std::size_t
main()::MyCustomerHash::operator()(
const Customer&) <near match>
13 std::size_t operator() (const Customer& c) {
14 ^~~~~~~~
15 errornovel2.cpp:22:17: note: passing 'const
main()::MyCustomerHash*' as 'this' argument
discards qualifiersimmediately followed by more than 20 other error messages:
16 In file included from /cygdrive/p/gcc/gcc61-
include/bits/move.h:57:0,
18 from /cygdrive/p/gcc/gcc61-
include/bits/stl_pair.h:59,
19 from /cygdrive/p/gcc/gcc61-
include/bits/stl_algobase.h:64,
20 from /cygdrive/p/gcc/gcc61-
include/bits/char_traits.h:39,
21 from /cygdrive/p/gcc/gcc61-include/string:40,
22 from errornovel2.cpp:1:
23 /cygdrive/p/gcc/gcc61-include/type_traits: In
instantiation of 'struct std::__not_<std::
__and_<std::__is_fast_hash<main()::MyCustomerHash>,
std::__detail::__is_noexcept_hash<
Customer, main()::MyCustomerHash> > >':
24 /cygdrive/p/gcc/gcc61-
include/bits/unordered_set.h:95:63: required from 'class
std::
unordered_set<Customer, main()::MyCustomerHash>'
25 errornovel2.cpp:28:47: required from here
26 /cygdrive/p/gcc/gcc61-include/type_traits:154:38: error:
'value' is not a member of 'std
::__and_<std::__is_fast_hash<main()::MyCustomerHash>,
std::__detail::__is_noexcept_hash<
Customer, main()::MyCustomerHash> >'
27 : public integral_constant<bool, !_Pp::value>
28 ^~~~
29 In file included from /cygdrive/p/gcc/gcc61-
include/unordered_set:48:0,
30 from errornovel2.cpp:2:
31 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h: In
instantiation of 'class std::
unordered_set<Customer, main()::MyCustomerHash>':
32 errornovel2.cpp:28:47: required from here

33 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h:95:63:
error: 'value' is not a member
of
'std::__not_<std::__and_<std::__is_fast_hash<main()::MyCustomerHash>,
std::__detail::
__is_noexcept_hash<Customer, main()::MyCustomerHash> >
>'
34 typedef __uset_hashtable<_Value, _Hash, _Pred, _Alloc>
_Hashtable;
35 ^~~~~~~~~~
36 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h:102:45:error: 'value' is not a member
of
'std::__not_<std::__and_<std::__is_fast_hash<main()::MyCustomerHash>,
std::__detail::
__is_noexcept_hash<Customer, main()::MyCustomerHash> >
>'
37 typedef typename _Hashtable::key_type key_type;
38 ^~~~~~~
...
Again, it’s hard to read the error message (even finding the beginning and end of
each message is a chore). The essence is that deep in header file
hashtable_policy.h in the instantiation of std::unordered_set<>
required by
Click here to view code image
std::unordered_set<Customer,MyCustomerHash> coll;
there is no match for the call to
Click here to view code image
const main()::MyCustomerHash (const Customer&)
in the instantiation of
Click here to view code image
noexcept(declval<const _Hash&>()(declval<const _Key&>()))>
~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~
(declval<const _Hash&>() is an expression of type
main()::MyCustomerHash). A possible “near match” candidate is
Click here to view code image
std::size_t main()::MyCustomerHash::operator()(const Customer&)
which is declared as
Click here to view code image
std::size_t operator() (const Customer& c) {
^~~~~~~
and the last note says something about the problem:
Click here to view code image
passing ’const main()::MyCustomerHash*’ as ’this’ argument
discards qualifiers
~~~~~~~~~~~~~~~~~~~~~~~

Can you see what the problem is? This implementation of thestd::unordered_set class template requires that the function call operator for
the hash object be a const member function (see also Section 11.1.1 on page 159).
When that’s not the case, an error arises deep in the guts of the algorithm.
All other error messages cascade from the first and go away when a const
qualifier is simply added to the hash function operator:
Click here to view code image
std::size_t operator() (const Customer& c) const {
… }
Clang 3.9 gives the slightly better hint at the end of the first error message that
operator() of the hash functor is not marked const:
Click here to view code image
… e
rrornovel2.cpp:28:47: note: in instantiation of template class
’std::unordered_set<Customer
, MyCustomerHash, std::equal_to<Customer>,
std::allocator<Customer> >’ requested here
std::unordered_set<Customer,MyCustomerHash> coll;
^
errornovel2.cpp:22:17: note: candidate function not viable:
’this’ argument has type ’const
MyCustomerHash’, but method is not marked const
std::size_t operator() (const Customer& c) {
^
Note that clang here mentions default template parameters such as
std::allocator<Customer>, while gcc skips them.
As you can see, it is often helpful to have more than one compiler available to test
your code. Not only does it help you write more portable code, but where one
compiler produces a particularly inscrutable error message, another might provide
more insig

9.5 后记

将源代码分成头文件和 CPP 文件是为了遵守唯一定义法则（one-definition rule, ODR）。附录 A 中对该法则有详实的介绍。

基于 C++编译器实现中的既有惯例，包含模式是一种很务实的解决方案。但是在最初的 C++实现中情况有所不同：模板定义的包含是隐式的，这就给人以源文件和头文件“分离”的错觉（参见第 14 章）。

C++98 通过导出模板（exported templates）支持了模板编译的分离模式（separation model）。这一分离模式允许被 export 标记的模板声明被声明在头文件里，相应的定义则被实现在 CPP 文件里，这一点和常规非 template 代码的情况很相似。不同于包含模式，这一模式是一种

不基于任何已有实现的理论模式，而且其实现也要远比 C++标准委员会所期待的要复杂。直到五年之后这一实现方式才被公布，这期间也没有其它实现方式出现。为了保持 C++标准和既有惯例的一致性，C++标准委员会在 C++11 中移除了 export 模式。对这一方面内容感兴趣的读者可以去读一下背书第一版的 6.3 节和 10.3 节。

It is sometimes tempting to imagine ways of extending the concept of precompiled headers so that more than one header could be loaded for a single compilation. This would in principle allow for a finer grained approach to precompilation. The obstacle here is mainly the preprocessor: Macros in one header file can entirely change the meaning of subsequent header files. However, once a file has been precompiled, macro processing is completed, and it is hardly practical to attempt to patch a precompiled header for the preprocessor effects induced by other headers. A new language feature known as modules (see Section 17.11 on page 366) is expected to be added to C++ in the not too distant future to address this issue (macro definitions cannot leak into module interfaces).

9.6 总结

- 模板的包含模式被广泛用来组织模板代码。第 14 章会介绍另一种替代方法。
- 当被定义在头文件中，且不在类或者结构体中时，函数模板的全特例化版本需要使用 inline。
- 为了充分发挥预编译的特性，要确保#include 指令的顺序相同。
- Debug 模板相关代码很有挑战性。

第 10 章 模板基本术语

到目前为止，我们介绍了一些 C++中模板的基本概念。在开始介绍更多细节内容之前，先来看一些将会被用到的术语。这是必要的，因为有时在 C++社区中（甚至实在之前的 C++标准中）会找不到某些术语的精确定义。

10.1 “类模板”还是“模板类”

在 C++中，structs，classes 以及 unions 都被称为 class types。如果没有特殊声明的话，“class”的字面意思是用关键字 class 或者 struct 声明的 class types。注意 class types 包含 unions，但是 class 不包含。

关于该如何称呼一个是模板的类，有一些困扰：

- 术语 class template 是指这个 class 是模板。也就是说它是一组 class 的参数化表达。
- 术语 template class 则被：
  - 用作 class template 的同义词。
  - 用来指代从 template 实例化出来的 classes。
  - 用来指代名称是一个 template-id（模板名 + <模板参数>）的类。

第二种和第三中意思的区别很小，对后续的讨论也不重要。

由于这一不确定性，在本书中会避免使用术语 template class。

同样地，我们会使用 function template，member template，member function template，以及 variable template，但不会使用 template function，template member，template member function，以及 template variable。

10.2 替换，实例化，和特例化

在处理模板相关的代码时，C++编译器必须经常去用模板实参替换模板参数。有时后这种替换只是试探性的：编译器需要验证这个替换是否有效（参见 8.4 节以及 15.7 节）。

用实际参数替换模板参数，以从一个模板创建一个常规类、类型别名、函数、成员函数或者变量的过程，被称为“模板实例化”。

不过令人意外的是，目前就该如何表示通过模板参数替换创建一个声明（不是定义）的过程，还没有相关标准以及基本共识。有人使用“部分实例化（partial instantiation）”或者“声明的实例化（instantiation of a declaration）”，但是这些用法都不够普遍。或许使用“不完全

实例化（incomplete instantiation）”会更直观一些（对于类模板，产生的是不完整类）。

通过实例化或者不完全实例化产生的实体通常被称为特例化（specialization）。

但是在 C++中，实例化过程并不是产生特例化的唯一方式。另外一些方式允许程序员显式的指定一个被关联到模板参数的、被进行了特殊替换的声明。正如 2.5 节介绍的那样，这一类特例化以一个 template<>开始：

```
template<typename T1, typename T2> // primary class template
class MyClass {
    …
};
template<> // explicit specialization
class MyClass<std::string,float> {
    …
};
```

严格来说，这被称为显式特例化（explicit specialization）。

正如在 2.6 节介绍的那样，如果特例化之后依然还有模板参数，就称之为部分特例化。

```
template<typename T> // partial specialization
class MyClass<T,T> {
    …
};
template<typename T> // partial specialization
class MyClass<bool,T> {
    …
};
```

在讨论（显式或者部分）特例化的时候，特例化之前的通用模板被称为主模板。

10.3 声明和定义

到目前为止，“声明”和“定义”只在本书中使用了几次。但是在标准 C++中，这些单词有着明确的定义，我们也将采用这些定义。

“声明”是一个 C++概念，它将一个名称引入或者再次引入到一个 C++作用域内。引入的过程中可能会包含这个名称的一部分类别，但是一个有效的声明并不需要相关名称的太多细节。比如：

```
class C; // a declaration of C as a class
void f(int p); // a declaration of f() as a function and p as a named
parameter
extern int v; // a declaration of v as a variable
```

注意，在 C++中虽然宏和 goto 标签也都有名字，但是它们并不是声明。

对于声明，如果其细节已知，或者是需要申请相关变量的存储空间，那么声明就变成了定义。对于 class 类型的定义和函数定义，意味着需要提供一个包含在{}中的主体，或者是对函数使用了=defaul/=delete。对于变量，如果进行了初始化或者没有使用 extern，那么声明也会变成定义。下面是一些“定义”的例子：

```
class C {}; // definition (and declaration) of class C
void f(int p) { //definition (and declaration) of function f()
    std::cout << p << ' \n' ;
}
extern int v = 1; // an initializer makes this a definition for v
int w; // global variable declarations not preceded by extern are also
definitions
```

作为扩展，如果一个类模板或者函数模板有包含在{}中的主体的话，那么声明也会变成定义。

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

是一个声明。而：

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

则是一个定义。

10.3.1 完整类型和非完整类型（complete versus incomplete types）

类型可以是完整的（complete）或者是不完整的（incomplete），这一名词和声明以及定义之间的区别密切相关。有些语言的设计要求完整类型，有一些也适用于非完整类型。

非完整类型是以下情况之一：

- 一个被声明但是还没有被定义的 class 类型。
- 一个没有指定边界的数组。
- 一个存储非完整类型的数组。
- Void 类型。
- 一个底层类型未定义或者枚举值未定义的枚举类型。
- 任何一个被 const 或者 volatile 修饰的以上某种类型。其它所有类型都是完整类型。比如：

```
class C; // C is an incomplete type
C const* cp; // cp is a pointer to an incomplete type
extern C elems[10]; // elems has an incomplete type
extern int arr[]; // arr has an incomplete type…c
```

```
lass C { }; // C now is a complete type (and therefore cpand elems
// no longer refer to an incomplete type)
int arr[10]; // arr now has a complete type
```

关于在模板中应该如何处理非完整类型，请参见 11.5 节。

10.4 唯一定义法则

C++语言中对实体的重复定义做了限制。这一限制就是“唯一定义法则（one-definition rule, ODR）”。相关细节非常复杂，涵盖的内容也比较多。接下来的章节中会涉及到各种应用上的情况，完整的关于 ODR 的介绍请参见附录 A。目前只要记住以下基础的 ODR 就够了：

- 常规（比如非模板）非 inline 函数和成员函数，以及非 inline 的全局变量和静态数据成员，在整个程序中只能被定义一次。
- Class 类型（包含 struct 和 union），模板（包含部分特例化，但不能是全特例化），以及 inline 函数和变量，在一个编译单元中只能被定义一次，而且不同编译单元间的定义应该相同。

编译单元是通过预处理源文件产生的一个文件；它包含通过#include 指令包含的内容以及宏展开之后的内容。

在后面的章节中，可链接实体（linkable entity）指的是下面的任意一种：一个函数或者成员函数，一个全局变量或者静态数据成员，以及通过模板产生的类似实体，只要对 linker 可见就行。

10.5 Template Arguments versus Template Parameters

考虑如下类模板：

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

和一个类似的类：

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

如果将前者中的模板参数 T 和 N 替换为 double 和 10，那么它将和后者相同。在 C++中这种类型的模板参数替换被表示为：

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

注意模板名称后面的尖括号以及其中的模板实参。

不管这些实参是否和模板参数有关，模板名称以及其后面的尖括号和其中的模板实参，被称为 template-id。

其用法和非模板类的用法非常相似。比如：

```
int main()
{
    ArrayInClass<double,10> ad; ad.array[0] = 1.0;
}
```

有必要对模板参数（template parameters）和模板实参（template arguments）进行区分。简单来讲可以说“模板参数是被模板实参初始化的”。或者更准确的说：

- 模板参数是那些在模板定义或者声明中，出现在 template 关键字后面的尖括号中的名称。
- 模板实参是那些用来替换模板参数的内容。不同于模板参数，模板实参可以不只是“名称”。

当指出模板的 template-id 的时候，用模板实参替换模板参数的过程就是显式的，但是在很多情况这一替换则是隐式的（比如模板参数被其默认值替换的情况）。

一个基本原则是：任何模板实参都必须是在编译期可知的。就如接下来会澄清的，这一要求对降低模板运行期间的成本很有帮助。由于模板参数最终都会被编译期的值进行替换，它们也可以被用于编译期表达式。在 ArrayInClass 模板中指定成员 array 的尺寸时就用到了这一特性。数组的尺寸必须是一个常量表达式，而模板参数 N 恰好满足这一要求。

对这一特性的使用可以更进一步：由于模板参数是编译期实体，它们也可以被用作模板实参。就像下面这个例子这样：

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

其中 T 既是模板参数也是模板实参。这样这一原理就可以被用来从简单模板构造更复杂的模板。当然，在原理上，这和我们构造类型和函数并没有什么不同。

10.6 总结

- 对那些是模板的类，函数和变量，我们称之为类模板，函数模板和变量模板。
- 模板实例化过程是一个用实参取代模板参数，从而创建常规类或者函数的过程。最终产

生的实体是一个特化。

- 类型可以是完整的或者非完整的。
- 根据唯一定义法则（ODR），非 inline 函数，成员函数，全局变量和静态数据成员在整个程序中只能被定义一次。

第 11 章 泛型库

到目前为止，关于模板的讨论主要是基于直接的任务和应用，集中在某些特性，能力和限制上。但是当模板被用于泛型库和框架设计时，其效果更明显，此时必须考虑到一些限制更少的潜在应用。虽然本书中几乎所有的内容都可以用于此类设计，接下来还是会重点介绍一些在设计可能会被用于未知类型的便捷组件时应该考虑的问题。

此处并没有穷尽所有的问题，只是总结了目前为止已经介绍的一些特性，引入了一些新的特性，同时引用了一些在接下来的章节中才会涉及到的特性。希望这能偶促使你继续阅读后面的某些章节。

11.1 可调用对象（Callables）

一些库包含这样一种接口，客户端代码可以向该类接口传递一个实体，并要求该实体必须被调用。相关的例子有：必须在另一个线程中被执行的操作，一个指定该如何处理 hash 值并将其存在 hash 表中的函数（hash 函数），一个指定集合中元素排序方式的对象，以及一个提供了某些默认参数值的泛型包装器。标准库也不例外：它定义了很多可以接受可调用对象作为参数的组件。

这里会用到一个叫做回调（callback）的名词。传统上这一名词被作为函数调用实参使用，我们将保持这一传统。比如一个排序函数可能会接受一个回调参数并将其用作排序标准，该回调参数将决定排序顺序。

在 C++中，由于一些类型既可以被作为函数调用参数使用，也可以按照 f(...)的形式调用，因此可以被用作回调参数：

- 函数指针类型
- 重载了 operator()的 class 类型（有时被称为仿函数（functors）），这其中包含 lambda 函数
- 包含一个可以产生一个函数指针或者函数引用的转换函数的 class 类型

这些类型被统称为函数对象类型（function object types），其对应的值被称为函数对象（function object）。

如果可以接受某种类型的可调用对象的话，泛型代码通常可以从中受益，而模板使其称为可能。

11.1.1 函数对象的支持

来看一下标准库中的 for_each()算法是如何实现的（为了避免名字冲突，这里使用“foreach”，为了简单也将不会返回任何值）：

```
template<typename Iter, typename Callable>
void foreach (Iter current, Iter end, Callable op)
{
    while (current != end) { //as long as not reached the end
        op(*current); // call passed operator for current element
        ++current; // and move iterator to next element
    }
}
```

下面的代码展示了将以上模板用于多种函数对象的情况：

```
#include <iostream>#include <vector>
#include "foreach.hpp"
// a function to call:
void func(int i)
{
    std::cout << "func() called for: " << i << ' \n' ;
}
// a function object type (for objects that can be used as functions):
class FuncObj {
    public:
        void operator() (int i) const { //Note: const member function
            std::cout << "FuncObj::op() called for: " << i << ' \n' ;
        }
};

int main()
{
    std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
    foreach(primes.begin(), primes.end(),  // range
        func); // function as callable (decays to pointer)
    foreach(primes.begin(), primes.end(), // range
        &func); // function pointer as callable
    foreach(primes.begin(), primes.end(),  // range
        FuncObj()); // function object as callable
    foreach(primes.begin(), primes.end(), // range
        [] (int i) { //lambda as callable
            std::cout << "lambda called for: " << i << ' \n' ;
          });
}
```

详细看一下以上各种情况：

- 当把函数名当作函数参数传递时，并不是传递函数本体，而是传递其指针或者引用。和数组情况类似（参见 7.4 节），在按值传递时，函数参数退化为指针，如果参数类型是模板参数，那么类型会被推断为指向函数的指针。

  和数组一样，按引用传递的函数的类型不会 decay。但是函数类型不能真正用 const 限制。如果将 foreach()的最后一个参数的类型声明为 Callable const &，const 会被省略。（通常而言，在主流 C++代码中很少会用到函数的引用。）
- 在第二个调用中，函数指针被显式传递（传递了一个函数名的地址）。这和第一中调用方式相同（函数名会隐式的 decay 成指针），但是相对而言会更清楚一些。
- 如果传递的是仿函数，就是将一个类的对象当作可调用对象进行传递。通过一个 class 类型进行调用通常等效于调用了它的 operator()。因此下面这样的调用：

  op(*current);

  会被转换成：

  ```
  op.operator()(*current); // call operator() with parameter *current
  for op
  ```

  注意在定义 operator()的时候最好将其定义成 const 成员函数。否则当一些框架或者库不希望该调用会改变被传递对象的状态时，会遇到很不容易 debug 的 error。

  对于 class 类型的对象，有可能会被转换为指向 surrogate call function（参见 C.3.5）的指针或者引用。此时，下面的调用：

  op(*current);

  会被转换为：

  ```
  (op.operator F())(*current);
  ```

  其中 F 就是 class 类型的对象可以转换为的，指向函数的指针或者指向函数的引用的类型。

- Lambda 表达式会产生仿函数（也称闭包），因此它与仿函数（重载了 operator()的类）的情况没有不同。不过 Lambda 引入仿函数的方法更为简便，因此它们从 C++11 开始变得很常见。

  有意思的是，以[]开始的 lambdas（没有捕获）会产生一个向函数指针进行转换的运算符。但是它从来不会被当作 surrogate call function，因为它的匹配情况总是比常规闭包的 operator()要差。

11.1.2 处理成员函数以及额外的参数

在以上例子中漏掉了另一种可以被调用的实体：成员函数。这是因为在调用一个非静态成员

函数的时候需要像下面这样指出对象：object.memfunc(...)或者 ptr->memfunc(...)，这和常规情况下的直接调用方式不同：func(...)。

幸运的是，从 C++17 开始，标准库提供了一个工具：std::invlke()，它非常方便的统一了上面的成员函数情况和常规函数情况，这样就可以用同一种方式调用所有的可调用对象。下面代码中 foreach()的实现使用了 std::invoke()：

```
#include <utility>
#include <functional>
template<typename Iter, typename Callable, typename… Args>
void foreach (Iter current, Iter end, Callable op, Args const&…args)
{
    while (current != end) { //as long as not reached the end of the
elements
        std::invoke(op, //call passed callable with
            args…, //any additional args
            *current); // and the current element
        ++current;
    }
}
```

这里除了作为参数的可调用对象，foreach()还可以接受任意数量的参数。然后 foreach()将参数传递给 std::invoke()。Std::invoke()会这样处理相关参数：

- 如果可调用对象是一个指向成员函数的指针，它会将 args...中的第一个参数当作 this 对象（不是指针）。Args...中其余的参数则被当做常规参数传递给可调用对象。
- 否则，所有的参数都被直接传递给可调用对象。

注意这里对于可调用对象和 agrs..都不能使用完美转发（perfect forward）：因为第一次调用可能会 steal(偷窃)相关参数的值，导致在随后的调用中出现错误。

现在既可以像之前那样调用 foreach()，也可以向它传递额外的参数，而且可调用对象可以是一个成员函数。正如下面的代码展现的那样：

```
#include <iostream>
#include <vector>
#include <string>
#include "foreachinvoke.hpp"
// a class with a member function that shall be called
class MyClass {
    public:
        void memfunc(int i) const {
            std::cout << "MyClass::memfunc() called for: " << i << '
\n' ;
        }
};
int main()
```

```
{
    std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
    // pass lambda as callable and an additional argument:
    foreach(primes.begin(), primes.end(), //elements for 2nd arg of
lambda
        [](std::string const& prefix, int i) { //lambda to call
            std::cout << prefix << i << ' \n' ;
        },
        "- value:"); //1st arg of lambda
    // call obj.memfunc() for/with each elements in primes passed as
argument
    MyClass obj;
    foreach(primes.begin(), primes.end(), //elements used as args
        &MyClass::memfunc, //member function to call
        obj); // object to call memfunc() for
}
```

第一次调用 foreach()时，第四个参数被作为 lambda 函数的第一个参数传递给 lambda，而 vector 中的元素被作为第二个参数传递给 lambda。第二次调用中，第三个参数 memfunc() 被第四个参数 obj 调用。

关于通过类型萃取判断一个可调用对象是否可以用于 std::invode()的内容，请参见 D.3.1 节。

11.1.3 函数调用的包装

Std::invoke()的一个常规用法是封装一个单独的函数调用（比如：记录相关调用，测量所耗时常，或者准备一些上下文信息（比如为此启动一个线程））。此时可以通过完美转发可调用对象以及被传递的参数来支持移动语义：

```
#include <utility> // for std::invoke()
#include <functional> // for std::forward()
template<typename Callable, typename… Args>
decltype(auto) call(Callable&& op, Args&&… args)
{
    return std::invoke(std::forward<Callable>(op), //passed callable
with
        std::forward<Args>(args)…); // any additional args
}
```

一个比较有意思的地方是该如何处理被调用函数的返回值，才能将其“完美转发”给调用者。为了能够返回引用（比如 std::ostream&），需要使用 decltype(auto)而不是 auto：

```
template<typename Callable, typename… Args>
decltype(auto) call(Callable&& op, Args&&… args)
```

decltype(auto)（在 C++14 中引入）是一个占位符类型，它根据相关表达式决定了变量、

返回值、或者模板实参的类型。详情请参考 15.10.3 节。

如果你想暂时的将 std::invoke()的返回值存储在一个变量中，并在做了某些别的事情后将其返回（比如处理该返回值或者记录当前调用的结束），也必须将该临时变量声明为 decltype(auto)类型：

```
decltype(auto) ret{std::invoke(std::forward<Callable>(op),
    std::forward<Args>(args)…)};
…
return ret;
```

注意这里将 ret 声明为 auto &&是不对的。Auto&&作为引用会将变量的生命周期扩展到作用域的末尾（参见 11.3 节），但是不会扩展到超出 return 的地方。

不过即使是使用 decltype(auto)也还是有一个问题：如果可调用对象的返回值是 void，那么将 ret 初始化为 decltype(auto)是不可以的，这是因为 void 是不完整类型。此时有如下选择：

- 在当前行前面声明一个对象，并在其析构函数中实现期望的行为。比如：

```
struct cleanup {
    ~cleanup() {
        … //code to perform on return
    }
} dummy;
return                          std::invoke(std::forward<Callable>(op),
std::forward<Args>(args)…);
```

- 分别实现 void 和非 void 的情况：

```
#include <utility> // for std::invoke()
#include <functional> // for std::forward()
#include <type_traits> // for std::is_same<> and
invoke_result<>
template<typename Callable, typename… Args>
decltype(auto) call(Callable&& op, Args&&… args)
{
    if constexpr(std::is_same_v<std::invoke_result_t<Callable,
Args…>, void>) {// return type is void:
        std::invoke(std::forward<Callable>(op),
        std::forward<Args>(args)…);
        …
        return;
    } else {
        // return type is not void:
        decltype(auto) ret{std::invoke(std::forward<Callable>(op),
std::forward<Args>(args)…)};
        …
```

```
        return ret;
    }
}
```

其中：

```
if constexpr(std::is_same_v<std::invoke_result_t<Callable, Args…>,
void>)
```

在编译期间检查使用 Args...的 callable 的返回值是不是 void 类型。关于 std::invoke_result<>的细节请参见 D.3.1 节。

后续的 C++版本可能会免除掉这种对 void 的特殊操作（参见 17.7 节）。

11.2 其他一些实现泛型库的工具

std::invoke()只是 C++标准库提供的诸多有用工具中的一个。在接下来的内容中，我们会介绍其他一些重要的工具。

11.2.1 类型萃取

标准库提供了各种各样的被称为类型萃取（type traits）的工具，它们可以被用来计算以及修改类型。这样就可以在实例化的时候让泛型代码适应各种类型或者对不同的类型做出不同的响应。比如：

```
#include <type_traits>
template<typename T>
class C
{
    // ensure that T is not void (ignoring const or volatile):
    static_assert(!std::is_same_v<std::remove_cv_t<T>,void>,
"invalid instantiation of class C for void  type");
    public:
        template<typename V>
        void f(V&& v) {
            if constexpr(std::is_reference_v<T>) {
                … // special code if T is a reference type
            }
            if constexpr(std::is_convertible_v<std::decay_t<V>,T>) {
                … // special code if V is convertible to T
            }
            if constexpr(std::has_virtual_destructor_v<V>) {
                … // special code if V has virtual destructor
```

```
            }
        }
};
```

如上所示，通过检查某些条件,可以在模板的不同实现之间做选择。在这里用到了编译期的 if 特性，该特性从 C++17 开始可用，作为替代选项，这里也可以使用 std::enable_if、部分特例化或者 SFINAE（参见第 8 章）。

但是使用类型萃取的时候需要额外小心：其行为可能和程序员的预期不同。比如：

```
std::remove_const_t<int const&> // yields int const&
```

这里由于引用不是 const 类型的（虽然你不可以改变它），这个操作不会有任何效果。

这样，删除引用和删除 const 的顺序就很重要了：

```
std::remove_const_t<std::remove_reference_t<int const&>> // int
std::remove_reference_t<std::remove_const_t<int const&>> // int const
```

另一种方法是，直接调用：

```
std::decay_t<int const&> // yields int
```

但是这同样会让裸数组和函数类型退化为相应的指针类型。

当然还有一些类型萃取的使用是有要求的。这些要求不被满足的话，其行为将是未定义的。比如：

```
make_unsigned_t<int> // unsigned int
make_unsigned_t<int const&> // undefined behavior (hopefully error)
```

某些情况下，结果可能会让你很意外。比如：

```
add_rvalue_reference_t<int const> // int const&&
add_rvalue_reference_t<int const&> // int const& (lvalueref remains
lvalue-ref)
```

这里我们期望 add_rvalue_reference 总是能够返回一个右值引用，但是 C++中的引用塌缩（reference-collapsing rules，参见 15.6.1 节）会令左值引用和右值引用的组合返回一个左值引用。

另一个例子是：

```
is_copy_assignable_v<int> // yields true (generally, you can assign an
int to an int)
is_assignable_v<int,int> // yields false (can't call 42 = 42)
```

其中 is_copy_assignable 通常只会检查是否能够将一个 int 赋值给另外一个（检查左值的相关操作），而 is_assignable 则会考虑值的种类（value category，会检查是否能将一个右值赋值给另外一个）。也就是说第一个语句等效于：

```
is_assignable_v<int&,int&> // yields true
```

对下面的例子也是这样：

```
is_swappable_v<int> // yields true (assuming lvalues)
is_swappable_v<int&,int&> // yields true (equivalent to the previous
check)
is_swappable_with_v<int,int> // yields false (taking value category
into account)
```

综上，在使用时需要额外注意类型萃取的精确定义。相关规则定义在附录 D 中。

11.2.2 std::addressoff()

函数模板 std::addressof<>()会返回一个对象或者函数的准确地址。即使一个对象重载了运算符&也是这样。虽然后者中的情况很少遇到，但是也会发生（比如在智能指针中）。因此，如果需要获得任意类型的对象的地址，那么推荐使用 addressof()：

```
template<typename T>
void f (T&& x)
{
    auto p = &x; // might fail with overloaded operator &
    auto q = std::addressof(x); // works even with overloaded operator
&
    ...
}
```

11.2.3 std::declval()

函数模板 std::declval()可以被用作某一类型的对象的引用的占位符。该函数模板没有定义，因此不能被调用（也不会创建对象）。因此它只能被用作不会被计算的操作数（比如 decltype 和 sizeof）。也因此，在不创建对象的情况下，依然可以假设有相应类型的可用对象。

比如在如下例子中，会基于模板参数 T1 和 T2 推断出返回类型 RT：

```
#include <utility>
template<typename T1, typename T2,
typename RT = std::decay_t<decltype(true ? std::declval<T1>() :
std::declval<T2>())>>
RT max (T1 a, T2 b)
{
    return b < a ? a : b;
}
```

为了避免在调用运算符?:的时候不得不去调用 T1 和 T2 的（默认）构造函数，这里使用了

std::declval，这样可以在不创建对象的情况下“使用”它们。不过该方式只能在不会做真正的计算时（比如 decltype）使用。

不要忘了使用 std::decay<>来确保返回类型不会是一个引用，因为 std::declval<>本身返回的是右值引用。否则，类似 max(1,2)这样的调用将会返回一个 int&&类型。相关细节请参见 19.3.4 节。

11.3 完美转发临时变量

正如 6.1 节介绍的那样，我们可以使用转发引用（forwarding reference）以及 std::forward<>来完美转发泛型参数：

```
template<typename T>
void f (T&& t) // t is forwarding reference
{
    g(std::forward<T>(t)); // perfectly forward passed argument t to g()
}
```

但是某些情况下，在泛型代码中我们需要转发一些不是通过参数传递进来的数据。此时我们可以使用 auto &&创建一个可以被转发的变量。比如，假设我们需要相继的调用 get()和 set()两个函数，并且需要将 get()的返回值完美的转发给 set():

```
template<typename T>void foo(T x)
{
    set(get(x));
}
```

假设以后我们需要更新代码对 get()的返回值进行某些操作，可以通过将 get()的返回值存储在一个被声明为 auto &&的变量中实现：

```
template<typename T>
void foo(T x)
{
    auto&& val = get(x);
    …
    // perfectly forward the return value of get() to set():
    set(std::forward<decltype(val)>(val));
}
```

这样可以避免对中间变量的多余拷贝。

11.4 作为模板参数的引用

虽然不是很常见，但是模板参数的类型依然可以是引用类型。

比如：

```
#include <iostream>
template<typename T>
void tmplParamIsReference(T) {
    std::cout << "T is reference: " << std::is_reference_v<T> << ' \n';
}

int main()
{
    std::cout << std::boolalpha;
    int i;
    int& r = i;
    tmplParamIsReference(i); // false
    tmplParamIsReference(r); // false
    tmplParamIsReference<int&>(i); // true
    tmplParamIsReference<int&>(r); // true
}
```

即使传递给 tmplParamIsReference()的参数是一个引用变量，T 依然会被推断为被引用的类型（因为对于引用变量 v，表达式 v 的类型是被引用的类型，表达式（expression）的类型永远不可能是引用类型）。不过我们可以显示指定 T 的类型化为引用类型：

```
tmplParamIsReference<int&>(r);
tmplParamIsReference<int&>(i);
```

这样做可以从根本上改变模板的行为，不过由于这并不是模板最初设计的目的，这样做可能会触发错误或者不可预知的行为。考虑如下例子：

```
template<typename T, T Z = T{}>
class RefMem {
    private:
        T zero;
    public:
        RefMem() : zero{Z} {
    }
};
int null = 0;

int main()
{
    RefMem<int> rm1, rm2;
    rm1 = rm2; // OK
    RefMem<int&> rm3; // ERROR: invalid default value for N
    RefMem<int&, 0> rm4; // ERROR: invalid default value for N extern
int null;
```

```
    RefMem<int&,null> rm5, rm6;
    rm5 = rm6; // ERROR: operator= is deleted due to reference member
}
```

此处模板的模板参数为 T，其非类型模板参数 z 被进行了零初始化。用 int 实例化该模板会获得预期的行为。但是如果尝试用引用对其进行实例化的话，情况就有点复杂了：

- 非模板参数的默认初始化不在可行。
- 不再能够直接用 0 来初始化非参数模板参数。
- 最让人意外的是，赋值运算符也不再可用，因为对于具有非 static 引用成员的类，其默赋值运算符会被删除掉。

而且将引用类型用于非类型模板参数同样会变的复杂和危险。考虑如下例子：

```
#include <vector>
#include <iostream>
template<typename T, int& SZ> // Note: size is reference
class Arr {
    private:
        std::vector<T> elems;
    public:
        Arr() : elems(SZ) { //use current SZ as initial vector size
        }
        void print() const {
            for (int i=0; i<SZ; ++i) { //loop over SZ elements
                std::cout << elems[i] << ' ' ;
            }
        }
};

int size = 10;
int main()
{
    Arr<int&,size> y; // compile-time ERROR deep in the code of class
std::vector<>
    Arr<int,size> x; // initializes internal vector with 10 elements
    x.print(); // OK
    size += 100; // OOPS: modifies SZ in Arr<>
    x.print(); // run-time ERROR: invalid memory access: loops over 120
elements
}
```

其中尝试将 Arr 的元素实例化为引用类型会导致 std::vector<>中很深层次的错误，因为其元素类型不能被实例化为引用类型：

```
Arr<int&,size> y; // compile-time ERROR deep in the code of class
std::vector<>
```

正如 9.4 节介绍的那样，这一类错误通常又臭又长，编译器会报出整个模板实例化过程中所有的错误：从模板一开始实例化的地方，一直到模板定义中真正触发错误的地方。

可能更糟糕的是将引用用于 size 这一类参数导致的运行时错误：可能在容器不知情的情况下，自身的 size 却发生了变化（比如 size 值变得无效）。如下这样使用 size 的操作（比如 print）就很可能会导致未定义的行为（导致程序崩溃甚至更糟糕）：

```
int size = 10;
…
Arr<int,size> x; // initializes internal vector with 10 elements
size += 100; // OOPS: modifies SZ in Arr<>
x.print(); // run-time ERROR: invalid memory access: loops over 120
elements
```

注意这里并不能通过将 SZ 声明为 int const &来修正这一错误，因为 size 本身依然是可变的。

看上去这一类问题根本就不会发生。但是在更复杂的情况下，确实会遇到此类问题。比如在 C++17 中，非类型模板参数可以通过推断得到：

```
template<typename T, decltype(auto) SZ>
class Arr;
```

使用 decltype(auto)很容易得到引用类型，因此在这一类上下文中应该尽量避免使用 auto。详情请参见 15.10.3 节。

基于这一原因，C++标准库在某些情况下制定了很特殊的规则和限制。比如：

- 在模板参数被用引用类型实例化的情况下，为了依然能够正常使用赋值运算符，std::pair<>和 std::tuple<>都没有使用默认的赋值运算符，而是做了单独的定义。比如：

```
namespace std {
template<typename T1, typename T2>
struct pair {
    T1 first;
    T2 second;
    …
    // default copy/move constructors are OK even with references:
    pair(pair const&) = default;
    pair(pair&&) = default;
    …
    // but assignment operator have to be defined to be available with
references:
    pair& operator=(pair const& p);
    pair& operator=(pair&& p) noexcept(…);
        …
    };
}
```

- 由于这些副作用可能导致的复杂性，在 C++17 中用引用类型实例化标准库模板 std::optional<>和 std::variant<>的过程看上去有些古怪。

  为了禁止用引用类型进行实例化，一个简单的 static_assert 就够了：

```
template<typename T>
class optional
{
    static_assert(!std::is_reference<T>::value, "Invalid
instantiation of optional<T> for references");
    …
};
```

通常引用类型和其他类型有很大不同，并且受一些语言规则的限制。这会影响对调用参数的声明（参见第 7 章）以及对类型萃取的定义（参见 19.6.1 节）。

11.5 推迟计算（Defer Evaluation）

在实现模板的过程中，有时候需要面对是否需要考虑不完整类型（参见 10.3.1 节）的问题。考虑如下的类模板：

```
template<typename T>
class Cont {
    private:
        T* elems;
    public:
        …
};
```

到目前为止，该 class 可以被用于不完整类型。这很有用，比如可以让其成员指向其自身的类型。

```
struct Node
{
    std::string value;
    Cont<Node> next; // only possible if Cont accepts incomplete types
};
```

但是，如果使用了某些类型萃取的话，可能就不能将其用于不完整类型了。比如：

```
template<typename T>
class Cont {
    private:
        T* elems;
    public:
        …
        typename
```

```
        std::conditional<std::is_move_constructible<T>::value, T&&,
T& >::type foo();
};
```

这里通过使用 std::conditional（参见 D.5）来决定 foo()的返回类型是 T&&还是 T&。决策标准是看模板参数 T 是否支持 move 语义。

问题在于 std::is_move_constructible 要求其参数必须是完整类型（参见 D。3.2 节）。使用这种类型的 foo()，struct node 的声明就会报错。

为了解决这一问题，需要使用一个成员模板代替现有 foo()的定义，这样就可以将 std::is_move_constructible 的计算推迟到 foo()的实例化阶段：

```
template<typename T>
class Cont {
    private:
        T* elems;
    public:
        template<typename D = T>
        typename
        std::conditional<std::is_move_constructible<D>::value, T&&,
T&>::type foo();
};
```

现在，类型萃取依赖于模板参数 D（默认值是 T），并且编译器会一直等到 foo()被以完整类型（比如 Node）为参数调用时，才会对类型萃取部分进行计算（此时 Node 是一个完整类型，其只有在定义时才是非完整类型）。

11.6 在写泛型库时需要考虑的事情

下面让我们列出在实现泛型库的过程中需要记住的一些事情：

- 在模板中使用转发引用来转发数值（参见 6.1 节）。如果数值不依赖于模板参数，就使用 auto &&（参见 11.3）。
- 如果一个参数被声明为转发引用，并且传递给它一个左值的话，那么模板参数会被推断为引用类型（参见 15.6.2 节或者《Effective Morden C++》）。
- 在需要一个依赖于模板参数的对象的地址的时候，最好使用 std::addressof()来获取地址，这样能避免因为对象被绑定到一个重载了 operator &的类型而导致的意外情况（参见 11.2.2）。
- 对于成员函数，需要确保它们不会比预定义的 copy/move 构造函数或者赋值运算符更能匹配某个调用（参见 6.4 节）。
- 如果模板参数可能是字符串常量，并且不是被按值传递的，那么请考虑使用 std::decay（参见 7.4 节以及附录 D.4）
- 如果你有被用于输出或者即用于输入也用于输出的、依赖于模板参数的调用参数，请为可能的、const 类型的模板参数做好准备（参见 7.2.2 节）。

- 请为将引用用于模板参数的副作用做好准备（参见 11.4 节）。尤其是在你需要确保返回类型不会是引用的时候（参见 7.5 节）。
- 请为将不完整类型用于嵌套式数据结构这一类情况做好准备（参见 11.5 节）。
- 为所有数组类型进行重载，而不仅仅是 T[SZ]（参见 5.4 节）。

11.7 总结

- 可以将函数，函数指针，函数对象，仿函数和 lambdas 作为可调用对象（callables）传递给模板。
- 如果需要为一个 class 重载 operator()，那么就将其声明为 const 的（除非该调用会修改它的状态）。
- 通过使用 std::invoke()，可以实现能够处理所有类型的、可调用对象（包含成员函数）的代码。
- 使用 decltype(auto)来完美转发返回值。
- 类型萃取是可以检查类型的属性和功能的类型函数。
- 当在模板中需要一个对象的地址时，使用 std::addressof().
- 在不经过表达式计算的情况下，可以通过使用 std::declval()创建特定类型的值。
- 在泛型代码中，如果一个对象不依赖于模板参数，那么就使用 auto&&来完美转发它。
- 可以通过模板来延迟表达式的计算（这样可以在 class 模板中支持不完整类型）。

第 18 章　模板的多态性

多态是一种用单个统一的符号将多种特定行为关联起来的能力。多态也是面向对象编程范式的基石，在 C++中它主要由继承和虚函数实现。由于这一机制主要（至少是一部分）在运行期间起作用，因此我们称之为动态多态（dynamic polymorphism）。它也是我们通常在讨论 C++中的简单多态时所指的多态。但是，模板也允许我们用单个统一符号将不同的特定行为关联起来，不过该关联主要发生在编译期间，我们称之为静态多态（static polymorphism）。在本章中我们将探讨这两种形式的多态，并讨论其各自所适用的情况。

第 22 章将讨论一些处理多态的方式，期间也会介绍并讨论一些设计相关的问题。

18.1 动态多态（dynamic polymorphism）

由于历史原因，C++在最开始的时候只支持通过继承和虚函数实现的多态。在此情况下，多态设计的艺术性主要体现在从一些相关的对象类型中提炼出一组统一的功能，然后将它们声明成一个基类的虚函数接口。

这一设计方式的范例之一是一种用来维护多种几何形状、并通过某些方式将其渲染的应用。在这样一种应用中，我们可以发现一个抽线基类（abstract base class，ABC），在其中声明了适用于几何对象的统一的操作和属性。其余适用于特定几何对象的类都从它做了继承（参见图 18.1）：

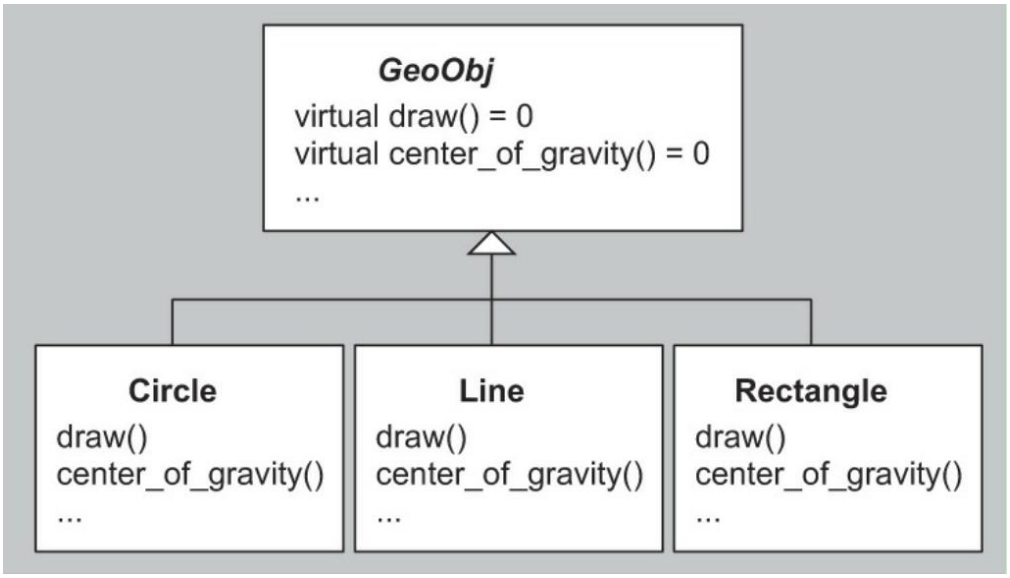

图 18.1　通过继承实现的多态

```
#include "coord.hpp"

// common abstract base class GeoObj for geometric objects
```

```
class GeoObj {
  public:
      // draw geometric object:
      virtual void draw() const = 0;
      // return center of gravity of geometric object:
      virtual Coord center_of_gravity() const = 0;
      ...
      virtual ~GeoObj() = default;
};

// concrete geometric object class Circle
// - derived from GeoObj
class Circle : public GeoObj {
  public:
      virtual void draw() const override;
      virtual Coord center_of_gravity() const override;
  ...
};

// concrete geometric object class Line
// - derived from GeoObj
class Line : public GeoObj {
  public:
      virtual void draw() const override;
      virtual Coord center_of_gravity() const override;
      ...
};
...
```

在创建了具体的对象之后，客户端代码可以通过指向公共基类的指针或者引用，使用虚函数的派发机制来操作它们。在通过基类的指针或者引用调用一个虚函数的时候，所调用的函数将是指针或者引用所指对象的真正类型中的相应函数。

在我们的例子中，具体的代码可以被简写成这样：

```
#include "dynahier.hpp"
#include <vector>
// draw any GeoObj
void myDraw (GeoObj const& obj)
{
  obj.draw(); // call draw() according to type of object
}

// compute distance of center of gravity between two GeoObjs
Coord distance (GeoObj const& x1, GeoObj const& x2)
```

```
{
  Coord c = x1.center_of_gravity() - x2.center_of_gravity();
  return c.abs(); // return coordinates as absolute values
}

// draw heterogeneous collection of GeoObjs
void drawElems (std::vector<GeoObj*> const& elems)
{
  for (std::size_type i=0; i<elems.size(); ++i) {
      elems[i]->draw(); // call draw() according to type of element
  }
}

int main(){
  Line l;
  Circle c, c1, c2;
  myDraw(l); // myDraw(GeoObj&) => Line::draw()
  myDraw(c); // myDraw(GeoObj&) => Circle::draw()
  distance(c1,c2); // distance(GeoObj&,GeoObj&)
  distance(l,c); // distance(GeoObj&,GeoObj&)
  std::vector<GeoObj*> coll; // heterogeneous collection
  coll.push_back(&l); // insert line
  coll.push_back(&c); // insert circle
  drawElems(coll); // draw different kinds of GeoObjs
}
```

关键的多态接口是函数 draw()和 center_of_gravity()。都是虚成员函数。上述例子在函数 mydraw()，distance()，以及 drawElems()中展示了这两个虚函数的用法。而后面这几个函数使用的都是公共基类 GeoObj。这一方式的结果是，在编译期间并不能知道将要被真正调用的函数。但是，在运行期间，则会基于各个对象的完整类型来决定将要调用的函数。因此，取决于集合对象的真正类型，适当的操作将会被执行：如果 mydraw()处理的是 Line 的对象，表达式 obj.draw()将调用 Line::draw()，如果处理的是 Circle 的对象，那么就会调用 Circle::draw()。类似地，对于 distance()，调用的也将是与参数对象对应的 center_of_gravity()。

能够处理异质集合中不同类型的对象，或许是动态多态最吸引人的特性。这一概念在 drawElems()函数中得到了体现：表达式

```
elems[i]->draw()
```

会调用不同的成员函数，具体情况取决于元素的动态类型。

18.2 静态多态

模板也可以被用来实现多态。不同的是，它们不依赖于对基类中公共行为的分解。取而代之

的是，这一“共性（commonality）”隐式地要求不同的“形状（shapes）”必须支持使用了相同语法的操作（比如，相关函数的名字必须相同）。在定义上，具体的 class 之间彼此相互独立（参见 18.2）。在用这些具体的 class 去实例化模板的时候，这一多态能力得以实现。

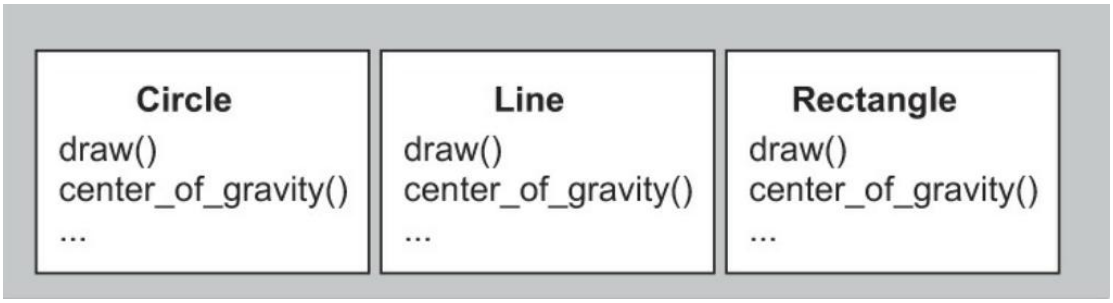

***Figure** 18.2. Polymorphism implemented via templates*

比如，上一节中的 myDraw():

```
void myDraw (GeoObj const& obj) // GeoObj is abstract base
class
{
  obj.draw();
}
```

也可以被实现成下面这样：

```
template<typename GeoObj>
void myDraw (GeoObj const& obj) // GeoObj is template
parameter
{
  obj.draw();
}
```

比较 myDraw()的两种实现，可以发现其主要的区别是将 GeoObj 用作模板参数而不是公共基类。但是，在表象之下还有很多区别。比如，使用动态多态的话，在运行期间只有一个 myDraw()函数，但是在使用模板的情况下，却会有多种不同的函数，例如 myDraw<Line>()和 myDraw<Circle>()。

我们可能希望用 static 多态重新实现上一节中的完整例子。首先，我们不再使用有层级结构的几何类，而是直接使用一些彼此独立的几何类：

```
#include "coord.hpp"
// concrete geometric object class Circle
// - not derived from any class
class Circle {
  public:
      void draw() const;
      Coord center_of_gravity() const;
```

```
        …
};

// concrete geometric object class Line
// - not derived from any class
class Line {
  public:
      void draw() const;
      Coord center_of_gravity() const;
      …
};
…
```

现在，可以像下面这样使用这些类：

```
#include "statichier.hpp"
#include <vector>
// draw any GeoObj
template<typename GeoObj>
void myDraw (GeoObj const& obj)
{
  obj.draw(); // call draw() according to type of object
}

// compute distance of center of gravity between two GeoObjs
template<typename GeoObj1, typename GeoObj2>
Coord distance (GeoObj1 const& x1, GeoObj2 const& x2)
{
  Coord c = x1.center_of_gravity() - x2.center_of_gravity();
  return c.abs(); // return coordinates as absolute values
}

// draw homogeneous collection of GeoObjs
template<typename GeoObj>
void drawElems (std::vector<GeoObj> const& elems)
{
  for (unsigned i=0; i<elems.size(); ++i) {
      elems[i].draw(); // call draw() according to type of element
  }
}

int main()
{
  Line l;
  Circle c, c1, c2;
```

```
    myDraw(l); // myDraw<Line>(GeoObj&) => Line::draw()
    myDraw(c); // myDraw<Circle>(GeoObj&) =>
    Circle::draw()
    distance(c1,c2); //distance<Circle,Circle>
    (GeoObj1&,GeoObj2&)
    distance(l,c); // distance<Line,Circle>(GeoObj1&,GeoObj2&)
    // std::vector<GeoObj*> coll; //ERROR: no heterogeneous
    collection possible
    std::vector<Line> coll; // OK: homogeneous collection
    possible
    coll.push_back(l); // insert line
    drawElems(coll); // draw all lines
}
```

和 myDraw()类似，我们不能够再将 GeoObj 作为具体的参数类型用于 distance()。我们引入了两个模板参数，GeoObj1 和 GeoObj2，来支持不同类型的集合对象之间的距离计算：

```
distance(l,c); // distance<Line,Circle>(GeoObj1&,GeoObj2&)
```

但是使用这种方式，我们将不再能够透明地处理异质容器。这也正是 static 多态中的 static 部分带来的限制：所有的类型必须在编译期可知。不过，我们可以很容易的为不同的集合对象类型引入不同的集合。这样就不再要求集合的元素必须是指针类型，这对程序性能和类型安全都会有帮助。

18.3 动态多态 VS 静态多态

让我们来对这两种多态性形式进行分类和比较。

术语

Static 和 dynamic 多态提供了对不同 C++编程术语的支持：

- 通过继承实现的多态是有界的（bounded）和动态的（dynamic）：
  - 有界的意思是，在设计公共基类的时候，参与到多态行为中的类型的相关接口就已经确定（该概念的其它一些术语是侵入的（invasive 和 intrusive））。
  - 动态的意思是，接口的绑定是在运行期间执行的。

- 通过模板实现的多态是无界的（unbounded）和静态的（static）：
  - 无界的意思是，参与到多态行为中的类型的相关接口是不可预先确定的（该概念的其它一些术语是非侵入的（noninvasive 和 nonintrusive））
  - 静态的意思是，接口的绑定是在编译期间执行的

因此，严格来讲，在 C++中，动态多态和静态多态分别是有界动态多态和无界静态多态的缩

写。在其它语言中还会有别的组合（比如在 Smakktalk 中的无界动态多态）。但是在 C++语境中，更简洁的动态多态和静态多态也不会带来困扰。

优点和缺点

C++中的动态多态有如下优点：

- 可以很优雅的处理异质集合。
- 可执行文件的大小可能会比较小（因为它只需要一个多态函数，不像静态多态那样，需要为不同的类型进行各自的实例化）。
- 代码可以被完整的编译；因此没有必须要被公开的代码（在发布模板库时通常需要发布模板的源代码实现）。

作为对比，下面这些可以说是 C++中 static 多态的优点：

- 内置类型的集合可以被很容易的实现。更通俗地说，接口的公共性不需要通过公共基类实现。
- 产生的代码可能会更快（因为不需要通过指针进行重定向，先验的（priori）非虚函数通常也更容易被 inline）。
- 即使某个具体类型只提供了部分的接口，也可以用于静态多态，只要不会用到那些没有被实现的接口即可。

通常认为静态多态要比动态多态更类型安全（type safe），因为其所有的绑定都在编译期间进行了检查。例如，几乎不用担心将一个通过模板实例化得到的、类型不正确的对象插入到一个已有容器中（编译期间会报错）。但是，对于一个存储了指向公共基类的指针的容器，其所存储的指针却有可能指向一个不同类型的对象。

在实际中，当相同的接口后面隐藏着不同的语义假设时，模板实例化也会带来一些问题。比如，当关联运算符 operator +被一个没实现其所需的关联操作的类型实例化时，就会遇到错误。在实际中，对于基于继承的设计层次，很少会遇到这一类的语义不匹配，这或许是因为相应的接口规格得到了较好的说明。

结合两种多态形式

当然，我们也可以结合这两种多态形式。比如，为了能够操作集合对象的异质集合，你可以从一个公共基类中派生出不同的集合对象。而且，你依然可以使用模板为某种形式的集合对象书写代码。

在第 21 章会更详细地讨论继承和模板的结合问题。我们会看到，一个成员函数的虚拟性是如何被参数化的，以及我们是如何通过 curiously recurring template pattern（CRTP）为 static 多态提供额外的灵活性的。

18.4 使用 concepts

针对使用了模板的静态多态的一个争议是，接口的绑定是通过实例化相应的模板执行的。也就是说没有可供编程的公共接口或者公共 class。取而代之的是，如果所有实例化的代码都是有效的，那么对模板的任何使用也都是有效的。否则，就会导致难以理解的错误信息，或者是产生了有效的代码却导致了意料之外的行为。

基于这一原因，C++语言的设计者们一直在致力于实现一种能够为模板参数显式地提供（或者是检查）接口的能力。在 C++中这一接口被称为 concept。它代表了为了能够成功的实例化模板，模板参数必须要满足的一组约束条件。

尽管已经在这一领域耕耘了很多年，concept 却依然没有被纳入 C++17。不过目前已有一些编译器对这一特性做了实验性的支持，它也很可能会被纳入到紧随 C++17 之后的标准中。

Concept 可以被理解成静态多态的一类“接口”。在我们的例子中，可能会像下面这样：

```
#include "coord.hpp"
template<typename T>
concept GeoObj = requires(T x) {
  { x.draw() } -> void;
  { x.center_of_gravity() } -> Coord;
  ...
};
```

在这里我们使用关键字 concept 定义了一个 GeoObj concept，它要求一个类型要有可被调用的成员函数 draw()和 center_of_gravity()，同时也对它们的返回类型做了限制。

现在我们可以重写样例模板中的一部分代码，以在其中使用 requires 子句要求模板参数满足 GeoObj concept：

```
#include "conceptsreq.hpp"
#include <vector>
// draw any GeoObj
template<typename T>
requires GeoObj<T>
void myDraw (T const& obj)
{
  obj.draw(); // call draw() according to type of object
}

// compute distance of center of gravity between two GeoObjs
template<typename T1, typename T2>
requires GeoObj<T1> && GeoObj<T2>
Coord distance (T1 const& x1, T2 const& x2)
{
  Coord c = x1.center_of_gravity() - x2.center_of_gravity();
```

```
  return c.abs(); // return coordinates as absolute values
}

// draw homogeneous collection of GeoObjs
template<typename T>
requires GeoObj<T>
void drawElems (std::vector<T> const& elems)
{
  for (std::size_type i=0; i<elems.size(); ++i) {
      elems[i].draw(); // call draw() according to type of element
  }
}
```

对于那些可以参与到静态多态行为中的类型，该方法依然是非侵入的：

```
// concrete geometric object class Circle
// - not derived from any class or implementing any interface
class Circle {
  public:
      void draw() const;
      Coord center_of_gravity() const;
      ...
};
```

也就是说，这一类类型的定义中依然不包含特定的基类，或者 require 子句，而且它们也依然可以是基础数据类型或者来自独立框架的类型。

附录 E 中包含了对于 C++中的 concept 更为详细的讨论，因为它们被期望能够出现在下一个 C++标准中。

18.5 新形势的设计模式

C++中的 static 多态给经典的设计模式提供了新的实现方式。以桥接模式为例（bridge pattern，它在很多 C++程序中扮演了重要的角色）。使用桥接模式的一个目的是在不同的接口实现之间做切换。

根据[DesignPatternsGoF]，桥接模式通常是通过使用一个接口类实现的，在这个接口类中包含了一个指向具体实现的指针，然后通过该指针委派所有的函数调用（参见图 18.3）。

但是，如果具体实现的类型在编译期间可知，我们也可以利用模板实现桥接模式（参见图 18.4）。这样做会更类型安全（一部分原因是避免了指针转换），而且性能也会更好。

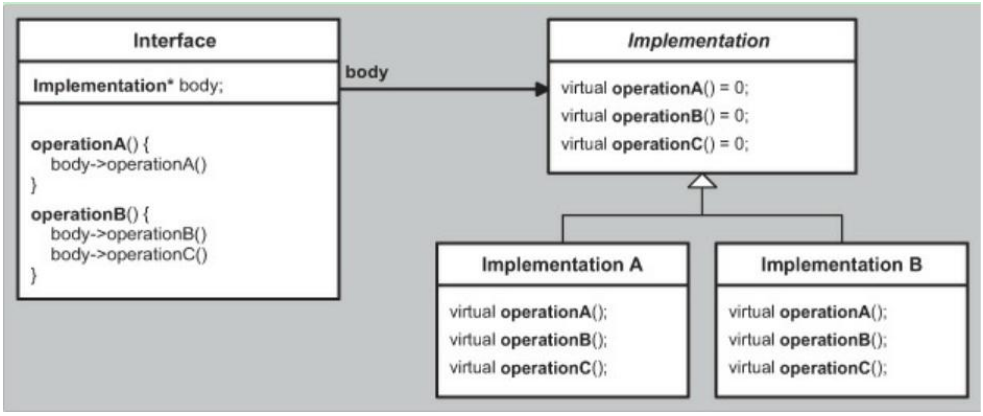

Figure 18.3. Bridge pattern implemented using inheritance

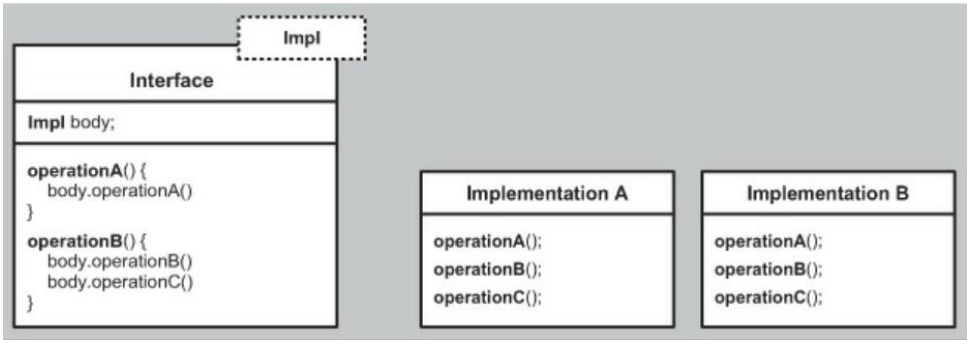

Figure 18.4. Bridge pattern implemented using templates

18.6 泛型编程（Generic Programming）

Static 多态的出现引入了泛型编程的概念。但是，到目前为止并没有一个世所公认的泛型编程的定义（就如同也没有一个世所公认的面向对象编程的定义一样）。根据[CzarneckiEiseneckerGenProg]，定义从针对泛型参数编程（programming with generic parameters）发展到发现有效算法的最佳抽象表达（finding the most abstract representation of efficient algorithms）。概述总结到：

泛型编程是计算机科学的一个分支，它主要处理的问题是寻找高效算法，数据结构，以及其它一些软件概念的抽象表达（结合它们自身的系统组织）。泛型编程专注于域概念族的表达。

在 C++的语境中，泛型编程有时候也被定义成模板编程（而面向对象编程被认为是基于虚函数的编程）。在这个意义上，几乎任何 C++模板的使用都可以被看作泛型编程的实例。但是，开发者通常认为泛型编程还应包含如下这一额外的要素：

该模板必须被定义于一个框架中，且必须能够适用于大量的、有用的组合。

到目前为止，在该领域中最重要的一个贡献是标准模板库（Standard Template Library, STL），随后它也被调整并合并进了 C++标准库（C++ standard library）。STL 是一个框架，它提供了许多有用的操作（称为算法），用于对象集合（称为容器）的一些线性数据结构。算法和容器都是模板。但是，关键的是算法本身并不是容器的成员函数。算法被以一种泛型的方式实

现，因此它们可以用于任意的容器类型（以及线性的元素集合）。为了实现这一目的，STL的设计者们找到了一种可以用于任意线性集合、称之为迭代器（iterators）抽象概念。从本质上来说，容器操作中针对于集合的某些方面已经被分解到迭代器的功能中。

这样，我们就可以在不知道元素的具体存储方式的情况下，实现一种求取序列中元素最大值的方法：

```
template<typename Iterator>
Iterator max_element (Iterator beg, //refers to start of collection
  Iterator end) //refers to end of collection
{
  // use only certain Iterator operations to traverse all elements
  // of the collection to find the element with the maximum value
  // and return its position as Iterator
  …
}
```

这样就可以不用去给所有的线性容器都提供一些诸如 max_element（）的操作，容器本身只要提供一个能够遍历序列中数值的迭代器类型，以及一些能够创建这些迭代器的成员函数就可以了：

```
namespace std {
  template<typename T, …>
  class vector {
     public:
        using const_iterator = …; // implementation-specific iterator
        … // type for constantvectors
        const_iterator begin() const; // iterator for start of
collection
        const_iterator end() const; // iterator for end of collection
        …
  };

  template<typename T, …>
  class list {
     public:
        using const_iterator = …; // implementation-specific iterator
           … // type for constant lists
           const_iterator begin() const; // iterator for start of
collection
           const_iterator end() const; // iterator for end of
collection
           …
  };
}
```

现在就可以通过调用泛型操作 max_element()（以容器的 beginning 和 end 伟参数）来寻找任意集合中的最大值了（省略了对空集合的处理）：

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include "MyClass.hpp"
template<typename T>
void printMax (T const& coll){
  // compute position of maximum value
  auto pos = std::max_element(coll.begin(),coll.end());
  // print value of maximum element of coll (if any):
  if (pos != coll.end()) {
      std::cout << *pos << ' \n' ;
  }
  else {
      std::cout << "empty" << ' \n' ;
  }
}

int main()
{
  std::vector<MyClass> c1;
  std::list<MyClass> c2;
  ...
  printMax(c1);
  printMax(c2);
}
```

通过用这些迭代器来参数化其操作，STL 避免了相关操作在定义上的爆炸式增长。我们并没有为每一种容器都把每一种操作定义一遍，而是只为一个算法进行一次定义，然后将其用于所有的容器。泛型的关键是迭代器，它由容器提供并被算法使用。这样之所以可行，是因为迭代器提供了特定的、可以被算法使用的接口。这些接口通常被称为 concept，它代表了为了融入该框架，模板必须满足的一组限制条件。此外，该概念还可用于其它一些操作和数据结构。

你可能会想起我们曾在第 18.4 节介绍过一个叫做 concept 的语言特性（更多的细节可以参见附录 E），而事实上，该语言特性刚好和这里的概念对应。实际上在当前上下文中，concept 这个名词最早是由 STL 的设计者为了规范化它们的代码而引入的概念。在那之后，我们开始努力使这些概念在我们的模板中明确化。

接下来的语言特性将帮助我们指出（以及检查）对迭代器的要求（因为有很多种迭代器类型，比如 forward 和 bidirectional 迭代器，因此也就会引入多种对应的 concept，参见 E.3.1）。不过在当今的 C++中，在泛型库（尤其是 C++标准库）的规格中这些 concept 通常都是隐式的。

幸运的是，确实有一些特性和技术（比如 static_assert 和 SFINAE）允许我们进行一定数量的自动化检查。

原则上，类似于 STL 方法的一类功能都可以用动态多态实现。但是在实际中，由于迭代器的 concept 相比于虚函数的调用过于轻量级，因此多态这一方法的用途有限。基于虚函数添加一个接口层，很可能会将我们的操作性能降低一个数量级（甚至更多）。

泛型编程之所以实用，正是因为它依赖于静态多态，这样就可以在编译期间就决定具体的接口。另一方面，需要在编译期间解析出接口的这一要求，又催生出了一些与面向对象设计原则（object oriented principles）不同的新原则。这些泛型设计原则（generic design principles）中最重要的一部分将会在本书剩余的章节中介绍。另外，附录 E 通过描述对 concept 概念的直接语言支持，将泛型编程当作一种开发范式进行了深入分析。

18.7 后记

处理容器类型是在 C++编程语言中引入模板的主要动机。在模板之前，多态层级是一种流行的容器方法。一个很流行的例子是 National Institutes of Health Class Library（NIHCL），它在很大程度上借鉴了 Smalltalk 中容器类的层级结构。

// 还没翻译完

第 19 章　萃取的实现

模板允许我们用多种类型对类和函数进行参数化。如果能够通过引入尽可能多的模板参数，去尽可能多的支持某种类型或者算法的各个方面，那将是一件很吸引人的事情。这样，我们“模板化”的代码就可以被针对客户的某种具体需求进行实例化。但是从实际的角度来看，我们并不希望引入过多的模板参数来实现最大化的参数化。因为让用户在客户代码中指出所有的模板参数是一件很繁杂的事情，而且每增加一个额外的模板参数，都会使模板和客户代码之间的协议变得更加复杂。

幸运的是，时间证明大部分被引入的额外的模板参数都有合理的默认值。在一些情况下，额外的模板参数可以完全由很少的、主要的模板参数决定，我们接下来会看到，这一类额外的模板参数可以被一起省略掉。在大多数情况下，其它一些参数的默认值可以从主模板参数得到，但是默认值需要偶尔被重载（针对特殊应用）。当然也有一些参数和主模板参数无关：在某种意义上，它们是其自身的主模板参数，只不过它们有适用于大多数情况的默认值。

萃取（或者叫萃取模板，traits/traits template）是 C++编程的组件，它们对管理那些在设计工业级应用模板时所需要管理的多余参数很有帮助。在本章中，我们会展示一些可以证明该类技术很有帮助的情况，同时也会展示各种各样的、可以让你写出更可靠也更为强大的工具的技术。

本章所展示的大部分萃取技术在 C++标准库中都有其对应的存在。但是，为了更为清晰，我们会将实现简化，删除只有在工业级实现中才会用到的细节。因此，我们会按照我们的命名规则来命名这些技术，但是你应该可以很容易的把它们和标准库对应起来。

19.1 一个例子：对一个序列求和

计算一个序列中所有元素的和是一个很常规的任务。也正是这个简单的问题，给我们提供了一个很好的、可以用来介绍各种不同等级的萃取应用的例子。

19.1.1 固定的萃取（Fixed Traits）

让我们先来考虑这样一种情况：待求和的数据存储在一个数组中，然后我们有一个指向数组中第一个元素的指针，和一个指向最后一个元素的指针。由于本书介绍的是模板，我们自然也希望写出一个适用于各种类型的模板。下面是一个看上去很直接的实现：

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
template<typename T>
T accum (T const* beg, T const* end)
```

```
{
  T total{}; // assume this actually creates a zero value
  while (beg != end) {
      total += *beg;
      ++beg;
  }
  return total;
}
#endif //ACCUM_HPP
```

例子中唯一有些微妙的地方是，如何创建一个类型正确的零值（zero value）来作为求和的起始值。此处我们使用了在第 5.2 节介绍的值初始化（value initialization，用到了{...}符号）。这就意味着这个局部的 total 对象要么被其默认值初始化，要么被零（zero）初始化（对应指针是用 nullptr 初始化，对应 bool 值是用 false 初始化）。

为了引入我们的第一个萃取模板，考虑下面这一个使用了 accum()的例子：

```
#include "accum1.hpp"
#include <iostream>
int main()
{
  // create array of 5 integer values
  int num[] = { 1, 2, 3, 4, 5 };
  // print average value
  std::cout << "the average value of the integer values is " << accum(num,
num+5) / 5 << ' \n' ;
  // create array of character values
  char name[] = "templates";
  int length = sizeof(name)-1;
  // (try to) print average character value
  std::cout << "the average value of the characters in \"" << name <<
"\" is " << accum(name, name+length) / length << ' \n' ;
}
```

在例子的前半部分，我们用 accum()对 5 个整型遍历求和：

```
int num[] = { 1, 2, 3, 4, 5 };
…
accum(num0, num+5)
```

接着就可以用这些变量的和除变量的数目得到平均值。

例子的第二部分试图为单词“templates”中所有的字符做相同的事情。结果应该是 a 到 z 之间的某一个值。在当今的大多数平台上，这个值都是通过 ASCII 码决定的：a 被编码成 97，z 被编码成 122。因此我们可能会期望能够得到一个介于 97 和 122 之间的返回值。但是在我们的平台上，程序的输出却是这样的：

```
the average value of the integer values is 3
the average value of the characters in "templates" is -5
```

问题在于我们的模板是被 char 实例化的，其数值范围即使是被用来存储相对较小的数值的和也是不够的。很显然，为了解决这一问题我们应该引入一个额外的模板参数 AccT，并将其用于返回值 total 的类型。但是这会给模板的用户增加负担：在调用这一模板的时候，他们必须额外指定一个类型。对于上面的例子，我们可能需要将其写称这个样子：

```
accum<int>(name,name+5)
```

这并不是一个过于严苛的要求，但是确实是可以避免的。

一个可以避免使用额外的模板参数的方式是，在每个被用来实例化 accum()的 T 和与之对应的应该被用来存储返回值的类型之间建立某种联系。这一联系可以被认为是 T 的某种属性。正如下面所展示的一样，可以通过模板的偏特化建立这种联系：

```
template<typename T>
struct AccumulationTraits;

template<>
struct AccumulationTraits<char> {
  using AccT = int;
};

template<>
struct AccumulationTraits<short> {
  using AccT = int;
};

template<>
struct AccumulationTraits<int> {
  using AccT = long;
};

template<>
struct AccumulationTraits<unsigned int> {
  using AccT = unsigned long;
};

template<>
struct AccumulationTraits<float> {
  using AccT = double;
};
```

AccumulationTraits 模板被称为萃取模板，因为它是提取了其参数类型的特性。（通常而言可以有不只一个萃取，也可以有不只一个参数）。我们选择不对这一模板进行泛型定义，因为

在不了解一个类型的时候，我们无法为其求和的类型做出很好的选择。但是，可能有人会辩解说 T 类型本身就是最好的待选类型（很显然对于我们前面的例子不是这样）。

有了这些了解之后，我们可以将 accum()按照下面的方式重写：

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include "accumtraits2.hpp"
template<typename T>
auto accum (T const* beg, T const* end)
{
  // return type is traits of the element type
  using AccT = typename AccumulationTraits<T>::AccT;
  AccT total{}; // assume this actually creates a zero value
  while (beg != end) {
      total += *beg;
      ++beg;
  }
  return total;
}
#endif //ACCUM_HPP
```

此时程序的输出就和我们所预期一样了：

```
the average value of the integer values is 3
the average value of the characters in "templates" is 108
```

考虑到我们为算法加入了很好的检查机制，总体而言这些变化不算太大。而且，如果要将 accum()用于新的类型的话，只要对 AccumulationTraits 再进行一次显式的偏特化，就会得到一个 AccT。值得注意的是，我们可以为任意类型进行上述操作：基础类型，声明在其它库中的类型，以及其它诸如此类的类型。

19.1.2 值萃取（Value Traits）

到目前为止我们看到的萃取，代表的都是特定“主”类型的额外的类型信息。在本节我们将会看到，这一“额外的信息”并不仅限于类型信息。还可以将常量以及其它数值类和一个类型关联起来。

在最原始的 accum()模板中，我们使用默认构造函数对返回值进行了初始化，希望将其初始化为一个类似零（zero like）的值：

```
AccT total{}; // assume this actually creates a zero value
...
return total;
```

很显然，这并不能保证一定会生成一个合适的初始值。因为 AccT 可能根本就没有默认构造

函数。

萃取可以再一次被用来救场。对于我们的例子，我们可以为 AccumulationTraits 添加一个新的值萃取（value trait，似乎翻译成值特性会更好一些）：

```
template<typename T>
struct AccumulationTraits;

template<>
struct AccumulationTraits<char> {
  using AccT = int;
  static AccT const zero = 0;
};

template<>
struct AccumulationTraits<short> {
  using AccT = int;
  static AccT const zero = 0;
};

template<>
struct AccumulationTraits<int> {
  using AccT = long;
  static AccT const zero = 0;
};

...
```

在这个例子中，新的萃取提供了一个可以在编译期间计算的，const 的 zero 成员。此时，accum() 的实现如下：

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include "accumtraits3.hpp"
template<typename T>
auto accum (T const* beg, T const* end)
{
  // return type is traits of the element type
  using AccT = typename AccumulationTraits<T>::AccT;
  AccT total = AccumulationTraits<T>::zero; // init total by trait value
  while (beg != end) {
      total += *beg;
      ++beg;
  }
  return total;
} #
```

```
endif // ACCUM_HPP
```

在上述代码中，存储求和结果的临时变量的初始化依然很直观：

```
AccT total = AccumulationTraits<T>::zero;
```

这一实现的一个不足之处是，C++只允许我们在类中对一个整形或者枚举类型的 static const 数据成员进行初始化。

Constexpr 的 static 数据成员会稍微好一些，允许我们对 float 类型以及其它字面值类型进行类内初始化：

```
template<>
struct AccumulationTraits<float> {
  using Acct = float;
  static constexpr float zero = 0.0f;
};
```

但是无论是 const 还是 constexpr 都禁止对非字面值类型进行这一类初始化。比如，一个用户定义的任意精度的 BigInt 类型，可能就不是字面值类型，因为它可能会需要将一部分信息存储在堆上（这会阻碍其成为一个字面值类型），或者是因为我们所需要的构造函数不是 constexpr 的。下面这个实例化的例子就是错误的：

```
class BigInt {
    BigInt(long long);
      ...
};

...

template<>
struct AccumulationTraits<BigInt> {
  using AccT = BigInt;
  static constexpr BigInt zero = BigInt{0}; // ERROR: not a literal type
构造函数不是 constexpr 的
};
```

一个比较直接的解决方案是，不再类中定义值萃取（只做声明）：

```
template<>
struct AccumulationTraits<BigInt> {
  using AccT = BigInt;
  static BigInt const zero; // declaration only
};
```

然后在源文件中对其进行初始化，像下面这样：

```
BigInt const AccumulationTraits<BigInt>::zero = BigInt{0};
```

这样虽然可以工作，但是却有些麻烦（必须在两个地方同时修改代码），这样可能还会有些低效，因为编译期通常并不知晓在其它文件中的变量定义。

在 C++17 中，可以通过使用 inline 变量来解决这一问题：

```
template<>
struct AccumulationTraits<BigInt> {
  using AccT = BigInt;
  inline static BigInt const zero = BigInt{0}; // OK since C++17
};
```

在 C++17 之前的另一种解决办法是，对于那些不是总是生成整型值的值萃取，使用 inline 成员函数。同样的，如果成员函数返回的是字面值类型，可以将该函数声明为 constexpr 的。

比如，我们可以像下面这样重写 AccumulationTraits：

```
template<typename T>
struct AccumulationTraits;

template<>
struct AccumulationTraits<char> {
  using AccT = int;
  static constexpr AccT zero() {
      return 0;
  }
};

template<>
struct AccumulationTraits<short> {
  using AccT = int;
  static constexpr AccT zero() {
      return 0;
  }
};

template<>
struct AccumulationTraits<int> {
  using AccT = long;
  static constexpr AccT zero() {
      return 0;
  }
};

template<>
struct AccumulationTraits<unsigned int> {
  using AccT = unsigned long;
```

```
    static constexpr AccT zero() {
        return 0;
    }
};

template<>
struct AccumulationTraits<float> {
    using AccT = double;
    static constexpr AccT zero() {
        return 0;
    }
};
...
```

然后针我们自定义的类型对这些萃取进行扩展：

```
template<>
struct AccumulationTraits<BigInt> {
    using AccT = BigInt;
    static BigInt zero() {
        return BigInt{0};
    }
};
```

在应用端，唯一的区别是函数的调用语法（不像访问一个 static 数据成员那么简洁）：

```
AccT total = AccumulationTraits<T>::zero(); // init total by trait
function
```

很明显，萃取可以不只是类型。在我们的例子中，萃取可以是一种能够提供所有在调用 accum()时所需的调用参数的信息的技术。这是萃取这一概念的关键：萃取为泛型编程提供了一种配置（configure）具体元素（通常是类型）的手段。

19.1.3 参数化的萃取

在前面几节中，在 accum()里使用的萃取被称为固定的（fixed），这是因为一旦定义了解耦合萃取，在算法中它就不可以被替换。但是在某些情况下，这一类重写（overriding）行为却又是我们所期望的。比如，我们可能碰巧知道某一组 float 数值的和可以被安全地存储在一个 float 变量中，而这样做可能又会带来一些性能的提升。

为了解决这一问题，可以为萃取引入一个新的模板参数 AT，其默认值由萃取模板决定：

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include "accumtraits4.hpp"
template<typename T, typename AT = AccumulationTraits<T>>
```

```
auto accum (T const* beg, T const* end)
{
  typename AT::AccT total = AT::zero();
  while (beg != end) {
      total += *beg;
      ++beg;
  }
      return total;
  }
#endif //ACCUM_HPP
```

采用这种方式，一部分用户可以忽略掉额外模板参数，而对于那些有着特殊需求的用户，他们可以指定一个新的类型来取代默认类型。但是可以推断，大部分的模板用户永远都不需要显式的提供第二个模板参数，因为我们可以为第一个模板参数的每一种（通过推断得到的）类型都配置一个合适的默认值。

19.2 萃取还是策略以及策略类（Traits versus Policies and Policies Classes）

到目前为止我们并没有区分累积（accumulation）和求和（summation）。但是我们也可以相像其它种类的累积。比如，我们可以对一组数值求积。或者说，如果这些值是字符串的话，我们可以将它们连接起来。即使是求一个序列中最大值的问题，也可以转化成一个累积问题。在所有这些例子中，唯一需要变得的操作是 accum()中的 total += *beg。我们可以称这一操作为累积操作的一个策略（policy）。

下面是一个在 accum()中引入这样一个策略的例子：

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include "accumtraits4.hpp"
#include "sumpolicy1.hpp"
template<typename T,
typename Policy = SumPolicy,
typename Traits = AccumulationTraits<T>>
auto accum (T const* beg, T const* end)
{
  using AccT = typename Traits::AccT;
  AccT total = Traits::zero();
  while (beg != end) {
      Policy::accumulate(total, *beg);
      ++beg;
  }
  return total;
```

```
}
#endif //ACCUM_HPP
```

在这一版的 accum()中，SumPolicy 是一个策略类，也就是一个通过预先商定好的接口，为算法实现了一个或多个策略的类。SumPolicy 可以被实现成下面这样：

```
#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP
class SumPolicy {
public:
  template<typename T1, typename T2>
  static void accumulate (T1& total, T2 const& value) {
      total += value;
  }
};
#endif //SUMPOLICY_HPP
```

如果提供一个不同的策略对数值进行累积的话，我们可以计算完全不同的事情。比如考虑下面这个程序，它试图计算一组数值的乘积：

```
#include "accum6.hpp"
#include <iostream>
class MultPolicy {
  public:
  template<typename T1, typename T2>
  static void accumulate (T1& total, T2 const& value) {
      total *= value;
  }
};

int main()
{
  // create array of 5 integer values
  int num[] = { 1, 2, 3, 4, 5 };
  // print product of all values
  std::cout << "the product of the integer values is " <<
accum<int,MultPolicy>(num, num+5) << ' \n' ;
}
```

但是这个程序的输出却和我们所期望的有所不同：

```
the product of the integer values is 0
```

问题出在我们对初始值的选取：虽然 0 能很好的满足求和的需求，但是却不适用于求乘积（初始值 0 会让乘积的结果也是 0）。这说明不同的萃取和策略可能会相互影响，也恰好强调了仔细设计模板的重要性。

在这种情况下，我们可能会认识到，累积循环的初始值应该是累计策略的一部分。这个策略可以使用也可以不使用其 zero()萃取。其它一些方法也应该被记住：不是所有的事情都要用萃取和策略才能够解决的。比如，C++标准库中的 std::accumulate()就将其初始值当作了第三个参数。

19.2.1 萃取和策略：有什么区别？（Traits and Policies: What's the Difference?）

可以设计一个合适的例子来证明策略只是萃取的一个特例。相反地，也可以认为萃取只是编码了一个特定的策略。

新的精简牛津词典（The New Shorter Oxford English Dictionary）有如下表述：

- 萃取 … 一个为物体所特有的属性（a distinctive feature characterizing a thing）。
- 策略 … 任何被作为有益因素或者权宜之计而采取的行动（any course of action adopted as advantageous or expedient）。

基于此，我们倾向于对策略类这一名词的使用做如下限制：它们应该是一些编码了某种与其它模板参数大致独立的行为的类。这和 Alexandrescu 在 Modern C++ Design 中的表述是一致的：

策略和萃取有很多相似之处，只是它们更侧重于行为，而不是类型。

引入了萃取技术的 Nathan Myers 则建议使用如下更为开放的定义：

萃取类：一个用来代替模板参数的类。作为一个类，它整合了有用的类型和常量；作为一个模板，它为实现一个可以解决所有软件问题的“额外的中间层”提供了方法。

总体而言，我们更倾向于使用如下（稍微模糊的）定义：

- 萃取代表的是一个模板参数的本质的、额外的属性。
- 策略代表的是泛型函数和类型（通常都有其常用地默认值）的可以配置的行为。

为了进一步阐明两者之间可能的差异，我们列出了如下和萃取有关的观察结果：

- 萃取在被当作固定萃取（fixed traits）的时候会比较有用（比如，当其不是被作为模板参数传递的时候）。
- 萃取参数通常都有很直观的默认参数（很少被重写，或者简单的说是不能被重写）。
- 萃取参数倾向于紧密的依赖于一个或者多个主模板参数。
- 萃取在大多数情况下会将类型和常量结合在一起，而不是成员函数。
- 萃取倾向于被汇集在萃取模板中。

对于策略类，我们有如下观察结果：

- 策略类如果不是被作为模板参数传递的话，那么其作用会很微弱。
- 策略参数不需要有默认值，它们通常是被显式指定的（虽有有些泛型组件通常会使用默认策略）。
- 策略参数通常是和其它模板参数无关的。

- 策略类通常会包含成员函数。
- 策略可以被包含在简单类或者类模板中。

但是，两者之间并没有一个清晰的界限。比如，C++标准库中的字符萃取就定义了一些函数行为（比如比较，移动和查找字符）。通过替换这些萃取，我们定义一个大小写敏感的字符类型，同时又可以保留相同的字符类型。因此，虽然它们被称为萃取，但是它们的一些属性和策略确实有联系的。

19.2.2 成员模板还是模板模板参数？（Member Templates versus Template Template Parameters）

为了实现累积策略（accumulation policy），我们选择将 SumPolicy 和 MultPolicy 实现为有成员模板的常规类。另一种使用类模板设计策略类接口的方式，此时就可以被当作模板模板参数使用（template template arguments，参见第 5.7 节和第 12.2.3 节）。比如，我们可以将 SumPolicy 重写为如下模板：

```
#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP
template<typename T1, typename T2>
class SumPolicy {
  public:
      static void accumulate (T1& total, T2 const& value) {
          total += value;
      }
};
#endif //SUMPOLICY_HPP
```

此时就可以调整 Accum，让其使用一个模板模板参数：

```
#ifndef ACCUM_HPP#define ACCUM_HPP
#include "accumtraits4.hpp"
#include "sumpolicy2.hpp"
template<typename T,
template<typename,typename> class Policy = SumPolicy,
  typename Traits = AccumulationTraits<T>>
auto accum (T const* beg, T const* end)
{
  using AccT = typename Traits::AccT;
  AccT total = Traits::zero();
  while (beg != end) {
      Policy<AccT,T>::accumulate(total, *beg);
      ++beg;
  }
  return total;
```

```
}
#endif//ACCUM_HPP
```

相同的转化也可以被用于萃取参数。（这一主题的其它变体也是有可能的：比如，相比于显式的将 Acct 的类型传递给策略类型，传递累积萃取并通过策略从萃取参数中获取类型，可能会更有优势一些。）

通过模板模板参数访问策略类的主要优势是，让一个策略类通过一个依赖于模板参数的类型携带一些状态信息会更容易一些（比如 static 数据成员）。（在我们的第一个方法中，static 数据成员必须要被嵌入到一个成员类模板中。）

但是，模板模板参数方法的一个缺点是，策略类必须被实现为模板，而且模板参数必须和我们的接口所定义的参数一样。这可能会使萃取本身的表达相比于非模板类变得更繁琐，也更不自然。

19.2.3 结合多个策略以及/或者萃取（Combining Multiple Policies and/or Traits）

正如我们的实现所展现的那样，萃取以及策略并不会完全摒除多模板参数的情况。但是，它们确实将萃取和模板的数量降低到了易于管理的水平。然后就有了一个很有意思的问题，该如何给这些模板参数排序？

一个简单的策略是，根据参数默认值被选择的可能型进行递增排序（也就是说，越是有可能使用一个参数的默认值，就将其排的越靠后）。比如说，萃取参数通常要在策略参数后面，因为在客户代码中，策略更可能被重写。（善于观察的读者应该已经在我们的代码中发现了这一规律。）

如果我们不介意增加代码的复杂性的话，还有一种可以按照任意顺序指定非默认参数的方法。具体请参见第 21.4 节。

19.2.4 通过普通迭代器实现累积（Accumulation with General Iterators）

在结束萃取和策略的介绍之前，最好再看下另一个版本的 accum()的实现，在该实现中添加了处理泛化迭代器的能力（不再只是简单的指针），这是为了支持工业级的泛型组件。有意思的是，我们依然可以用指针来调用这一实现，因为 C++标准库提供了迭代器萃取。此时我们就可以像下面这样定义我们最初版本的 accum()了（请暂时忽略后面的优化）：

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
```

```
#include <iterator>
template<typename Iter>
auto accum (Iter start, Iter end)
{
  using VT = typename std::iterator_traits<Iter>::value_type;
  VT total{}; // assume this actually creates a zero value
  while (start != end) {
      total += *start;
      ++start;
  }
  return total;
}
#endif //ACCUM_HPP
```

这里的 std::iterator_traits 包含了所有迭代器相关的属性。由于存在一个针对指针的偏特化，这些萃取可以很方便的被用于任意常规的指针类型。标准库对这一特性的支持可能会像下面这样：

```
namespace std {
    template<typename T>
    struct iterator_traits<T*> {
      using difference_type = ptrdiff_t;
      using value_type = T;
      using pointer = T*;
      using reference = T&;
      using iterator_category = random_access_iterator_tag ;
  };
}
```

但是，此时并没有一个适用于迭代器所指向的数值的累积的类型；因此我们依然需要设计自己的 AccumulationTraits。

19.3 类型函数（Type Function）

最初的示例说明我们可以基于类型定义行为。传统上我们在 C 和 C++里定义的函数可以被更明确的称为值函数（value functions）：它们接收一些值作为参数并返回一个值作为结果。对于模板，我们还可以定义类型函数（type functions）：它们接收一些类型作为参数并返回一个类型或者常量作为结果。

一个很有用的内置类型函数是 sizeof，它返回了一个代表了给定类型大小（单位是 byte）的常数。类模板依然可以被用作类型函数。此时类型函数的参数是模板参数，其结果被提取为成员类型或者成员常量。比如，sizeof 运算符可以被作为如下接口提供：

```
#include <cstddef>
#include <iostream>
```

```
template<typename T>
struct TypeSize {
  static std::size_t const value = sizeof(T);
};

int main()
{
  std::cout << "TypeSize<int>::value = " << TypeSize<int>::value << '
\n' ;
}
```

这看上去可能没有那么有用，因为我们已经有了一个内置的 sizeof 运算符，但是请注意此处的 TypeSize<T>是一个类型，它可以被作为类模板参数传递。或者说，TypeSize 是一个模板，也可以被作为模板模板参数传递。

在接下来的内容中，我们设计了一些更为通用的类型函数，可以按照上述方式将它们用作萃取类。

19.3.1 元素类型（Element Type）

假设我们有很多的容器模板，比如 std::vector<>和 std::list<>，也可以包含内置数组。我们希望得到这样一个类型函数，当给的一个容器类型时，它可以返回相应的元素类型。这可以通过偏特化实现：

```
#include <vector>
#include <list>
template<typename T>
struct ElementT; // primary template

template<typename T>
struct ElementT<std::vector<T>> { //partial specialization for
std::vector
  using Type = T;
};

template<typename T>
struct ElementT<std::list<T>> { //partial specialization for std::list
  using Type = T;
};

...

template<typename T, std::size_t N>
struct ElementT<T[N]> { //partial specialization for arrays of known
```

```
bounds
  using Type = T;
};

template<typename T>
struct ElementT<T[]> { //partial specialization for arrays of unknown
bounds
  using Type = T;
};

...
```

注意此处我们应该为所有可能的数组类型提供偏特化（详见第 5.4 节）。

我们可以想下面这样使用这些类型函数：

```
#include "elementtype.hpp"
#include <vector>
#include <iostream>
#include <typeinfo>
template<typename T>
void printElementType (T const& c)
{
std::cout << "Container of " <<
typeid(typename ElementT<T>::Type).name() << " elements.\n";
}

int main()
{
  std::vector<bool> s;
  printElementType(s);
  int arr[42];
  printElementType(arr);
}
```

偏特化的使用使得我们可以在容器类型不知道具体类型函数存在的情况下去实现类型函数。但是在某些情况下，类型函数是和其所适用的类型一起被设计的，此时相关实现就可以被简化。比如，如果容器类型定义了 value_type 成员类型（标准库容器都会这么做），我们就可以有如下实现：

```
template<typename C>
  struct ElementT {
  using Type = typename C::value_type;
};
```

这个实现可以是默认实现，它不会排除那些针对没有定义成员类型 value_type 的容器的偏特

化实现。

虽然如此，我们依然建议为类模板的类型参数提供相应的成员类型定义，这样在泛型代码中就可以更容易的访问它们（和标准库容器的处理方式类似）。下面的代码体现了这一思想：

```
template<typename T1, typename T2, …>
class X {
  public:
      using … = T1;
      using … = T2;
      …
};
```

那么类型函数的作用体现在什么地方呢？它允许我们根据容器类型参数化一个模板，但是又不需要提供代表了元素类型和其它特性的参数。比如，相比于使用

```
template<typename T, typename C>
T sumOfElements (C const& c);
```

这一需要显式指定元素类型的模板（sumOfElements<int> list），我们可以定义这样一个模板：

```
template<typename C>
typename ElementT<C>::Type sumOfElements (C const& c);
```

其元素类型是通过类型函数得到的。

注意观察萃取是如何被实现为已有类型的扩充的；也就是说，我们甚至可以为基本类型和封闭库的类型定义类型函数。

在上述情况下，ElementT 被称为萃取类，因为它被用来访问一个已有容器类型的萃取（通常而言，在这样一个类中可以有多个萃取）。因此萃取类的功能并不仅限于描述容器参数的特性，而是可以描述任意“主参数”的特性。

为了方便，我们可以伟类型函数创建一个别名模板。比如，我们可以引入：

```
template<typename T>
using ElementType = typename ElementT<T>::Type;
```

这可以让 sumOfEkements 的定义变得更加简单：

```
template<typename C>
ElementType<C> sumOfElements (C const& c);
```

19.3.2 转换萃取（Transformation Traits）

除了可以被用来访问主参数类型的某些特性，萃取还可以被用来做类型转换，比如为某个类型添加或移除引用、const 以及 volatile 限制符。

删除引用

比如，我们可以实现一个 RemoveReferenceT 萃取，用它将引用类型转换成其底层对象或者函数的类型，对于非引用类型则保持不变：

```
template<typename T>
struct RemoveReferenceT {
  using Type = T;
};

template<typename T>
struct RemoveReferenceT<T&> {
  using Type = T;
};

template<typename T>
struct RemoveReferenceT<T&&> {
  using Type = T;
};
```

同样地，引入一个别名模板可以简化上述萃取的使用：

```
template<typename T>
using RemoveReference = typename RemoveReference<T>::Type;
```

当类型是通过一个有时会产生引用类型的构造器获得的时候，从一个类型中删除引用会很有意义，比如对于在第 15.6 节介绍的关于函数参数类型 T&&的特殊推断规则。

C++标准库提供了一个相应的 std::remove_reference<>萃取，详见附录 D.4。

添加引用

我们也可以给一个已有类型添加左值或者右值引用：

```
template<typename T>
struct AddLValueReferenceT {
  using Type = T&;
};

template<typename T>
using AddLValueReference = typename AddLValueReferenceT<T>::Type;

template<typename T>
struct AddRValueReferenceT {
  using Type = T&&;
```

```
};

template<typename T>
using AddRValueReference = typename AddRValueReferenceT<T>::Type;
```

引用折叠的规则在这一依然适用（参见第 15.6 节）。比如对于 AddLValueReference<int &&>，返回的类型是 int&，因为我们不需要对它们进行偏特化实现。

如果我们只实现 AddLValueReferenceT 和 AddRValueReferenceT，而又不对它们进行偏特化的话，最方便的别名模板可以被简化成下面这样：

```
template<typename T>
using AddLValueReferenceT = T&;

template<typename T>
using AddRValueReferenceT = T&&;
```

此时不通过类模板的实例化就可以对其进行实例化（因此称得上是一个轻量级过程）。但是这样做是由风险的，因此我们依然希望能够针对特殊的情况对这些模板进行特例化。比如，如果适用上述简化实现，那么我们就不能将其用于 void 类型。一些显式的特化实现可以被用来处理这些情况：

```
template<>
struct AddLValueReferenceT<void> {
  using Type = void;
};

template<>
struct AddLValueReferenceT<void const> {
  using Type = void const;
};

template<>
struct AddLValueReferenceT<void volatile> {
  using Type = void volatile;
};

template<>
struct AddLValueReferenceT<void const volatile> {
  using Type = void const volatile;
};
```

AddRValueReferenceT 的情况与之类似。

有了这些偏特化之后，上文中的别名模板必须被实现为类模板的形式（不能适用最简单的那种形式），这样才能保证相应的篇特换在需要的时候被正确选取（因为别名模板不能被特化）。

C++ 标准库中也提供了与之相应的类型萃取：std::add_lvalue_reference<> 和 std::add_rvalue_reference<>，在附录 D.4 中对它们有专门的介绍。该标准模板也包含了对 void 类型的特化。

移除限制符

转换萃取可以分解或者引入任意种类的复合类型，并不仅限于引用。比如，如果一个类型中存在 const 限制符，我们可以将其移除：

```
template<typename T>
struct RemoveConstT {
  using Type = T;
};

template<typename T>
struct RemoveConstT<T const> {
  using Type = T;
};

template<typename T>
using RemoveConst = typename RemoveConstT<T>::Type;
```

而且，转换萃取可以是多功能的，比如创建一个可以被用来移除 const 和 volatile 的 RemoveCVT 萃取：

```
#include "removeconst.hpp"
#include "removevolatile.hpp"
template<typename T>
struct RemoveCVT : RemoveConstT<typename RemoveVolatileT<T>::Type>
{
};

template<typename T>
using RemoveCV = typename RemoveCVT<T>::Type;
```

RemoveCVT 中有两个需要注意的地方。第一个需要注意的地方是，它同时使用了 RemoveConstT 和相关的 RemoveVolitleT，首先移除类型中可能存在的 volatile，然后将得到了类型传递给 RemoveConstT。第二个需要注意的地方是，它没有定义自己的和 RemoveConstT 中 Type 类似的成员，而是通过使用元函数转发（metafunction forwarding）从 RemoveConstT 中继承了 Type 成员。这里元函数转发被用来简单的减少 RemoveCVT 中的类型成员。但是，即使是对于没有为所有输入都定义了元函数的情况，元函数转发也会很有用，在第 19.4 节中会进一步介绍这一技术。

RemoveCVT 的别名模板可以被进一步简化成：

```
template<typename T>
using RemoveCV = RemoveConst<RemoveVolatile<T>>;
```

同样地，这一简化只适用于 RemoveCVT 没有被特化的情况。但是和 AddLValueReference 以及 AddRValueReference 的情况不同的是，我们想不出一种对其进行特化的原因。

C++ 标准库也提供了与之对应的 std::remove_volatile<>，std::remove_const<>，以及 std::remove_cv<>。在附录 D.4 中有对它们的讨论。

退化（Decay）

为了使对转换萃取的讨论变得更完整，我们接下来会实现一个模仿了按值传递参数时的类型转化行为的萃取。该类型转换继承自 C 语言，这意味着参数类型会发生退化（数组类型退化成指针类型，函数类型退化成指向函数的指针类型），而且会删除相应的顶层 const，volatile 以及引用限制符（因为在解析一个函数调用时，会会忽略掉参数类型中的顶层限制符）。

下面的程序展现了按值传递的效果，它会打印出经过编译器退化之后的参数类型：

```
#include <iostream>
#include <typeinfo>
#include <type_traits>
template<typename T>
void f(T)
{}

template<typename A>
void printParameterType(void (*)(A))
{
  std::cout << "Parameter type: " << typeid(A).name() << ' \n' ;
  std::cout << "- is int: " <<std::is_same<A,int>::value << ' \n' ;
  std::cout << "- is const: " <<std::is_const<A>::value << ' \n' ;
  std::cout << "- is pointer: " <<std::is_pointer<A>::value << ' \n' ;
}

int main()
{
  printParameterType(&f<int>);
  printParameterType(&f<int const>);
  printParameterType(&f<int[7]>);
  printParameterType(&f<int(int)>);
}
```

在程序的输出中，除了 int 参数保持不变外，其余 int const，int[7]，以及 int(int)参数分别退化成了 int，int*，以及 int(*)(int)。

我们可以实现一个与之功能类似的萃取。为了和 C++标准库中的 std::decay 保持匹配，我们称之为 DecayT。它的实现结合了上文中介绍的多种技术。首先我们对非数组、非函数的情况进行定义，该情况只需要删除 const 和 volatile 限制符即可：

```
template<typename T>
struct DecayT : RemoveCVT<T>
{
};
```

然后我们处理数组到指针的退化，这需要用偏特化来处理所有的数组类型（有界和无界数组）：

```
template<typename T>
struct DecayT<T[]> {
  using Type = T*;
};

template<typename T, std::size_t N>
struct DecayT<T[N]> {
  using Type = T*;
};
```

最后来处理函数到指针的退化，这需要应对所有的函数类型，不管是什么返回类型以及有多数参数。为此，我们适用了变参模板：

```
template<typename R, typename… Args>
struct DecayT<R(Args…)> {
  using Type = R (*)(Args…);
};

template<typename R, typename… Args>
struct DecayT<R(Args…, …)> {
  using Type = R (*)(Args…, …);
};
```

注意，上面第二个偏特化可以匹配任意使用了 C-style 可变参数的函数。下面的例子展示了 DecayT 主模板以及其全部四种偏特化的使用：

```
#include <iostream>
#include <typeinfo>
#include <type_traits>
#include "decay.hpp"
template<typename T>
void printDecayedType()
```

```
{
  using A = typename DecayT<T>::Type;
  std::cout << "Parameter type: " << typeid(A).name() << ' \n' ;
  std::cout << "- is int: " << std::is_same<A,int>::value << ' \n' ;
  std::cout << "- is const: " << std::is_const<A>::value << ' \n' ;
  std::cout << "- is pointer: " << std::is_pointer<A>::value << ' \n';
}

int main()
{
  printDecayedType<int>();
  printDecayedType<int const>();
  printDecayedType<int[7]>();
  printDecayedType<int(int)>();
}
```

和往常一样，我们也提供了一个很方便的别名模板：

```
template typename T>
using Decay = typename DecayT<T>::Type;
```

C++标准库也提供了相应的类型萃取 std::decay<>，在附录 D.4 中有相应的介绍。

19.3.3 预测型萃取（Predicate Traits）

到目前为止，我们学习并开发了适用于单个类型的类型函数：给定一个类型，产生另一些相关的类型或者常量。但是通常而言，也可以设计基于多个参数的类型函数。这同样会引出另外一种特殊的类型萃取--类型预测（产生一个 bool 数值的类型函数）。

IsSameT

IsSameT 将判断两个类型是否相同：

```
template<typename T1, typename T2>
struct IsSameT {
  static constexpr bool value = false;
};

template<typename T>
struct IsSameT<T, T> {
  static constexpr bool value = true;
};
```

这里的主模板说明通常我们传递进来的两个类型是不同的，因此其 value 成员是 false。但是，通过使用偏特化，当遇到传递进来的两个相同类型的特殊情况，value 成员就是 true 的。

比如，如下表达式会判断传递进来的模板参数是否是整型：

```
if (IsSameT<T, int>::value) …
```

对于产生一个常量的萃取，我们没法为之定义一个别名模板，但是可以为之定义一个扮演可相同角色的 constexpr 的变量模板：

```
template<typename T1, typename T2>
constexpr bool isSame = IsSameT<T1, T2>::value;
```

C++标准库提供了与之相应的 std::is_same<>，在附录 D.3.3 中有相应的介绍。

true_type 和 false_type

通过为可能的输出结果 true 和 false 提供不同的类型，我们可以大大的提高对 IsSameT 的定义。事实上，如果我们声明一个 BoolConstant 模板以及两个可能的实例 TrueType 和 FalseType：

```
template<bool val>
struct BoolConstant {
  using Type = BoolConstant<val>;
  static constexpr bool value = val;
};

using TrueType = BoolConstant<true>;
using FalseType = BoolConstant<false>;
```

就可以基于两个类型是否匹配，让相应的 IsSameT 分别继承自 TrueType 和 FalseType:

```
#include "boolconstant.hpp"
template<typename T1, typename T2>
struct IsSameT : FalseType{};

template<typename T>
struct IsSameT<T, T> : TrueType{};
```

现在 IsSameT<T, int>的返回类型会被隐式的转换成其基类 TrueType 或者 FalseType，这样就不仅提供了相应的 value 成员，还允许在编译期间将相应的需求派发到对应的函数实现或者类模板的偏特化上。比如：

```
#include "issame.hpp"
#include <iostream>
template<typename T>
void fooImpl(T, TrueType)
{
  std::cout << "fooImpl(T,true) for int called\n";
```

```
}

template<typename T>
void fooImpl(T, FalseType)
{
  std::cout << "fooImpl(T,false) for other type called\n";
}

template<typename T>
void foo(T t)
{
fooImpl(t, IsSameT<T,int>{}); // choose impl. depending on whether T
is int
}

int main()
{
  foo(42); // calls fooImpl(42, TrueType)
  foo(7.7); // calls fooImpl(42, FalseType)
}
```

这一技术被称为标记派发（tag dispatching），在第 20.2 节有相关介绍。

注意在 BoolConstant 的实现中还有一个 Type 成员，这样就可以通过它为 IsSameT 引入一个别名模板：

```
template<typename T>
using isSame = typename IsSameT<T>::Type;
```

这里的别名模板可以和之前的变量模板 isSame 并存。

通常而言，产生 bool 值的萃取都应该通过从诸如 TrueType 和 FalseType 的类型进行派生来支持标记派发。但是为了尽可能的进行泛化，应该只有一个类型代表 true，也应该只有一个类型代表 false，而不是让每一个泛型库都为 bool 型常量定义它自己的类型。

幸运的是，从 C++11 开始 C++ 标准库在<type_traits>中提供了相应的类型：std::true_type 和 std::false_type。在 C++11 和 C++14 中其定义如下：

```
namespace std {
  using true_type = integral_constant<bool, true>;
  using false_type = integral_constant<bool, false>;
}
```

在 C++17 中，其定义如下：

```
namespace std {
  using true_type = bool_constant<true>;
```

```
  using false_type = bool_constant<false>;
}
```

其中 bool_constant 的定义如下：

```
namespace std {
  template<bool B>
  using bool_constant = integral_constant<bool, B>;
}
```

更多细节请参见附录 D1.1。

由于这一原因，在本书接下来的部分，我们将直接使用 std::true_type 和 std::false_type，尤其是在定义类型预测的时候。

19.3.4 返回结果类型萃取（Result Type Traits）

另一个可以被用来处理多个类型的类型函数的例子是返回值类型萃取。在编写操作符模板的时候它们会很有用。为了引出这一概念，我们来写一个可以对两个 Array 容器求和的函数模板：

```
template<typename T>
Array<T> operator+ (Array<T> const&, Array<T> const&);
```

这看上去很好，但是由于语言本身允许我们对一个 char 型数值和一个整形数值求和，我们自然也很希望能够对 Array 也执行这种混合类型（mixed-type）的操作。这样我们就要处理该如何决定相关模板的返回值的问题：

```
template<typename T1, typename T2>
Array<???> operator+ (Array<T1> const&, Array<T2> const&);
```

除了在第 1.3 节介绍的各种方法外，里一个可以解决上述问题的方式就是返回值类型模板：

```
template<typename T1, typename T2>
Array<typename PlusResultT<T1, T2>::Type>
operator+ (Array<T1> const&, Array<T2> const&);
```

如果有便捷别名模板可用的话，还可以将其写称这样：

```
template<typename T1, typename T2>
Array<PlusResult<T1, T2>>
operator+ (Array<T1> const&, Array<T2> const&);
```

其中的 PlusResultT 萃取会自行判断通过+操作符对两种类型（可能是不同类型）的数值求和所得到的类型：

```
template<typename T1, typename T2>
struct PlusResultT {
  using Type = decltype(T1() + T2());
```

```
};

template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

这一萃取模板通过使用 decltype 来计算表达式 T1()+T2()的类型，将决定结果类型这一艰巨的工作（包括处理类型增进规则（promotion rules）和运算符重载）留给了编译器。

但是对于我们的例子而言，decltype 却保留了过多的信息（参见第 15.10.2 节中关于 decltype 行为的介绍）。比如，我们的 PlusResultT 可能会返回一个引用类型，但是我们的 Array 模板却很可能不是为引用类型设计的。更为实际的例子是，重载的 operator+可能会返回一个 const 类型的数值：

```
class Integer { ··· };
Integer const operator+ (Integer const&, Integer const&);
```

对两个 Array<Integer>的值进行求和却得到了一个存储了 Integer const 数值的 Array，这很可能不是我们所期望的结果。事实上我们所期望的是将返回值类型中的引用和限制符移除之后所得到的类型，正如我们在上一小节所讨论的那样：

```
template<typename T1, typename T2>
Array<RemoveCV<RemoveReference<PlusResult<T1, T2>>>>
operator+ (Array<T1> const&, Array<T2> const&);
```

这一萃取的嵌套形式在模板库中很常见，在元编程中也经常被用到。元编程的内容会在第 23 章进行介绍。（便捷别名模板在这一类多层级嵌套中会很有用。如果没有它的话，我们就必须为每一级嵌套都增加一个 typename 和一个::Type。）

到目前为止，数组的求和运算符可以正确地计算出对两个元素类型可能不同的 Array 进行求和的结果类型。但是上述形式的 PlusResultT 却对元素类型 T1 和 T2 施加了一个我们所不期望的限制：由于表达式 T1() + T2()试图对类型 T1 和 T2 的数值进行值初始化，这两个类型必须要有可访问的、未被删除的默认构造函数（或者是非 class 类型）。Array 类本身可能并没有要求其元素类型可以被进行值初始化，因此这是一个额外的、不必要的限制。

declval

好在我们可以很简单的在不需要构造函数的情况下计算+表达式的值，方法就是使用一个可以为一个给定类型 T 生成数值的函数。为了这一目的，C++标准提供了 std::declval<>，在第 11.2.3 节有对其进行介绍。在<utility>中其定义如下：

```
namespace std {
  template<typename T>
  add_rvalue_reference_t<T> declval() noexcept;
}
```

表达式 declval<>可以在不需要使用默认构造函数（或者其它任意操作）的情况下为类型 T 生成一个值。

该函数模板被故意设计成未定义的状态，因为我们只希望它被用于 decltype，sizeof 或者其它不需要相关定义的上下文中。它有两个很有意思的属性：

- 对于可引用的类型，其返回类型总是相关类型的右值引用，这能够使 declval 适用于那些不能够正常从函数返回的类型，比如抽象类的类型（包含纯虚函数的类型）或者数组类型。因此当被用作表达式时，从类型 T 到 T&&的转换对 declval<T>()的行为是没有影响的：其结果都是右值（如果 T 是对象类型的话），对于右值引用，其结果之所以不会变是因为存在引用塌缩（参见第 15.6 节）。
- 在 noexcept 异常规则中提到，一个表达式不会因为使用了 declval 而被认成是会抛出异常的。当 declval 被用在 noexcept 运算符上下文中时，这一特性会很有帮助（参见第 19.7.2 节）。

有了 declval，我们就可以不用在 PlusResultT 中使用值初始化了：

```
#include <utility>
template<typename T1, typename T2>
struct PlusResultT {
  using Type = decltype(std::declval<T1>() + std::declval<T2>());
};

template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

返回值类型萃取提供了一种从特定操作中获取准确的返回值类型的方式，在确定函数模板的返回值的类型的时候，它会很有用。

19.4 基于 SFINAE 的萃取（SFINAE-Based Traits）

SFINAE（参见第 8.4 节和第 15.7 节）会将在模板参数推断过程中，构造无效类型和表达式的潜在错误（会导致程序出现语法错误）转换成简单的推断错误，这样就允许重载解析继续在其它待选项中间做选择。虽然 SFINAE 最开始是被用来避免与函数模板重载相关的伪错误，我们也可以用它在编译期间判断特定类型和表达式的有效性。比如我们可以通过萃取来判断一个类型是否有某个特定的成员，是否支持某个特定的操作，或者该类型本身是不是一个类。

基于 SFINAE 的两个主要技术是：用 SFINAE 排除某些重载函数，以及用 SFINAE 排除某些偏特化。

10.4.1 用 SFINAE 排除某些重载函数

我们触及到的第一个基于 SFINAE 的例子是将 SFINAE 用于函数重载，以判断一个类型是否是

默认可构造的，对于可以默认构造的类型，就可以不通过值初始化来创建对象。也就是说，对于类型 T，诸如 T()的表达式必须是有效的。

一个基础的实现可能会像下面这样：

```
#include "issame.hpp"
template<typename T>
struct IsDefaultConstructibleT {
    private:
        // test() trying substitute call of a default constructor for
        //T passed as U :
        template<typename U, typename = decltype(U())>
        static char test(void*);// test() fallback:

        template<typename>
        static long test(…);
    public:
        static constexpr bool value =
IsSameT<decltype(test<T>(nullptr)), char>::value;
};
```

通过函数重载实现一个基于 SFINAE 的萃取的常规方式是声明两个返回值类型不同的同名（test()）重载函数模板：

```
template<…> static char test(void*);
template<…> static long test(…);
```

第一个重载函数只有在所需的检查成功时才会被匹配到（后文会讨论其实现方式）。第二个重载函数是用来应急的：它会匹配任意调用，但是由于它是通过"…"（省略号）进行匹配的，因此其它任何匹配的优先级都比它高（参见附录 C.2）。

返回值 value 的具体值取决于最终选择了哪一个 test 函数：

```
static constexpr bool value
  = IsSameT<decltype(test<…>(nullptr)), char>::value;
```

如果选择的是第一个 test()函数，由于其返回值类型是 char，value 会被初始化伟 isSame<char, char>，也就是 true。否则，value 会被初始化伟 isSame<long, char>，也就是 false。

现在，到了该处理我们所需要检测的属性的时候了。目标是只有当我们所关心的测试条件被满足的时候，才可以使第一个 test()有效。在这个例子中，我们想要测试的条件是被传递进来的类型 T 是否是可以被默认构造的。为了实现这一目的，我们将 T 传递给 U，并给第一个 test()声明增加一个无名的（dummy）模板参数，该模板参数被一个只有在这一转换有效的情况下才有效的构造函数进行初始化。在这个例子中，我们使用的是只有当存在隐式或者显式的默认构造函数 U()时才有效的表达式。我们对 U()的结果施加了 deltype 操作，这样就可以用其结果初始化一个类型参数了。

第二个模板参数不可以被推断，因为我们不会为之传递任何参数。而且我们也不会为之提供显式的模板参数。因此，它只会被替换，如果替换失败，基于 SFINAE，相应的 test()声明会被丢弃掉，因此也就只有应急方案可以匹配相应的调用。

因此，我们可以像下面这样使用这一萃取：

```
IsDefaultConstructibleT<int>::value //yields true

struct S {
  S() = delete;
};

IsDefaultConstructibleT<S>::value //yields false
```

但是需要注意，我们不能在第一个 test()声明里直接使用模板参数 T：

```
template<typename T>
struct IsDefaultConstructibleT {
  private:
      // ERROR: test() uses T directly:
      template<typename, typename = decltype(T())>
      static char test(void*);
      // test() fallback:
      template<typename>
      static long test(…);
  public:
      static constexpr bool value
          = IsSameT<decltype(test<T>(nullptr)), char>::value;
};
```

但是这样做并不可以，因为对于任意的 T，所有模板参数为 T 的成员函数都会被执行模板参数替换，因此对一个不可以默认构造的类型，这些代码会遇到编译错误，而不是忽略掉第一个 test()。通过将类模板的模板参数 T 传递给函数模板的参数 U，我们就只为第二个 test()的重载创建了特定的 SFINAE 上下文。

另一种基于 SFINAE 的萃取的实现策略

远在 1998 年发布第一版 C++标准之前，基于 SFINAE 的萃取的实现就已经成为了可能。该方法的核心一致都是实现两个返回值类型不同的重载函数模板：

```
template<…> static char test(void*);
template<…> static long test(…);
```

但是，在最早的实现技术中，会基于返回值类型的大小来判断使用了哪一个重载函数（也会用到 0 和 enum，因为在当时 nullptr 和 constexpr 还没有被引入）：

```
enum { value = sizeof(test<···>(0)) == 1 };
```

在某些平台上，sizeof(char)的值可能会等于 sizeof(long)的值。比如，在数字信号处理器（digital signal processors，DSP）或者旧的 Cray 机器上，所有内部的基础类型的大小都可以相同。比如根据定义 sizeof(char) == 1，但是在这些机器上，sizeof(long)，甚至是 sizeof(long long)的值也都是 1。

基于此，我们希望能够确保 test()的返回值类型在所有的平台上都有不同的值。比如，在定义了：

```
using Size1T = char;
using Size2T = struct { char a[2]; };
```

或者：

```
using Size1T = char(&)[1];
using Size2T = char(&)[2];
```

之后，可以像下面这样定义 test()的两个重载版本：

```
template<···> static Size1T test(void*); // checking test()
template<···> static Size2T test(···); // fallback
```

这样，我们要么返回 Size1T，其大小为 1，要么返回 Size2T，在所有的平台上其值都至少是 2。

使用了上述某一种方式的代码目前依然很常见。

但是要注意，传递给 test()的调用参数的类型并不重要。我们所要保证的是被传递的参数和所期望的类型能够匹配。比如，可以将其定义成能够接受整型常量 42 的形式：

```
template<···> static Size1T test(int); // checking test()
template<···> static Size2T test(···); // fallback
···
enum { value = sizeof(test<···>(42)) == 1 };
```

将基于 SFINAE 的萃取变参预测型萃取

正如在第 19.3.3 节介绍的那样，返回 bool 值的萃取，应该返回一个继承自 std::true_type 或者 std::false_type 的值。使用这一方式，同样可以解决在某些平台上 sizeof(char) == sizeof(long)的问题。

为了这一目的，我们需要间接定义一个 IsDefaultConstructibleT。该萃取本身需要继承自一个辅助类的 Type 成员，该辅助类会返回所需的基类。幸运的是，我们可以简单地将 test()的返回值类型用作对应的基类：

```
template<···> static std::true_type test(void*); // checking test()
```

```
template<…> static std::false_type test(…); // fallback
```

然后可以将基类的 Type 成员声明为：

```
using Type = decltype(test<FROM>(nullptr));
```

此时我们也不再需要使用 IsSameT 萃取了。

优化之后，完整的 IsDefaultConstructibleT 的实现如下：

```
#include <type_traits>
template<typename T>
struct IsDefaultConstructibleHelper {
  private:
      // test() trying substitute call of a default constructor for
      T passed as U:
      template<typename U, typename = decltype(U())>
      static std::true_type test(void*);
      // test() fallback:
      template<typename>
      static std::false_type test(…);
  public:
      using Type = decltype(test<T>(nullptr));
};

template<typename T>
struct IsDefaultConstructibleT :
  IsDefaultConstructibleHelper<T>::Type {
};
```

现在，如果第一个 test()函数模板是有效的话，那么它就将是被选择的重载函数，因此成员 IsDefaultConstructibleHelper::Type 会被其返回值类型 std::true_type 初始化。这样的话 IsConvertibleT<>就会继承自 std::true_type。

如果第一个 test() 函数模板是无效的话，那么它就会被 SFINAE 剔除掉，IsDefaultConstructibleHelper::Type 也就会被应急 test() 的返回值类型初始化，也就是 std::false_type。这样的话 IsConvertibleT<>就会继承自 std::false_type。

19.4.2 用 SFINAE 排除偏特化

另一种实现基于 SFINAE 的萃取的方式会用到偏特化。这里，我们同样可以使用上文中用来判断类型 T 是否是可以被默认初始化的例子：

```
#include "issame.hpp"
#include <type_traits> //defines true_type and false_type
```

```
// 别名模板，helper to ignore any number of template parameters:
template<typename …> using VoidT = void;

// primary template:
template<typename, typename = VoidT<>>
struct IsDefaultConstructibleT : std::false_type
{ };

// partial specialization (may be SFINAE’d away):
template<typename T>
struct IsDefaultConstructibleT<T, VoidT<decltype(T())>> :
std::true_type
{ }
;
```

和上文中优化之后的 IsDefaultConstructibleT 预测萃取类似，我们让适用于一般情况的版本继承自 std::false_type，因为默认情况下一个类型没有 size_type 成员。

此处一个比较有意思的地方是，第二个模板参数的默认值被设定为一个辅助别名模板 VoidT。这使得我们能够定义各种使用了任意数量的编译期类型构造的偏特化。

针对我们的例子，只需要一个类型构造：

```
decltype(T())
```

这样就可以检测类型 T 是否是可以被默认初始化的。如果对于某个特定的类型 T，其默认构造函数是无效的，此时 SIFINEAE 就是使该偏特化被丢弃掉，并最终使用主模板。否则该偏特化就是有效的，并且会被选用。

在 C++17 中，C++标准库引入了与 VoidT 对应的类型萃取 std::void_t<>。在 C++17 之前，向上面那样定义我们自己的 std::void_t 是很有用的，甚至可以将其定义在 std 命名空间里：

```
#include <type_traits>
#ifndef __cpp_lib_void_t
namespace std {
  template<typename…> using void_t = void;
}
#endif
```

从 C++14 开始，C++标准委员会建议通过定义预先达成一致的特征宏（feature macros）来标识那些标准库的内容以及被实现了。这并不是标准的强制性要求，但是实现者通常都会遵守这一建议，以为其用户提供方便。__cpp_lib_void_t 就是被建议用来标识在一个库中是否实现了 std::void_t 的宏，所以在上面的 code 中我们将其用于了条件判断。

很显然，这一定义类型萃取的方法看上去要比之前介绍的使用了函数模板重载的方法精简的多。但是该方法要求要能够将相应的条件放进模板参数的声明中。而使用了函数重载的类模

板则使得我们能够使用额外的辅助函数或者辅助类。

19.4.3 将泛型 Lambdas 用于 SFINAE（Using Generic Lambdas for SFINAE）

无论使用哪一种技术，在定义萃取的时候总是需要用到一些样板代码：重载并调用两个 test() 成员函数，或者实现多个偏特化。接下来我们会展示在 C++17 中，如何通过指定一个泛型 lambda 来做条件测试，将样板代码的数量最小化。

作为开始，先介绍一个用两个嵌套泛型 lambda 表达式构造的工具：

```
#include <utility>
// helper: checking validity of f (args…) for F f and Args… args:
template<typename F, typename… Args,
typename = decltype(std::declval<F>() (std::declval<Args&&>()…))>
std::true_type isValidImpl(void*);

// fallback if helper SFINAE'd out:
template<typename F, typename… Args>
std::false_type isValidImpl(…);

// define a lambda that takes a lambda f and returns whether calling
f with args is valid
inline constexpr
auto isValid = [](auto f) {
                return [](auto&&… args) {
                    return decltype(isValidImpl<decltype(f),
decltype(args)&&…>(nullptr)){};
                };
            };


// helper template to represent a type as a value
template<typename T>
struct TypeT {
  using Type = T;
};

// helper to wrap a type as a value
template<typename T>
constexpr auto type = TypeT<T>{};

// helper to unwrap a wrapped type in unevaluated contexts
```

```
template<typename T>
T valueT(TypeT<T>); // no definition needed
```

先从 isValid 的定义开始：它是一个类型为 lambda 闭包的 constexpr 变量。声明中必须要使用一个占位类型（placeholder type，代码中的 auto），因为 C++没有办法直接表达一个闭包类型。在 C++17 之前，lambda 表达式不能出现在 const 表达式中，因此上述代码只有在 C++17 中才有效。因为 isValid 是闭包类型的，因此它可以被调用，但是它被调用之后返回的依然是一个闭包类型，返回结果由内部的 lambda 表达式生成。

在深入讨论内部的 lambda 表达式之前，先来看一个 isValid 的典型用法：

```
constexpr auto isDefaultConstructible
  = isValid([](auto x) -> decltype((void)decltype(valueT(x))() {});
```

我们已经知道 isDefaultConstructible 的类型是闭包类型，而且正如其名字所暗示的那样，它是一个可以被用来测试某个类型是不是可以被默认构造的函数对象。也就是说，isValid 是一个萃取工厂（traits factory）：它会为其参数生成萃取，并用生成的萃取对对象进行测试。

辅助变量模板 type 允许我们用一个值代表一个类型。对于通过这种方式获得的数值 x，我们可以通过使用 decltype(valueT(x))得到其原始类型，这也正是上面被传递给 isValid 的 lambda 所做的事情。如果提取的类型不可以被默认构造，我们要么会得到一个编译错误，要么相关联的声明就会被 SFINAE 掉（得益于 isValid 的具体定义，我们代码中所对应的情况是后者）。

可以像下面这样使用 isDefaultConstructible：

```
isDefaultConstructible(type<int>) //true (int is
defaultconstructible)
isDefaultConstructible(type<int&>)  //false  (references  are  not
default-constructible)
```

为了理解各个部分是如何工作的，先来看看当 isValid 的参数 f 被绑定到 isDefaultConstructible 的泛型 lambda 参数时，isValid 内部的 lambda 表达式会变成什么样子。通过对 isValid 的定义进行替换，我们得到如下等价代码：

```
constexpr auto isDefaultConstructible= [](auto&&… args) {
  return decltype(isValidImpl<decltype([](auto x) ->
decltype((void)decltype(valueT(x))()))>,
                              decltype(args)&&…> (nullptr)){};
};
```

如果我们回头看看第一个 isValidImpl()的定义，会发现它还有一个如下形式的默认模板参数：

```
decltype(std::declval<F>()(std::declval<Args&&>()…))>
```

它试图对第一个模板参数的值进行调用，而这第一个参数正是 isDefaultConstructible 定义中的 lambda 的闭包类型，调用参数为传递给 isDefaultConstructible 的(decltype(args)&&...)类型的值。由于 lambda 中只有一个参数 x，因此 args 就需要扩展成一个

参数：在我们上面的 static_assert 例子中，参数类型伟 TypeT<int>或者 TypeT<int&>。对于 TypeT<int&>的情况，decltype(valueT(x))的结果是 int&，此时 decltype(valueT(x))()是无效的，因此在第一个 isValidImpl()的声明中默认模板参数的替换会失败，从而该 isValidImpl()声明会被 SFINAE 掉。这样就只有第二个声明可用，且其返回值类型为 std::false_type。

整体而言，在传递 type<int&>的时候，`isDefaultConstructible` 会返回 false_type。而如果传递的是 type<int>的话，替换不会失败，因此第一个 isValidImpl()的声明会被选择，返回结果也就是 true_type 类型的值。

会议一下，为了使 SFINAE 能够工作，替换必须发生在被替换模板的立即上下文（immediate context，参见第 15.7.1 节）中。在我们这个例子中，被替换的模板是 isValidImpl 的第一个声明，而且泛型 lambda 的调用运算符被传递给了 isValid。因此，被测试的内容必须出现在 lambda 的返回类型中，而不是函数体中。

我们的 `isDefaultConstructible` 萃取和之前的萃取在实现上有一些不同，主要体现在它需要执行函数形式的调用，而不是指定模板参数。这可能是一种更为刻度的方式，但是也可以按照之前的方式实现：

```
template<typename T>using IsDefaultConstructibleT
  = decltype(isDefaultConstructible(std::declval<T>()));
```

虽然这是传统的模板声明方式，但是它只能出现在 namespace 作用域内，然而 `isDefaultConstructible` 的定义却很可能被在一个块作用域内引入。

到目前为止，这一技术看上去好像并没有那么有竞争力，因为无论是实现中涉及的表达式还是其使用方式都要比之前的技术复杂得多。但是，一旦有了 isValid，并且对其进行了很好的理解，有很多萃取都可以只用一个声明实现。比如，对是否能够访问名为 first 的成员进行测试，就非常简洁（完整的例子请参见 19.6.4 节）：

```
constexpr auto hasFirst
  = isValid([](auto x) -> decltype((void)valueT(x).first) {});
```

19.4.4 SFINAE 友好的萃取

通常，类型萃取应该可以在不使程序出现问题的情况下回答特定的问题。基于 SFINAE 的萃取解决这一难题的方式是“小心地将潜在的问题捕获进一个 SFINAE 上下文中”，将可能出现的错误转变成相反的结果。

但是，到目前为止我们所展示的一些萃取（比如在第 19.3.4 节介绍的 PlusResultT 萃取），在应对错误的时候表现的并不是那么好。会议一下之前关于 PlusResultT 的定义：

```
template<typename T1, typename T2>
struct PlusResultT {
  using Type = decltype(std::declval<T1>() + std::declval<T2>());
};
```

```
template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

在这一定义中，用到+的上下文并没有被 SFINAE 保护。因此，如果程序试着对不支持+运算符的类型执行 PlusResultT 的话，那么 PlusResultT 计算本身就会使成勋遇到错误，比如下面这个例子中，试着为两个无关类型 A 和 B 的数组的求和运算声明返回类型的情况：

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

// declare + for arrays of different element types:
template<typename T1, typename T2>
Array<typename PlusResultT<T1, T2>::Type> operator+ (Array<T1> const&,
Array<T2> const&);
```

很显然，如果没有为数组元素定义合适的+运算符的话，使用 PlusResultT<>就会遇到错误。

```
class A {
};

class B {
};
void addAB(Array<A> arrayA, Array<B> arrayB) {
auto sum = arrayA + arrayB; // ERROR: fails in instantiation of
PlusResultT<A, B>
  ...
}
```

这里的问题并不是错误会发生在代码明显有问题的地方（没办法对元素类型分别为 A 和 B 的数组进行求和），而是错误会发生在对 operator+进行模板参数推断的时候，在很深层次的 PlusResultT<A,B>的实例化中。

这会导致一个很值得注意的结果：即使我们为 A 和 B 的数组重载一个求和函数，程序依然可能会遇到编译错误，because C++ does not specify whether the types in a function template are actually instantiated if another overload would be better：

```
// declare generic + for arrays of different element types:
template<typename T1, typename T2>
Array<typename PlusResultT<T1, T2>::Type>
  operator+ (Array<T1> const&, Array<T2> const&);

// overload + for concrete types:
Array<A> operator+(Array<A> const& arrayA, Array<B> const& arrayB);

void addAB(Array<A> const& arrayA, Array<B> const& arrayB) {
```

```
auto sum = arrayA + arrayB; // ERROR?: depends on whether the compiler
  … // instantiates PlusResultT<A,B>
}
```

如果编译器可以在不对第一个 operator+模板声明进行推断和替换的情况下，就能够判断出第二个 operator+声明会更加匹配的话，上述代码也不会有问题。

但是，在推断或者替换一个备选函数模板的时候，任何发生在类模板定义的实例化过程中的事情都不是函数模板替换的立即上下文（immediate context），SFINAE 也不会保护我们不会在其中构建无效类型或者表达式。此时并不会丢弃这一函数模板待选项，而是会立即报出试图在 PlusResult<>中为 A 和 B 调用 operator + 的错误：

```
template<typename T1, typename T2>
struct PlusResultT {
  using Type = decltype(std::declval<T1>() + std::declval<T2> ());
}
```

为了解决这一问题，我们必须要将 PlusResultT 变成 SFINAR 友好的，也就是说需要为之提供更恰当的定义，以使其即使会在 decltype 中遇到错误，也不会诱发编译错误。

参考在之前章节中介绍的 HassLessT，我们可以通过定义一个 HasPlusT 萃取，来判断给定的类型是有一个可用的+运算符：

```
#include <utility> // for declval
#include <type_traits> // for true_type, false_type, and void_t//
primary template:
template<typename, typename, typename = std::void_t<>>
struct HasPlusT : std::false_type
{};

// partial specialization (may be SFINAE’d away):
template<typename T1, typename T2>
struct HasPlusT<T1, T2, std::void_t<decltype(std::declval<T1>() +
std::declval<T2> ())>>
: std::true_type
{};
```

如果其返回结果为 true，PlusResultT 就可以使用现有的实现。否则，PlusResultT 就需要一个安全的默认实现。对于一个萃取，如果对某一组模板参数它不能生成有意义的结果，那么最好的默认行为就是不为其提供 Type 成员。这样，如果萃取被用于 SFINAE 上下文中（比如之前代码中 array 类型的 operator+的返回值类型），缺少 Type 成员会导致模板参数推断出错，这也正是我们所期望的、array 类型的 operator+模板的行为。

下面这一版 PlusResultT 的实现就提供了上述的行为：

```
#include "hasplus.hpp"
template<typename T1, typename T2, bool = HasPlusT<T1, T2>::value>
```

```
struct PlusResultT { //primary template, used when HasPlusT yields true
  using Type = decltype(std::declval<T1>() + std::declval<T2>());
};

template<typename T1, typename T2>
struct PlusResultT<T1, T2, false> { //partial specialization, used otherwise
};
```

在这一版的实现中，我们引入了一个有默认值的模板参数，它会使用上文中的 HasPlusT 来判断前面的两个模板参数是否支持求和操作。然后我们对于第三个模板参数的值为 false 的情况进行了偏特化，而且在该偏特化中没有任何成员，从而避免了我们所描述过的问题。对与支持求和操作的情况，第三个模板参数的值是 true，因此会选用主模板，也就是定义了 Type 成员的那个模板。这样就保证了只有对支持+操作的类型，PlusResultT 才会提供返回类型。（注意，被进行求和的模板参数不应该被显式的指定参数）

再次考虑 Array<A>和 Array<B>的求和：如果使用最新的 PlusResultT 实现，那么 PlusResultT<A, B>的实例化将不会有 Type 成员，因为不能够对 A 和 B 进行求和。因此对应的 operator+模板的返回值类型是无效的，该函数模板也就会被 SFINAE 掉。这样就会去选择专门为 Array<A>和 Array<B>指定的 operator+的重载版本。

作为一般的设计原则，在给定了合理的模板参数的情况下，萃取模板永远不应该在实例化阶段出错。其实先方式通常是执行两次相关的检查：

1. 一次是检查相关操作是否有效
2. 一次是计算其结果

在 PlusResultT 中我们已经见证了这一原则，在那里我们通过调用 HasPlusT<>来判断 PlusResultImpl<>中对 operator+的调用是否有效。

让我们将这一原则用于在第 19.3.1 节介绍的 ElementT：它从一个容器类型生成该容器的元素类型。同样的，由于其结果依赖于该容器类型所包含的成员类型 value_type，因此主模板应该只有在容器类型包含 value_type 成员的时候，才去定义成员类型 Type：

```
template<typename C, bool = HasMemberT_value_type<C>::value>
struct ElementT {
  using Type = typename C::value_type;
};

template<typename C>
struct ElementT<C, false> {
};
```

第三个能够让萃取变得 SFINAE 友好的例子将在第 19.7.2 节介绍，在那里对于 IsNothrowMoveConstructibleT 我们需要首先检测是否存在游动构造函数，然后检测它是否被声明为 noexcept 的。

19.5 IsConvertibleT

细节很重要。因此基于 SIFINAE 萃取的常规方法在实际中会变得更加复杂。为了展示这一复杂性，我们将定义一个能够判断一种类型是否可以被转化成另外一种类型的萃取，比如当我们期望某个基类或者其某一个子类作为参数的时候。IsConvertibleT 就可以判断其第一个类型参数是否可以被转换成第二个类型参数：

```
#include <type_traits> // for true_type and false_type
#include <utility> // for declval
template<typename FROM, typename TO>
struct IsConvertibleHelper {
  private:
      // test() trying to call the helper aux(TO) for a FROM passed as F :
      static void aux(TO);

      template<typename F, typename T,
      typename = decltype(aux(std::declval<F>()))>
      static std::true_type test(void*);
      // test() fallback:
      template<typename, typename>
      static std::false_type test(…);
  public:
      using Type = decltype(test<FROM>(nullptr));
};

template<typename FROM, typename TO>
struct IsConvertibleT : IsConvertibleHelper<FROM, TO>::Type {
};
template<typename FROM, typename TO>
using IsConvertible = typename IsConvertibleT<FROM, TO>::Type;

template<typename FROM, typename TO>
constexpr bool isConvertible = IsConvertibleT<FROM, TO>::value;
```

这里我们使用了在第 19.4.1 节介绍的函数重载的方法。也就是说，我们在一个辅助类中定义了两个名为 test()的返回值类型不同的重载函数，并为该辅助类声明了 Type 成员类型：

```
template<…> static std::true_type test(void*);
template<…> static std::false_type test(…);
…
using Type = decltype(test<FROM>(nullptr));
…
template<typename FROM, typename TO>
struct IsConvertibleT : IsConvertibleHelper<FROM, TO>::Type {
};
```

和往常一样，第一个 test()只有在所需的检查成功的时候才会被匹配到，第二个 test()则是应急方案。因此问题的关键就是让第一个 test()只有在类型 FROM 可以被转换成 TO 的时候才有效。为了实现这一目的，我们再次给第一个 test()分配了一个 dummy（并且无名）的模板参数，并将其初始化成只有当转换又消失才有效的内容。该模板参数不可以被推断，我们也不会为之提供显式的模板参数。因此它会被替换，而且当替换失败之后，该 test()声明会被丢弃掉。

请再次注意，下面这种声明是不可以的：

```
static void aux(TO);
template<typename = decltype(aux(std::declval<FROM>()))>
static char test(void*);
```

这样当成员函数模板被解析的时候，FROM 和 TO 都已经完全确定了，因此对一组不适合做相应转换的类型，在调用 test()之前就会立即触发错误。

由于这一原因，我们引入了作为成员函数模板参数的 F：

```
static void aux(TO);
template<typename F, typename = decltype(aux(std::declval<F> ()))>
static char test(void*);
```

并在 value 的初始化中将 FROM 类型用作调用 test()时的显式模板参数：

```
static constexpr bool value
  = isSame<decltype(test<FROM>(nullptr)), char>;
```

请注意这里是如何在不调用任何构造函数的情况下，通过使用在第 19.3.4 节介绍的 std::declval 生成一个类型的值的。如果这个值可以被转换成 TO，对 aux()的调用就是有效的，相应的 test()调用也就会被匹配到。否则，会触发 SFINAE 错误，导致应急 test()被调用。

然后，我们就可以像下面这样使用该萃取了：

```
IsConvertibleT<int, int>::value //yields true
IsConvertibleT<int, std::string>::value //yields false
IsConvertibleT<char const*, std::string>::value //yields true
IsConvertibleT<std::string, char const*>::value //yields false
```

处理特殊情况

下面 3 种情况还不能被上面的 IsConvertibleT 正确处理：

1. 向数组类型的转换要始终返回 false，但是在上面的代码中，aux()声明中的类型为 TO 的参数会退化成指针类型，因此对于某些 FROM 类型，它会返回 true。
2. 向指针类型的转换也应该始终返回 false，但是和 1 中的情况一样，上述实现只会将它们当作退化后的类型。
3. 向（被 const/volatile 修饰）的 void 类型的转换需要返回 true。但是不幸的是，在 TO 是

void 的时候，上述实现甚至不能被正确实例化，因为参数类型不能包含 void 类型（而且 aux()的定义也用到了这一参数）。

对于这几种情况，我们需要对它们进行额外的偏特化。但是，为所有可能的与 const 以及 volatile 的组合情况都分别进行偏特化是很不明智的。相反，我们为辅助类模板引入了一个额外的模板参数:

```
template<typename FROM, typename TO, bool = IsVoidT<TO>::value ||
IsArrayT<TO>::value || IsFunctionT<TO>::value>
struct IsConvertibleHelper {
  using Type = std::integral_constant<bool, IsVoidT<TO>::value &&
IsVoidT<FROM>::value>;
};

template<typename FROM, typename TO>
struct IsConvertibleHelper<FROM,TO,false> {
  ··· //previous implementation of IsConvertibleHelper here
};
```

额外的 bool 型模板参数能够保证，对于上面的所有特殊情况，都会最终使用主辅助萃取（而不是偏特化版本）。如果我们试图将 FROM 转换为数组或者函数，或者 FROM 是 void 而 TO 不是，都会得到 false_type 的结果，不过对于 FROM 和 TO 都是 false_type 的情况，它也会返回 false_type。其它所有的情况，都会使第三个模板参数为 false，从而选择偏特化版本的实现（对应于我们之前介绍的实现）。

至于 IsArrayT 和 IsFunctionT 的实现，请分别参见第 19.8.2 节和第 19.8.3 节。

C++标准库中也提供了与之对应的 std::is_convertible<>，具体请参见第 D.3.3 节。

19.6 探测成员（Detecting Members）

另一种对基于 SFINAE 的萃取的应用是，创建一个可以判断一个给定类型 T 是否含有名为 X 的成员（类型或者非类型成员）的萃取。

19.6.1 探测类型成员（Detecting Member Types）

首先定义一个可以判断给定类型 T 是否含有类型成员 size_type 的萃取：

```
#include <type_traits>
// defines true_type and false_type
// helper to ignore any number of template parameters:
template<typename ···> using VoidT = void;
```

```
// primary template:
template<typename, typename = VoidT<>>
struct HasSizeTypeT : std::false_type
{};

// partial specialization (may be SFINAE'd away):
template<typename T>
struct HasSizeTypeT<T, VoidT<typename T::size_type>> : std::true_type
{} ;
```

这里用到了在第 19.4.2 节介绍的剔除偏特化的方法。

和往常已有，对于预测萃取，我们让一般情况派生自 std::false_type，因为某人情况下一个类型是没有 size_type 成员的。

在这种情况下，我们只需要一个条件：

```
typename T::size_type
```

该条件只有在 T 含有类型成员 size_type 的时候才有效，这也正是我们所想要做的。如果对于某个类型 T，该条件无效，那么 SFINAE 会使偏特化实现被丢弃，我们就退回到主模板的情况。否则，偏特化有效并且会被有限选取。

可以像下面这样使用萃取：

```
std::cout << HasSizeTypeT<int>::value; // false

struct CX {
  using size_type = std::size_t;
};

std::cout << HasSizeType<CX>::value; // true
```

需要注意的是，如果类型成员 size_type 是 private 的，HasSizeTypeT 会返回 false，因为我们的萃取模板并没有访问该类型的特殊权限，因此 typename T::size_type 是无效的（触发 SFINAE）。也就是说，该萃取所做的事情是测试我们是否能够访问类型成员 size_type。

处理引用类型

作为编程人员，应该已经很熟悉我们所考虑的主要问题之外的边边角角的问题了。诸如 HasSizeTypeT 一类的萃取，在处理引用类型的时候可能会遇到让人意外的事情。比如，虽然如下的代码可以正常工作：

```
struct CXR {
  using size_type = char&; // Note: type size_type is a reference type
};
```

```
std::cout << HasSizeTypeT<CXR>::value; // OK: prints true
```

但是与之类似的代码却不会输出我们所期望的结果：

```
std::cout << HasSizeTypeT<CXR&>::value; // OOPS: prints false
std::cout << HasSizeTypeT<CXR&&>::value; // OOPS: prints false
```

这或许会让人感到意外。引用类型确实没有成员，但是当我们使用引用的时候，结果表达式的类型是引用所指向的类型，因此我们可能会希望，当我们传递进来的模板参数是引用类型的时候，依然根据其指向的类型做判断。为了这一目的，可以在 HasSizeTypeT 的偏特化中使用我们之前介绍的 RemoveReference 萃取：

```
template<typename T>
struct HasSizeTypeT<T, VoidT<RemoveReference<T>::size_type>> :
std::true_type {
};
```

注入类的名字（Injected Class Names）

同样值得注意的是，对于注入类的名字（参见第 13.2.3 节），我们上述检测类型成员的萃取也会返回 true。比如对于：

```
struct size_type {
};

struct Sizeable : size_type {
};

static_assert(HasSizeTypeT<Sizeable>::value, "Compiler bug: Injected
class name missing");
```

后面的 static_assert 会成功，因为 size_type 会将其自身的名字当作类型成员，而且这一成员会被继承。如果 static_assert 不会成功的话，那么我就发现了一个编译器的问题。

19.6.2 探测任意类型成员

在定义了诸如 HasSizeTypeT 的萃取之后，我们会很自然的想到该如何将该萃取参数化，以对任意名称的类型成员做探测。

不幸的是，目前这一功能只能通过宏来实现，因为还没有语言机制可以被用来描述“潜在”的名字。当前不使用宏的、与该功能最接近的方法是使用泛型 lambda，正如在第 19.6.4 节介绍的那样。

如下的宏可以满足我们的需求：

```
#include <type_traits> // for true_type, false_type, and void_t
#define
DEFINE_HAS_TYPE(MemType) \
template<typename, typename = std::void_t<>> \
struct HasTypeT_##MemType \
: std::false_type {
}; \
template<typename T> \
struct HasTypeT_##MemType<T, std::void_t<typename T::MemType>> \
: std::true_type { } // ; intentionally skipped
```

每一次对 DEFINE_HAS_TYPE(MemberType) 的使用都相当于定义了一个新的 HasTypeT_MemberType 萃取。比如，我们可以用之来探测一个类型是否有 value_type 或者 char_type 类型成员：

```
#include "hastype.hpp"
#include <iostream>
#include <vector>

DEFINE_HAS_TYPE(value_type);
DEFINE_HAS_TYPE(char_type);
int main()
{
  std::cout << "int::value_type: " << HasTypeT_value_type<int>::value
<< ' \n' ;
  std::cout << "std::vector<int>::value_type: " <<
HasTypeT_value_type<std::vector<int>>::value << ' \n' ;
  std::cout << "std::iostream::value_type: " <<
HasTypeT_value_type<std::iostream>::value << ' \n' ;
  std::cout << "std::iostream::char_type: " <<
HasTypeT_char_type<std::iostream>::value << ' \n' ;
}
```

19.6.3 探测非类型成员

可以继续修改上述萃取，以让其能够测试数据成员和（单个的）成员函数：

```
#include <type_traits> // for true_type, false_type, and void_t
#define
DEFINE_HAS_MEMBER(Member) \
template<typename, typename = std::void_t<>> \
struct HasMemberT_##Member \
: std::false_type { }; \
```

```
template<typename T> \

struct HasMemberT_##Member<T,
std::void_t<decltype(&T::Member)>> \
: std::true_type { } // ; intentionally skipped
```

当&::Member 无效的时候，偏特化实现会被 SFINAE 掉。为了使条件有效，必须满足如下条件：

- Member 必须能够被用来没有歧义的识别出 T 的一个成员（比如，它不能是重载成员你函数的名字，也不能是多重继承中名字相同的成员的名字）。
- 成员必须可以被访问。
- 成员必须是非类型成员以及非枚举成员（否则前面的&会无效）。
- 如果 T::Member 是 static 的数据成员，那么与其对应的类型必须没有提供使得&T::Member 无效的 operator&（比如，将 operator&设成不可访问的）。

所有以上条件都满足之后，我们可以像下面这样使用该模板：

```
#include "hasmember.hpp"
#include <iostream>
#include <vector>
#include <utility>
DEFINE_HAS_MEMBER(size);
DEFINE_HAS_MEMBER(first);
int main()
{
  std::cout << "int::size: " << HasMemberT_size<int>::value << ' \n' ;
  std::cout << "std::vector<int>::size: " <<
HasMemberT_size<std::vector<int>>::value << ' \n' ;
  std::cout << "std::pair<int,int>::first: " <<
HasMemberT_first<std::pair<int,int>>::value << ' \n' ;
}
```

修改上面的偏特化实现以排除那些&T::Member 不是成员指针的情况（比如排除 static 数据成员的情况）并不会很难。类似地，也可以限制该偏特化仅适用于数据成员或者成员函数。

探测成员函数

注意，HasMember 萃取只可以被用来测试是否存在“唯一”一个与特定名称对应的成员。如果存在两个同名的成员的话，该测试也会失败，比如当我们测试某些重载成员函数是否存在的时候：

```
DEFINE_HAS_MEMBER(begin);
std::cout << HasMemberT_begin<std::vector<int>>::value; // false
```

但是，正如在第 8.4.1 节所说的那样，SFINAE 会确保我们不会在函数模板声明中创建非法的类型和表达式，从而我们可以使用重载技术进一步测试某个表达式是否是病态的。

也就是说，可以很简单地测试我们能否按照某种形式调用我们所感兴趣的函数，即使该函数被重载了，相关调用可以成功。正如在第 19.5 节介绍的 IsConvertibleT 一样，此处的关键是能否构造一个表达式，以测试我们能否在 decltype 中调用 begin()，并将该表达式用作额外的模板参数的默认值：

```
#include <utility> // for declval
#include <type_traits> // for true_type, false_type, and void_t
// primary template:
template<typename, typename = std::void_t<>>
struct HasBeginT : std::false_type {
};

// partial specialization (may be SFINAE'd away):
template<typename T>
struct HasBeginT<T, std::void_t<decltype(std::declval<T>
().begin())>> : std::true_type {
};
```

这里我们使用 decltype(std::declval<T> ().begin()) 来测试是否能够调用 T 的 begin()。

探测其它的表达式

相同的技术还可以被用于其它的表达式，甚至是多个表达式的组合。比如，我们可以测试对类型为 T1 和 T2 的对象，是否有合适的<运算符可用：

```
#include <utility> // for declval
#include <type_traits> // for true_type, false_type, and void_t
// primary template:
template<typename, typename, typename = std::void_t<>>
struct HasLessT : std::false_type
{};

// partial specialization (may be SFINAE'd away):
template<typename T1, typename T2>
struct HasLessT<T1, T2, std::void_t<decltype(std::declval<T1>() <
std::declval<T2>())>>: std::true_type
{};
```

和往常一样，问题的难点在于该如果为所要测试的条件定义一个有效的表达式，并通过使用 decltype 将其放入 SFINAE 的上下文中，在该表达式无效的时候，SFINAE 机制会让我们最终

选择主模板：

```
decltype(std::declval<T1>() < std::declval<T2>())
```

采用这种方式探测表达式有效性的萃取是很稳健的：如果表达式没有问题，它会返回 true，而如果<运算符有歧义，被删除，或者不可访问的话，它也可以准确的返回 false。

我们可以像下面这样使用萃取：

```
HasLessT<int, char>::value //yields true
HasLessT<std::string, std::string>::value //yields true
HasLessT<std::string, int>::value //yields false
HasLessT<std::string, char*>::value //yields
trueHasLessT<std::complex<double>,
std::complex<double>>::value //yields false
```

正如在第 2.3.1 节介绍的那样，我们也可以通过使用该萃取去要求模板参数 T 必须要支持< 运算符：

```
template<typename T>
class C
{
  static_assert(HasLessT<T>::value, "Class C requires comparable
elements");
  ...
};
```

值得注意的是，基于 std::void_t 的特性，我们可以将多个限制条件放在同一个萃取中：

```
#include <utility> // for declval
#include <type_traits> // for true_type, false_type, and void_t
// primary template:
template<typename, typename = std::void_t<>>
struct HasVariousT : std::false_type
{};

// partial specialization (may be SFINAE'd away):
template<typename T>
struct HasVariousT<T, std::void_t<decltype(
                                  std::declval<T> ().begin()),
                                  typename T::difference_type,
                                  typename T::iterator>> :
                 std::true_type
{};
```

能够测试某一语法特性有效性的萃取是很有用的，基于有或者没有某一特定操作，可以用该萃取去客制化模板的行为。这一类萃取既可以被用于 SFINAE 友好的萃取（第 19.4.4 节）的一部分，也可以为基于类型特性的重载（第 20 章）提供帮助。

19.6.4 用泛型 Lambda 探测成员

在第 19.4.3 节介绍的 isValid lambda，提供了一种定义可以被用来测试成员的更为紧凑的技术，其主要的好处是在处理名称任意的成员时，不需要使用宏。

下面这个例子展示了定义可以检测数据或者类型成员是否存在（比如 first 或者 size_type），或者有没有为两个不同类型的对象定义 operator <的萃取的方式：

```
#include "isvalid.hpp"
#include<iostream>
#include<string>
#include<utility>
int main()
{
  using namespace std;
  cout << boolalpha;
  // define to check for data member first:
  constexpr auto hasFirst = isValid([](auto x) ->
decltype((void)valueT(x).first) {});
  cout << "hasFirst: " << hasFirst(type<pair<int,int>>) << ' \n' ; //
true

  // define to check for member type size_type:
  constexpr auto hasSizeType = isValid([](auto x) -> typename
decltype(valueT(x))::size_type { });

  struct CX {
      using size_type = std::size_t;
  };

  cout << "hasSizeType: " << hasSizeType(type<CX>) << ' \n' ; // true

  if constexpr(!hasSizeType(type<int>)) {
      cout << "int has no size_type\n";
      ...
  }

  // define to check for <:
  constexpr auto hasLess = isValid([](auto x, auto y) ->
decltype(valueT(x) < valueT(y)) {});

  cout << hasLess(42, type<char>) << ' \n' ; //yields true
  cout << hasLess(type<string>, type<string>) << ' \n' ; //yields true
```

```
    cout << hasLess(type<string>, type<int>) << ' \n' ; //yields false
    cout << hasLess(type<string>, "hello") << ' \n' ; //yields true
}
```

请再次注意，hasSizeType 通过使用 std::decay 将参数 x 中的引用删除了，因为我们不能访问引用中的类型成员。如果不这么做，该萃取（对于引用类型）会始终返回 false，从而导致第二个重载的 isValidImpl<>被使用。

为了能够使用统一的泛型语法（将类型用于模板参数），我们可以继续定义额外的辅助工具。比如：

```
#include "isvalid.hpp"
#include<iostream>
#include<string>
#include<utility>
constexpr auto hasFirst
  = isValid([](auto&& x) -> decltype((void)&x.first) {});

template<typename T>
using HasFirstT = decltype(hasFirst(std::declval<T>()));

constexpr auto hasSizeType = isValid([](auto&& x) -> typename
std::decay_t<decltype(x)>::size_type {});

template<typename T>
using HasSizeTypeT = decltype(hasSizeType(std::declval<T>()));

constexpr auto hasLess = isValid([](auto&& x, auto&& y) -> decltype(x
< y) { });

template<typename T1, typename T2>
using HasLessT = decltype(hasLess(std::declval<T1>(),
std::declval<T2>()));

int main()
{
  using namespace std;
  cout << "first: " << HasFirstT<pair<int,int>>::value << ' \n' ;
  // true
  struct CX {
      using size_type = std::size_t;
  };

  cout << "size_type: " << HasSizeTypeT<CX>::value << ' \n' ; // true
  cout << "size_type: " << HasSizeTypeT<int>::value << ' \n' ; // false
```

```
    cout << HasLessT<int, char>::value << ' \n' ; // true
    cout << HasLessT<string, string>::value << ' \n' ; // true
    cout << HasLessT<string, int>::value << ' \n' ; // false
    cout << HasLessT<string, char*>::value << ' \n' ; // true
}
```

现在可以像下面这样使用 HasFirstT：

```
HasFirstT<std::pair<int,int>>::value
```

它会为一个包含两个 int 的 pair 调用 hasFirst，其行为和之前的讨论一致。

19.7 其它的萃取技术

最后让我们来介绍其它一些在定义萃取时可能会用到的方法。

19.7.1 If-Then-Else

在上一小节中，PlusResultT 的定义采用了和之前完全不同的实现方法，该实现方法依赖于另一个萃取（HasPlusT）的结果。我们可以用一个特殊的类型模板 IfThenElse 来表达这一 if-then-else 的行为，它接受一个 bool 型的模板参数，并根据该参数从另外两个类型参数中间做选择：

```
#ifndef IFTHENELSE_HPP
#define IFTHENELSE_HPP
// primary template: yield the second argument by default and rely on
// a partial specialization to yield the third argument
// if COND is false
template<bool COND, typename TrueType, typename FalseType>
struct IfThenElseT {
  using Type = TrueType;
};

// partial specialization: false yields third argument
template<typename TrueType, typename FalseType>
struct IfThenElseT<false, TrueType, FalseType> {
  using Type = FalseType;
};

template<bool COND, typename TrueType, typename FalseType>
using IfThenElse = typename IfThenElseT<COND, TrueType,
FalseType>::Type;
#endif //IFTHENELSE_HPP
```

下面的例子展现了该模板的一种应用，它定义了一个可以为给定数值选择最合适的整形类型的函数：

```
#include <limits>
#include "ifthenelse.hpp"
template<auto N>
struct SmallestIntT {
  using Type =
  typename IfThenElseT<N <= std::numeric_limits<char> ::max(), char,
                       typename IfThenElseT<N <=
std::numeric_limits<short> ::max(), short,
                           typename IfThenElseT<N <=
std::numeric_limits<int> ::max(), int,
                               typename IfThenElseT<N <=
std::numeric_limits<long>::max(), long,
                                   typename IfThenElseT<N <=
std::numeric_limits<long long>::max(), long long, //then
                                       void //fallback
                                          >::Type
                                   >::Type
                               >::Type
                           >::Type
                       >::Type;
};
```

需要注意的是，和常规的 C++ if-then-else 语句不同，在最终做选择之前，then 和 else 分支中的模板参数都会被计算，因此两个分支中的代码都不能有问题，否则整个程序就会有问题。考虑下面这个例子，一个可以为给定的有符号类型生成与之对应的无符号类型的萃取。已经有一个标准萃取（std::make_unsigned）可以做这件事情，但是它要求传递进来的类型是有符号的整形，而且不能是 bool 类型；否则它将使用未定义行为的结果（参见第 D.4 节）。这一萃取不够安全，因此最好能够实现一个这样的萃取，当可能的时候，它就正常返回相应的无符号类型，否则就原样返回被传递进来的类型（这样，当传递进来的类型不合适时，也能避免触发未定义行为）。下面这个简单的实现是不行的：

```
// ERROR: undefined behavior if T is bool or no integral type:
template<typename T>
struct UnsignedT {
  using Type = IfThenElse<std::is_integral<T>::value
&& !std::is_same<T,bool>::value, typename std::make_unsigned<T>::type,
T>;
};
```

因为在实例化 UnsingedT<bool>的时候，行为依然是未定义的，编译期依然会试图从下面的代码中生成返回类型：

```
typename std::make_unsigned<T>::type
```

为了解决这一问题，我们需要再引入一层额外的间接层，从而让 IfThenElse 的参数本身用类型函数去封装结果：

```
// yield T when using member Type:
template<typename T>
struct IdentityT {
  using Type = T;
};

// to make unsigned after IfThenElse was evaluated:
template<typename T>
struct MakeUnsignedT {
  using Type = typename std::make_unsigned<T>::type;
};

template<typename T>
struct UnsignedT {
  using Type = typename IfThenElse<std::is_integral<T>::value
&& !std::is_same<T,bool>::value,
                                   MakeUnsignedT<T>,
                                   IdentityT<T>
                                       >::Type;
};
```

在这一版 UnsignedT 的定义中，IfThenElse 的类型参数本身也都是类型函数的实例。只不过在最终 IfThenElse 做出选择之前，类型函数不会真正被计算。而是由 IfThenElse 选择合适的类型实例（MakeUnsignedT 或者 IdentityT）。最后由::Type 对被选择的类型函数实例进行计算，并生成结果 Type。

此处值得强调的是，之所以能够这样做，是因为 IfThenElse 中未被选择的封装类型永远不会被完全实例化。下面的代码也不能正常工作：

```
template<typename T>
struct UnsignedT {
  using Type = typename IfThenElse<std::is_integral<T>::value
&& !std::is_same<T,bool>::value,
                                   MakeUnsignedT<T>::Type,
                                   T
                                >::Type;
};
```

我们必须要延后对 MakeUnsignedT<T>使用::Type，也就是意味着，我们同样需要为 else 分支中的 T 引入 IdentyT 辅助模板，并同样延后对其使用::Type。

我们同样不能在当前语境中使用如下代码：

```
template<typename T>
  using Identity = typename IdentityT<T>::Type;
```

我们当然可以定义这样一个别名模板，在其它地方它可能也很有用，但是我们唯独不能将其用于 IfThenElse 的定义中，因为任意对 Identity<T>的使用都会立即触发对 IdentityT<T>的完全实例化，不然无法获取其 Type 成员。

在 C++标准库中有与 IfThenElseT 模板对应的模板（std::conditional<>，参见第 D.5 节）。使用这一标准库模板实现的 UnsignedT 萃取如下：

```
template<typename T>
struct UnsignedT {
  using Type = typename std::conditional_t<std::is_integral<T>::value
&& !std::is_same<T,bool>::value,
                                          MakeUnsignedT<T>,
                                          IdentityT<T>
                                        >::Type;
};
```

19.7.2 探测不抛出异常的操作

我们可能偶尔会需要判断某一个操作会不会抛出异常。比如，在可能的情况下，移动构造函数应当被标记成 noexcept 的，意思是它不会抛出异常。但是，某一特定 class 的 move constructor 是否会抛出异常，通常决定于其成员或者基类的移动构造函数会不会抛出异常。比如对于下面这个简单类模板（Pair）的移动构造函数：

```
template<typename T1, typename T2>
class Pair {
  T1 first;
  T2 second;
  public:
      Pair(Pair&& other)
      : first(std::forward<T1>(other.first)),
      second(std::forward<T2>(other.second)) {
      }
};
```

当 T1 或者 T2 的移动操作会抛出异常时，Pair 的移动构造函数也会抛出异常。如果有一个叫做 IsNothrowMoveConstructibleT 的萃取，就可以在 Pair 的移动构造函数中通过使用 noexcept 将这一异常的依赖关系表达出来：

```
Pair(Pair&& other)
noexcept(IsNothrowMoveConstructibleT<T1>::value &&
IsNothrowMoveConstructibleT<T2>::value)
: first(std::forward<T1>(other.first)),
second(std::forward<T2>(other.second))
```

```
{}
```

现在剩下的事情就是去实现 IsNothrowMoveConstructibleT 萃取了。我们可以直接用 noexcept 运算符实现这一萃取，这样就可以判断一个表达式是否被进行 nothrow 修饰了：

```
#include <utility> // for declval
#include <type_traits> // for bool_constant
template<typename T>
struct IsNothrowMoveConstructibleT
: std::bool_constant<noexcept(T(std::declval<T>()))>
{};
```

这里使用了运算符版本的 noexcept，它会判断一个表达式是否会抛出异常。由于其结果是 bool 型的，我们可以直接将它用于 std::bool_constant<>基类的定义（std::bool_constant 也被用来定义 std::true_type 和 sts::false_type）。

但是该实现还应该被继续优化，因为它不是 SFINAE 友好的：如果它被一个没有可用移动或者拷贝构造函数的类型（这样表达式 T(std::declval<T&&>())就是无效的）实例化，整个程序就会遇到问题：

```
class E {
  public:
      E(E&&) = delete;
};
…
std::cout << IsNothrowMoveConstructibleT<E>::value; // compiletime
ERROR
```

在这种情况下，我们所期望的并不是让整个程序奔溃，而是获得一个 false 类型的值。

就像在第 19.4.4 节介绍的那样，在真正做计算之前，必须先对被用来计算结果的表达式的有效性进行判断。在这里，我们要在检查移动构造函数是不是 noexcept 之前，先对其有效性进行判断。因此，我们要重写之前的萃取实现，给其增加一个默认值是 void 的模板参数，并根据移动构造函数是否可用对其进行偏特化：

```
#include <utility> // for declval
#include <type_traits> // for true_type, false_type, and
bool_constant<>
// primary template:
template<typename T, typename = std::void_t<>>
struct IsNothrowMoveConstructibleT : std::false_type
{ };

// partial specialization (may be SFINAE'd away):
template<typename T>
struct IsNothrowMoveConstructibleT<T,
std::void_t<decltype(T(std::declval<T>()))>>
```

```
  : std::bool_constant<noexcept(T(std::declval<T>()))>
  {};
```

如果在偏特化中对 std::void_t<...>的替换有效，那么就会选择该偏特化实现，在其父类中的 noexcept(...)表达式也可以被安全的计算出来。否则，偏特化实现会被丢弃（也不会对其进行实例化），被实例化的也将是主模板（产生一个 std::false_type 的返回值）。

值得注意的是，除非真正能够调用移动构造函数，否则我们无法判断移动构造函数是不是会抛出异常。也就是说，移动构造函数仅仅是 public 和未被标识为 delete 的还不够，还要求对应的类型不能是抽象类（但是抽象类的指针或者引用却可以）。因此，该类型萃取被命名伟 IsNothrowMoveConstructible，而不是 HasNothrowMoveConstructor。对于其它所有的情况，我们都需要编译期支持。

C++标准库提供了与之对应的萃取 std::is_move_constructible<>，在第 D.3.2 节有对其进行介绍。

19.7.3 萃取的便捷性（Traits Convenience）

一个关于萃取的普遍不满是它们相对而言有些繁琐，因为对类型萃取的使用通需要提供一个::Type 尾缀，而且在依赖上下文中（dependent context），还需要一个 typename 前缀，两者几成范式。当同时使用多个类型萃取时，会让代码形式变得很笨拙，就如同在我们的 operator+例子中一样，如果想正确的对其进行实现，需要确保不会返回 const 或者引用类型：

```
  template<typename T1, typename T2>
  Array< typename RemoveCVT<typename RemoveReferenceT<typename
  PlusResultT<T1, T2>::Type >::Type >::Type>
  operator+ (Array<T1> const&, Array<T2> const&);
```

通过使用别名模板（alias templates）和变量模板（variable templates），可以让对产生类型或者数值的萃取的使用变得很方便。但是也需要注意，在某些情况下这一简便方式并不使用，我们依然要使用最原始的类模板。我们已经讨论过一个这一类的例子（MemberPointerToIntT），但是更详细的讨论还在后面。

别名模板和萃取（Alias Templates and Traits）

正如在第 2.8 节介绍的那样，别名模板为降低代码繁琐性提供了一种方法。相比于将类型萃取表达成一个包含了 Type 类型成员的类模板，我们可以直接使用别名模板。比如，下面的三个别名模板封装了之前的三种类型萃取：

```
  template<typename T>
  using RemoveCV = typename RemoveCVT<T>::Type;

  template<typename T>
```

```
using RemoveReference = typename RemoveReferenceT<T>::Type;

template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

有了这些别名模板，我们可以将 operator+的声明简化成：

```
template<typename T1, typename T2>
Array<RemoveCV<RemoveReference<PlusResultT<T1, T2>>>>
operator+ (Array<T1> const&, Array<T2> const&);
```

这一版本的实现明显更简洁，也让人更容易分辨其组成。这一特性使得别名模板非常适用于某些类型萃取。

但是，将别名模板用于类型萃取也有一些缺点：

1. 别名模板不能够被进行特化（在第 16.3 节有过提及），但是由于很多编写萃取的技术都依赖于特化，别名模板最终可能还是需要被重新导向到类模板。
2. 有些萃取是需要由用户进行特化的，比如描述了一个求和运算符是否是可交换的萃取，此时在很多使用都用到了别名模板的情况下，对类模板进行特换会很让人困惑。
3. 对别名模板的使用最会让该类型被实例化（比如，底层类模板的特化），这样对于给定类型我们就很难避免对其进行无意义的实例化（正如在第 19.7.1 节讨论的那样）。

对最后一点的另外一种表述方式是，别名模板不可以和元函数转发一起使用（参见第 19.3.2 节）。

由于将别名模板用于类型萃取既有优点也有缺点，我们建议像我们在本小节以及 C++标准库中那样使用它：同时提供根据遵守特定命名管理的类模板（我们选择使用 T 后缀以及 Type 类型成员）和遵守了稍微不同命名惯例的别名模板（我们丢弃了 T 尾缀），而且让每一个别名模板都基于底层的类模板进行定义。这样，在别名模板能够使代码变得更简洁的地方就是用别名模板，否则，对于更为高阶的用户就让他们使用类模板。

注意，由于某些历史原因，C++标准库选择了不同的命名惯例。其类型萃取会包含一个 type 类型成员，但是不会有特定的后缀（在 C++11 中为某些类型萃取引入了后缀）。从 C++14 开始，为之引入了相应的别名模板（直接生成 type），该别名模板会有一个_t 后缀，因为没有后缀的名字已经被标准化了（参见第 D.1 节）。

变量模板和萃取（Variable Templates and Traits）

对于返回数值的萃取需要使用一个::value（或者类似的成员）来生成萃取的结果。在这种情况下，constexpr 修饰的变量模板（在第 5.6 节有相关介绍）提供了一种简化代码的方法。

比如，下面的变量模板封装了在第 19.3.3 节介绍的 IsSameT 萃取和在第 19.5 节介绍的 IsConvertibleT 萃取：

```
template<typename T1, typename T2>
```

```
constexpr bool IsSame = IsSameT<T1,T2>::value;

template<typename FROM, typename TO>
constexpr bool IsConvertible = IsConvertibleT<FROM, TO>::value;
```

此时我们可以将这一类代码：

```
if (IsSameT<T,int>::value || IsConvertibleT<T,char>::value) …
```

简化成：

```
if (IsSame<T,int> || IsConvertible<T,char>) …
```

同样由于历史原因，C++标准库也采用了不同的命名惯例。产生 result 结果的萃取类模板并没有特殊的后缀，而且它们中的一些在 C++11 中就已经被引入进来了。在 C++17 中引入的与之对应的变量模板则有一个_v 后缀（参见第 D.1 节）。

19.8 类型分类（Type Classification）

在某些情况下，如果能够知道一个模板参数的类型是内置类型，指针类型，class 类型，或者是其它什么类型，将会很有帮助。在接下来的章节中，我们定义了一组类型萃取，通过它们我们可以判断给定类型的各种特性。这样我们就可以单独为特定的某些类型编写代码：

```
if (IsClassT<T>::value) {
  …
}
```

或者是将其用于编译期 if（在 C++17 中引入）以及某些为了萃取的便利性而引入的特性（参见第 19.7.3 节）：

```
if constexpr (IsClass<T>) {
  …
}
```

或者时将其用于偏特化：

```
template<typename T, bool = IsClass<T>>
class C { //primary template for the general case
  …
};

template<typename T>
class C<T, true> { //partial specialization for class types
  …
};
```

此外，诸如 IsPointerT<T>::value 一类的表达式的结果是 bool 型常量，因此它们也将是有效的非类型模板参数。这样，就可以构造更为高端和强大的模板，这些模板可以被基于它们的

类型参数的特性进行特化。

C++标准库定义了一些类似的萃取，这些萃取可以判断一个类型的主要种类或者是该类型被复合之后的种类。更多细节请参见第 D.2.2 节和第 D.2.1 节。

19.8.1 判断基础类型（Determining Fundamental Types）

作为开始，我们先定义一个可以判断某个类型是不是基础类型的模板。默认情况下，我们认为类型不是基础类型，而对于基础类型，我们分别进行了特化：

```
#include <cstddef> // for nullptr_t
#include <type_traits> // for true_type, false_type, and
bool_constant<>
// primary template: in general T is not a fundamental type
template<typename T>
struct IsFundaT : std::false_type {
};
// macro to specialize for fundamental types
#define MK_FUNDA_TYPE(T) \
template<> struct IsFundaT<T> : std::true_type { \
};
MK_FUNDA_TYPE(void)
MK_FUNDA_TYPE(bool)
MK_FUNDA_TYPE(char)
MK_FUNDA_TYPE(signed char)
MK_FUNDA_TYPE(unsigned char)
MK_FUNDA_TYPE(wchar_t)
MK_FUNDA_TYPE(char16_t)
MK_FUNDA_TYPE(char32_t)
MK_FUNDA_TYPE(signed short)
MK_FUNDA_TYPE(unsigned short)
MK_FUNDA_TYPE(signed int)
MK_FUNDA_TYPE(unsigned int)
MK_FUNDA_TYPE(signed long)
MK_FUNDA_TYPE(unsigned long)
MK_FUNDA_TYPE(signed long long)
MK_FUNDA_TYPE(unsigned long long)
MK_FUNDA_TYPE(float)
MK_FUNDA_TYPE(double)
MK_FUNDA_TYPE(long double)
MK_FUNDA_TYPE(std::nullptr_t)
#undef MK_FUNDA_TYPE
```

主模板定义了常规情况。也就是说，通常而言 IfFundaT<T>::value 会返回 false：

```
template<typename T>
struct IsFundaT : std::false_type {
  static constexpr bool value = false;
};
```

对于每一种基础类型，我们都进行了特化，因此 IsFundaT<T>::value 的结果也都会返回 true。为了简单，我们定义了一个可以扩展成所需代码的宏。比如：

```
MK_FUNDA_TYPE(bool)
```

会扩展成：

```
template<> struct IsFundaT<bool> : std::true_type {
  static constexpr bool value = true;
};
```

下面的例子展示了该模板的一种可能的应用场景：

```
#include "isfunda.hpp"
#include <iostream>
template<typename T>
void test (T const&)
{
  if (IsFundaT<T>::value) {
      std::cout << "T is a fundamental type" << ' \n' ;}
  else {
      std::cout << "T is not a fundamental type" << ' \n' ;
  }
}

int main()
{
  test(7);
  test("hello");
}
```

其输出如下：

```
T is a fundamental type
T is not a fundamental type
```

采用同样会的方式，我们也可以定义类型函数 IsIntegralT 和 IsFloatingT 来区分哪些类型是整形标量类型以及浮点型标量类型。

C++标准库采用了一种更为细粒度的方法来测试一个类型是不是基础类型。它先定义了主要的类型种类，每一种类型都被匹配到一个相应的种类（参见第 D.2.1 节），然后合成诸如 std::is_integral 和 std::is_fundamental 类型种类（参见第 D2.2.节）。

19.8.2 判断复合类型

复合类型是由其它类型构建出来的类型。简单的复合类型包含指针类型，左值以及右值引用类型，指向成员的指针类型（pointer-to-member types），和数组类型。它们是由一种或者两种底层类型构造的。Class 类型以及函数类型同样也是复合类型，但是它们可能是由任意数量的类型组成的。在这一分类方法中，枚举类型同样被认为是复杂的符合类型，虽然它们不是由多种底层类型构成的。简单的复合类型可以通过偏特化来区分。

指针

我们从指针类型这一简单的分类开始：

```
template<typename T>
struct IsPointerT : std::false_type { //primary template: by default
not a pointer
};

template<typename T>
struct IsPointerT<T*> : std::true_type { //partial specialization for
pointers
  using BaseT = T; // type pointing to
};
```

主模板会捕获所有的非指针类型，和往常一样，其值为 fase 的 value 成员是通过基类 std::false_type 提供的，表明该类型不是指针。偏特化实现会捕获所有的指针类型（T*），其为 true 的成员 value 表明该类型是一个指针。偏特化实现还额外提供了类型成员 BaseT，描述了指针所指向的类型。注意该类型成员只有在原始类型是指针的时候才有，从其使其变成 SFINAE 友好的类型萃取。

C++标准库也提供了相对应的萃取 std::is_pointer<>，但是没有提供一个成员类型来描述指针所指向的类型。相关描述详见第 D.2.1 节。

引用

相同的方法也可以被用来识别左值引用：

```
template<typename T>
struct IsLValueReferenceT : std::false_type { //by default no lvalue
reference
};

template<typename T>
```

```
struct IsLValueReferenceT<T&> : std::true_type { //unless T is lvalue
references
  using BaseT = T; // type referring to
};
```

以及右值引用：

```
template<typename T>
struct IsRValueReferenceT : std::false_type { //by default no rvalue
reference
};

template<typename T>
struct IsRValueReferenceT<T&&> : std::true_type { //unless T is rvalue
reference
  using BaseT = T; // type referring to
};
```

它俩又可以被组合成 IsReferenceT<>萃取：

```
#include "islvaluereference.hpp"
#include "isrvaluereference.hpp"
#include "ifthenelse.hpp"
template<typename T>
class IsReferenceT
: public IfThenElseT<IsLValueReferenceT<T>::value,
                     IsLValueReferenceT<T>,
                     IsRValueReferenceT<T>
                   >::Type {
};
```

在这一实现中，我们用 IfThenElseT 从 ISLvalueReference<T>和 IsRValueReferenceT<T>中选择基类，这里还用到了元函数转发（参见第 19.3.2 节）。如果 T 是左值引用，我们会从 IsLReference<T>做继承，并通过继承得到相应的 value 和 BaseT 成员。否则，我们就从 IsRValueReference<T>做继承，它会判断一个类型是不是右值引用（并未相应的情况提供对应的成员）。

C++标准库也提供了相应的 std::is_lvalue_reference<>和 std::is_rvalue_reference<>萃取（相关介绍请参见第 D.2.1 节），还有 std::is_reference<>（相关介绍请参见第 D.2.2 节）。同样的，这些萃取也没有提供代表其所引用的类型的类型成员。

数组

在定义可以判断数组的萃取时，让人有些意外的是偏特化实现中的模板参数数量要比主模板

多：

```
#include <cstddef>
template<typename T>
struct IsArrayT : std::false_type { //primary template: not an array
};

template<typename T, std::size_t N>
struct IsArrayT<T[N]> : std::true_type { //partial specialization for
arrays
  using BaseT = T;
  static constexpr std::size_t size = N;
};

template<typename T>
struct IsArrayT<T[]> : std::true_type { //partial specialization for
unbound arrays
  using BaseT = T;
  static constexpr std::size_t size = 0;
};
```

在这里，多个额外的成员被用来描述被用来分类的数组的信息：数组的基本类型和大小（0被用来标识未知大小的数组的尺寸）。

C++标准库提供了相应的 std::is_array<>来判断一个类型是不是数组，在第 D.2.1 节有其相关介绍。除此之外，诸如 std::rank<>和 std::extent<>之类的萃取还允许我们去查询数组的维度以及某个维度的大小（参见第 D.3.1 节）。

指向成员的指针（Pointers to Members）

也可以用相同的方式处理指向成员的指针：

```
template<typename T>
struct IsPointerToMemberT : std::false_type { //by default no
pointer-to-member
};

template<typename T, typename C>
struct IsPointerToMemberT<T C::*> : std::true_type { //partial
specialization
  using MemberT = T;
  using ClassT = C;
};
```

这里额外的成员（MemberT 和 ClassT）提供了与成员的类型以及 class 的类型相关的信息。

C++ 标准库提供了更为具体的萃取，std::is_member_object_pointer<> 和 std::is_member_function_pointer<>，详见第 D.2.1 节，还有在第 D.2.2 节介绍的 std::is_member_pointer<>。

19.8.3 识别函数类型（Identifying Function Types）

函数类型比较有意思，因为它们除了返回类型，还可能会有任意数量的参数。因此，在匹配一个函数类型的偏特化实现中，我们用一个参数包来捕获所有的参数类型，就如同我们在 19.3.2 节中对 DecayT 所做的那样：

```
#include "../typelist/typelist.hpp"
template<typename T>
struct IsFunctionT : std::false_type { //primary template: no function
};

template<typename R, typename… Params>
struct IsFunctionT<R (Params…)> : std::true_type
{ //functions
  using Type = R;
  using ParamsT = Typelist<Params…>;
  static constexpr bool variadic = false;
};

template<typename R, typename… Params>
struct IsFunctionT<R (Params…, …)> : std::true_type { //variadic
functions
  using Type = R;
  using ParamsT = Typelist<Params…>;
  static constexpr bool variadic = true;
};
```

上述实现中函数类型的每一部分都被暴露了出来：返回类型被 Type 标识，所有的参数都被作为 ParamsT 捕获进了一个 typelist 中（在第 24 章有关于 typelist 的介绍），而可变参数（...）表示的是当前函数类型使用的是不是 C 风格的可变参数。

不幸的是，这一形式的 IsFunctionT 并不能处理所有的函数类型，因为函数类型还可以包含 const 和 volatile 修饰符，以及左值或者右值引用修饰符（参见第 C.2.1 节），在 C++17 之后，还有 noexcept 修饰符。比如：

```
using MyFuncType = void (int&) const;
```

这一类函数类型只有在被用于非 static 成员函数的时候才有意义，但是不管怎样都算得上是函数类型。而且，被标记为 const 的函数类型并不是真正意义上的 const 类型，因此

RemoveConst 并不能将 const 从函数类型中移除。因此，为了识别有限制符的函数类型，我们需要引入一大批额外的偏特化实现，来覆盖所有可能的限制符组合（每一个实现都需要包含 C 风格和非 C 风格的可变参数情况）。这里，我们只展示所有偏特化实现中的 5 中情况：

```
template<typename R, typename… Params>
struct IsFunctionT<R (Params…) const> : std::true_type {
  using Type = R;
  using ParamsT = Typelist<Params…>;
  static constexpr bool variadic = false;
};

template<typename R, typename… Params>
struct IsFunctionT<R (Params…, …) volatile> : std::true_type {
  using Type = R;
  using ParamsT = Typelist<Params…>;
  static constexpr bool variadic = true;
};

template<typename R, typename… Params>
struct IsFunctionT<R (Params…, …) const volatile> : std::true_type {
  using Type = R;
  using ParamsT = Typelist<Params…>;
  static constexpr bool variadic = true;
};

template<typename R, typename… Params>
struct IsFunctionT<R (Params…, …) &> : std::true_type {
  using Type = R;
  using ParamsT = Typelist<Params…>;
  static constexpr bool variadic = true;
};

template<typename R, typename… Params>
struct IsFunctionT<R (Params…, …) const&> : std::true_type {
  using Type = R;
  using ParamsT = Typelist<Params…>;
  static constexpr bool variadic = true;
};
…
```

当所有这些都准备完毕之后，我们就可以识别除 class 类型和枚举类型之外的所有类型了。我们会在接下来的章节中除了这两种例外情况。

C++标准库也提供了相应的 std::is_function<>萃取，详细介绍请参见第 D.2.1 节。

19.8.4 判断 class 类型（Determining Class Types）

和到目前为止我们已经处理的各种复合类型不同，我们没有相应的偏特化模式来专门匹配 class 类型。也不能像处理基础类型一样一一列举所有的 class 类型。相反，我们需要用一种间接的方法来识别 class 类型，为此我们需要找出一些适用于所有 class 类型的类型或者表达式（但是不能适用于其它类型）。有着这样的类型或者表达式之后，我们就可以使用在第 19.4 节介绍的 SFINAE 萃取技术了。

Class 中可以被我们用来识别 class 类型的最为方便的特性是：只有 class 类型可以被用于指向成员的指针类型（pointer-to-member types）的基础。也就是说，对于 X Y::*一类的类型结构，Y 只能是 class 类型。下面的 IsClassT<>就利用了这一特性（将 X 随机选择为 int）：

```
#include <type_traits>
template<typename T, typename = std::void_t<>>
struct IsClassT : std::false_type { //primary template: by default no
class
};

template<typename T>
struct IsClassT<T, std::void_t<int T::*>> // classes can have
pointer-to-member
: std::true_type {
};
```

C++语言规则指出，lambda 表达式的类型是“唯一的，未命名的，非枚举 class 类型”。因此在将 IsClassT 萃取用于 lambda 表达时，我们得到的结果是 true：

```
auto l = [](){};
static_assert<IsClassT<decltype(l)>::value, "">; //succeeds
```

需要注意的是，int T::*表达式同样适用于 unit 类型（更具 C++标准，枚举类型也是 class 类型）。

C++标准库提供了 std::is_class<>和 std::is_union 萃取，在第 D.2.1 节有关于它们的介绍。但是，这些萃取需要编译期进行专门的支持，因为目前还不能通过任何核心的语言技术（standard core language techniques）将 class 和 struct 从 union 类型中分辨出来。

19.8.5 识别枚举类型（Determining Enumeration Types）

目前通过我们已有的萃取技术还唯一不能识别的类型是枚举类型。我们可以通过编写基于 SFINAE 的萃取来实现这一功能，这里首先需要测试是否可以像整形类型（比如 int）进行显式转换，然后依次排除基础类型，class 类型，引用类型，指针类型，还有指向成员的指针类型（这些类型都可以被转换成整形类型，但是都不是枚举类型）。但是也有更简单的方法，因为我们发现所有不属于其它任何一种类型的类型就是枚举类型，这样就可以像下面这样实

现该萃取：

```
template<typename T>
struct IsEnumT {
static constexpr bool value = !IsFundaT<T>::value
&& !IsPointerT<T>::value &&
                          !IsReferenceT<T>::value
&& !IsArrayT<T>::value &&
                          !IsPointerToMemberT<T>::value
&& !IsFunctionT<T>::value &&
                           !IsClassT<T>::value;
};
```

C++标准库提供了相对应的 std::is_enum<>萃取，在第 D.2.1 节有对其进行介绍。通常，为了提高编译性能，编译期会直接提供这一类萃取，而不是将其实现为其它的样子。

19.9 策略萃取（Policy Traits）

到目前为止，我们例子中的萃取模板被用来判断模板参数的特性：它们代表的是哪一种类型，作用于该类型数值的操作符的返回值的类型，以及其它特性。这一类萃取被称为特性萃取（property traits）。

最为对比，某些萃取定义的是该如何处理某些类型。我们称之为策略萃取（policy traits）。这里会对之前介绍的策略类（policy class，我们已经指出，策略类和策略萃取之间的界限并不青霞）的概念进行回顾，但是策略萃取更倾向于是模板参数的某一独有特性（而策略类却通常和其它模板参数无关）。

虽然特性萃取通常都可以被实现为类型函数，策略萃取却通常将策略包装进成员函数中。为了展示这一概念，先来看一下一个定义了特定策略（必须传递只读参数）的类型函数。

19.9.1 只读参数类型

在 C++和 C 中，函数的调用参数（call parameters）默认情况下是按照值传递的。这意味着，调用函数计算出来的参数的值，会被拷贝到由被调用函数控制的位置。大部分程序员都知道，对于比较大的结构体，这一拷贝的成本会非常高，因此对于这一类结构体最好能够将其按照常量引用（reference-to-const）或者是 C 中的常量指针（pointer-to-const）进行传递。对于小的结构体，到底该怎样实现目前还没有定论，从性能的角度来看，最好的机制依赖于代码所运行的具体架构。在大多数情况下这并没有那么关键，但是某些情况下，即使是对小的结构体我们也要仔细应对。

当然，有了模板之后事情要变得更加微妙一些：我们事先并不知道用来替换模板参数的类型将会是多大。而且，事情也并不是仅仅依赖于结构体的大小：即使是比较小的结构体，其拷

贝构造函数的成本也可能会很高，这种情况下我们应对选择按常量引用传递。

正如之前暗示的那样，这一类问题通常应当用策略萃取模板（一个类型函数）来处理：该函数将预期的参数类型 T 映射到最佳的参数类型 T 或者是 T const&。作为第一步的近似，主模板会将大小不大于两个指针的类型按值进行传递，对于其它所有类型都按照常量引用进行传递：

```
template<typename T>
struct RParam {
using Type = typename IfThenElseT<sizeof(T) <=2*sizeof(void*),
                                  T,
                                  T const&>::Type;
};
```

另一方面，对于那些另 sizeof 运算符返回一个很小的值，但是拷贝构造函数成本却很高的容器类型，我们可能需要分别对它们进行特化或者偏特化，就像下面这样：

```
template<typename T>
struct RParam<Array<T>> {
  using Type = Array<T> const&;
};
```

由于这一类类型在 C++中很常见，如果只将那些拥有简单拷贝以及移动构造函数的类型按值进行传递，当需要考虑性能因素时，再选择性的将其它一些 class 类型加入按值传递的行列（C++标准库中包含了 std::is_trivially_copy_constructible 和 std::is_trivially_move_constructible 类型萃取）。

```
#ifndef RPARAM_HPP
#define RPARAM_HPP
#include "ifthenelse.hpp"
#include <type_traits>
template<typename T>
struct RParam {
  using Type = IfThenElse<(sizeof(T) <= 2*sizeof(void*)
                && std::is_trivially_copy_constructible<T>::value
                && std::is_trivially_move_constructible<T>::value),
            T,
            T const&>;
};
#endif //RPARAM_HPP
```

无论采用哪一种方式，现在该策略都可以被集成到萃取模板的定义中，客户也可以用它们去实现更好的效果。比如，假设我们有两个 class，对于其中一个 class 我们指明要按值传递只读参数：

```
#include "rparam.hpp"
#include <iostream>
class MyClass1 {
```

```
        public:
            MyClass1 () {
            }

            MyClass1 (MyClass1 const&) {
                std::cout << "MyClass1 copy constructor called\n";}
};

class MyClass2 {
  public:
      MyClass2 () {
      }

      MyClass2 (MyClass2 const&) {
          std::cout << "MyClass2 copy constructor called\n";
      }
};

// pass MyClass2 objects with RParam<> by value
template<>
class RParam<MyClass2> {
  public:
      using Type = MyClass2;
};
```

现在，我们就可以定义将 PParam<>用于只读参数的函数了，并对其进行调用：

```
#include "rparam.hpp"
#include "rparamcls.hpp"
// function that allows parameter passing by value or by reference
template<typename T1, typename T2>
void foo (typename RParam<T1>::Type p1, typename RParam<T2>::Type p2)
{
  ...
}

int main()
{
  MyClass1 mc1;
  MyClass2 mc2;
  foo<MyClass1,MyClass2>(mc1,mc2);
}
```

不幸的是，PParam 的使用有一些很大的缺点。第一，函数的声明很凌乱。第二，可能也是更有异议的地方，就是在调用诸如 foo()一类的函数时不能使用参数推断，因为模板参数只

出现在函数参数的限制符中。因此在调用时必须显式的指明所有的模板参数。

一个稍显笨拙的权宜之计是：使用提供了完美转发的 inline 封装函数（inline wrapper function），但是需要假设编译器将省略 inline 函数：

```
#include "rparam.hpp"
#include "rparamcls.hpp"
// function that allows parameter passing by value or by reference
template<typename T1, typename T2>
void foo_core (typename RParam<T1>::Type p1, typename RParam<T2>::Type
p2)
{
  ...
}

// wrapper to avoid explicit template parameter passing
template<typename T1, typename T2>
void foo (T1 && p1, T2 && p2)
{
  foo_core<T1,T2>(std::forward<T1>(p1),std::forward<T2>(p2));
}

int main()
{
  MyClass1 mc1;
  MyClass2 mc2;
  foo(mc1,mc2); // same as foo_core<MyClass1,MyClass2> (mc1,mc2)
}
```

19.10 在标准库中的情况

在 C++11 中，类型萃取变成了 C++标准库中固有的一部分。它们或多或少的构成了在本章中讨论的所有的类型函数和类型萃取。但是，对于它们中的一部分，比如个别的操作探测，以及有过讨论的 std::is_union，目前都还没有已知的语言解决方案。而是由编译器为这些萃取提供了支持。同样的，编译器也开始支持一些已经由语言本身提供了解决方案的萃取，这主要是为了减少编译时间。

因此，如果你需要类型萃取，我们建议在可能的情况下都尽量使用由 C++标准库提供的萃取。在附录 D 中对它们有详细的讨论。

需要注意的是，某些萃取的行为可能会让人很意外（至少对于新手程序员）。除了我们在第 11.2.1 节和第 D.1.2 节暗示过的东西，也请参考我们在附录 D 中给出的相关描述。

C++标准库也定义了一些策略和属性萃取：

- 类模板 std::char_traits 被 std::string 和 I/O stream 当作策略萃取使用。
- 为了将算法简单的适配于标准迭代器的种类，标准库提供了一个很简单的 std::iterator_traits 属性萃取模板。
- 模板 std::numeric_limits 作为属性萃取模板也会很有帮助。
- 最后，为标准库容器类型进行的内存分配是由策略萃取类处理的（参见 std::shared_ptr 的实现）。从 C++98 开始，标准库专门为了这一目的提供了 std::allocator 模板。从 C++11 开始，标准库引入了 std::allocator_traits 模板，这样就能够修改内存分配器的策略或者行为了。

19.11 后记

Nathan Myers 是第一个提出萃取参数这一概念的人。他最初将该想法作为在标准库组件中定义处理类型的方式提交给 C++标准委员会。在当时，他称其为 baggage templates，并注意到它们包含了萃取。但是，C++委员会的一部分成员不喜欢 baggage 这个名字，并最终促成了 traits 这一名字的使用。后者在那之后被广泛使用。

客户代码通常不会和萃取有任何交集：默认萃取类的行为满足了大部分的常规需求，而且由于它们是默认模板参数，它们根本就不需要出现在客户代码中。这有利于为默认萃取模板使用很长的名字。当客户代码通过提供客制化的萃取参数适应了模板的行为后，最好能够为最终的特化提供一个类型别名。

萃取可以被作为一种反射（reflection）使用，在其中程序看到了其自身的更为高阶的属性。诸如 IsClassT 和 PlusResult 的萃取，以及其它一些窥测了程序中类型的类型萃取，都实现了一种编译期的反射，这被证明是元编程的一个很好的手段。

将类型属性作为模板特化成员存储的相反至少可以追溯到 1990 年代中期。一种比较严肃的早期的类型分类模板的应用是由 SGI（Silicon Graphics）发布的 STL 实现。SGI 模板被用来代表其模板参数的一些属性。这些信息又被用来为特定的类型进行 STL 算法优化。

Boost 提供了更为完整的一组类型分类模板，它们构成了 2011C++标准库中<type_triats>的基础。虽然其中一些萃取可以根据本章介绍的技术实现，其它一些却需要编译器的支持，这一点和由 SGI 编译期提供的__type_traits 特化实现很类似。

使用诸如 isValid 的泛型模板提取 SFINAE 条件的本质信息这一技术是由 Louis Dionne 在 2015 年提出的，并在 Boost.Hana 中得到应用。

策略类显然是由很多程序员一起开发的，但是其中只有一部分得到了署名。Andrei Alexandrescu 使 policy classes 这一名词变得流行，他在其《Modern C++ Design》中对其有更为详细的介绍。

第 20 章 基于类型属性的重载（Overloading on Type Properties）

函数重载使得相同的函数名能够被多个函数使用，只要能够通过这些函数的参数类型区分它们就行。比如：

```
void f (int);
void f (char const*);
```

对于函数模板，可以在类型模式上进行重载，比如针对指向 T 的指针或者 Array<T>：

```
template<typename T> void f(T*);
template<typename T> void f(Array<T>);
```

在类型萃取（参考第 19 章）的概念流行起来之后，很自然地会想到基于模板参数对函数模板进行重载。比如：

```
template<typename Number> void f(Number); // only for numbers
template<typename Container> void f(Container);// only for containers
```

但是，目前 C++还没有提供任何可以直接基于类型属性进行重载的方法。事实上，上面的两个模板声明的是完全相同的函数模板，而不是进行了重载，因为在比较两个函数模板的时候不会比较模板参数的名字。

幸运的是，有比较多的基于类型特性的技术，可以被用来实现类似于函数模板重载的功能。本章将会讨论这些相关技术，以及为什么要实现这一类重载的原因。

20.1 算法特化（我更愿意称之为算法重载，见注释）

函数模板重载的一个动机是，基于算法适用的类型信息，为算法提供更为特化的版本。考虑一个交换两个数值的 swap()操作：

```
template<typename T>
void swap(T& x, T& y)
{
  T tmp(x);
  x = y;
  y = tmp;
}
```

这一实现用到了三次拷贝操作。但是对于某些类型，可以有一种更为高效的 swap()实现，比如对于存储了指向具体数组内容的指针和数组长度的 Array<T>:

```
template<typename T>
```

```
void swap(Array<T>& x, Array<T>& y)
{
  swap(x.ptr, y.ptr);
  swap(x.len, y.len);
}
```

俩种 swap()实现都可以正确的交换两个 Array<T>对象的内容。但是，后一种实现方式的效率要高很多，因为它利用了 Array<T>中额外的（具体而言，是 ptr 和 len 以及它们各自的职责）、不为其它类型所有的特性。因此后一种实现方式要（在概念上）比第一种实现方式更为“特化”，这是因为它只为适用于前一种实现的类型的一个子集提供了交换操作。幸运的是，基于函数模板的部分排序规则（partial ordering rules，参见 16.2.2 节），第二种函数模板也是更为特化的，在有更为特化的版本（也更高效）可用的时候，编译器会优先选择该版本，在其不适用的时候，会退回到更为泛化的版本（可能会不那么高效）。

在一个泛型算法中引入更为特化的变体，这一设计和优化方式被称为算法特化（algorithm specialization）。更为特化的变体适用于泛型算法诸多输入中的一个子集，这个子集可以通过特定的类型或者是类型的属性来区分，针对这个子集，该特化版本通常要比泛型算法的一般版本高效的多。

在适用的情况下更为特化的算法变体会自动的被选择，这一点对算法特化的实现至关重要，调用者甚至都不需要知道具体变体的存在。在我们的 swap()例子中，具体实现方式是用（在概念上）更为特化的函数模板去重载最泛化的模板，同时确保了在 C++的部分排序规则（partial ordering rules）中更为特化的函数模板也是更为特化的。

并不是所有的概念上更为特化的算法变体，都可以被直接转换成提供了正确的部分排序行为（partial ordering behavior）的函数模板。比如我们下面的这个例子。

函数模板 advanceIter()（类似于 C++标准库中的 std::advance()）会将迭代器 x 向前迭代 n 步。这一算法可以用于输入的任意类型的迭代器：

```
template<typename InputIterator, typename Distance>
void advanceIter(InputIterator& x, Distance n)
{
  while (n > 0) { //linear time
      ++x;
      --n;
  }
}
```

对于特定类型的迭代器（比如提供了随机访问操作的迭代器），我们可以为该操作提供一个更为高效的实现方式：

```
template<typename RandomAccessIterator, typename Distance>
void advanceIter(RandomAccessIterator& x, Distance n) {
  x += n; // constant time
}
```

但是不幸的是，同时定义以上两种函数模板会导致编译错误，正如我们在序言中介绍的那样，这是因为只有模板参数名字不同的函数模板是不可以被重载的。本章剩余的内容会讨论能够允许我们实现类似上述函数模板重载的一些技术。

20.2 标记派发（Tag Dispatching）

算法特化的一个方式是，用一个唯一的、可以区分特定变体的类型来标记（tag）不同算法变体的实现。比如为了解决上述 advanceIter()中的问题，可以用标准库中的迭代器种类标记类型，来区分 advanceIter()算法的两个变体实现：

```
template<typename Iterator, typename Distance>
void advanceIterImpl(Iterator& x, Distance n,
std::input_iterator_tag)
{
  while (n > 0) { //linear time
      ++x;
      --n;
  }
}

template<typename Iterator, typename Distance>
void advanceIterImpl(Iterator& x, Distance n,
std::random_access_iterator_tag)
{
  x += n; // constant time
}
```

然后，通过 advanceIter()函数模板将其参数连同与之对应的 tag 一起转发出去：

```
template<typename Iterator, typename Distance>
void advanceIter(Iterator& x, Distance n)
{
    advanceIterImpl(x, n, typename
       std::iterator_traits<Iterator>::iterator_category())
}
```

萃取类模板 std::iterator_traits 通过其成员类型 iterator_category 返回了迭代器的种类。迭代器种类是前述_tag 类型中的一种，它指明了相关类型的具体迭代器种类。在 C++标准库中，可用的 tags 被定义成了下面这样，在其中使用了继承来反映出一个用 tag 表述的种类是不是从另一个种类派生出来的：

```
namespace std {
  struct input_iterator_tag { };
  struct output_iterator_tag { };
  struct forward_iterator_tag : public input_iterator_tag { };
```

```
  struct bidirectional_iterator_tag : public forward_iterator_tag
  { };

  struct random_access_iterator_tag : public
      bidirectional_iterator_tag { };
}
```

有效使用标记派发（tag dispatching）的关键在于理解 tags 之间的内在关系。我们用来标记两个 advanceIterImpl 变体的标记是 std::input_iterator_tag 和 std::random_access_iterator_tag，而由于 std::random_access_iterator_tag 继承自 std::input_iterator_tag，对于随机访问迭代器，会优先选择更为特化的 advanceIterImpl()变体（使用了 std::random_access_iterator_tag 的那一个）。因此，标记派发依赖于将单一的主函数模板的功能委托给一组_impl 变体，这些变体都被进行了标记，因此正常的函数重载机制会选择适用于特定模板参数的最为特化的版本。

当被算法用到的特性具有天然的层次结构，并且存在一组为这些标记提供了值的萃取机制的时候，标记派发可以很好的工作。而如果算法特化依赖于专有（ad hoc）类型属性的话（比如依赖于类型 T 是否含有拷贝赋值运算符），标记派发就没那么方便了。对于这种情况，我们需要一个更强大的技术。

20.3 Enable/Disable 函数模板

算法特化需要提供可以基于模板参数的属性进行选择的、不同的函数模板。不幸的是，无论是函数模板的部分排序规则（参见 16.2.2 节）还是重载解析（参见附录 C），都不能满足更为高阶的算法特化的要求。

C++标准库为之提供的一个辅助工具是 std::enable_if，我们曾在第 6.3 节对其进行了介绍。本节将介绍通过引入一个对应的模板别名，实现该辅助工具的方式，为了避免名称冲突，我们将称之称为 EnableIf。

和 std::enable_if 一样，EnableIf 模板别名也可以被用来基于特定的条件 enable(或 disable)特定的函数模板。比如，随机访问版本的 advanceIter()算法可以被实现成这样：

```
template<typename Iterator>
constexpr bool IsRandomAccessIterator =
  IsConvertible< typename
std::iterator_traits<Iterator>::iterator_category,
            std::random_access_iterator_tag>;

template<typename Iterator, typename Distance>
  EnableIf<IsRandomAccessIterator<Iterator>>
advanceIter(Iterator& x, Distance n){
  x += n; // constant time
}
```

这里使用了基于 EnableIf 的偏特化，在迭代器是随机访问迭代器的时候启用特定的 advanceIter()变体。EnableIf 包含两个参数，一个是标示着该模板是否应该被启用的 bool 型条件参数，另一个是在第一个参数为 true 时，EnableIf 应该包含的类型。在我们上面的例子中，用在第 19.5 节和第 19.7.3 节介绍的 IsConvertible 类型萃取定义了一个新的类型萃取 IsRandomAccessIterator。这样，这一特殊版本的 advanceIter()实现只有在模板参数 Iterator 是被一个随机访问迭代器替换的时候才会被启用。

EnableIf 的实现非常简单：

```
template<bool, typename T = void>
struct EnableIfT {
};

template< typename T>
struct EnableIfT<true, T> {
using Type = T;
};

template<bool Cond, typename T = void>
using EnableIf = typename EnableIfT<Cond, T>::Type;
```

EnableIf 会扩展成一个类型，因此它被实现成了一个别名模板（alias template）。我们希望为之使用偏特化（参见第 16 章），但是别名模板（alias template）并不能被偏特化。幸运的是，我们可以引入一个辅助类模板（helper class template）EnableIfT，并将真正要做的工作委托给它，而别名模板 EnableIf 所要做的只是简单的从辅助模板中选择结果类型。当条件是 true 的时候，EnableIfT<...>::Type（也就是 EnableIf<...>）的计算结果将是第二个模板参数 T。当条件是 false 的时候，EnableIf 不会生成有效的类型，因为主模板 EnableIfT 没有名为 Type 的成员。通常这应该是一个错误，但是在 SFINAE（参见第 15.7 节）上下文中（比如函数模板的返回类型），它只会导致模板参数推断失败，并将函数模板从待选项中移除。

对于 advanceIter()，EnableIf 的使用意味着只有当 Iterator 参数是随机访问迭代器的时候，函数模板才可以被使用（而且返回类型是 void），而当 Iterator 不是随机访问迭代器的时候，函数模板则会被从待选项中移除。我们可以将 EnableIf 理解成一种在模板参数不满足特定需求的时候，防止模板被实例化的防卫手段。由于 advanceIter()需要一些只有随机访问迭代器才有操作，因此只能被随机访问迭代器实例化。有时候这样使用 EnableIf 也不是绝对安全的----用户可能会断言一个类型是随机访问迭代器，却又没有为之提供相应的操作----此时 EnableIf 可以被用来帮助尽早的发现这一类错误。

现在我们已经可以显式的为特定的类型激活其所适用的更为特化的模板了。但是这还不够：我们还需要“去激活（de-activate）”不够特化的模板，因为在两个模板都适用的时候，编译期没有办法在两者之间做决断（order），从而会报出一个模板歧义错误。幸运的是，实现这一目的方法并不复杂：我们为不够特化的模板使用相同模式的 EnableIf，只是适用相反的判断条件。这样，就可以确保对于任意 Iterator 类型，都只有一个模板会被激活。因此，适用于非随机访问迭代器的 advanceIter()会变成下面这样：

```
template<typename Iterator, typename Distance>
  EnableIf<!IsRandomAccessIterator<Iterator>>
advanceIter(Iterator& x, Distance n)
{
  while (n > 0) {//linear time
      ++x;
      --n;
  }
}
```

20.3.1 提供多种特化版本

上述模式可以被继续泛化以满足有两种以上待选项的情况：可以为每一个待选项都配备一个EnableIf，并且让它们的条件部分，对于特定的模板参数彼此互斥。这些条件部分通常会用到多种可以用类型萃取（type traits）表达的属性。

比如，考虑另外一种情况，第三种 advanceIter()算法的变体：允许指定一个负的距离参数，以让迭代器向“后”移动。很显然这对一个“输入迭代器（input itertor）”是不适用的，对一个随机访问迭代器却是适用的。但是，标准库也包含一种双向迭代器（bidirectional iterator）的概念，这一类迭代器可以向后移动，但却不要求必须同时是随机访问迭代器。实现这一情况需要稍微复杂一些的逻辑：每个函数模板都必须使用一个包含了在所有函数模板间彼此互斥 EnableIf 条件，这些函数模板代表了同一个算法的不同变体。这样就会有下面一组条件：

- 随机访问迭代器：适用于随机访问的情况（常数时间复杂度，可以向前或向后移动）
- 双向迭代器但又不是随机访问迭代器：适用于双向情况（线性时间复杂度，可以向前或向后移动）
- 输入迭代器但又不是双向迭代器：适用于一般情况（线性时间复杂度，只能向前移动）

相关函数模板的具体实现如下：

```
#include <iterator>
// implementation for random access iterators:
template<typename Iterator, typename Distance>
EnableIf<IsRandomAccessIterator<Iterator>>
advanceIter(Iterator& x, Distance n) {
  x += n; // constant time
}

template<typename Iterator>
constexpr bool IsBidirectionalIterator =
IsConvertible< typename
std::iterator_traits<Iterator>::iterator_category,
std::bidirectional_iterator_tag>;

// implementation for bidirectional iterators:
```

```
template<typename Iterator, typename Distance>
EnableIf<IsBidirectionalIterator<Iterator>
&& !IsRandomAccessIterator<Iterator>>
advanceIter(Iterator& x, Distance n) {
  if (n > 0) {
      for ( ; n > 0; ++x, --n) { //linear time
      }
  } else {
      for ( ; n < 0; --x, ++n) { //linear time
      }
  }
}

// implementation for all other iterators:
template<typename Iterator, typename Distance>
  EnableIf<!IsBidirectionalIterator<Iterator>>
advanceIter(Iterator& x, Distance n) {
  if (n < 0) {
      throw "advanceIter(): invalid iterator category for negative n";
  }
  while (n > 0) { //linear time
      ++x;
      --n;
  }
}
```

通过让每一个函数模板的 EnableIf 条件与其它所有函数模板的条件互相排斥，可以保证对于一组参数，最多只有一个函数模板可以在模板参数推断中胜出。

上述例子已体现出通过 EnableIf 实现算法特化的一个缺点：每当一个新的算法变体被加入进来，就需要调整所有算法变体的 EnableIf 条件，以使得它们之间彼此互斥。作为对比，当通过标记派发（tag dispatching）引入一个双向迭代器的算法变体时，则只需要使用标记 std::bidirectional_iterator_tag 重载一个 advanceIterImpl()即可。

标记派发（tag dispatching）和 EnableIf 两种技术所适用的场景有所不同：一般而言，标记派发可以基于分层的 tags 支持简单的派发，而 EnableIf 则可以基于通过使用类型萃取（type trait）获得的任意一组属性来支持更为复杂的派发。

20.3.2 EnableIf 所之何处（where does the EnableIf Go）？

EnableIf 通常被用于函数模板的返回类型。但是，该方法不适用于构造函数模板以及类型转换模板，因为它们都没有被指定返回类型。而且，使用 EnableIf 也会使得返回类型很难被读懂。对于这一问题，我们可以通过将 EnableIf 嵌入一个默认的模板参数来解决，比如：

```
#include <iterator>
#include "enableif.hpp"
#include "isconvertible.hpp"
template<typename Iterator>
constexpr bool IsInputIterator = IsConvertible< typename
std::iterator_traits<Iterator>::iterator_category,
std::input_iterator_tag>;

template<typename T>
class Container {
  public:
      // construct from an input iterator sequence:
      template<typename Iterator, typename =
EnableIf<IsInputIterator<Iterator>>>
      Container(Iterator first, Iterator last);
      // convert to a container so long as the value types are convertible:
      template<typename U, typename = EnableIf<IsConvertible<T, U>>>
      operator Container<U>() const;
};
```

但是，这样做也有一个问题。如果我们尝试再添加一个版本的重载的话，会导致错误：

```
// construct from an input iterator sequence:
template<typename Iterator,
typename = EnableIf<IsInputIterator<Iterator>
&& !IsRandomAccessIterator<Iterator>>>
Container(Iterator first, Iterator last);

template<typename Iterator, typename =
EnableIf<IsRandomAccessIterator<Iterator>>>
Container(Iterator first, Iterator last); // ERROR: redeclaration //
of constructor template
```

问题在于这两个模板唯一的区别是默认模板参数，但是在判断两个模板是否相同的时候却又不会考虑默认模板参数。

该问题可以通过引入另外一个模板参数来解决，这样两个构造函数模板就有数量不同的模板参数了：

```
// construct from an input iterator sequence:
template<typename Iterator, typename =
EnableIf<IsInputIterator<Iterator>
&& !IsRandomAccessIterator<Iterator>>>
Container(Iterator first, Iterator last);

template<typename Iterator, typename =
```

```
EnableIf<IsRandomAccessIterator<Iterator>>, typename = int> // extra
dummy parameter to enable both constructors
Container(Iterator first, Iterator last); //OK now
```

20.3.3 编译期 if

值得注意的是，C++17 的 constexpr if 特性（参见第 8.5 节）使得某些情况下可以不再使用 EnableIf。比如在 C++17 中可以像下面这样重写 advanceIter():

```
template<typename Iterator, typename Distance>
void advanceIter(Iterator& x, Distance n) {
  if constexpr(IsRandomAccessIterator<Iterator>) {
      // implementation for random access iterators:
      x += n; // constant time
  }else if constexpr(IsBidirectionalIterator<Iterator>) {
      // implementation for bidirectional iterators:
      if (n > 0)
          {for ( ; n > 0; ++x, --n) { //linear time for positive n
          }
      } else {
          for ( ; n < 0; --x, ++n) { //linear time for negative n
          }
      }
  }else {
      // implementation for all other iterators that are at least input iterators:
      if (n < 0) {
          throw "advanceIter(): invalid iterator category for negative n";
      }
      while (n > 0) { //linear time for positive n only
          ++x;
          --n;
      }
  }
}
```

这样会更好一些。更为特化的代码分支只会被那些支持它们的类型实例化。因此，对于使用了不被所有的迭代器都支持的代码的情况，只要它们被放在合适的 constexpr if 分支中，就是安全的。

但是，该方法也有其缺点。只有在泛型代码组件可以被在一个函数模板中完整的表述时，这一使用 constexpr if 的方法才是可能的。在下面这些情况下，我们依然需要 EnableIf：

- 需要满足不同的“接口”需求
- 需要不同的 class 定义
- 对于某些模板参数列表，不应该存在有效的实例化。

对于最后一种情况，下面这种做法看上去很有吸引力：

```
template<typename T>
void f(T p) {
  if constexpr (condition<T>::value) {
      // do something here…
  }
  else {
      // not a T for which f() makes sense:
      static_assert(condition<T>::value, "can't call f() for such a T");
  }
}
```

但是我们并不建议这样做，因为它对 SFINAE 不太友好：函数 f<T>()并不会被从待选项列表中移除，因此它有可能会屏蔽掉另一种重载解析结果。作为对比，使用 EnableIf f<T>()则会在 EnableIf<...>替换失败的时候将该函数从待选项列表中移除。

20.3.4 Concepts

上述技术到目前为止都还不错，但是有时候却稍显笨拙，它们可能会占用很多的编译器资源，以及在某些情况下，可能会产生难以理解的错误信息。因此某些泛型库的作者一直都在盼望着一种能够更简单、直接地实现相同效果的语言特性。为了满足这一需求，一个被称为 conceptes 的特性很可能会被加入到 C++语言中；具体请参见第 6.5 节，第 18.4 节以及附录 E。

比如，我们可能希望被重载的 container 的构造函数可以像下面这样：

```
template<typename T>
class Container {
  public:
  //construct from an input iterator sequence:
  template<typename Iterator>
  requires IsInputIterator<Iterator>
  Container(Iterator first, Iterator last);

  // construct from a random access iterator sequence:
  template<typename Iterator>
  requires IsRandomAccessIterator<Iterator>
  Container(Iterator first, Iterator last);
  // convert to a container so long as the value types are convertible:

  template<typename U>
  requires IsConvertible<T, U>
  operator Container<U>() const;
};
```

其中 requires 条款（参见第 E.1 节）描述了使用当前模板的要求。如果某个要求不被满足，那么相应的模板就不会被当作备选项考虑。因此它可以被当作 EnableIf 这一想法的更为直接的表达方式，而且是被语言自身支持的。

Requires 条款还有另外一些优于 EnableIf 的地方。约束包容（constraint subsumption，参见第 E.3.1 节）为只有 requires 不同的模板进行了排序，这样就不再需要标记派发了（tag dispatching）。而且，requires 条款也可以被用于非模板。比如只有在 T 的对象可以被<运算符比较的时候，才为容器提供 sort()成员函数：

```
template<typename T>
class Container {
  public:
      ...
      requires HasLess<T>
      void sort() {
          ...
      }
};
```

20.4 类的特化（Class Specialization）

类模板的偏特化可以被用来提供一个可选的、为特定模板参数进行了特化的实现，这一点和函数模板的重载很相像。而且，和函数模板的重载类似，如果能够基于模板参数的属性对各种偏特化版本进行区分，也会很有意义。考虑一个以 key 和 value 的类型为模板参数的泛型 Dictionary 类模板。只要 key 的类型提供了 operator==()运算符，就可以实现一个简单（但是低效）的 Dictionary：

```
template<typename Key, typename Value>
class Dictionary
{
  private:
      vector<pair<Key const, Value>> data;
  public:
      //subscripted access to the data:
      value& operator[](Key const& key)
      {
          // search for the element with this key:
          for (auto& element : data) {
              if (element.first == key){
                  return element.second;
              }
          }
          // there is no element with this key; add one
          data.push_back(pair<Key const, Value>(key, Value()));
```

```
            return data.back().second;
        }
        …
};
```

如果 key 的类型提供了 operator <()运算符的话，则可以基于标准库的 map 容器提供一种相对高效的实现方式。类似的，如果 key 的类型提供了哈希操作的话，则可以基于标准库的 unordered_map 提供一种更为高效的实现方式。

20.4.1 启用/禁用类模板

启用/禁用类模板的不同实现方式的方法是使用类模板的偏特化。为了将 EnableIf 用于类模板的偏特化，需要先为 Dictionary 引入一个未命名的、默认的模板参数：

```
template<typename Key, typename Value, typename = void>
class Dictionary
{
  … //vector implementation as above
};
```

这个新的模板参数将是我们使用 EnableIf 的入口，现在它可以被嵌入到基于 map 的偏特化 Dictionary 的模板参数例表中：

```
template<typename Key, typename Value>
class Dictionary<Key, Value, EnableIf<HasLess<Key>>>
{
  private:
      map<Key, Value> data;
  public:value& operator[](Key const& key) {
      return data[key];
  }
  …
};
```

和函数模板的重载不同，我们不需要对主模板的任意条件进行禁用，因为对于类模板，任意偏特化版本的优先级都比主模板高。但是，当我们针对支持哈希操作的另一组 keys 进行特化时，则需要保证不同偏特化版本间的条件是互斥的：

```
template<typename Key, typename Value, typename = void>
class Dictionary
{
  … // vector implementation as above
};

template<typename Key, typename Value>
class Dictionary<Key, Value, EnableIf<HasLess<Key> && !HasHash<Key>>> {
```

```
{
  … // map implementation as above
};

template typename Key, typename Value>
class Dictionary Key, Value, EnableIf HasHash Key>>>
{
  private:
      unordered_map Key, Value> data;
  public:
  value& operator[](Key const& key) {
      return data[key];
  }
  …
};
```

20.4.2 类模板的标记派发

同样地，标记派发也可以被用于在不同的模板特化版本之间做选择。为了展示这一技术，我们定义一个类似于之前章节中介绍的 advanceIter()算法的函数对象类型 Advance<Iterator>，它同样会以一定的步数移动迭代器。会同时提供基本实现（用于 input iterators）和适用于双向迭代器和随机访问迭代器的特化版本，并基于辅助萃取 BestMatchInSet（下面会讲到）为相应的迭代器种类选择最合适的实现版本：

```
// primary template (intentionally undefined):
template<typename Iterator,
  typename Tag = BestMatchInSet< typename
std::iterator_traits<Iterator> ::iterator_category,
                                    std::input_iterator_tag,
                                    std::bidirectional_iterator_tag,
                                    std::random_access_iterator_tag>>
class Advance;

// general, linear-time implementation for input iterators:
template<typename Iterator>
class Advance<Iterator, std::input_iterator_tag>
{
  public:
      using DifferenceType = typename
std::iterator_traits<Iterator>::difference_type;
      void operator() (Iterator& x, DifferenceType n) const
      {
          while (n > 0) {
              ++x;
```

```
                --n;
            }
        }
};

// bidirectional, linear-time algorithm for bidirectional iterators:
template<typename Iterator>
class Advance<Iterator, std::bidirectional_iterator_tag>
{
  public:
      using DifferenceType =typename
std::iterator_traits<Iterator>::difference_type;
      void operator() (Iterator& x, DifferenceType n) const
      {
          if (n > 0) {
              while (n > 0) {
                  ++x;
                  --n;
              }
          } else {
              while (n < 0) {
                  --x;
                  ++n;
              }
          }
      }
};

// bidirectional, constant-time algorithm for random access iterators:
template<typename Iterator>
class Advance<Iterator, std::random_access_iterator_tag>
{
  public:
      using DifferenceType =
      typename std::iterator_traits<Iterator>::difference_type;
      void operator() (Iterator& x, DifferenceType n) const
      {
          x += n;
      }
}
```

这一实现形式和函数模板中的标记派发很相像。但是，比较困难的是 BestMatchInSet 的实现，它主要被用来为一个给定的迭代器选择选择最匹配 tag。本质上，这个类型萃取所做的是，当给定一个迭代器种类标记的值之后，要判断出该从以下重载函数中选择哪一个，并返回其

参数类型：

```
void f(std::input_iterator_tag);
void f(std::bidirectional_iterator_tag);
void f(std::random_access_iterator_tag);
```

模拟重载解析最简单的方式就是使用重载解析，就像下面这样：

```
// construct a set of match() overloads for the types in Types…:
template<typename… Types>
struct MatchOverloads;

// basis case: nothing matched:
template<>
struct MatchOverloads<> {
  static void match(…);
};

// recursive case: introduce a new match() overload:
template<typename T1, typename… Rest>
struct MatchOverloads<T1, Rest…> : public MatchOverloads<Rest…>
{
  static T1 match(T1); // introduce overload for T1
  using MatchOverloads<Rest…>::match;// collect overloads from bases
};

// find the best match for T in Types…
template<typename T, typename… Types>
struct BestMatchInSetT {
  using Type = decltype(MatchOverloads<Types…>::match(declval<T> ()));
};

template<typename T, typename… Types>
using BestMatchInSet = typename BestMatchInSetT<T, Types…>::Type;
```

MatchOverloads 模板通过递归继承为输入的一组 Types 中的每一个类型都声明了一个 match()函数。每一次递归模板 MatchOverloads 偏特化的实例化都为列表中的下一个类型引入了一个新的 match()函数。然后通过使用 using 声明将基类中的 match()函数引入当前作用域。当递归地使用该模板的时候，我们就有了一组和给定类型完全对应的 match()函数的重载，每一个重载函数返回的都是其参数的类型。然后 BestMatchInSetT 模板会将 T 类型的对象传递给一组 match()的重载函数，并返回最匹配的 match()函数的返回类型。如果没有任何一个 match()函数被匹配上，那么返回基本情况对应的 void（使用省略号来捕获任意参数）将代表出现了匹配错误。总结来讲，BestMatchInSetT 将函数重载的结果转化成了类型萃取，这样可以让通过标记派发，在不同的模板偏特化之间做选择的情况变得相对容易一些。

20.5 实例化安全的模板（Instantiation-Safe Templates）

EnableIf 技术的本质是：只有在模板参数满足某些条件的情况下才允许使用某个模板或者某个偏特化模板。比如，最为高效的 advanceIter()算法会检查迭代器的参数种类是否可以被转化成 std::random_access_iterator_tag，也就意味着各种各样的随机访问迭代器都适用于该算法。

如果我们将这一概念发挥到极致，将所有模板用到的模板参数的操作都编码进 EnableIf 的条件，会怎样呢？这样一个模板的实例化永远都不会失败，因为那些没有提供 EnableIf 所需操作的模板参数会导致一个推断错误，而不是任由可能会出错的实例化继续进行。我们称这一类模板为“实例化安全（instantiation-safe ）”的模板，接下来会对其进行简单介绍。

先从一个计算两个数之间的最小值的简单模板 min()开始。我们可能会将其实现成下面这样：

```
template<typename T>
T const& min(T const& x, T const& y)
{
  if (y < x) {
      return y;
  }
  return x;
}
```

这个模板要求类型为 T 的两个值可以通过<运算符进行比较，并将比较结果转换成 bool 类型给 if 语句使用。可以检查类型是否支持<操作符，并计算其返回值类型的类型萃取，在形式上和我们第 19.4.4 节介绍的 SFINAE 友好的 PlusResultT 萃取类似。为了方便，我们此处依然列出 LessResultT 的实现：

```
#include <utility> // for declval()
#include <type_traits> // for true_type and false_type
template<typename T1, typename T2>
class HasLess {
  template<typename T> struct Identity;

  template<typename U1, typename U2>
  static std::true_type
  test(Identity<decltype(std::declval<U1>() < std::declval<U2>())>*);

  template<typename U1, typename U2>
  static std::false_type
  test(…);

  public:
      static constexpr bool value = decltype(test<T1, T2> (nullptr))::value;
};
```

```
template<typename T1, typename T2, bool HasLess>
class LessResultImpl {
  public:
  using Type = decltype(std::declval<T1>() < std::declval<T2>());
};

template<typename T1, typename T2>
class LessResultImpl<T1, T2, false> {
};

template<typename T1, typename T2>
class LessResultT
: public LessResultImpl<T1, T2, HasLess<T1, T2>::value> {
};

template<typename T1, typename T2>
using LessResult = typename LessResultT<T1, T2>::Type;
```

现在就可以通过将该萃取和 IsConvertible 一起使用，使 min()变成实例化安全的：

```
#include "isconvertible.hpp"
#include "lessresult.hpp"
template<typename T>
EnableIf<IsConvertible<LessResult<T const&, T const&>, bool>, T const&>
min(T const& x, T const& y)
{
  if (y < x) {
      return y;
  }
  return x;
}
```

通过各种实现了不同<运算符的类型来调用 min()，要更能说明问题一些，就像下面这样：

```
#include"min.hpp"
struct X1 { };
bool operator< (X1 const&, X1 const&) { return true; }

struct X2 { };
bool operator<(X2, X2) { return true; }

struct X3 { };
bool operator<(X3&, X3&) { return true; }

struct X4 { };
```

```
struct BoolConvertible {
  operator bool() const { return true; } // implicit conversion to bool
};

struct X5 { };

BoolConvertible operator< (X5 const&, X5 const&)
{
  return BoolConvertible();
}

struct NotBoolConvertible { // no conversion to bool
};

struct X6 { };

NotBoolConvertible operator< (X6 const&, X6 const&)
{
  return NotBoolConvertible();
}

struct BoolLike {
  explicit operator bool() const { return true; } // explicit conversion to bool
};

struct X7 { };

BoolLike operator< (X7 const&, X7 const&) { return BoolLike(); }

int main()
{
  min(X1(), X1()); // X1 can be passed to min()
  min(X2(), X2()); // X2 can be passed to min()
  min(X3(), X3()); // ERROR: X3 cannot be passed to min()
  min(X4(), X4()); // ERROR: X4 cannot be passed to min()
  min(X5(), X5()); // X5 can be passed to min()
  min(X6(), X6()); // ERROR: X6 cannot be passed to min()
  min(X7(), X7()); // UNEXPECTED ERROR: X7 cannot be passed to min()
}
```

在编译上述程序的时候，要注意虽然针对 min()函数会报出 4 个错误（X3，X4，X6，以及 X7），但它们都不是从 min()的函数体中报出来的（如果不是实例化安全的话，则会从函数体中报出错误）。相反，编译器只会抱怨说没有合适的 min()函数，因为唯一的选择已经被 SFINAE

排除了。Clang 会报出如下错误：

```
min.cpp:41:3: error: no matching function for call to ' min'
min(X3(), X3()); // ERROR: X3 cannot be passed to min
^~~
./min.hpp:8:1: note: candidate template ignored: substitution
failure
[with T = X3]: no type named ' Type'  in
' LessResultT<const X3 &, const X3 &>'
min(T const& x, T const& y)
```

g++报出的部分错误信息如下：

```
min.cpp: In function 'int main()':
min.cpp:83:19: error: no matching function for call to 'min(X3, X3)'
     min(X3(), X3()); // ERROR: X3 cannot be passed to min()
                  ^
min.cpp:72:1: note: candidate: template<class T>
std::enable_if_t<std::is_convertible<typename LessResultT<const T&, const
T&>::Type, bool>::value, const T&> min(const T&, const T&)
 min(T const& x, T const& y)
 ^
min.cpp:72:1: note:   template argument deduction/substitution failed:
min.cpp: In substitution of 'template<class T>
std::enable_if_t<std::is_convertible<typename LessResultT<const T&, const
T&>::Type, bool>::value, const T&> min(const T&, const T&) [with T = X3]':
min.cpp:83:19:   required from here
min.cpp:72:1: error: no type named 'Type' in 'class LessResultT<const X3&, const X3&>'
```

因此可以看出，EnableIf 只允许针对那些满足了模板要求的类型（X1，X2，和 X5）进行实例化，也就永远不会从 min()的函数体中报出错误。

例子中的最后一个类型（X7），体现了实现实例化安全模板过程中的一些很微妙的地方。如果 X7 是被传递给非实例化安全的 min()，那么可以成功实例化。但是对于实例化安全的 min()，实例化却会失败，因为 BoolLike 不可以被隐式的转换成 bool 类型。这里的区别很微妙：在某些情况下，显式的向 bool 的转换可以被隐式的使用，比如控制语句（if，while，for 以及 do）的布尔型条件，内置的！，&&以及||运算符，还有三元运算符?:。在这些情况下，该值被认为是“语境上可以转换成 bool”

但是，我们对一般的、可以隐式地向 bool 转换这一条件的坚持，导致实例化安全的模板被过分限制了；也就是说，在 EnableIf 中指定的条件要比我们实际需要的条件更为严格（正确实例化模板所需要的条件）。另一方面，如果我们完全忘记了可以向 bool 转换这一要求，那么对于 min()模板的要求就过于宽松了，这样的话对于某些类型可能会遇到实例化错误（比如 X6）。

为了解决 min()中这一由实例化安全带来的问题，我们需要一个可以判断某个类型是否是“语

境上可以转换成 bool”的萃取技术。控制流程语句对该萃取技术的实现没有帮助，因为语句不可以出现在 SFINAE 上下文中，同样的，可以被任意类型重载的逻辑操作也不可以。幸运的是，三元运算符?:是一个表达式，而且不可以被重载，因此它可以被用来测试一个类型是否是“语境上可以转换成 bool”的：

```
#include <utility> // for declval()
#include <type_traits> // for true_type and false_type
template<typename T>
class IsContextualBoolT {
  private:
      template<typename T> struct Identity;

      template<typename U>
      static std::true_type test(Identity<decltype(declval<U>()? 0 : 1)>*);

      template<typename U>
      static std::false_type test(…);
  public:
      static constexpr bool value = decltype(test<T> (nullptr))::value;
};

template<typename T>
constexpr bool IsContextualBool = IsContextualBoolT<T>::value;
```

有了这一萃取，我们就可以实现一个使用了正确的 EnableIf 条件且实例化安全的 min()了：

```
#include "iscontextualbool.hpp"
#include "lessresult.hpp"
template<typename T>
EnableIf<IsContextualBool<LessResult<T const&, T const&>>, T const&>
min(T const& x, T const& y)
{
  if (y < x) {
      return y;
  }
  return x;
}
```

将各种各样的条件检查，组合进描述了类型种类（比如前向迭代器）的萃取技术，并将这些萃取技术一起放在 EnableIf 的条件检查中，这一使 min()变得实例化安全的技术可以被推广到用于描述其它重要模板的条件。这样做一方面可以获得更好的重载行为，另一方面也可以避免在实例化深层次嵌套的模板时，编译器遇到错误后会产生过于冗长的错误信息的问题。但是此时的类错误信息通常不会指出具体是哪一个操作出了错误。而且，正如我们在 min()中展现的那样，准确的判断并编码相关的条件可能是让人抓狂的。我们在第 28.2 节探讨了使用了这些萃取的 debug 技术。

20.6 在标准库中的情况

C++标准库为输入，输出，前向，双向以及随机访问迭代器提供了迭代器标记，我们对这些都已经做了展示。这些迭代器标记是标准迭代器萃取（std::iterator_traits）技术以及施加于迭代器的需求的一部分，因此它们可以被安全得用于标记派发。

C++11 标准库中的 std::enable_if 模板提供了和我们所展示的 EnableIf 相同的行为。唯一的不同是标准库用了一个小写的成员类型 type，而我们使用的是 Type。

算法的偏特化在 C++标准库中被用在了很多地方。比如，std::advance()以及 std::distance()基于其迭代器参数的种类的不同，都有很多变体。虽然很多标准库的实现都倾向于使用标记派发（tag dispatch），但是最近其中一些实现也已经使用 std::enable_if 来进行算法特化了。而且，很多的 C++标准库的实现，在内部也都用这些技术去实现各种标准库算法的偏特化。比如，当迭代器指向连续内存且它们所指向的值有拷贝赋值运算符的时候，std::copy()可以通过调用 std::memory()和 std::memmove()来进行偏特化。同样的，std::fill()也可以通过调用 std::memset 进行优化，而且在知晓一个类型有一个普通的析构函数（trivial destructor）的情况下，很多算法都可以避免去调用析构函数。C++标准并没有对这些算法特化的实现方式进行统一（比如统一采用 std::advance()和 std::distance()的方式），但是实现者还是为了性能而选择类似的方式。

正如第 8.4 节介绍的那样，C++标准库强烈的建议在其所需要施加的条件中使用 std::enable_if<>或者其它类似 SFINAE 的技术。比如，std::vector 就有一个允许其从迭代器序列进行构造的构造函数模板：

```
template<typename InputIterator>
vector(InputIterator first, InputIterator second,
allocator_type const& alloc = allocator_type());
```

它要求“当通过类型 InputIterator 调用构造函数的时候，如果该类型不属于输入迭代器（input iterator），那么该构造函数就不能参与到重载解析中”（参见第 23.2.3 节）。这一措辞并没有精确到足以使当前最高效的技术被应用到实现当中，但是在其被引入到标准中的时候，std::enable_if<>确实被寄予了这一期望。

20.7 后记

标记派发（tag dispatch）在 C++中已经存在很久了。它被用于最初版本的 STL 中，而且通常被和萃取（traits）一起使用。SFINAE 和 std::enable_if 的使用则要晚上很多：本书的第一版中介绍了 SFINAE 的概念，并展示了其在判断某个成员类型是否存在中的使用。

“enable if”这一技术最早是由 Jaakko J ¨ arvi, Jeremiah Will-cock, Howard Hinnant, 以及 Andrew Lumsdaine 在[OverloadingProperties]中发布的，在其中他们介绍了 EnableIf 模板，如何通过 EnableIf（和 DisableIf）实现函数重载，以及如何通过使用 EnableIf 实现类模板的偏特化。从那时起，EnableIf 以及类似的技术在高端模板库（包含 C++标准库）的实现中就已

经变得无处不在了。而且这些技术的流行也促使 C++11 对 SFINAE 进行了扩展（参见第 15.7 节）。Peter Dimov 是第一个注意到在不引入新的模板参数的情况下，函数模板的默认模板参数（C++11 新特性）可以让 EnableIf 在构造函数模板中的使用变成可能。

Concepts 这一语言特性预期会在 C++17 之后的标准中被引入。它很可能会使一些技术（包含 EnableIf）被废弃掉。与此同时，C++17 的 constexpr if 语句（参见第 8.5 节和第 20.3.3 节）也正在慢慢渗透进现代模板库中。

第 21 章 模板和继承

直觉上，模板和继承之间似乎并不应该存在什么有意思的交互。如果有的话，那么也应该是在第 13 章中介绍的，当从一个和模板参数有关的基类做继承的时候，必须仔细地对待那些不受限制的变量名。但是事实证明，一些有意思的技术恰恰结合了这两种技术，比如 Curiously Recurring Template Pattern（CRTP）和 MIXINS。本章将介绍其中的一些相关技术。

21.1 空基类优化（The Empty Class Optimization，EBCO）

C++中的类经常是“空”的，也就是说它们的内部表征在运行期间不占用内存。典型的情况是那写只包含类型成员，非虚成员函数，以及静态数据成员的类。而非静态数据成员，虚函数，以及虚基类，在运行期间则是需要占用内存的。

然而即使是空的类，其所占用的内存大小也不是零。如果愿意的话，运行下面的程序可以证明这一点：

```
#include <iostream>
class EmptyClass {
};

int main()
{
  std::cout << "sizeof(EmptyClass):" << sizeof(EmptyClass) << ' \n' ;
}
```

在某些平台上，这个程序会打印出 1。在少数对 class 类型实施了严格内存对齐要求的平台上，则可能会打印出其它结果（典型的结果是 4）.

21.1.1 布局原则

C++的设计者有很多种理由不去使用内存占用为零的 class。比如，一个存储了内存占用为零的 class 的数组，其内存占用也将是零，这样的话常规的指针运算规则都将不在适用。假设 ZeroSizedT 是一个内存占用为零的类型：

```
ZeroSizedT z[10];
...
&z[i] - &z[j] //compute distance between pointers/addresses
```

正常情况下，上述例子中的结果可以用两个地址之间的差值，除以该数组中元素类型的大小得到，但是如果元素所占用内存为零的话，上述结论显然不再成立。

虽然在 C++中没有内存占用为零的类型，但是 C++标准却指出，在空 class 被用作基类的时候，如果不给它分配内存并不会导致其被存储到与其它同类型对象或者子对象相同的地址上，那么就可以不给它分配内存。下面通过一些例子来看看实际应用中空基类优化（empty class optimization，EBCO）的意义。考虑如下程序：

```
#include <iostream>
class Empty {
  using Int = int;// type alias members don't make a class nonempty
};

class EmptyToo : public Empty {
};

class EmptyThree : public EmptyToo {
};

int main()
{
  std::cout << "sizeof(Empty): " << sizeof(Empty) << '\n';
  std::cout << "sizeof(EmptyToo): " << sizeof(EmptyToo) << '\n';
  std::cout << "sizeof(EmptyThree): " << sizeof(EmptyThree) << '\n';
}
```

如果你所使用的编译器实现了 EBCO 的话，它打印出来的三个 class 的大小将是相同的，但是它们的结果也都不会是零（参见图 21.1）。这意味着在 EmptyToo 中，Empty 没有被分配内存。注意一个继承自优化后的空基类（且只有这一个基类）的空类依然是空的。这就解释了为什么 EmptyThree 的大小和 Empty 相同。如果你所用的编译器没有实现 EBCO 的话，那么它打印出来的各个 class 的大小将是不同的（参见图 21.2）。

Figure 21.1. Layout of EmptyThree by a compiler that implements the EBCO

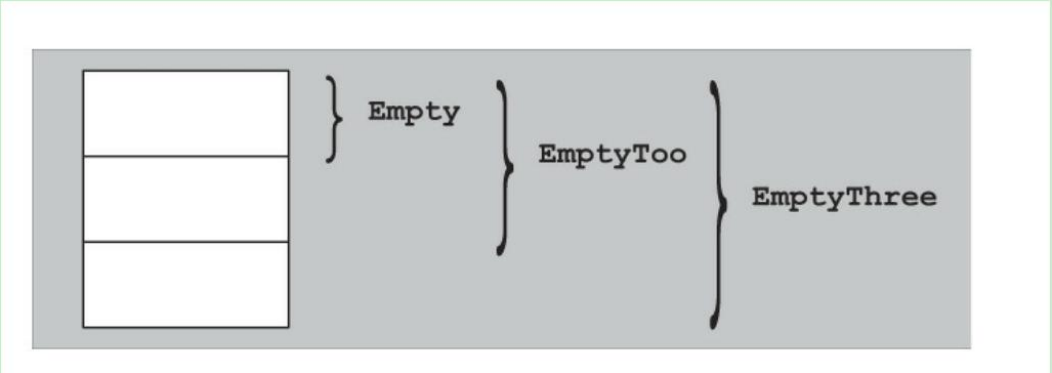

Figure 21.2. Layout of EmptyThree by a compiler that does not implement the EBCO

考虑一种 EBCO 不适用的情况：

```
#include <iostream>!
class Empty {
  using Int = int; // type alias members don't make a class nonempty
};

class EmptyToo : public Empty {
};

class NonEmpty : public Empty, public EmptyToo {
};

int main(){
  std::cout <<"sizeof(Empty): " << sizeof(Empty) <<' \n' ;
  std::cout <<"sizeof(EmptyToo): " << sizeof(EmptyToo) <<' \n' ;
  std::cout <<"sizeof(NonEmpty): " << sizeof(NonEmpty) <<' \n' ;
}
```

可能有点意外的是，NonEmpty 不再是一个空的类。毕竟它以及它的基类都没有任何数据成员。但是 NonEmpty 的基类 Empty 和 EmptyToo 不可以被分配到相同的地址上，因为这会导致 EmptyToo 的基类 Empty 和 NonEmpty 的基类 Empty 被分配到相同的地址。或者说两个类型相同的子对象会被分配到相同的地址上，而这在 C++布局规则中是不被允许的。你可能会想到将其中一个 Empty 基类的子对象放在偏移量为“0 字节”的地方，将另一个放在偏移量为“1 字节”的地方，但是完整的 NonEmpty 对象的内存占用依然不能是 1 字节，因为在一个包含了两个 NonEmpty 对象的数组中，第一个元素的 Empty 子对象不能和第二个元素中的 Empty 子对象占用相同的地址（参见图 21.3）。

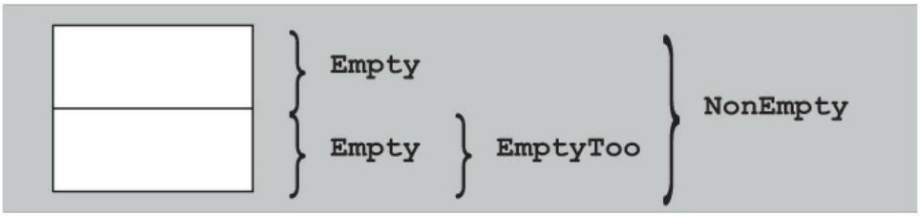

Figure 21.3. Layout of NonEmpty by a compiler that implements the EBCO

EBCO 之所以会有这一限制，是因为我们希望能够通过比较两个指针来确定它们所指向的是不是同一个对象。由于指针在程序中几乎总是被表示为单纯的地址，因此就需要我们来确保两个不同的地址（比如指针的值）指向的总是两个不同的对象。

这一限制可能看上去并不是那么重要。但是，在实践中却经常遇到，因为有些类会倾向于从一组空的、定义了某些基本类型别名的类做继承。当两个这一类 class 的子对象被用于同一个完整类型中的时候，这一优化方案会被禁止。

即使有这一限制，对于模板库而言 EBCO 也依然是一个重要的优化方案，因为有很多技术在引入基类的时候都只是为了引入一些新的类型别名或者额外的函数功能，而不会增加新的数据成员。在本章中会对其中些这一类的技术进行讨论。

21.1.2 将数据成员实现为基类

EBCO 和数据成员之间没有对等关系，因为（其中一个问题是）它会在用指针指向数据成员的表示上造成一些问题。结果就是，在有些情况下会期望将其实现为一个 private 的基类，这样粗看起来就可以将其视作成员变量。但是，这样做也并不是没有问题。

由于模板参数经常会被空 class 类型替换，因此在模板上下文中这一问题要更有意思一些，但是通常我们不能依赖这一规则。如果我们对类型参数一无所知，就不能很容易的使用 EBCO。考虑下面的例子：

```
template<typename T1, typename T2>
class MyClass {
  private:
      T1 a;
      T2 b;
  ...
};
```

其中的一个或者两个模板参数完全有可能被空 class 类型替换。如果真是这样，那么 MyClass<T1, T2>这一表达方式可能不是最优的选择，它可能会为每一个 MyClass<T1,T2>的实例都浪费一个字的内存。

这一内存浪费可以通过把模板参数作为基类使用来避免：

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

但是这一直接的替代方案也有其自身的缺点：

- 当 T1 或者 T2 被一个非 class 类型或者 union 类型替换的时候，该方法不再适用。
- 在两个模板参数被同一种类型替换的时候，该方法不再适用（虽然这一问题简单地通过增加一层额外的继承来解决，参见 513 页）。
- 用来替换 T1 或者 T2 的类型可能是 final 的，此时尝试从其派生出新的类会触发错误。

即使这些问题能够很好的解决，也还有一个严重的问题存在：给一个 class 添加一个基类，可能会从根本上改变该 class 的接口。对于我们的 MyClass 类，由于只有很少的接口会被影响到，这可能看上去不是一个重要的问题。但是正如在本章接下来的内容中将要看到的，从一个模板参数做继承，会影响到一个成员函数是否可以是 virtual 的。很显然，EBCO 的这一适用方式会带来各种各样的问题。

当已知模板参数只会被 class 类型替换，以及需要支持另一个模板参数的时候，可以使用另一种更实际的方法。其主要思想是通过使用 EBCO 将可能为空的类型参数与别的参数“合并”。比如，相比于这样：

```
template<typename CustomClass>
class Optimizable {
  private:
      CustomClass info; // might be empty
      void* storage;
  …
};
```

一个模板开发者会使用如下方式：

```
template<typename CustomClass>
class Optimizable {
  private:
      BaseMemberPair<CustomClass, void*> info_and_storage;
  …
};
```

虽然还没有看到 BaseMemberPari 的具体实现方式，但是可以肯定它的引入会使 Optimizable 的实现变得更复杂。但是很多的模板开发者都反应，相比于复杂度的增加，它带来的性能提升是值得的。我们会在第 25.5.1 节对这一内容做进一步讨论。

BaseMemberPair 的实现可以非常简洁：

```
#ifndef BASE_MEMBER_PAIR_HPP
#define BASE_MEMBER_PAIR_HPP
template<typename Base, typename Member>
class BaseMemberPair : private Base {
```

```
    private:
        Member mem;
    public:// constructor
        BaseMemberPair (Base const & b, Member const & m)
        : Base(b), mem(m) {
        }

        // access base class data via first()
        Base const& base() const {
        return static_cast<Base const&>(*this);
        }

        Base& base() {
        return static_cast<Base&>(*this);
        }

        // access member data via second()
        Member const& member() const {
        return this->mem;
        }

        Member& member() {
        return this->mem;
        }
};
#endif // BASE_MEMBER_PAIR_HPP
```

相应的实现需要使用 base()和 member()成员函数来获取被封装的（或者被执行了内存优化的）数据成员。

21.2 The Curiously Recurring Template Pattern (CRTP)

另一种模式是 CRTP。这一个有着奇怪名称的模式指的是将派生类作为模板参数传递给其某个基类的一类技术。该模式的一种最简单的 C++实现方式如下：

```
template<typename Derived>
class CuriousBase {
  ...
};

class Curious : public CuriousBase<Curious> {
  ...
};
```

上面的 CRTP 的例子使用了非依赖性基类（nondependent base class 参见 13.4 节）：Curious 不是一个模板类，因此它对在依赖性基类中遇到的名称可见性问题是免疫的。但是这并不是 CRTP 的固有特征。事实上，我们同样可以使用下面的这一实现方式：

```
template<typename Derived>
class CuriousBase {
  ...
};

template<typename T>
class CuriousTemplate : public CuriousBase<CuriousTemplate<T>> {
  ...
};
```

将派生类通过模板参数传递给其基类，基类可以在不使用虚函数的情况下定制派生类的行为。这使得 CRTP 对那些只能被实现为成员函数的情况（比如构造函数，析构函数，以及下表运算符）或者依赖于派生类的特性的情况很有帮助（This makes CRTP useful to factor out implementations that can only be member functions (e.g., constructor, destructors, and subscript operators) or are dependent on the derived class's identity.）。

一个 CRTP 的简单应用是将其用于追踪从一个 class 类型实例化出了多少对象。这一功能也可以通过在构造函数中递增一个 static 数据成员、并在析构函数中递减该数据成员来实现。但是给不同的 class 都提供相同的代码是一件很无聊的事情，而通过一个基类（非 CRTP）实现这一功能又会将不同派生类实例的数目混杂在一起。事实上，可以实现下面这一模板：

```
#include <cstddef>
template<typename CountedType>
class ObjectCounter {
  private:
      inline static std::size_t count = 0; // number of existing objects
  protected:
      // default constructor
      ObjectCounter() {
          ++count;
      }
      // copy constructor
      ObjectCounter (ObjectCounter<CountedType> const&) {
          ++count;
      }
      // move constructor
      ObjectCounter (ObjectCounter<CountedType> &&) {
          ++count;
      }

      // destructor
      ~ObjectCounter() {
```

```
            --count;
        }
    public:
        // return number of existing objects:
        static std::size_t live() {
            return count;
        }
};
```

注意这里为了能够在 class 内部初始化 count 成员，使用了 inline。在 C++17 之前，必须在 class 模板外面定义它：

```
template<typename CountedType>
class ObjectCounter {
  private:
    static std::size_t count; // number of existing objects
    ...
};

// initialize counter with zero:
template<typename CountedType>
std::size_t ObjectCounter<CountedType>::count = 0;
```

当我们想要统计某一个 class 的对象（未被销毁）数目时，只需要让其派生自 ObjectCounter 即可。比如，可以按照下面的方式统计 MyString 的对象数目：

```
#include "objectcounter.hpp"
#include <iostream>

template<typename CharT>
class MyString : public ObjectCounter<MyString<CharT>> {
  ...
};

int main()
{
  MyString<char> s1, s2;
  MyString<wchar_t> ws;
  std::cout << "num of MyString<char>: "
  << MyString<char>::live() << '\n';
  std::cout << "num of MyString<wchar_t>: "
  << ws.live() << '\n';
}
```

21.2.1 The Barton-Nackman Trick

在 1994 年，John J.Barton 和 Lee R.Nackman 提出了一种被称为 restricted template expansion 的技术。该技术产生的动力之一是：在当时，函数模板的重载是严重受限的，而且 namespace 在当时也不为大多数编译器所支持。

为了说明这一技术，假设我们有一个需要为之定义 operator ==的类模板 Array。一个可能的方案是将该运算符定义为类模板的成员，但是由于其第一个参数（绑定到 this 指针上的参数）和第二个参数的类型转换规则不同（为什么？一个是指针？一个是 Arry 类型？）。由于我们希望 operator ==对其参数是对称的，因此更倾向与将其定义为某一个 namespace 中的函数。一种很直观的实现方式可能会像下面这样：

```
template<typename T>
class Array {
  public:
      …
};

template<typename T>bool operator== (Array<T> const& a, Array<T> const&
b)
{
  …
}
```

不过如果函数模板不可以被重载的话，这会引入一个问题：在当前作用域内不可以再声明其它的 operator ==模板，而其它的类模板却又很可能需要这样一个类似的模板。Barton 和 Nackman 通过将 operator ==定义成 class 内部的一个常规友元函数解决了这一问题：

```
template<typename T>
class Array {
static bool areEqual(Array<T> const& a, Array<T> const& b);
  public:
      …
      friend bool operator== (Array<T> const& a, Array<T> const& b)
      {
          return areEqual(a, b);
      }
};
```

假设我们用 float 实例化了该 Array 类。作为实例化的结果，该友元运算符函数也会被连带声明，但是请注意该函数本身并不是一个函数模板的实例。作为实例化过程的一个副产品，它是一个被注入到全局作用域的常规非模板函数。由于它是非模板函数，即使在重载函数模板的功能被引入之前，也可以用其它的 operator ==对其进行重载。由于这样做避免了去定义一个适用于所有类型 T 的 operator ==(T, T)模板，Barton 和 Nackman 将其称为 restricted template expansion。

由于

```
operator== (Array<T> const&, Array<T> const&)
```

被定义在一个 class 的定义中，它会被隐式地当作 inline 函数，因此我们决定将其实现委托给一个 static 成员函数（不需要是 inline 的）。

从 1994 年开始，friend 函数定义的查找方式就已经变了，因此在标准 C++中，Barton-Nackman 的方法就不再那么有用了。在其刚被发明出来的时候，如果要通过 friend name injection 实例化模板，就需要 friend 函数的声明在类模板的闭合作用域内是可见的。而标准 C++则通过参数依赖（argument-dependent lookup，参见第 13.2.2 节）来查找 friend 函数。这意味着在函数的调用参数中，至少要有一个参数需要有一个包含了 friend 函数的关联类。如果参数的类型是无关的 class 类型，即使该类型可以被转成包含了 friend 函数的 class 类型，也无法找到该 friend 函数。比如：

```
class S {
};

template<typename T>
class Wrapper {
  private:
      T object;
  public:
      Wrapper(T obj) : object(obj) { //implicit conversion from T to Wrapper<T>
      }

      friend void foo(Wrapper<T> const&) {
      }
};

int main()
{
  S s;
  Wrapper<S> w(s);
  foo(w); // OK: Wrapper<S> is a class associated withw
  foo(s); // ERROR: Wrapper<S> is not associated with s
}
```

此处的 foo(w)调用是有效的，因为 foo()是被定义于 Wrapper<S>中的友元，而 Wrapper<s>又是与参数 w 有关的类。但是在 foo(s)的调用中，friend foo(Wrapper<S> const &)的声明并不可见，这是因为定义了 foo(Wrapper<S> const &)的类 Wrapper<S>并没有和 S 类型的参数 s 关联起来。因此，虽然在类型 S 和 Wrapper<S>之间有一个隐式的类型转换（通过 Wrapper<S>的构造函数），但是由于一开始就没有找到这个 foo()函数，所以这个转换函数永远都不会被考虑。而在 Barton 和 Nackman 发明它们这个方法的时候，friend 名称注射机制会让 freind foo()可见，因此也就可以成功调用 foo(s)。

在 modern C++中，相比于直接定义一个函数模板，在类模板中定义一个 friend 函数的好处是：友元函数可以访问该类模板的 private 成员以及 protected 成员，并且无需再次申明该类

模板的所有模板参数。但是，在与 Curiously Recurring Template Pattern(CRTP)结合之后，friend 函数定义可以变的更有用一些，就如在下面一节中节介绍的那样。

21.2.2 运算符的实现（Operator Implementations）

在给一个类重载运算符的时候，通常也需要重载一些其它的（当然也是相关的）运算符。比如，一个实现了 operator ==的类，通常也会实现 operator !=，一个实现了 operator <的类，通常也会实现其它的关系运算符（>，<=，>=）。在很多情况下，这些运算符中只有一个运算符的定义比较有意思，其余的运算符都可以通过它来定义。例如，类 X 的 operator !=可以通过使用 operator ==来定义：

```
bool operator!= (X const& x1, X const& x2) {
  return !(x1 == x2);
}
```

对于那些 operator !=的定义类似的类型，可以通过模板将其泛型化：

```
template<typename T>
bool operator!= (T const& x1, T const& x2) {
  return !(x1 == x2);
}
```

事实上，在 C++标准库的<utility>头文件中已经包含了类似的定义。但是，一些别的定义（比如！=， >，<=和>=）在标准化过程中则被放到了 namespace std::rel_ops 中，因为当时可以确定如果让它们在 std 中可见的话，会导致一些问题。实际上，如果让这些定义可见的话，会使得任意类型都有一个!= operator (虽然实例化有可能失败)，而且对于其两个参数而言该 operator 也总会是最匹配的。

虽然上述第一个问题可以通过 SFINAE 技术解决（参见 19.4 节），这样的话这个!= operator 的定义只会在某种类型有合适的== operator 时才会被进行相应的实例化。但是第二个问题依然存在：相比于用户定义的需要进行从派生类到基类的转化的!= operator，上述通用的!= operator 定义总是会被优先选择，这有时会导致意料之外的结果。

另一种基于 CRTP 的运算符模板形式，则允许程序去选择泛型的运算符定义（假设为了增加对代码的重用不会引入过度泛型化的问题）：

```
template<typename Derived>
class EqualityComparable
{
  public:
      friend bool operator!= (Derived const& x1, Derived const& x2)
      {
          return !(x1 == x2);
      }
};
```

```
class X : public EqualityComparable<X>
{
  public:
  friend bool operator== (X const& x1, X const& x2) {
      // implement logic for comparing two objects of type X
  }
};

int main()
{
  X x1, x2;
  if (x1 != x2) { }
}
```

此处我们结合使用了 CRTP 和 Barton-Nackman 技术。EqualityComparable<>为了基于派生类中定义的 operator==给其派生类提供 operator !=，使用了 CRTP。事实上这一定义是通过 friend 函数定义的形式提供的（Barton-Nackman 技术），这使得两个参数在类型转换时的 operator != 行为一致。

当需要将一部分行为分解放置到基类中，同时需要保存派生类的标识时，CRTP 会很有用。结合 Barton-Nackman，CRTP 可以基于一些简单的运算符为大量的运算符提供统一的定义。这些特性使得 CRTP 和 Barton-Nackman 技术被 C++模板库的开发者所钟爱。

21.2.3 Facades

将 CRTP 和 Barton-Nackman 技术用于定义某些运算符是一种很简便的方式。我们可以更近一步，这样 CRTP 基类就可以通过由 CRTP 派生类暴露出来的相对较少（但是会更容易实现）的接口，来定义大部分甚至是全部 public 接口。这一被称为 facade 模式的技术，在定义需要支持一些已有接口的新类型（数值类型，迭代器，容器等）时非常有用。

为了展示 facade 模式，我们为迭代器实现了一个 facade，这样可以大大简化一个符合标准库要求的迭代器的编写。一个迭代器类型（尤其是 random access iterator）所需要支持的接口是非常多的。下面的一个基础版的 IteratorFacade 模板展示了对迭代器接口的要求：

```
template<typename Derived, typename Value, typename Category,
typename Reference = Value&, typename Distance = std::ptrdiff_t>
class IteratorFacade
{
  public:
      using value_type = typename std::remove_const<Value>::type;
      using reference = Reference;
      using pointer = Value*;
      using difference_type = Distance;
      using iterator_category = Category;
```

```
        // input iterator interface:
        reference operator *() const { … }
        pointer operator ->() const { … }
        Derived& operator ++() { … }
        Derived operator ++(int) { … }

        friend bool operator== (IteratorFacade const& lhs,
        IteratorFacade const& rhs) { … }
        …

        // bidirectional iterator interface:
        Derived& operator --() { … }
        Derived operator --(int) { … }

        // random access iterator interface:
        reference operator [](difference_type n) const { … }
        Derived& operator +=(difference_type n) { … }
        …

        friend difference_type operator -(IteratorFacade const& lhs,
        IteratorFacade const& rhs) {
            …
        }

        friend bool operator <(IteratorFacade const& lhs,
        IteratorFacade const& rhs) { … }
        …
};
```

为了简洁，上面代码中已经省略了一部分声明，但是即使只是给每一个新的迭代器实现上述代码中列出的接口，也是一件很繁杂的事情。幸运的是，可以从这些接口中提炼出一些核心的运算符：

- 对于所有的迭代器，都有如下运算符：
  - 解引用（dereference）：访问由迭代器指向的值（通常是通过 operator *和->）。
  - 递增（increment）：移动迭代器以让其指向序列中的下一个元素。
  - 相等（equals）：判断两个迭代器指向的是不是序列中的同一个元素。

- 对于双向迭代器，还有：
  - 递减（decrement）：移动迭代器以让其指向列表中的前一个元素。

- 对于随机访问迭代器，还有：
  - 前进（advance）：将迭代器向前或者向后移动 n 步。
  - 测距（measureDistance）：测量一个序列中两个迭代器之间的距离。

Facade 的作用是给一个只实现了核心运算符（core operations）的类型提供完整的迭代器接口。IteratorFacade 的实现就涉及到到将迭代器语法映射到最少量的接口上。在下面的例子中，我们通过成员函数 asDerived()访问 CRTP 派生类：

```
Derived& asDerived() {
  return *static_cast<Derived*>(this);
}

Derived const& asDerived() const {
  return *static_cast<Derived const*>(this);
}
```

有了以上定义，facade 中大部分功能的实现就变得很直接了。下面只展示一部分的迭代器接口，其余的实现都很类似：

```
reference operator*() const {
  return asDerived().dereference();
}

Derived& operator++() {
  asDerived().increment();
  return asDerived();
}

Derived operator++(int) {
  Derived result(asDerived());
  asDerived().increment();
  return result;
}

friend bool operator== (IteratorFacade const& lhs, IteratorFacade const& rhs) {
  return lhs.asDerived().equals(rhs.asDerived());
}
```

定义一个链表的迭代器

结合以上 IteratorFacade 的定义，可以容易地定义一个指向简单链表的迭代器。比如，链表中节点的定义如下：

```
template<typename T>
class ListNode
{
  public:
      T value;
      ListNode<T>* next = nullptr;
```

```
        ~ListNode() { delete next; }
};
```

通过使用 IteratorFacade，可以以一种很直接的方式定义指向这样一个链表的迭代器：

```
template<typename T>
class ListNodeIterator
: public IteratorFacade<ListNodeIterator<T>, T,
std::forward_iterator_tag>
{
  ListNode<T>* current = nullptr;
  public:
      T& dereference() const {
          return current->value;
      }

      void increment() {
          current = current->next;
      }

      bool equals(ListNodeIterator const& other) const {
      return current == other.current;
      }

      ListNodeIterator(ListNode<T>* current = nullptr) :
      current(current) { }
};
```

ListNodeIterator 在使用很少量代码的情况下，提供了一个前向迭代器（forward iterator）所需要的所有运算符和嵌套类型。接下来会看到，即使是实现一个比较复杂的迭代器（比如，随机访问迭代器），也只需要再额外执行少量的工作。

隐藏接口

上述 ListNodeIterator 实现的一个缺点是，需要将 dereference()，advance()和 equals()运算符暴露成 public 接口。为了避免这一缺点，可以重写 IteratorFacade：通过一个单独的访问类（access class），来执行其所有作用于 CRTP 派生类的运算符操作。我们称这个访问类为 IteratorFacadeAccess：

```
// ‘friend’ this class to allow IteratorFacade access to core iterator operations:
class IteratorFacadeAccess
{
  // only IteratorFacade can use these definitions
  template<typename Derived, typename Value, typename Category,
  typename Reference, typename Distance>
```

```
    friend class IteratorFacade;

    // required of all iterators:
    template<typename Reference, typename Iterator>
    static Reference dereference(Iterator const& i) {
        return i.dereference();
    }

    ...
    // required of bidirectional iterators:
    template<typename Iterator>
    static void decrement(Iterator& i) {
        return i.decrement();
    }

    // required of random-access iterators:
    template<typename Iterator, typename Distance>
    static void advance(Iterator& i, Distance n) {
        return i.advance(n);
    }
    ...
};
```

该 class 为每一个核心迭代器操作都提供了对应的 static 成员函数，它们会调用迭代器中相应的（nonstatic）成员函数。所有的 static 成员函数都是 private 的，只有 IteratorFacade 才可以访问它们。因此，我们的 ListNodeIterator 可以将 IteratorFacadeAccess 当作 friend，并把 facade 所需要的接口继续保持为 private 的：

```
friend class IteratorFacadeAccess;
```

迭代器的适配器（Iterator Adapters）

使用我们的 IteratorFacade 可以很容易的创建一个迭代器的适配器，这样就可以基于已有的迭代器生成一个提供了对底层序列进行了视角转换的新的迭代器。比如，可能有一个存储了 Person 类型数值的容器：

```
struct Person {
  std::string firstName;
  std::string lastName;

  friend std::ostream& operator<<(std::ostream& strm, Person const& p) {
      return strm << p.lastName << ", " << p.firstName;
  }
};
```

但是，相对于编译容器中所有 Person 元素的值，我们可能只是想得到其 first name。在本节中我们会开发一款迭代器的适配器（称之为 ProjectionIterator），通过它可以将底层迭代器（base）“投射”到一些指向数据成员的指针（pointer-to-data member）上，比如：Person::firstName。

ProjectionIterator 是依据基类迭代器以及将要被迭代器暴露的数值类型定义的一种迭代器：

```
template<typename Iterator, typename T>
class ProjectionIterator
: public IteratorFacade<ProjectionIterator<Iterator, T>, T, typename
std::iterator_traits<Iterator>::iterator_category, T&, typename
std::iterator_traits<Iterator>::difference_type>
{
  using Base = typename std::iterator_traits<Iterator>::value_type;
  using Distance = typename std::iterator_traits<Iterator>::difference_type;
  Iterator iter;
  T Base::* member;
  friend class IteratorFacadeAccess
  ...
  //implement core iterator operations for IteratorFacade
  public:
      ProjectionIterator(Iterator iter, T Base::* member)
      : iter(iter), member(member) { }
};

template<typename Iterator, typename Base, typename T>
auto project(Iterator iter, T Base::* member) {
  return ProjectionIterator<Iterator, T>(iter, member);
}
```

每一个 projection iterator 都定存储了两个值：iter（指向底层序列的迭代器），以及 member（一个指向数据成员的指针，表示将要投射到的成员）。有了这一认知，我们来考虑传递给基类 IteratorFacade 的模板参数。第一个是 ProjectionIterator 本身（为了使用 CRTP）。第二个参数（T）和第四个参数（T&）是我们的 projection iterator 的数值和引用类型，将其定义成 T 类型数值的序列。第三和第五个参数仅仅只是传递了底层迭代器的种类的不同类型。因此，如果 Iterator 是 input iterator 的话，我们的 projection iterator 也将是 input iterator，如果 Iterator 是双向迭代器的话，我们的 projection iterator 也将是双向迭代器，以此类推。Project()函数则使得 projection iterator 的构建变得很简单。

唯一缺少的是对 IteratorFacade 核心需求的实现。最有意思的是 dereference()，它会解引用底层迭代器并投射到指向数据成员的指针：

```
T& dereference() const {
  return (*iter).*member;
}
```

其余操作是依照底层迭代器实现的：

```
  void increment() {
    ++iter;
  }

  bool equals(ProjectionIterator const& other) const {
    return iter == other.iter;
  }

  void decrement() {
    --iter;
  }
```

为了简单起见，我们忽略了对随机访问迭代器的定义，但是其实现是类似的。

就这些！通过使用 projection iterator，我们可以打印出存储在 vector 中的 Person 数值的 first name：

```
  #include <vector>
  #include <algorithm>
  #include <iterator>
  int main()
  {
    std::vector<Person> authors = { {"David", "Vandevoorde"},
        {"Nicolai", "Josuttis"},
        {"Douglas", "Gregor"} };

    std::copy(project(authors.begin(), &Person::firstName),
        project(authors.end(), &Person::firstName),
        std::ostream_iterator<std::string>(std::cout, "\n"));
  }
```

Facade 模式在创建需要符合特定接口的新类型时异常有用。新的类型只需要向 facade 暴露出少量和核心操作，后续 facade 会通过结合使用 CRTP 和 Barton-Nackman 技术提供完整且正确的 public 接口。

21.3 Mixins（混合？）

考虑一个包含了一组点的简单 Polygon 类：

```
  class Point
  {
    public:
        double x, y;
```

```
        Point() : x(0.0), y(0.0) { }
        Point(double x, double y) : x(x), y(y) { }
};

class Polygon
{
  private:
        std::vector<Point> points;
  public:
        … //public operations
};
```

如果可以扩展与每个 Point 相关联的一组信息的话（比如包含特定应用中每个点的颜色，或者给每个点加个标签），那么 Polygon 类将变得更为实用。实现该扩展的一种方式是用点的类型对 Polygon 进行参数化：

```
template<typename P>
class Polygon
{
  private:
        std::vector<P> points;
  public:
        … //public operations
};
```

用户可以通过继承创建与 Point 类似，但是包含了特定应用所需数据，并且提供了与 Point 相同的接口的类型：

```
class LabeledPoint : public Point
{
  public:
        std::string label;
        LabeledPoint() : Point(), label("") { }
        LabeledPoint(double x, double y) : Point(x, y), label("") {
        }
};
```

这一实现方式有其自身的缺点。比如，首先需要将 Point 类型暴露给用户，这样用户才能从它派生出自己的类型。而且 LablePoint 的作者也需要格外小心地提供与 Point 完全一样的接口（比如，继承或者提供所有与 Point 相同的构造函数），否则在 Polygon 中使用 LabledPoint 的时候会遇到问题。这一问题在 Point 随 Polygon 模板版本发生变化时将会变得更加严重：如果给 Point 新增一个构造函数，就需要去更新所有的派生类。

Mixins 是另一种可以客制化一个类型的行为但是不需要从其进行继承的方法。事实上，Mixins 反转了常规的继承方向，因为新的类型被作为类模板的基类“混合进”了继承层级中，而不是被创建为一个新的派生类。这一方式允许在引入新的数据成员以及某些操作的时候，不需

要去复制相关接口。

一个支持了 mixins 的类模板通常会接受一组任意数量的 class，并从之进行派生：

```
template<typename… Mixins>
class Point : public Mixins…
{
  public:
      double x, y;
      Point() : Mixins()…, x(0.0), y(0.0) { }
      Point(double x, double y) : Mixins()…, x(x), y(y) { }
};
```

现在，我们就可以通过将一个包含了 label 的基类“混合进来（mix in）”来生成一个 LabledPoint：

```
class Label
{
  public:
      std::string label;
      Label() : label("") { }
};

using LabeledPoint = Point<Label>;
```

甚至是“mix in”几个基类：

```
class Color
{
  public:
      unsigned char red = 0, green = 0, blue = 0;
};

using MyPoint = Point<Label, Color>;
```

有了这个基于 mixin 的 Point，就可以在不改变其接口的情况下很容易的为 Point 引入额外的信息，因此 Polygon 的使用和维护也将变得相对简单一些。为了访问相关数据和接口，用户只需进行从 Point 到它们的 mixin 类型（Label 或者 Color）之间的隐式转化即可。而且，通过提供给 Polygon 类模板的 mixins，Point 类甚至可以被完全隐藏：

```
template<typename… Mixins>
class Polygon
{
  private:
      std::vector<Point<Mixins…>> points;
  public:
      … //public operations
};
```

当需要对模板进行少量客制化的时候，Mixins 会很有用，比如在需要用用户指定的数据去装饰内部存储的对象时，使用 mixins 就不需要将内部数据类型和接口暴露出来并写进文档。

21.3.1 Curious Mixins

在和第 21.2 节介绍的 CRTP 一起使用的时候，Mixins 会变得更强大。此时每一个 mixins 都是一个以派生类为模板参数的类模板，这样就允许对派生类做额外的客制化。一个 CRTP-mixin 版本的 Point 可以被下称下面这样：

```
template<template<typename>… Mixins>
class Point : public Mixins<Point>…
{
  public:
      double x, y;
      Point() : Mixins<Point>()…, x(0.0), y(0.0) { }
      Point(double x, double y) : Mixins<Point>()…, x(x), y(y) { }
};
```

这一实现方式需要对那些将要被混合进来（mix in）的类做一些额外的工作，因此诸如 Label 和 Color 一类的 class 需要被调整成类模板。但是，现在这些被混合进来的 class 的行为可以基于其降要被混合进的派生类进行调整。比如，我们可以将前述的 ObjectCounter 模板混合进 Point，这样就可以统计在 Polygon 中创建的点的数目。

21.3.2 Parameterized Virtuality（虚拟性的参数化）

Minxins 还允许我们去间接的参数化派生类的其它特性，比如成员函数的虚拟性。下面的简单例子展示了这一令人称奇的技术：

```
#include <iostream>
class NotVirtual {
};

class Virtual {
  public:
      virtual void foo() {
      }
};

template<typename… Mixins>
class Base : public Mixins…
{
  public:
```

```
        // the virtuality of foo() depends on its declaration
        // (if any) in the base classes Mixins…
        void foo() {
            std::cout << "Base::foo()" << '\n';
        }
};

template<typename… Mixins>
class Derived : public Base<Mixins…> {
  public:
        void foo() {
            std::cout << "Derived::foo()" << '\n';
        }
};

int main()
{
  Base<NotVirtual>* p1 = new Derived<NotVirtual>;
  p1->foo(); // calls Base::foo()
  Base<Virtual>* p2 = new Derived<Virtual>;
  p2->foo(); // calls Derived::foo()
}
```

该技术提供了这样一种工具，使用它可以设计出一个既可以用来实例化具体的类，也可以通过继承对其进行扩展的类模板。但是，要获得一个可以为某些更为特化的功能产生一个更好的基类的类，仅仅是针对某些成员函数进行虚拟化还是不够的。这一类开发方法需要更为基础的设计决策。更为实际的做法是设计两个不同的工具（类或者类模板层级），而不是将它们集成进一个模板层级。

21.4 Named Template Arguments（命名的模板参数）

不少模板技术有时会导致类模板包含很多不同的模板类型参数。但是，其中一些模板参数通常都会有合理的默认值。其中一种这一类模板的定义方式可能会向下面这样：

```
template<typename Policy1 = DefaultPolicy1,
        typename Policy2 = DefaultPolicy2,
        typename Policy3 = DefaultPolicy3,
        typename Policy4 = DefaultPolicy4>
class BreadSlicer {
  …
};
```

可以想象，在使用这样一个模板时通常都可以使用模板参数的默认值。但是，如果需要指定某一个非默认参数的值的话，那么也需要指定该参数前面的所有参数的值（虽然使用的可能

是它们的默认值）。

很显然，我们更倾向于使用 BreadSlicer<Policy3 = Custom> 的形式，而不是 BreadSlicer<DefaultPolicy1, DefaultPolicy2, Custom>。在下面的内容在，我们开发了一种几乎可以完全实现以上功能的技术。

我们的技术方案是将默认类型放在一个基类中，然后通过派生将其重载。相比与直接指定类型参数，我们会通过辅助类（helper classes）来提供相关信息。比如我们可以将其写成这样 BreadSlicer<Policy3_is<Custom>>。由于每一个模板参数都可以表述任一条款，默认值就不能不同。或者说，在更高的层面上，每一个模板参数都是等效的：

```
template<typename PolicySetter1 = DefaultPolicyArgs,
  typename PolicySetter2 = DefaultPolicyArgs,
  typename PolicySetter3 = DefaultPolicyArgs,
  typename PolicySetter4 = DefaultPolicyArgs>
class BreadSlicer {
  using Policies = PolicySelector<PolicySetter1,
                  PolicySetter2,
                  PolicySetter3,
                  PolicySetter4>;
  // use Policies::P1, Policies::P2, … to refer to the various policies
  …
};
```

剩余的挑战就是该如何设计 PolicySelector 模板了。必须将不同的模板参数融合进一个单独的类型，而且这个类型需要用那个没有指定默认值的类型去重载默认的类型别名成员。可以通过继承实现这一融合：

```
// PolicySelector<A,B,C,D> creates A,B,C,D as base classes
// Discriminator<> allows having even the same base class more than once
template<typename Base, int D>
class Discriminator : public Base {
};

template<typename Setter1, typename Setter2,
  typename Setter3, typename Setter4>
class PolicySelector : public Discriminator<Setter1,1>,
                       public Discriminator<Setter2,2>,
                       public Discriminator<Setter3,3>,
                       public Discriminator<Setter4,4>
{
};
```

注意此处对中间的 Discriminator 模板的使用。其要求不同的 Setter 类型是类似的（不能使用多个类型相同的直接基类。而非直接基类，则可以使用和其它基类类似的类型）。

正如之前提到的，我们将全部的默认值收集到基类中：

```
// name default policies as P1, P2, P3, P4
class DefaultPolicies {
  public:
      using P1 = DefaultPolicy1;
      using P2 = DefaultPolicy2;
      using P3 = DefaultPolicy3;
      using P4 = DefaultPolicy4;
};
```

但是，如果我们最终会从该基类继承很多次的话，需要额外小心的避免歧义。因此，此处需要确保对基类使用虚继承：

```
// class to define a use of the default policy values
// avoids ambiguities if we derive from DefaultPolicies more than once
class DefaultPolicyArgs : virtual public DefaultPolicies {
};
```

最后，我们也需要一些模板来重载掉那些默认的策略值：

```
template<typename Policy>
class Policy1_is : virtual public DefaultPolicies {
  public:
      using P1 = Policy; // overriding type alias
};

template<typename Policy>
class Policy2_is : virtual public DefaultPolicies {
  public:
      using P2 = Policy; // overriding type alias
};

template<typename Policy>
class Policy3_is : virtual public DefaultPolicies {
  public:
      using P3 = Policy; // overriding type alias
};

template<typename Policy>
class Policy4_is : virtual public DefaultPolicies {
  public:
      using P4 = Policy; // overriding type alias};
}
```

有了 Discriminator<>类模板的帮助，这就会产生一种层级关系，在其中所有的模板参数都是基类（参见图 21.4）。重要的一点是，所有的这些基类都有一个共同的虚基类 DefaultPolicies，

也正是它定义了 P1，P2，P3 和 P4 的默认值。但是 P3 在某一个派生类中被重新定义了（比如在 Policy3_is<>中）。根据作用域规则，该定义会隐藏掉在基类中定义的相应定义。这样，就不会有歧义了。

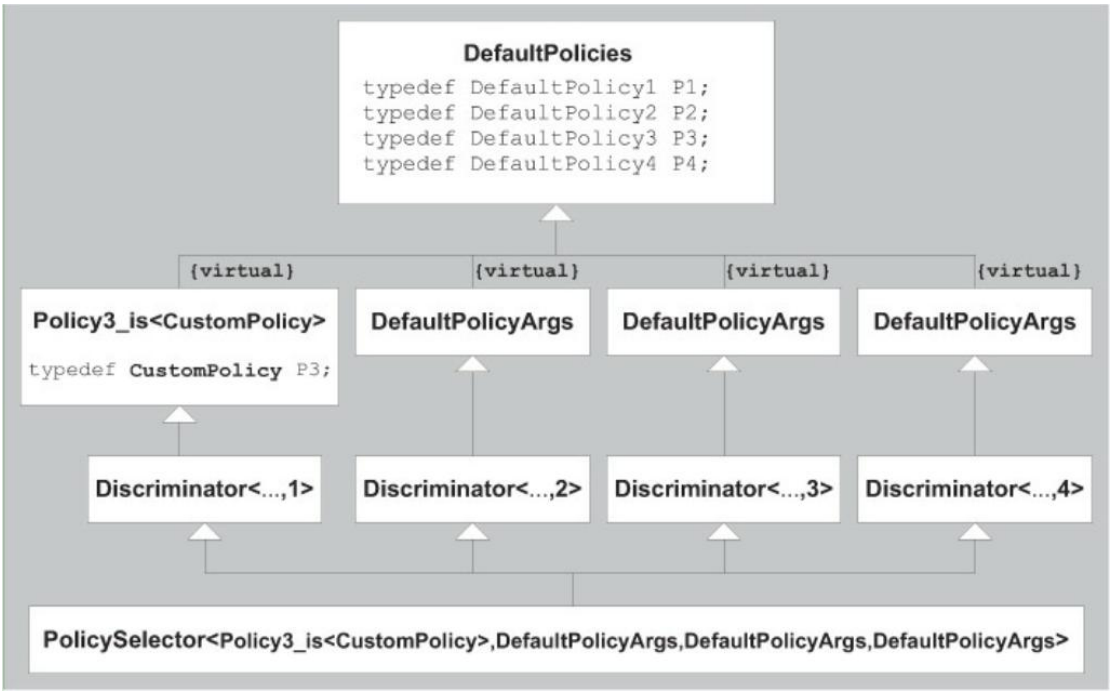

Figure 21.4. Resulting type hierarchy of BreadSlicer<>::Policies

在模板 BreadSlicer 中，可以使用 Policies::P3 的形式引用以上 4 中策略。比如:

```
template<…>
class BreadSlicer {
  …
  public:
      void print () {
          Policies::P3::doPrint();
      }
  …
};
```

在 inherit/namedtmpl.cpp 中，可以找到完整的例子。

虽然在上面开发的例子中只用到了四个模板类型参数，但是很显然该技术适用于任意数量的模板参数。注意，我们实际上永远不会对包含虚基类的辅助类进行实例化。因此，虽然它们是虚基类，但是并不会导致性能或者内存消耗的问题。

21.5 后记

Bill Gibbons 是将 EBCO 引入 C++背后的主要推手。Nathan Myers 则使其变得更加流行，并且提出了一个能够很好的利用 EBCO 特性的、类似于我们的 BaseMemberPair 的模板。在 Boost

库中有一个更为高端的模板（被称为 compressed_pair），其解决了我们在本章中提到的 MyClass 模板的一些问题。Boost::compressed_pair 也可以作为 BaseMemberPair 的替代品使用。

至少从 1991 年开始，CRTP 就已经被使用了。但是 Coplien 是第一个将其正式表述成一种设计模式的人。然后很多 CRTP 的应用就被发布了出来。短语参数化继承（parameterized inheritance ）有时候被错误的等同于 CRTP。如我们所展现的那样，CRTP 并不要求对派生进行参数化，而且某些形式的参数化继承也不符合 CRTP 规则。有时候 CRTP 也会和 Barton-Nackman 技术混淆，这是因为 Barton 和 Nackman 总是将 CRTP 和友元名称注入（friend name injection）一起使用（而后者才是 Barton-Nackman 技术的主要组成部分）。我们使用 CRTP 和 Barton-Nackman 技术实现运算符的方式，遵照了 Boost.Operators 库中用到的基本方法，该库提供了大量的运算符定义。类似的，我们实现迭代器的方法也遵照了 Boost.Iterator 库中用到的基本方法，该库为提供了一些核心迭代器运算符（相等，解引用，移动）的派生类提供了丰富的、符合标准库规范的迭代器接口。我们的 ObjectCounter 例子几乎和 Scott Meyers 在[MeyersCounting]中开发的技术相同。

在面向对象编程中，Mixins 的概念至少在 1986 年就已经存在，它被用作一种想 OO 类中引入一小部分功能的方式。在 C++中将模板用于 mixins 这一方式在第一版 C++标准发布之后开始变得流行。从那时开始，在 C++库的设计中，它就变成了一种流行的技术。

命名模板参数（named template arguments）被用来简化 Boost 库中的某些类模板。Boost 使用元编程技术创建了一种类似于 PolicySelector 的类型（但是没有使用虚继承）。这里介绍的一种更为简单的替代品是由我们中的一员开发的（Vandevoorde）。

第 22 章 桥接 static 和 dynamic 多态

在第 18 章中介绍了 C++中 static 多态（通过模板实现）和 dynamic 多态（通过继承和 virtual 函数实现）的本质。两种多态都给程序编写提供了功能强大的抽象，但是也都各有其不足：static 多态提供了和非多态代码一样的性能，但是其在运行期间所使用的类型在编译期就已经决定。而通过继承实现的 dynamic 多态，则允许单一版本的多态函数适用于在编译期未知的类型，但是该方式有点不太灵活，因为相关类型必须从统一的基类做继承。

本章将介绍在 C++中把 static 多态和 dynamic 多态桥接起来的方式，该方式具备了在第 18.3 节中介绍的各种模型的部分优点：比较小的可执行代码量，几乎全部的动态多态的编译期特性，以及（允许内置类型无缝工作的）静态多态的灵活接口。作为例子，我们将创建一个简化版的 std::function<>模板。

22.1 函数对象，指针，以及 std:function<>

在给模板提供定制化行为的时候，函数对象会比较有用。比如，下面的函数模板列举了从 0 到某个值之间的所有整数，并将每一个值都提供给了一个已有的函数对象 f：

```
#include <vector>
#include <iostream>
template<typename F>
void forUpTo(int n, F f){
  for (int i = 0; i != n; ++i)
  {
      f(i); // call passed function f for i
  }
}

void printInt(int i)
{
  std::cout << i << ' ' ;
}

int main()
{
  std::vector<int> values;
  // insert values from 0 to 4:
  forUpTo(5,
          [&values](int i) {
          values.push_back(i);
      });
```

```
    // print elements:
    forUpTo(5, printInt); // prints 0 1 2 3 4
    std::cout << ' \n' ;
}
```

其中 forUpTo()函数模板适用于所有的函数对象，包括 lambda，函数指针，以及任意实现了合适的 operator()运算符或者可以转换为一个函数指针或引用的类，而且每一次对 forUpTo()的使用都很可能产生一个不同的函数模板实例。上述例子中的函数模板非常小，但是如果该模板非常大的话，这些不同应用导致的实例化很可能会导致代码量的增加。

一个缓解代码量增加的方式是将函数模板转变为非模板的形式，这样就不再需要实例化。比如，我们可能会使用函数指针：

```
void forUpTo(int n, void (*f)(int))
{
  for (int i = 0; i != n; ++i)
  {
      f(i); // call passed function f for i
  }
}
```

但是，虽然在给其传递 printInt()的时候该方式可以正常工作，给其传递 lambda 却会导致错误：

```
forUpTo(5,
  printInt); //OK: prints 0 1 2 3 4

forUpTo(5,
      [&values](int i) { //ERROR: lambda not convertible to a function
pointer
      values.push_back(i);
  });
```

标准库中的类模板 std::functional<>则可以用来实现另一种类型的 forUpTo():

```
#include <functional>
void forUpTo(int n, std::function<void(int)> f)
{
  for (int i = 0; i != n; ++i)
  {
      f(i) // call passed function f for i
  }
}
```

Std::functional<>的模板参数是一个函数类型，该类型体现了函数对象所接受的参数类型以及其所需要产生的返回类型，非常类似于表征了参数和返回类型的函数指针。

这一形式的 forUpTo()提供了 static 多态的一部分特性：适用于一组任意数量的类型（包含函数指针，lambda，以及任意实现了适当 operator()运算符的类），同时又是一个只有一种实现的非模板函数。为了实现上述功能，它使用了一种称之为类型消除（type erasure）的技术，该技术将 static 和 dynamic 多态桥接了起来。

22.2 广义函数指针

Std::functional<>类型是一种高效的、广义形式的 C++函数指针，提供了与函数指针相同的基本操作：

- 在调用者对函数本身一无所知的情况下，可以被用来调用该函数。
- 可以被拷贝，move 以及赋值。
- 可以被另一个（函数签名一致的）函数初始化或者赋值。
- 如果没有函数与之绑定，其状态是“null”。

但是，与 C++函数指针不同的是，std::functional<>还可以被用来存储 lambda，以及其它任意实现了合适的 operator()的函数对象，所有这些情况对应的类型都可能不同。

在本章接下来的内容中，我们会实现一版自己的广义函数指针类模板（FunctionPtr），会给其提供相同的关键操作以及能力，并用之替换 std::functional:

```
#include "functionptr.hpp"
#include <vector>
#include <iostream>
void forUpTo(int n, FunctionPtr<void(int)> f)
{
  for (int i = 0; i != n; ++i)
  {
      f(i); // call passed function f for i
  }
}

void printInt(int i)
{
  std::cout << i << ' ';
}

int main()
{
  std::vector<int> values;
  // insert values from 0 to 4:
  forUpTo(5,[&values](int i) {
          values.push_back(i);
      });
  // print elements:
```

```
    forUpTo(5, printInt); // prints 0 1 2 3 4
    std::cout << ' \n' ;
}
```

FunctionPtr 的接口非常直观的提供了构造，拷贝，move，析构，初始化，以及从任意函数对象进行赋值，还有就是要能够调用其底层的函数对象。接口中最有意思的一部分是如何在一个类模板的偏特化中对其进行完整的描述，该偏特化将模板参数（函数类型）分解为其组成部分（返回类型以及参数类型）：

```
// primary template:
template<typename Signature>
class FunctionPtr;

// partial specialization:
template<typename R, typename… Args>
class FunctionPtr<R(Args…)>
{
  private:
      FunctorBridge<R, Args…>* bridge;
  public:
      // constructors:
      FunctionPtr() : bridge(nullptr) {
      }

      FunctionPtr(FunctionPtr const& other); // see
functionptrcpinv.hpp

      FunctionPtr(FunctionPtr& other)
      : FunctionPtr(static_cast<FunctionPtr const&>(other)) {
      }

      FunctionPtr(FunctionPtr&& other) : bridge(other.bridge) {
          other.bridge = nullptr;
      }

      //construction from arbitrary function objects:
      template<typename F> FunctionPtr(F&& f); // see
functionptrinit.hpp// assignment operators:
      FunctionPtr& operator=(FunctionPtr const& other) {
          FunctionPtr tmp(other);
          swap(*this, tmp);
          return *this;
      }

      FunctionPtr& operator=(FunctionPtr&& other) {
```

```
            delete bridge;
            bridge = other.bridge;
            other.bridge = nullptr;
            return *this;
        }

        //construction and assignment from arbitrary function objects:
        template<typename F> FunctionPtr& operator=(F&& f) {
            FunctionPtr tmp(std::forward<F>(f));
            swap(*this, tmp);
            return *this;
        }

        // destructor:
        ~FunctionPtr() {
            delete bridge;
        }

        friend void swap(FunctionPtr& fp1, FunctionPtr& fp2) {
            std::swap(fp1.bridge, fp2.bridge);
        }

        explicit operator bool() const {
            return bridge == nullptr;
        }

        // invocation:
        R operator()(Args… args) const; // see functionptr-cpinv.hpp
};
```

该实现包含了唯一一个非 static 的成员变量，bridge，它将负责被存储函数对象的储存和维护。该指针的所有权被绑定到了一个 FunctionPtr 的对象上，因此相关的大部分实现都只需要去操纵这个指针即可。代码中未被实现的、也是比较有意思的一部分，将在接下来的章节中进行介绍。

22.3 桥接接口（Bridge Interface）

FunctorBridge 类模板负责持有以及维护底层的函数对象，它被实现为一个抽象基类，为 FunctionPtr 的动态多态打下基础：

```
template<typename R, typename… Args>
class FunctorBridge
{
  public:
```

```
        virtual ~FunctorBridge() {
        }
        virtual FunctorBridge* clone() const = 0;
        virtual R invoke(Args… args) const = 0;
};
```

FunctorBridge 通过虚函数提供了用来操作被存储函数对象的必要操作：一个析构函数，一个用来执行 copy 的 clone()操作，以及一个用来调用底层函数对象的 invoke()操作。不要忘记将 clone()和 invoke()声明为 const 的成员函数。

有了这些虚函数，就可以继续实现 FunctionPtr 的拷贝构造函数和函数调用运算符了：

```
template<typename R, typename… Args>
FunctionPtr<R(Args…)>::FunctionPtr(FunctionPtr const& other)
: bridge(nullptr)
{
  if (other.bridge) {
  bridge = other.bridge->clone();
}
}

template<typename R, typename… Args>
R FunctionPtr<R(Args…)>::operator()(Args&&… args) const
{
  return bridge->invoke(std::forward<Args>(args)…);
}
```

22.4 类型擦除（Type Erasure）

FunctorBridge 的每一个实例都是一个抽象类，因此其虚函数功能的具体实现是由派生类负责的。为了支持所有可能的函数对象（一个无界集合），我们可能会需要无限多个派生类。幸运的是，我们可以通过用其所存储的函数对象的类型对派生类进行参数化：

```
template<typename Functor, typename R, typename… Args>
class SpecificFunctorBridge : public FunctorBridge<R, Args…> {
  Functor functor;
  public:
      template<typename FunctorFwd>
      SpecificFunctorBridge(FunctorFwd&& functor)
      : functor(std::forward<FunctorFwd>(functor)) {
      }

      virtual SpecificFunctorBridge* clone() const override {
          return new SpecificFunctorBridge(functor);
      }
```

```
        virtual R invoke(Args&&… args) const override {
            return functor(std::forward<Args>(args)…);
        }
};
```

每一个 SpecificFunctorBridge 的实例都存储了函数对象的一份拷贝（类型为 Functor），它可以被调用，拷贝（通过 clone()），以及销毁（通过隐式调用析构函数）。SpecificFunctorBridge 实例会在 FunctionPtr 被实例化的时候顺带产生，FunctionPtr 的剩余实现如下：

```
template<typename R, typename… Args>
template<typename F>
FunctionPtr<R(Args…)>::FunctionPtr(F&& f)
: bridge(nullptr)
{
  using Functor = std::decay_t<F>;
  using Bridge = SpecificFunctorBridge<Functor, R, Args…>;
  bridge = new Bridge(std::forward<F>(f));
}
```

注意，此处由于 FunctionPtr 的构造函数本身也被函数对象类型模板化了，该类型只为 SpecificFunctorBridge 的特定偏特化版本（以 Bridge 类型别名表述）所知。一旦新开辟的 Bridge 实例被赋值给数据成员 bridge，由于从派生类到基类的转换（Bridge* --> FunctorBridge<R, Args...>*），特定类型 F 的额外信息将会丢失。类型信息的丢失，解释了为什么名称“类型擦除”经常被用于描述用来桥接 static 和 dynamic 多态的技术。

该实现的一个特点是在生成 Functor 的类型的时候使用了 std::decay，这使得被推断出来的类型 F 可以被存储，比如它会将指向函数类型的引用 decay 成函数指针类型，并移除了顶层 const，volatile 和引用。

22.5 可选桥接（Optional Bridging）

上述 FunctionPtr 实现几乎可以被当作一个函数指针的非正式替代品适用。但是它并没有提供对下面这一函数指针操作的支持：检测两个 FunctionPtr 的对象是否会调用相同的函数。为了实现这一功能，需要在 FunctorBridge 中加入 equals 操作：

```
virtual bool equals(FunctorBridge const* fb) const = 0;
```

在 SpecificFunctorBridge 中的具体实现如下：

```
virtual bool equals(FunctorBridge<R, Args…> const* fb) const override
{
  if (auto specFb = dynamic_cast<SpecificFunctorBridge const*> (fb))
{
      return functor == specFb->functor;
  }
```

```
    //functors with different types are never equal:
    return false;
  }
```

最后可以为 FunctionPtr 实现 operator==，它会先检查对应内容是否是 null，然后将比较委托给 FunctorBridge:

```
  friend bool
  operator==(FunctionPtr const& f1, FunctionPtr const& f2) {
    if (!f1 || !f2) {
        return !f1 && !f2;
    }
    return f1.bridge->equals(f2.bridge);
  }

  friend bool
  operator!=(FunctionPtr const& f1, FunctionPtr const& f2) {
    return !(f1 == f2);
  }
```

该实现是正确的，但是不幸的是，它也有一个缺点：如果 FunctionPtr 被一个没有实现合适的 operator==的函数对象（比如 lambdas）赋值，或者是被这一类对象初始化，那么这个程序会遇到编译错误。这可能会很让人意外，因为 FunctionPtrs 的 operator==可能根本就没有被使用，却遇到了编译错误。而诸如 std::vector 之类的模板，只要它们的 operator==没有被使用，它们就可以被没有相应 operator==的类型实例化。

这一 operator==相关的问题是由类型擦除导致的：因为在给 FunctionPtr 赋值或者初始化的时候，我们会丢失函数对象的类型信息，因此在赋值或者初始化完成之前，就需要捕捉到所有所需要知道的该类型的信息。该信息就包含调用函数对象的 operator==所需要的信息，因为我们并不知道它在什么时候会被用到。

幸运的是，我们可以使用基于 SFINAE 的萃取技术（见 19.4 节），在调用 operator==之前，确认它是否可用，如下：

```
  #include <utility> // for declval()
  #include <type_traits> // for true_type and false_type
  template<typename T>
  class IsEqualityComparable
  {
    private:
        // test convertibility of == and ! == to bool:
        static void* conv(bool); // to check convertibility to bool

        template<typename U>
        static std::true_type test(decltype(conv(std::declval<U
  const&>() == std::declval<U const&>())),
```

```
    decltype(conv(!(std::declval<U const&>() == std::declval<U
    const&>())));

        // fallback:
        template<typename U>
        static std::false_type test(…);
    public:
        static constexpr bool value = decltype(test<T>(nullptr,
    nullptr))::value;
    };
```

上述 IsEqualityComparable 技术使用了在 19.4.1 节介绍的表达式测试萃取的典型形式：两个 test()重载，其中一个包含了被封装在 decltype 中的用来测试的表达式，另一个通过省略号接受任意数量的参数。第一个 test()试图通过==去比较两个 T const 类型的对象，然后确保两个结果都可以被隐式的转换成 bool，并将可以转换为 bool 的结果传递给 operator!=()。如果两个运算符都正常的话，参数类型都将是 void *。

使用 IsEqualityComparable，可以构建一个 TryEquals 类模板，它要么会调用==运算符（如果可用的话），要么就在没有可用的 operator==的时候抛出一个异常：

```
#include <exception>
#include "isequalitycomparable.hpp"
template<typename T, bool EqComparable =
IsEqualityComparable<T>::value>
struct TryEquals
{
  static bool equals(T const& x1, T const& x2) {
  return x1 == x2;
  }
};

class NotEqualityComparable : public std::exception
{ };

template<typename T>
struct TryEquals<T, false>
{
  static bool equals(T const& x1, T const& x2) {
      throw NotEqualityComparable();
  }
}
```

最后，通过在 SpecificFunctorBridge 中使用 TryEquals，当被存储的函数对象类型一致，而且支持 operator==的时候，就可以在 FunctionPtr 中提供对 operator==的支持：

```
virtual bool equals(FunctorBridge<R, Args…> const* fb) const override
```

```
{
  if (auto specFb = dynamic_cast<SpecificFunctorBridge const*>(fb)) {
      return TryEquals<Functor>::equals(functor, specFb->functor);
  }

  //functors with different types are never equal:
  return false;
}
```

22.6 性能考量

类型擦除技术提供了 static 和 dynamic 多态的一部分优点，但是并不是全部。尤其是，使用类型擦除技术产生的代码的性能更接近于动态多态，因为它们都是用虚函数实现了动态分配。因此某些 static 多态的传统优点（比如编译期将函数调用进行 inline 的能力）可能就被丢掉了。这一性能损失是否能够被察觉到，取决于具体的应用，但是通过比较被调用函数的运算量以及相关虚函数的运算量，有时候也很容易就能判断出来：如果二者比较接近，（比如 FunctionPtr 所作的只是对两个整数进行求和），类型擦除可能会比 static 多态要满很多。而如果函数调用执行的任务量比较大的话（比如访问数据库，对容器进行排列），那么 type erasure 带来的性能损失就很难被察觉到。

22.7 后记

通过引入 any 类型，Kevlin Henney 使得类型擦除在 C++中变得流行起来，随后演变成 Boost 中一个流行的库，以及 C++17 标准库的一部分。该技术在 Boost.Function 库中得到优化（主要是进行了性能和代码量的优化），并最终变成了 std::function<>。但是之前所有的库都只解决了一组操作相关的问题：任何只有 copy 和 case 操作的简单类型，函数都可以对其进行调用。

随后的一些工作，比如 Boost.TypeErasue 库以及 Adobe 的 Poly 库，通过使用模板元编程技术，让用户可以生成一个支持某些特定操作的、被擦除了类型的值。比如下面的类型（通过 Boost.TypeErase 库构建）就支持拷贝构造，类似 typeid 的操作，以及输出用来打印的输出 stream：

```
using AnyPrintable = any<mpl::vector<copy_constructible<>, typeid_<>,
ostreamable<> >>;
```

第 23 章 元编程

元编程的意思是“编写一个程序”。也就是说，我们构建了可以被编程系统用来产生新代码的代码，而且新产生的代码实现了我们真正想要的功能。通常名词“元编程”暗示了一种自反的属性：元编程组件是其将要为之产生一部分代码的程序的一部分（比如，程序中一些附加的或者不同的部分）。

为什么需要元编程？和其它编程技术一样，目的是用尽可能少的“付出”，换取尽可能多的功能，其中“付出”可以用代码长度、维护成本之类的事情来衡量。元编程的特性之一是在编译期间（at translation time，翻译是否准确？）就可以进行一部分用户定义的计算。其动机通常是性能（在 translation time 执行的计算通常可以被优化掉）或者简化接口（元-程序通常要比其展开后的结果短小一些），或者两者兼而有之。

元编程通常依赖于萃取和类型函数等概念，详见第 19 章。我们建议在继续接下来的内容之前，最好先熟悉下第 19 章的内容。

23.1 现代 C++元编程的现状

C++元编程是随着时间发展逐渐成形的。我们先来分类讨论多种在现代 C++中经常使用的元编程方法。

2.3.1.1 值元编程（Value Metaprogramming）

在本书第一版中，我们局限于原始 C++标准的特性（发布于 1988 年，在 2003 年做了修订）。在当时，构建简单的编译期（“meta-”）计算程序也会是一个小的挑战。因此我们曾在本章中花了很大篇幅来做这些事情；一个非常复杂的例子是在编译期间用递归的模板实例化来计算一个整数的平方根。不过正如 8.2 节介绍的那样，在 C++11，尤其是 C++14 中通过使用 constexpr 函数，可以大大降低这一挑战的难度。比如在 C++14 中，一个在编译期计算平方根的函数可以被简单的写成这样：

```
template<typename T>
constexpr T sqrt(T x)
{
    // handle cases where x and its square root are equal as a special
case to simplify
    // the iteration criterion for larger x:
    if (x <= 1) {
        return x;
    }
```

```
    // repeatedly determine in which half of a [lo, hi] interval the square
root of x is located,
    // until the interval is reduced to just one value:
    T lo = 0, hi = x;
    for (;;) {
        auto mid = (hi+lo)/2, midSquared = mid*mid;
        if (lo+1 >= hi || midSquared == x) {
          // mid must be the square root:
          return mid;
        }
        //continue with the higher/lower half-interval:
        if (midSquared < x) {
            lo = mid;
        } else {
            hi = mid;
        }
    }
}
```

该算法通过反复地选取一个包含 x 的平方根的中间值来计算结果（为了让收敛标准比较简单，对 0 和 1 的平方根做了特殊处理）。该 sqrt()函数可以被在编译期或者运行期间计算：

```
static_assert(sqrt(25) == 5, ""); //OK (evaluated at compile time)
static_assert(sqrt(40) == 6, ""); //OK (evaluated at compile time)
std::array<int, sqrt(40)+1> arr; //declares array of 7 elements (compile
time)
long long l = 53478;
std::cout << sqrt(l) << ' \n' ; //prints 231 (evaluated at run time)
```

在运行期间这一实现方式可能不是最高效的（在这里去开发机器的各种特性通常是值得的），但是由于该函数意在被用于编译期计算，绝对的效率并没有可移植性重要。注意在这个例子中并没有什么“模板魔法”，只是用到了常规的模板参数推断。相关代码是“纯 C++”的，并没有特别难以理解的地方。

上面介绍的值元编程（比如在编译期间计算某些数值）偶尔会非常有用，但是在现代 C++中还有另外两种可用的元编程方式（在 C++14 和 C++17 中）：类型元编程和混合元编程。

23.1.2 类型元编程

在第 19 章中讨论某些萃取模板的时候已经遇到过一种类型元编程，它接受一个类型作为输入并输出一个新的类型。比如 RemoveReferenceT 类模板会计算引用类型所引用对象的真正类型。但是在第 19 章中实现的例子只会计算很初级的类型操作。通过递归的模板实例化--这也是主要的基于模板的元编程手段--我们可以实现更复杂的类型计算。

考虑如下例子：

```
// primary template: in general we yield the given type:
template<typename T>
struct RemoveAllExtentsT {
  using Type = T;
};
// partial specializations for array types (with and without bounds):
template<typename T, std::size_t SZ>
struct RemoveAllExtentsT<T[SZ]> {
    using Type = typename RemoveAllExtentsT<T>::Type;
};
template<typename T>
struct RemoveAllExtentsT<T[]> {
    using Type = typename RemoveAllExtentsT<T>::Type;
};
template<typename T>
using RemoveAllExtents = typename RemoveAllExtentsT<T>::Type;
```

这里 RemoveAllExtents 就是一种类型元函数（比如一个返回类型的计算设备），它会从一个类型中移除掉任意数量的顶层“数组层”。就像下面这样：

```
RemoveAllExtents<int[]> // yields int
RemoveAllExtents<int[5][10]> // yields int
RemoveAllExtents<int[][10]> // yields int
RemoveAllExtents<int(*)[5]> // yields int(*)[5]
```

元函数通过偏特化来匹配高层次的数组，递归地调用自己并最终完成任务。

如果数值计算的功能只适用于标量，那么其应用会很受限制。幸运的是，几乎有所得语言都至少有一种数值容器，这可以大大的提高该语言的能力（而且很多语言都有各种各样的容器，比如 array/vector，hash table 等）。对于元编程也是这样：增加一个“类型容器”会大大的提高其自身的适用范围。幸运的是，现代 C++提供了可以用来开发类似容器的机制。第 24 章开发的 Typelist<...>类模板，就是这一类型的类型容器。

23.1.3 混合元编程

通过使用数值元编程和类型元编程，可以在编译期间计算数值和类型。但是最终我们关心的还是在运行期间的效果，因此在运行期间的代码中，我们将元程序用在那些需要类型和常量的地方。不过元编程能做的不仅仅是这些：我们可以在编译期间，以编程的方式组合一些有运行期效果的代码。我们称之为混合元编程。

下面通过一个简单的例子来说明这一原理：计算两个 std::array 的点乘结果。回忆一下，sta::array 是具有固定长度的容器模板，其声明如下：

```
namespace std {
```

```
    template<typename T, size_t N> struct array;
}
```

其中 N 是 std::array 的长度。假设有两个类型相同的 sta::array 对象，其点乘结果可以通过如下方式计算：

```
template<typename T, std::size_t N>
auto dotProduct(std::array<T, N> const& x, std::array<T, N>
const& y)
{
    T result{};
    for (std::size_t k = 0; k<N; ++k) {
        result += x[k]*y[k];
    }
    return result;
}
```

如果对 for 循环进行直接编译的话，那么就会生成分支指令，相比于直接运行如下命令，这在一些机器上可能会增加运行成本：

```
result += x[0]*y[0];
result += x[1]*y[1];
result += x[2]*y[2];
result += x[3]*y[3];
…
```

幸运的是，现代编译器会针对不同的平台做出相应的最为高效的优化。但是为了便于讨论，下面重新实现一版不需要 loop 的 dotProduct():

```
template<typename T, std::size_t N>
struct DotProductT {
    static inline T result(T* a, T* b)
    {
        return *a * *b + DotProduct<T, N-1>::result(a+1,b+1);
    }
};
// partial specialization as end criteria
template<typename T>
struct DotProductT<T, 0> {
    static inline T result(T*, T*) {
        return T{};
    }
};
template<typename T, std::size_t N>
auto dotProduct(std::array<T, N> const& x,
    std::array<T, N> const& y)
{
```

```
    return DotProductT<T, N>::result(x.begin(), y.begin());
}
```

新的实现将计算放在了类模板 DotProductT 中。这样做的目的是为了使用类模板的递归实例化来计算结果，并能够通过部分特例化来终止递归。注意例子中 DotProductT 的每一次实例化是如何计算点乘中的一项结果、以及所有剩余结果的。对于 std::arrat<T,N>，会对主模板进行 N 次实例化，对部分特例化的模板进行一次实例化。为了保证效率，编译期需要将每一次对静态成员函数 result()的调用内联（inline）。幸运的是，即使使用的时中等优化选项，编译器也会这样做。

这段代码的主要特点是它融合了编译期计算（这里通过递归的模板实例化实现，这决定了代码的整体结构）和运行时计算（通过调用 result()，决定了具体的运行期间的效果）。

我们之前提到过，“类型容器”可以大大提高元编程的能力。我们同样看到固定长度的 array 在混合元编程中也非常有用。但是混合元编程中真正的“英雄容器”是 tuple（元组）。Tuple 是一串数值，且其中每个值的类型可以分别指定。C++标准库中包含了支持这一概念的类模板 std::tuple。比如：

```
std::tuple<int, std::string, bool> tVal{42, "Answer", true};
```

定义的变量 tVal 包含了三个类型分别为 int, std::string 和 bool 的值。因为 tuple 这一类容器在现代 C++编程中非常重要，我们将在第 25 章对其进行更深入的讨论。tVal 的类型和下面这个简单的 struct 类型非常类似：

```
struct MyTriple {
    int v1;
    std::string v2;
    bool v3;
};
```

既然对于 array 类型和（简单）的 struct 类型，我们有比较灵活的 std::array 和 std::tuple 与之对应，那么你可能会问，与简单的 union 对应的类似类型是否对混合元编程也很有益。答案是“yes”。C++标准库在 C++17 中为了这一目的引入了 std::variant 模板，在第 26 章中我们会介绍一个类似的组件。

由于 std::tuple 和 std::variant 都是异质类型（与 struct 类似），使用这些类型的混合元编程有时也被称为“异质元编程”。

23.1.4 将混合元编程用于“单位类型”（Units Types，可能翻译的不恰当）

另一个可以展现混合元编程威力的例子是那些实现了不同单位类型的数值之间计算的库。相应的数值计算发生在程序运行期间，而单位计算则发生在编译期间。

下面会以一个极度精简的例子来做讲解。我们将用一个基于主单位的分数来记录相关单位。比如如果时间的主单位是秒，那么就用 1/1000 表示 1 微秒，用 60/1 表示一分钟。因此关键点就是要定义一个比例类型，使得每一个数值都有其自己的类型：

```
template<unsigned N, unsigned D = 1>
struct Ratio {
    static constexpr unsigned num = N; // numerator
    static constexpr unsigned den = D; // denominator
    using Type = Ratio<num, den>;
};
```

现在就可以定义在编译期对两个单位进行求和之类的计算：

```
// implementation of adding two ratios:
template<typename R1, typename R2>
struct RatioAddImpl
{
    private:
        static constexpr unsigned den = R1::den * R2::den;
        static constexpr unsigned num = R1::num * R2::den + R2::num *
R1::den;
    public:
        typedef Ratio<num, den> Type;
};
// using declaration for convenient usage:
template<typename R1, typename R2>
using RatioAdd = typename RatioAddImpl<R1, R2>::Type;
```

这样就可以在编译期计算两个比率之和了：

```
using R1 = Ratio<1,1000>;
using R2 = Ratio<2,3>;
using RS = RatioAdd<R1,R2>; //RS has type Ratio<2003,2000>
std::cout << RS::num << ' /' << RS::den << ' \n' ; //prints 2003/3000
using RA = RatioAdd<Ratio<2,3>,Ratio<5,7>>; //RA has type
Ratio<29,21>
std::cout << RA::num << ' /' << RA::den << ' \n' ; //prints 29/21
```

然后就可以为时间段定义一个类模板，用一个任意数值类型和一个 Ratio<>实例化之后的类型作为其模板参数：

```
// duration type for values of type T with unit type U:
template<typename T, typename U = Ratio<1>>
class Duration {public:
    using ValueType = T;
    using UnitType = typename U::Type;
    private:
        ValueType val;
```

```
    public:
        constexpr Duration(ValueType v = 0)
        : val(v) {
        }
        constexpr ValueType value() const {
            return val;
        }
};
```

比较有意思的地方是对两个 Durations 求和的 operator+运算符的定义：

```
// adding two durations where unit type might differ:
template<typename T1, typename U1, typename T2, typename U2>
auto constexpr operator+(Duration<T1, U1> const& lhs,
Duration<T2, U2> const& rhs)
{
    // resulting type is a unit with 1 a nominator and
    // the resulting denominator of adding both unit type fractions
    using VT = Ratio<1,RatioAdd<U1,U2>::den>;
    // resulting value is the sum of both values
    // converted to the resulting unit type:
    auto val = lhs.value() * VT::den / U1::den * U1::num +
    rhs.value() * VT::den / U2::den * U2::num;
    return Duration<decltype(val), VT>(val);
}
```

这里参数所属的单位类型可以不同，比如分别为 U1 和 U2。然后可以基于 U1 和 U2 计算最终的时间段，其类型为一个新的分子为 1 的单位类型。基于此，可以编译如下代码：

```
int x = 42;
int y = 77;
auto a = Duration<int, Ratio<1,1000>>(x); // x milliseconds
auto b = Duration<int, Ratio<2,3>>(y); // y 2/3 seconds
auto c = a + b; //computes resulting unit type 1/3000 seconds//and
generates run-time code for c = a*3 + b*2000
```

此处“混合”的效果体现在，在计算 c 的时候，编译器会在编译期决定结果的单位类型 Ratio<1,3000>，并产生出可以在程序运行期间计算最终结果的代码（结果会被根据单位类型进行调整）。

由于数值类型是由模板参数决定的，因此可以将 int 甚至是异质类型用于 Duration 类：

```
auto d = Duration<double, Ratio<1,3>>(7.5); // 7.5 1/3 seconds
auto e = Duration<int, Ratio<1>>(4); // 4 seconds
auto f = d + e; //computes resulting unit type 1/3 seconds
// and generates code for f = d + e*3
```

而且如果相应的数值在编译期是已知的话，编译器甚至可以在编译期进行以上计算（因为上文中的 operator+是 constexpr）。

C++中的 std::chrono 类模板使用了类似于上文中的内容，但是做了一些优化，比如使用已定义的单位（比如 std::chrono::miliseconds），支持时间段常量（比如 10ms），以及能够处理溢出。

23.2 反射元编程的维度

上文中介绍了基于 constexpr 的“值元编程”和基于递归实例化的“类型元编程”。这两种在现代 C++中可用的选项采用了明显不同的方式来驱动计算。事实证明“值元编程”也可以通过模板的递归实例化来实现，在引入 C++11 的 constexpr 函数之前，这也正是其实现方式。比如下面的代码使用递归实例化来计算一个整数的平方根：

```
// primary template to compute sqrt(N)
template<int N, int LO=1, int HI=N>
struct Sqrt {
    // compute the midpoint, rounded up
    static constexpr auto mid = (LO+HI+1)/2;
    // search a not too large value in a halved interval
    static constexpr auto value = (N<mid*mid) ?
      Sqrt<N,LO,mid-1>::value : Sqrt<N,mid,HI>::value;
};
// partial specialization for the case when LO equals HI
template<int N, int M>
struct Sqrt<N,M,M> {
    static constexpr auto value = M;
};
```

这个源程序使用了几乎和 23.1.1 节中的 constexpr 函数完全一样的算法，不断的二分查找包含平方根的中间值。但是，这里元函数的输入是一个非类型模板参数，而不是一个函数参数，用来追踪中间值边界的“局部变量”也是非类型模板参数。显然这个方法远不如 constexpr 函数友好，但是我接下来依然会探讨这段代码是如何消耗编译器资源的。

无论如何，我们已经看到元编程的计算引擎可以有多种潜在的选择。但是计算不是唯一的一个我们应该在其中考虑相关选项的维度。一个综合的元编程解决方案应该在如下 3 个维度中间做选择：

- 计算维度（Compution）
- 反射维度（Reflection）
- 生成维度（Generation）

反射维度指的是以编程的方式检测程序特性的能力。生成维度指的是为程序生成额外代码的能力。

我们已经见过计算维度中的两个选项：递归实例化和 constexpr 计算。对于反射维度，在类型萃取（参见第 19.6.1 节）相关章节中也介绍了其部分解决方案。虽然一些可用的类型萃取使得某些高端的模板技术变得可能，但是这远没有包含所有的、我们所期望能够从反射机制中获得的特性。比如给定一个类，一些应用总是倾向于在程序中访问其某些成员。目前已有的类型萃取是基于模板实例化的，而且 C++总是会提供额外的语言特性或者是“固有的”库元素来在编译期生成包含反射信息的类模板实例。这一方法和基于模板递归实例化进行的计算比较相似。但是不幸的是，类模板实例会占用比较多的编译器内存，而且这部分内存要直到编译结束才会被释放（否则的话编译时间会大大延长）。另一个被期望可以在“计算维度”和 constexpr 运算选项组合的很好的选项是，引入一个新的标准类型来代表“反射信息”。在第 17.9 节中对这一选项进行了讨论（目前 C++标准委员会正在对相关内容进行探讨）。

第 17.9 节中还讨论了另一种有潜力的、可以提供强大的代码生成能力的方法。在已有的 C++语言中创建一个灵活的、通用的、用户友好的代码生成机制依然是一个被很多组织研究的、颇有挑战的事情。但是模板实例化又总是各种代码生成机制中的一种。另外，编译器在将函数调用扩展成小函数的 inline 方面已经足够可靠，该机制可以被用作产生代码的一种手段。这些内容是上文中 DotProductT 的基础，并且结合强大的反射工具，现阶段已有的技术已经可以获得优异的元编程效果了。

23.3 递归实例化的代价

现在来分析第 23.2 节中介绍的 Sqrt<>模板。主模板是由模板参数 N（被计算平方根的值）和其它两个可选参数触发的、常规的递归计算。两个可选的参数分别是结果的上限和下限。如果只用一个参数调用该模板，那么其平方根最小是 1，最大是其自身。

递归会按照二分查找的方式进行下去。在模板内部会计算 value 是在从 LO 到 HI 这个区间的上半部还是下半部。这一分支判断是通过运算符?:实现的。如果 mid^2 比 N 大，那么就继续在上半部分查找，否在就在下半部分查找。

偏特例化被用来在 LO 和 HI 的值都为 M 的时候结束递归，这个值也就是我们最终所要计算的结果。

实例化模板的成本并不低廉：即使是比较适中的类模板，其实例依然有可能占用数 KB 的内存，而且这部分被占用的内存在编译完成之前不可以被回收利用。我们先来分析一个使用了 Sqrt 模板的简单程序：

```
#include <iostream>
#include "sqrt1.hpp"
int main()
{
    std::cout << "Sqrt<16>::value = " << Sqrt<16>::value << ' \n' ;
    std::cout << "Sqrt<25>::value = " << Sqrt<25>::value << ' \n' ;
    std::cout << "Sqrt<42>::value = " << Sqrt<42>::value << ' \n' ;
    std::cout << "Sqrt<1>::value = " << Sqrt<1>::value << ' \n' ;
}
```

表达式

```
Sqrt<16>::value
```

被扩展成

```
Sqrt<16,1,16>::value
```

在模板内部，元程序按照如下方式计算 Sqrt<16,1,16>::value 的值：

```
mid = (1+16+1)/2
    = 9

value = (16<9*9) ? Sqrt<16,1,8>::value
       : Sqrt<16,9,16>::value
      = (16<81) ? Sqrt<16,1,8>::value
        : Sqrt<16,9,16>::value
      = Sqrt<16,1,8>::value
```

接着这个值会被以 Sqrt<16,1,8>::value 的形式计算，其会被接着展开为：

```
mid = (1+8+1)/2
    = 5
value = (16<5*5) ? Sqrt<16,1,4>::value
       : Sqrt<16,5,8>::value
      = (16<25) ? Sqrt<16,1,4>::value
        : Sqrt<16,5,8>::value
      = Sqrt<16,1,4>::value
```

类似的，Sqrt<16,1,4>::value 被分解为如下形式：

```
mid = (1+4+1)/2
    = 3
value = (16<3*3) ? Sqrt<16,1,2>::value
       : Sqrt<16,3,4>::value
      = (16<9) ? Sqrt<16,1,2>::value
        : Sqrt<16,3,4>::value
      = Sqrt<16,3,4>::value
```

最终，Sqrt<16,3,4>::value 产生出如下结果：

```
mid = (3+4+1)/2
    = 4
value = (16<4*4) ? Sqrt<16,3,3>::value
       : Sqrt<16,4,4>::value
      = (16<16) ? Sqrt<16,3,3>::value
        : Sqrt<16,4,4>::value
      = Sqrt<16,4,4>::value
```

然后这一递归过程会被 Sqrt<16,4,4>::value 终结，因为对它的调用会匹配到模板的特化版本上（上限和下限相同）。因此最终的结果是：

```
value = 4
```

23.3.1 追踪所有的实例化过程

上文中主要分析了被用来计算 16 的平方根的实例化过程。但是当编译期计算：

```
(16<=8*8) ? Sqrt<16,1,8>::value
    : Sqrt<16,9,16>::value
```

的时候，它并不是只计算真正用到了的分支，同样也会计算没有用到的分支（Sqrt<16,9,16>）。而且，由于代码试图通过运算符::访问最终实例化出来的类的成员，该类中所有的成员都会被实例化。也就是说 Sqrt<16,9,16>的完全实例化会导致 Sqrt<16,9,12>和 Sqrt<16,13,16>都会被完全实例化。仔细分析以上过程，会发现最终会实例化出很多的实例，数量上几乎是 N 的两倍。

幸运的是，有一些技术可以被用来降低实例化的数目。为了展示其中一个重要的技术，我们按照如下方式重写了 Sqrt 元程序：

```
#include "ifthenelse.hpp"
// primary template for main recursive step
template<int N, int LO=1, int HI=N>
struct Sqrt {
    // compute the midpoint, rounded up
    static constexpr auto mid = (LO+HI+1)/2;
    // search a not too large value in a halved interval
    using SubT = IfThenElse<(N<mid*mid),
        Sqrt<N,LO,mid-1>,
        Sqrt<N,mid,HI>>;
    static constexpr auto value = SubT::value;
};
// partial specialization for end of recursion criterion
template<int N, int S>
struct Sqrt<N, S, S> {
    static constexpr auto value = S;
};
```

代码中主要的变化是使用了 IfThenElse 模板，在第 19.7.1 节有对它的介绍。回忆一下，IfThenElse 模板被用来基于一个布尔常量在两个类型之间做选择。如果布尔型常量是 true，那么会选择第一个类型，否则就选择第二个类型。一个比较重要的、需要记住的点是：为一个类模板的实例定义类型别名，不会导致 C++编译器去实例化该实例。因此使用如下代码时：

```
using SubT = IfThenElse<(N<mid*mid),
    Sqrt<N,LO,mid-1>,
```

```
    Sqrt<N,mid,HI>>;
```

既不会完全实例化 `Sqrt<N,LO,mid-1>`也不会完全实例化 `Sqrt<N,mid,HI>`。

在调用 `SubT::value` 的时候，只有真正被赋值给 SubT 的那一个实例才会被完全实例化。和之前的方法相比，这会让实例化的数量和 $\log_2 N$ 成正比：当 N 比较大的时候，这会大大降低元程序实例化的成本。

23.4 计算完整性

从以上的 Sqrt<>的例子可以看出，一个模板元程序可能会包含以下内容：

- 状态变量：模板参数
- 循环结构：通过递归实现
- 执行路径选择：通过条件表达式或者偏特例化实现
- 整数运算

如果对递归实例化的数量和使用的状态变量的数量不做限制，那么就可以用之来计算任何可以计算的事情。尽管这样做可能不是很方便。而且，由于模板实例化需要大量的编译器资源，大量的递归实例化会很快地降低编译器地编译速度，甚至是耗尽可用地硬件资源。C++标准建议最少应该支持 1024 层的递归实例化，但是并没有强制如此，但是这应该足够大部分（当然不是全部）模板元编程任务使用了。

因此在实际中，不应该滥用模板元编程。但是在少数情况下，在实现便捷模板方面它又不可替代。在一些特殊情况下，它们可能被隐藏在常规模板的深处，以尽可能地提高关键算法地计算性能。

23.5 递归实例化和递归模板参数

考虑如下递归模板：

```
template<typename T, typename U>
struct Doublify {
};
template<int N>
struct Trouble {
    using LongType = Doublify<typename Trouble<N-1>::LongType,
                              typename Trouble<N-1>::LongType>;
};
template<>
struct Trouble<0> {
    using LongType = double;
};
```

```
Trouble<10>::LongType ouch;
```

Trouble<10>::LongType 的使用并不是简单地触发形如 Trouble<9>, Trouble<8>, …, Trouble<0>地递归实例化，还会用越来越复杂地类型实例化 Doublify。表 23.1 展示了其快速地增长方式：

类型别名	底层类型
Trouble<0>::LongType	double
Trouble<1>::LongType	Doublify<double,double>
Trouble<2>::LongType	Doublify<Doublify<double,double>, Doublify<double,double>>
Trouble<3>::LongType	Doublify<Doublify<Doublify<double,double>, Doublify<double,double>>, <Doublify<double,double>, Doublify<double,double>>>

表 23.1 Trouble<3>::LongType 的变化趋势

就如从表 23.1 中看到的那样，Trouble<N>::LongType 类型的复杂度与 N 成指数关系。通常这种情况给 C++编译器带来的压力要远比有递归实例化但是没有递归模板参数的情况要大。这里的问题之一是，编译器会用一些支离破碎的名字来表达这些类型。这些支离破碎的名字会用相同的方式去编码模板的特例化，在早期 C++中，这一编码方式的实现和模板签名（template-id）的长度成正比。这些编译器会使用大于 10,000 个字符来表达 Trouble<N>::LongType。

基于嵌套模板在现在 C++中非常常见的事实，新的 C++实现使用了一种聪明的压缩技术来大大降低名称编码（比如对于 Trouble<N>::LongType，只需要用数百个字符）的增长速度。如果没有为某些模板实例生成低层级的代码，那么相关类型的名字就是不需要的，新的编译器就不会为这些类型产生名字。除此之外，其它情况都没有改善，因此在组织递归实例化代码的时候，最好不要让模板参数也嵌套递归。

23.6 枚举值还是静态常量

在早期 C++中，枚举值是唯一可以用来在类的声明中、创建可用于类成员的“真正的常量”（也称常量表达式）的方式。比如通过它们可以定义 Pow3 元程序来计算 3 的指数：

```
// primary template to compute 3 to the Nth
template<int N>
struct Pow3 {
    enum { value = 3 * Pow3<N-1>::value };
};
// full specialization to end the recursion
template<>
struct Pow3<0> {
    enum { value = 1 };
};
```

在 C++98 标准中引入了类内静态常量初始化的概念，因此 Pow3 元程序可以被写成这样：

```
// primary template to compute 3 to the Nth
template<int N
struct Pow3 {
    static int const value = 3 * Pow3<N-1>::value;
};
// full specialization to end the recursion
template<>
struct Pow3<0> {
    static int const value = 1;
};
```

但是上面代码中有一个问题：静态常量成员是左值（参见附录 B）。因此如果我们有如下函数：

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

然后我们将元程序的结果传递给它：

```
foo(Pow3<7>::value);
```

编译器需要传递 Pow3<7>::value 的地址，因此必须实例化静态成员并为之开辟内存。这样该计算就不是一个纯正的“编译期”程序了。

枚举值不是左值（也就是说它们没有地址）。因此当将其按引用传递时，不会用到静态内存。几乎等效于将被计算值按照字面值传递。因此本书第一版建议在这一类应用中使用枚举值，而不是静态常量。

不过在 C++中，引入了 constexpr 静态数据成员，并且其使用不限于整型类型。这并没有解决上文中关于地址的问题，但是即使如此，它也是用来产生元程序结果的常规方法。其优点是，它可以有正确的类型（相对于人工的枚举类型而言），而且当用 auto 声明静态成员的类型时，可以对其类型进行推断。C++17 则引入了 inline 的静态数据成员，这解决了上面提到的地址问题，而且可以和 constexpr 一起使用。

23.7 后记

最早的文档可查的元编程的例子是由 Erwin Unruh 实现的，并代表西门子在 C++标准委员会上做了展示。他注意到了模板实例化过程的计算完整性，并开发了第一个元程序来证明了自己的观点。他使用的是 Metaware 编译器，让编译器在错误信息中输出了连续的素数。在 1994 年 C++委员会上流传的代码如下（做了些修改，以使其能够在标准编译器上编译）：

```
// prime number computation // (modified from original from 1994 by Erwin
Unruh)
template<int p, int i>
struct is_prime {
```

```
    enum { pri = (p==2) || ((p%i) && is_prime<(i>2? p:0),i-1>::pri) };
};

template<>
struct is_prime<0,0> {
    enum {pri=1};
};

template<>
struct is_prime<0,1> {
    enum {pri=1};
};

template<int i>
struct D {
    D(void*);
};

template<int i>
struct CondNull {
    static int const value = i;
};

template<>
struct CondNull<0> {
    static void* value;
};

void* CondNull<0>::value = 0;

template<int i>
struct Prime_print { // primary template for loop to print prime numbers
    Prime_print<i-1> a;
    enum { pri = is_prime<i,i-1>::pri };
    void f() {
        D<i> d = CondNull<pri ? 1 : 0>::value; // 1 is an error, 0 is
ne
        a.f();
    }
};

template<>
struct Prime_print<1> { // full specialization to end the loop
    enum {pri=0};
```

```
    void f() {
        D<1> d = 0;
    };
};

#ifndef LAST
#define LAST 18
#endif

int main()
{
    Prime_print<LAST> a;
    a.f();
}
```

如果你试着编译以上程序，编译器会打印错误说在 `Prime_print::f()`中，初始化 d 时遇到错误。错误发生在初始值时 1 的时候，因为只有一个参数为 `void*`的构造函数，而又只有 0 可以被转换成 `void*`。下面是一个编译器报的错误的（包含在其它一些信息中）：

```
unruh.cpp:39:14: error: no viable conversion from ' const int' to
' D<17>'
unruh.cpp:39:14: error: no viable conversion from ' const int' to
' D<13>'
unruh.cpp:39:14: error: no viable conversion from ' const int' to
' D<11>'
unruh.cpp:39:14: error: no viable conversion from ' const int' to
' D<7>'
unruh.cpp:39:14: error: no viable conversion from ' const int' to
' D<5>'
unruh.cpp:39:14: error: no viable conversion from ' const int' to
' D<3>'
unruh.cpp:39:14: error: no viable conversion from ' const int' to
' D<2>'
```

C++模板元编程的概念，作为一个严肃的编程工具最早是由 Todd Veldhuizen 在其文章 Using C++ Template Metaprograms (see [VeldhuizenMeta95]) 中推广开来的（并做了一些规范化）。Todd 在其关于 Blitz++的工作（一个 C++数值数组库，参见 Blitz++）中也对元编程（和表达式模板技术）做了一些优化和扩展。

本书第一版和 Andrei Alexandrescu 的 Modern C++ Design 都为 C++库的爆发做出了贡献，书中通过总结一些至今还在使用的基础技术探索了基于模板的元编程。Boost 项目则为这一爆发带了了秩序。在早期，它引入了 MPL（元编程库，meta-programming library），这为类型元编程（同样由 Abrahams 和 Gurtovoy 的书 “C++Template Metaprogramming” 带火）。Template Metaprogramming” ）定义了一致的框架。

另一个重要的进步是由 Louis Dionne 贡献的，在他的 Boost.Hana 库中，使得元编程语法变得更容易被接受。在标准委员会中，Louis 正在和 Andrew Sutton，Herb Sutter，David Vandevoorde 以及其他一些人一起，努力使元编程在语言中得到更好的支持。该工作中一个主要的部分是什么样的程序特性应该在反射中得到支持。Matúš Chochlík, Axel Naumann 以及 David Sankel 是相关领域的主要贡献者。

在 BartonNackman 中，John J. Barton 和 Lee R. Nackman 展示了在执行计算时该如何追踪维度成员。SIunits 库是由 Walter Brown 开发的一个用来处理物理单元的、更全面的库。而我们在第 23.1.4 节中作为灵感来源使用的 std::chrono，则是由 Howard Hinnant 开发，专门用来处理时间和日期的库。

第 24 章 类型列表（Typelists）

高效的编程通常需要用到各种各样的数据结构，元编程也不例外。对于类型元编程，核心的数据结构是 typelist，和其名字的意思一样，它指的是一个包含了类型的列表。模板元编程可以操作 typelist 并最终生成可执行程序的一部分。接下来会在本章中讨论使用 typelist 的技术。由于大多数 typelist 相关的操作都用到了模板元编程，这里假设你已经对模板元编程比较熟悉（参见第 23 章）。

24.1 类型列表剖析（Anatomy of a Typelist）

类型列表指的是一种代表了一组类型，并且可以被模板元编程操作的类型。它提供了典型的列表操作方法：遍历列表中的元素，添加元素或者删除元素。但是类型列表和大多数运行期间的数据结构都不同（比如 std::list），它的值不允许被修改。向类型列表中添加一个元素并不会修改原始的类型列表，只是会创建一个新的、包含了原始类型列表和新添加元素的类型列表。对函数式编程语言（比如 Scheme，ML 以及 Haskell）比较熟悉的读者应该会意识到 C++中的类型列表和这些函数式编程语言中的列表之间的相似性。

类型列表通常是按照类模板特例的形式实现的，它将自身的内容（包含在模板参数中的类型以及类型之间的顺序）编码到了参数包中。一种将其内容编码到参数包中的类型列表的直接实现方式如下：

```
template<typename… Elements>
class Typelist
{};
```

Typelist 中的元素被直接写成其模板参数。一个空的类型列表被写为 `Typelist<>`，一个只包含 int 的类型列表被写为 `Typelist<int>`。下面是一个包含了所有有符号整型的类型列表：

```
using SignedIntegralTypes =
    Typelist<signed char, short, int, long, long long>;
```

操作这个类型列表需要将其拆分，通常的做法是将第一个元素（the head）从剩余的元素中分离（the tail）。比如 Front 元函数会从类型列表中提取第一个元素：

```
template<typename List>
class FrontT;

template<typename Head, typename… Tail>
class FrontT<Typelist<Head, Tail…>>
{
    public:
        using Type = Head;
```

```
};

template<typename List>
    using Front = typename FrontT<List>::Type;
```

这样 FrontT<SignedIntegralTypes>::Type（或者更简洁的记作 FrontT<SignedIntegralTypes>）返回的就是 signed char。同样 PopFront 元函数会删除类型列表中的第一个元素。在实现上它会将类型列表中的元素分为头（head）和尾（tail）两部分，然后用尾部的元素创建一个新的 Typelist 特例。

```
template<typename List>
class PopFrontT;

template<typename Head, typename… Tail>
class PopFrontT<Typelist<Head, Tail…>> {
    public:
    using Type = Typelist<Tail…>;
};

template<typename List>
    using PopFront = typename PopFrontT<List>::Type;
```

PopFront<SignedIntegralTypes>会产生如下类型列表：

```
Typelist<short, int, long, long long>
```

同样也可以向类型列表中添加元素，只需要将所有已经存在的元素捕获到一个参数包中，然后在创建一个包含了所有元素的 TypeList 特例就行：

```
template<typename List, typename NewElement>
class PushFrontT;

template<typename… Elements, typename NewElement>
class PushFrontT<Typelist<Elements…>, NewElement> {
    public:
        using Type = Typelist<NewElement, Elements…>;
};

template<typename List, typename NewElement>
    using PushFront = typename PushFrontT<List, NewElement>::Type;
```

和预期的一样，

```
PushFront<SignedIntegralTypes, bool>
```

会生成：

```
Typelist<bool, signed char, short, int, long, long long>
```

24.2 类型列表的算法

基础的类型列表操作 Front，PopFront 和 PushFront 可以被组合起来实现更有意思的列表操作。比如通过将 PushFront 作用于 PopFront 可以实现对第一个元素的替换：

```
using Type = PushFront<PopFront<SignedIntegralTypes>, bool>;
// equivalent to Typelist<bool, short, int, long, long long>
```

更近一步，我们可以按照模板原函数的实现方式，实现作用于类型列表的诸如搜索、转换和反转等操作。

24.2.1 索引（Indexing）

类型列表的一个非常基础的操作是从列表中提取某个特定的类型。第 24.1 节展示了提取第一个元素的实现方式。接下来我们将这一操作推广到可以提取第 N^{th} 个元素。比如，为了提取给定类型列表中的第 2 个元素，可以这样：

```
using TL = NthElement<Typelist<short, int, long>, 2>;
```

这相当于将 TL 作为 long 的别名使用。NthElement 操作的实现方式是使用一个递归的元程序遍历 typelist 中的元素，直到找到所需元素为止：

```
// recursive case:
template<typename List, unsigned N>
class NthElementT : public NthElementT<PopFront<List>, N-1>
{};

// basis case:
template<typename List>
class NthElementT<List, 0> : public FrontT<List>
{ };

template<typename List, unsigned N>
using NthElement = typename NthElementT<List, N>::Type;
```

首先来看由 N = 0 部分特例化出来的基本情况。这一特例化会通过返回类型列表中的第一个元素来终止递归。其方法是对 FrontT<List>进行 public 继承，这样 FrontT<List>作为类型列表中第一个元素的 Type 类型别名，就可以被作为 NthElement 的结果使用了（这里用到了元函数转发，参见 19.3.2 节，但是译者没找到具体内容）。

作为模板主要部分的递归代码，会遍历类型列表。由于偏特化部分保证了 N > 0，递归部分的代码会不断地从剩余列表中删除第一个元素并请求第 N-1 个元素。在我们的例子中：

```
NthElementT<Typelist<short, int, long>, 2>
```

继承自：

```
NthElementT<Typelist<int, long>, 1>
```

而它又继承自：

```
NthElementT<Typelist<long>, 0>
```

这里遇到了最基本的 N = 0 的情况，它继承自提供了最终结果 Type 的 FrontT<Typelist<long>>。

24.2.2 寻找最佳匹配

有些类型列表算法会去查找类型列表中的数据。例如可能想要找出类型列表中最大的类型（比如为了开辟一段可以存储类型列表中任意类型的内存）。这同样可以通过递归模板元程序实现：

```
template<typename List>
class LargestTypeT;

// recursive case:
template<typename List>
class LargestTypeT
{
    private:
        using First = Front<List>;
        using Rest = typename LargestTypeT<PopFront<List>>::Type;
    public:
        using Type = IfThenElse<(sizeof(First) >= sizeof(Rest)), First,
Rest>;
};

// basis case:
template<>
class LargestTypeT<Typelist<>>
{
    public:
        using Type = char;
};

template<typename List>
using LargestType = typename LargestTypeT<List>::Type;
```

LargestType 算法会返回类型列表中第一个最大的类型。比如对于 Typelist<bool, int, long, short>，该算法会返回第一个大小和 long 相同的类型，可能是 int 也可能是 long，取决于你的平台。

由于递归算法的使用，对 LargestTypeT 的调用次数会翻倍。它使用了 first/rest 的概念，分三步完成任务。在第一步中，它先只基于第一个元素计算出部分结果，在本例中是将第一个元素放置到 First 中。接下来递归地计算类型列表中剩余部分的结果，并将结果放置在 Rest 中。比如对于类型列表 Typelist<bool, int, long, short>，在递归的第一步中 First 是 bool，而 Rest 是该算法作用于 Typelist<int, long, short>得到的结果。最后在第三步中综合 First 和 Rest 得到最终结果。此处，IfThenElse 会选出列表中第一个元素（First）和到目前为止的最优解（Rest）中类型最大的那一个。>=的使用会倾向于选择第一个出现的最大的类型。

递归会在类型列表为空时终结。默认情况下我们将 char 用作哨兵类型来初始化该算法，因为任何类型都不会比 char 小。

注意上文中的基本情况显式的用到了空的类型列表 Typelist<>。这样有点不太好，因为它可能会妨碍到其它类型的类型列表（我们会在第 24.3 节和第 24.5 节中讲到这一类类型列表）的使用。为了解决这一问题，引入了 IsEmpty 元函数，它可以被用来判断一个类型列表是否为空：

```
template<typename List>
class IsEmpty
{
    public:
        static constexpr bool value = false;
};

template<>
class IsEmpty<Typelist<>> {
    public:
        static constexpr bool value = true;
};
```

结合 IsEmpty，可以像下面这样将 LargestType 实现成适用于任意支持了 Front，PopFront 和 IsEmpty 的类型：

```
template<typename List, bool Empty = IsEmpty<List>::value>
class LargestTypeT;

// recursive case:
template<typename List>
class LargestTypeT<List, false>
{
    private:
        using Contender = Front<List>;
        using Best = typename LargestTypeT<PopFront<List>>::Type;
    public:
        using Type = IfThenElse<(sizeof(Contender) >=
sizeof(Best)),Contender, Best>;
```

```
};
// basis case:
template<typename List>
class LargestTypeT<List, true>
{
    public:
        using Type = char;
};

template<typename List>
using LargestType = typename LargestTypeT<List>::Type;
```

默认的 LargestTypeT 的第二个模板参数 Empty 会检查一个类型列表是否为空。如果不为空，就递归地继续在剩余的列表中查找。如果为空，就会终止递归并返回作为初始结果的 char。

24.2.3 向类型类表中追加元素

通过 PushFront 可以向类型列表的头部添加一个元素，并产生一个新的类型列表。除此之外我们还希望能够像在程序运行期间操作 std::list 和 std::vector 那样，向列表的末尾追加一个元素。对于我们的 Typelist 模板，为实现支持这一功能的 PushBack，只需要对 24.1 节中的 PushFront 做一点小的修改：

```
template<typename List, typename NewElement>
class PushBackT;

template<typename… Elements, typename NewElement>
class PushBackT<Typelist<Elements…>, NewElement>
{
    public:
        using Type = Typelist<Elements…, NewElement>;
};

template<typename List, typename NewElement>
using PushBack = typename PushBackT<List, NewElement>::Type;
```

不过和实现 LargestType 的算法一样，可以只用 Front，PushFront，PopFront 和 IsEmpty 等基础操作实现一个更通用的 PushBack 算法：

```
template<typename List, typename NewElement, bool =
IsEmpty<List>::value>
class PushBackRecT;

// recursive case:
template<typename List, typename NewElement>
class PushBackRecT<List, NewElement, false>
```

```
{
    using Head = Front<List>;
    using Tail = PopFront<List>;
    using NewTail = typename PushBackRecT<Tail, NewElement>::Type;
    public:
        using Type = PushFront<Head, NewTail>;
};

// basis case:
template<typename List, typename NewElement>
class PushBackRecT<List, NewElement, true>
{
    public:
        using Type = PushFront<List, NewElement>;
};

// generic push-back operation:
template<typename List, typename NewElement>
class PushBackT : public PushBackRecT<List, NewElement> { };

template<typename List, typename NewElement>
using PushBack = typename PushBackT<List, NewElement>::Type;
```

PushBackRecT 会自行管理递归。对于最基本的情况，用 PushFront 将 NewElement 添加到空的类型列表中。递归部分的代码则要有意思的多：它首先将类型列表分成首元素（Head）和一个包含了剩余元素的新的类型列表（Tail）。新元素则被追加到 Tail 的后面，这样递归的进行下去，就会生成一个 NewTail。然后再次使用 PushFront 将 Head 添加到 NewTail 的头部，生成最终的类型列表。

接下来以下面这个简单的例子为例展开递归的调用过程：

```
PushBackRecT<Typelist<short, int>, long>
```

在最外层的递归代码中，Head 会被解析成 short，Tail 则被解析生 Typelist<int>。然后递归到：

```
PushBackRecT<Typelist<int>, long>
```

其中 Head 会被解析成 int，Tail 则被解析成 Typelist<>。

然后继续递归计算：

```
PushBackRecT<Typelist<>, long>
```

这会触发最基本的情况并返回 `PushFront<Typelist<>, long>`，其结果是 `Typelist<long>`。然后返回上一层递归，将之前的 Head 添加到返回结果的头部：

```
PushFront<int, Typelist<long>>
```

它会返回 Typelist<int, long>。然后继续返回上一层递归，将最外层的 Head（short）添加到返回结果的头部：

```
PushFront<short, Typelist<int, long>>
```

然后就得到了最终的结果：

```
Typelist<short, int, long>
```

通用版的 PushBackRecT 适用于任何类型的类型列表。和本节中之前实现的算法一样，计算过程中它需要的模板实例的数量和类型列表的长度 N 成正比（如果类型列表的长度为 N，那么 PushBackRecT 实例和 PushFrontT 实例的数目都是 N+1，FrontT 和 PopFront 实例的数量为 N）。由于模板实例化对于编译器而言是一个很复杂的过程，因此通过计算模板实例的数目，可以大致估算出编译器编译某个元程序所需要的时间。

对于比较大的模板元程序，编译时间可能会是一个问题，因此有必要设法去降低算法所需要的模板实例的数目。事实上，第一版 PushBack 的实现（用 Typelist 进行了部分特例化）只需要固定数量的模板实例化，这使得它要比通用版本的实现（在编译期）更高效。而且，由于它被描述成 PushBackT 的一种偏特化，在对一个 Typelist 执行 PushBack 的时候这一高效的实现会被自动选择，从而为模板元程序引入了“算法特化”的概念（参见 20.1 节）。该章节中介绍的很多技术都可以被模板元程序用来降低算法所需模板实例的数量。

24.2.4 类型列表的反转

当类型列表的元素之间有某种顺序的时候，对于某些算法而言，如果能够反转该顺序的话，事情将会变得很方便。比如在 24.1 节介绍的 SignedIntegralTypes 中元素是按整型大小的等级递增的。但是对其元素反转之后得到的 Typelist<long, long, long, int, short, signed char>可能会更有用。下面的 Reverse 算法实现了相应的元函数：

```
template<typename List, bool Empty = IsEmpty<List>::value>
class ReverseT;

template<typename List>
using Reverse = typename ReverseT<List>::Type;

// recursive case:
template<typename List>
class ReverseT<List, false>:public PushBackT<Reverse<PopFront<List>>,
Front<List>> { };

// basis case:
template<typename List>
class ReverseT<List, true>{
    public:
        using Type = List;
};
```

该元函数的基本情况是一个作用于空的类型列表的函数。递归的情况则将类型列表分割成第一个元素和剩余元素两部分。比如对于 Typelist<short, int, long>，递归过程会先将第一个元素（short）从剩余元素（Typelist<int, long>）中分离开。然后递归得反转列表中剩余的元素（生成 Typelist<long, int>），最后通过调用 PushBackT 将首元素追加到被反转的列表的后面（生成 Typelist<long, int, short>）.

结合 Reverse，可以实现移除列表中最后一个元素的 PopBackT 操作：

```
template<typename List>
class PopBackT {
    public:
    u   sing Type = Reverse<PopFront<Reverse<List>>>;
};

template<typename List>
using PopBack = typename PopBackT<List>::Type;
```

该算法先反转整个列表，然后删除首元素并将剩余列表再次反转，从而实现删除末尾元素的目的。

24.2.5 类型列表的转换

之前介绍的类型列表的相关算法允许我们从类型列表中提取任意元素，在类型列表中做查找，构建新的列表以及反转列表。但是我们还需要对类型列表中的元素执行一些其它的操作。比如可能希望对类型列表中的所有元素做某种转换，例如通过 AddConst 给列表中的元素加上 const 修饰符：

```
template<typename T>
struct AddConstT
{
    using Type = T const;
};

template<typename T>
using AddConst = typename AddConstT<T>::Type;
```

为了实现这一目的，相应的算法应该接受一个类型列表和一个元函数作为参数，并返回一个将该元函数作用于类型列表中每个元素之后，得到的新的类型列表。比如：

```
Transform<SignedIntegralTypes, AddConstT>
```

返回的是一个包含了 signed char const, short const, int const, long const 和 long long const 的类型列表。元函数被以模板参数模板（参见 5.7 节）的形式提供，它负责将一种类型转换为另一种类型。Transform 算法本身和预期的一样是一个递归算法：

```
template<typename List, template<typename T> class MetaFun, bool Empty
```

```
= IsEmpty<List>::value>
class TransformT;

// recursive case:
template<typename List, template<typename T> class MetaFun>
class TransformT<List, MetaFun, false>
: public PushFrontT<typename TransformT<PopFront<List>,
MetaFun>::Type, typename MetaFun<Front<List>>::Type>
{};

// basis case:
template<typename List, template<typename T> class MetaFun>
class TransformT<List, MetaFun, true>
{
    public:
        using Type = List;
};

template<typename List, template<typename T> class MetaFun>
using Transform = typename TransformT<List, MetaFun>::Type;
```

此处的递归情况虽然句法比较繁琐，但是依然很直观。最终转换的结果是第一个元素的转换结果，加上对剩余元素执行执行递归转换后的结果。

在第 24.4 节介绍了一种更为高效的 Transform 的实现方法。

24.2.6 类型列表的累加（Accumulating Typelists）

转换（Transform）算法在需要对类型列表中的元素做转换时很有帮助。通常将它和累加（Accumulate）算法一起使用，它会将类型列表中的所有元素组合成一个值。Accumulate 算法以一个包含元素 T_1，T_2，…，T_N 的类型列表 T，一个初始类型 I，和一个接受两个类型作为参数的元函数 F 为参数，并最终返回一个类型。它的返回值是 `F (F (F (…F(I, T1), T2), …, TN-1), TN )`，其中在第 i^{th} 步，F 将作用于前 i-1 步的结果以及 T_i。

取决于具体的类型列表，F 的选择以及初始值 I 的选择，可以通过 Accumulate 产生各种不同的输出。比如如果 F 可以被用来在两种类型中选择较大的那一个，Accumulate 的行为就和 LargestType 差不多。而如果 F 接受一个类型列表和一个类型作为参数，并且将类型追加到类型列表的后面，其行为又和 Reverse 算法差不多。

Accumulate 的实现方式遵循了标准的递归元编程模式：

```
template<typename List,
         template<typename X, typename Y> class F,
         typename I,
```

```
        bool = IsEmpty<List>::value>
class AccumulateT;

// recursive case:
template<typename List,
        template<typename X, typename Y> class F,
        typename I>
class AccumulateT<List, F, I, false>
    : public AccumulateT<PopFront<List>, F,
                typename F<I, Front<List>>::Type>
{};

// basis case:
template<typename List,
       template<typename X, typename Y> class F,
       typename I>
class AccumulateT<List, F, I, true>
{
   public:
       using Type = I;
};

template<typename List,
       template<typename X, typename Y> class F,
       typename I>
using Accumulate = typename AccumulateT<List, F, I>::Type;
```

这里初始类型 I 也被当作累加器使用，被用来捕捉当前的结果。因此当递归到类型列表末尾的时候，递归循环的基本情况会返回这个结果。在递归情况下，算法将 F 作用于之前的结果（I）以及当前类型列表的首元素，并将 F 的结果作为初始类型继续传递，用于下一级对剩余列表的求和（Accumulating）。

有了 Accumulate，就可以通过将 PushFrontT 作为元函数 F，将空的类型列表（TypeList<T>）作为初始类型 I，反转一个类型列表：

```
using Result = Accumulate<SignedIntegralTypes, PushFrontT,
Typelist<>>;
// produces TypeList<long long, long, int, short, signed char>
```

如果要实现基于 Accumulate 的 LargestType（称之为 LargestTypeAcc），还需要做一些额外的工作，因为首先要实现一个返回两种类型中类型较大的那一个的元函数：

```
template<typename T, typename U>
class LargerTypeT
    : public IfThenElseT<sizeof(T) >= sizeof(U), T, U>
{ };
```

```
template<typename Typelist>
class LargestTypeAccT
    : public AccumulateT<PopFront<Typelist>, LargerTypeT,
Front<Typelist>>
{ };

template<typename Typelist>
using LargestTypeAcc = typename LargestTypeAccT<Typelist>::Type;
```

值得注意的是，由于这一版的 LargestType 将类型列表的第一个元素当作初始类型，因此其输入不能为空。我们可以显式地处理空列表的情况，要么是返回一个哨兵类型（char 或者 void），要么让该算法很好的支持 SFINASE，就如同 19.4.4 节讨论的那样：

```
template<typename T, typename U>
class LargerTypeT
: public IfThenElseT<sizeof(T) >= sizeof(U), T, U>
{ };

template<typename Typelist, bool = IsEmpty<Typelist>::value>
class LargestTypeAccT;

template<typename Typelist>
class LargestTypeAccT<Typelist, false>
: public AccumulateT<PopFront<Typelist>, LargerTypeT,
Front<Typelist>>
{ };

template<typename Typelist>
class LargestTypeAccT<Typelist, true>
{ };

template<typename Typelist>
using LargestTypeAcc = typename LargestTypeAccT<Typelist>::Type;
```

Accumulate 是一个非常强大的类型列表算法，利用它可以实现很多种操作，因此可以将其看作类型列表操作相关的基础算法。

24.2.7 插入排序

作为最后一个类型列表相关的算法，我们来介绍插入排序。和其它算法类似，其递归过程会将类型列表分成第一个元素（Head）和剩余的元素（Tail）。然后对 Tail 进行递归排序，并将 Head 插入到排序后的类型列表中的合适的位置。该算法的实现如下：

```
template<typename List, template<typename T, typename U>  class Compare,
```

```
          bool = IsEmpty<List>::value>
class InsertionSortT;

template<typename List, template<typename T, typename U> class Compare>
using InsertionSort = typename InsertionSortT<List, Compare>::Type;

// recursive case (insert first element into sorted list):
template<typename List, template<typename T, typename U> class Compare>
class InsertionSortT<List, Compare, false>
    : public InsertSortedT<InsertionSort<PopFront<List>, Compare>,
                           Front<List>, Compare>
{};

// basis case (an empty list is sorted):
template<typename List, template<typename T, typename U> class Compare>
class InsertionSortT<List, Compare, true>
{
   public:
      using Type = List;
};
```

在对类型列表进行排序时，参数 Compare 被用来作比较。它接受两个参数并通过其 value 成员返回一个布尔值。将其用来处理空列表的情况会稍嫌繁琐。

插入排序算法的核心时元函数 InsertSortedT，它将一个值插入到一个已经排序的列表中（插入到第一个可能的位置）并保持列表依然有序：

```
#include "identity.hpp"
template<typename List, typename Element,
template<typename T, typename U> class Compare, bool =
IsEmpty<List>::value>
class InsertSortedT;

// recursive case:
template<typename List, typename Element, template<typename T,
typename U> class Compare>
class InsertSortedT<List, Element, Compare, false>
{
   // compute the tail of the resulting list:
   using NewTail = typename IfThenElse<Compare<Element,
Front<List>>::value, IdentityT<List>,
                                    InsertSortedT<PopFront<List>,
Element, Compare>>::Type;
   // compute the head of the resulting list:
   using NewHead = IfThenElse<Compare<Element, Front<List>>::value,
```

```
Element, Front<List>>;
    public:
        using Type = PushFront<NewTail, NewHead>;
};

// basis case:
template<typename List, typename Element, template<typename T,
typename U> class Compare>
class InsertSortedT<List, Element, Compare, true>
: public PushFrontT<List, Element>
{};

template<typename List, typename Element,template<typename T, typename
U> class Compare>
using InsertSorted = typename InsertSortedT<List, Element,
Compare>::Type;
```

由于只有一个元素的列表是已经排好序的，因此相关代码不是很复杂。对于递归情况，基于元素应该被插入到列表头部还是剩余部分，其实现也有所不同。如果元素应该被插入到（已经排序的）列表第一个元素的前面，那么就用 PushFront 直接插入。否则，就将列表分成 head 和 tail 两部分，这样递归的尝试将元素插入到 tail 中，成功之后再用 PushFront 将 head 插入到 tail 的前面。

上述实现中包含了一个避免去实例化不会用到的类型的编译期优化，在第 19.7.1 节对该技术（去看看）进行了讨论。下面这个实现在技术上也是正确的：

```
template<typename List, typename Element, template<typename T,
typename U> class Compare>
class InsertSortedT<List, Element, Compare, false>
: public IfThenElseT<Compare<Element, Front<List>>::value,
PushFront<List, Element>,
                PushFront<InsertSorted<PopFront<List>, Element,
Compare>, Front<List>>>
{};
```

但是由于这种递归情况的实现方式会计算 IfThenElseT 的两个分支（虽然只会用到一个），其效率会受到影响。在这个实现中，在 IfThenElseT 的 then 分支中使用 PushFront 的成本非常低，但是在 else 分支中递归地使用 InsertSorted 的成本则很高。

在我们的优化实现中，第一个 IfThenElse 会计算出列表的 tail（NewTail）。其第二和第三个参数是用来计算特定结果的元函数。Then 分支中的参数使用 IdentityT（参见 19.7.1 节）来计算未被修改的 List。Else 分支中的参数用 InsertSortedT 来计算将元素插入到已排序列表之后的结果。在较高层面上，Identity 和 InsertSortedT 两者中只有一个会被实例化，因此不会有太多的额外工作。

第二个 IfThenElse 会计算上面获得的 list 的 head，其两个分支的计算代价都很低，因此都会被立即计算。最终的结果由 NewHead 和 NewTail 计算得到。

这一实现方案所需要的实例化数目，与被插入元素在一个已排序列表中的插入位置成正比。这表现为更高级别的插入排序属性：排序一个已经有序的列表，所需要实例化的数目和列表的长度成正比（如果已排序列表的排列顺序和预期顺序相反的话，所需要的实例化数目和列表长度的平方成正比）。

下面的程序会基于列表中元素的大小，用插入排序对其排序。比较函数使用了 sizeof 运算符并比较其结果：

```
template<typename T, typename U>
struct SmallerThanT {
    static constexpr bool value = sizeof(T) < sizeof(U);
};

void testInsertionSort()
{
    using Types = Typelist<int, char, short, double>;
    using ST = InsertionSort<Types, SmallerThanT>;
    std::cout << std::is_same<ST,Typelist<char, short, int,
double>>::value << ' \n' ;
}
```

24.3 非类型类型列表（Nontype Typelists）

通过类型列表，有非常多的算法和操作可以用来描述并操作一串类型。某些情况下，还会希望能够操作一串编译期数值，比如多维数组的边界，或者指向另一个类型列表中的索引。

有很多种方法可以用来生成一个包含编译期数值的类型列表。一个简单的办法是定义一个类模板 CTValue（compile time value），然后用它表示类型列表中某种类型的值：

```
template<typename T, T Value>
struct CTValue
{
    static constexpr T value = Value;
};
```

用它就可以生成一个包含了最前面几个素数的类型列表：

```
using Primes = Typelist<CTValue<int, 2>, CTValue<int, 3>,
             CTValue<int, 5>, CTValue<int, 7>,
             CTValue<int, 11>>;
```

这样就可以对类型列表中的数值进行数值计算，比如计算这些素数的乘积。

首先 MultiPlyT 模板接受两个类型相同的编译期数值作为参数，并生成一个新的、类型相同的编译期数值：

```
template<typename T, typename U>
struct MultiplyT;

template<typename T, T Value1, T Value2>
struct MultiplyT<CTValue<T, Value1>, CTValue<T, Value2>> {
    public:
        using Type = CTValue<T, Value1 * Value2>;
};

template<typename T, typename U>
using Multiply = typename MultiplyT<T, U>::Type;
```

然后结合 MultiplyT，下面的表达式就会返回所有 Primes 中素数的乘积：

```
Accumulate<Primes, MultiplyT, CTValue<int, 1>>::value
```

不过这一使用 Typelist 和 CTValue 的方式过于复杂，尤其是当所有数值的类型相同的时候。可以通过引入 CTTypelist 模板别名来进行优化，它提供了一组包含在 Typelist 中、类型相同的数值：

```
template<typename T, T... Values>
using CTTypelist = Typelist<CTValue<T, Values>...>;
```

这样就可以使用 CTTypelist 来定义一版更为简单的 Primes（素数）：

```
using Primes = CTTypelist<int, 2, 3, 5, 7, 11>;
```

这一方式的唯一缺点是，别名终归只是别名，当遇到错误的时候，错误信息可能会一直打印到 CTValueTypes 中的底层 Typelist，导致错误信息过于冗长。为了解决这一问题，可以定义一个能够直接存储数值的、全新的类型列表类 Valuelist：

```
template<typename T, T... Values>
struct Valuelist {
};

template<typename T, T... Values>
struct IsEmpty<Valuelist<T, Values...>> {
    static constexpr bool value = sizeof...(Values) == 0;
};

template<typename T, T Head, T... Tail>
struct FrontT<Valuelist<T, Head, Tail...>> {
    using Type = CTValue<T, Head>;
    static constexpr T value = Head;
};
```

```
template<typename T, T Head, T… Tail>
struct PopFrontT<Valuelist<T, Head, Tail…>> {
    using Type = Valuelist<T, Tail…>;
};

template<typename T, T… Values, T New>
struct PushFrontT<Valuelist<T, Values…>, CTValue<T, New>> {
    using Type = Valuelist<T, New, Values…>;
};

template<typename T, T… Values, T New>
struct PushBackT<Valuelist<T, Values…>, CTValue<T, New>> {
    using Type = Valuelist<T, Values…, New>;
};
```

通过代码中提供的 IsEmpty，FrontT，PopFrontT 和 PushFrontT，Valuelist 就可以被用于本章中介绍的各种算法了。PushBackT 被实现为一种算法的特例化，这样做可以降低编译期间该操作的计算成本。比如 Valuelist 可以被用于前面定义的算法 InsertionSort：

```
template<typename T, typename U>
struct GreaterThanT;

template<typename T, T First, T Second>
struct GreaterThanT<CTValue<T, First>, CTValue<T, Second>> {
    static constexpr bool value = First > Second;
};

void valuelisttest()
{
    using Integers = Valuelist<int, 6, 2, 4, 9, 5, 2, 1, 7>;
    using SortedIntegers = InsertionSort<Integers, GreaterThanT>;
    static_assert(std::is_same_v<SortedIntegers, Valuelist<int, 9, 7,
6, 5, 4, 2, 2, 1>>, "insertion sort failed");
}
```

注意在这里可以提供一种用字面值常量来初始化 CTValue 的功能，比如：

```
auto a = 42_c; // initializes a as CTValue<int,42>
```

相关细节请参见 25.6 节。

24.3.1 可推断的非类型参数

在 C++17 中，可以通过使用一个可推断的非类型参数（结合 auto）来进一步优化 CTValue 的实现：

```
template<auto Value>
struct CTValue
{
    static constexpr auto value = Value;
};
```

这样在使用 CTValue 的时候就可以不用每次都去指定一个类型了，从而简化了使用方式：

```
using Primes = Typelist<CTValue<2>, CTValue<3>, CTValue<5>, CTValue<7>,
CTValue<11>>;
```

在 C++17 中也可以对 Valuelist 执行同样的操作，但是结果可能不一定会变得更好。正如在第 15.10.1 节提到的那样，对一个非类型参数包进行类型推断时，各个参数可以不同：

```
template<auto… Values>
class Valuelist { };
int x;
using MyValueList = Valuelist<1,' a' , true, &x>;
```

虽然这样一个列表可能也很有用，但是它和之前要求元素类型必须相同的 Valuelist 已经不一样了。虽然我们也可以要求其所有元素的类型必须相同（参见 15.10.1 节的讨论），但是对于一个空的 Valuelist<>而言，其元素类型却是未知的。

24.4 对包扩展相关算法的优化（Optimizing Algorithms with Pack Expansions ）

通过使用包展开（参见 12.4.1 节），可以将类型列表迭代的任务转移给编译器。在第 24.2.5 节开发的 Transform 就是一个天生的、适用于包展开的算法，因为它会对所有列表中的元素执行相同的操作。这样就能够用一种偏特例化的算法对一个类型列表进行转换：

```
template<typename… Elements, template<typename T> class MetaFun>
class TransformT<Typelist<Elements…>, MetaFun, false>
{
    public:
        using Type = Typelist<typename MetaFun<Elements>::Type…>;
};
```

这一实现方式用以一个参数包 Elements 捕获了类型列表中的所有元素。接着它通过将 typename MetaFun<Elements>::Type 用于包展开，将元函数作用于 Elements 中的各个元素，并生成一个新的类型列表。可以认为这一实现方式更简单一些，因为它不需要递归，而是非常直观地使用了一些语言特性。除此之外，由于只需要实例化一个 Transform 模板的实例，它需要的模板实例的数目也更少。不过这个算法需要的 MetaFun 实例的数量依然是和列表长度成正比的，因为这些实例是该算法的基础，不可能被省略。

其它算法也可以从包展开中获益。比如在第 24.2.4 节介绍的 Reverse 算法，其需要对 PushBack

实例化的次数和列表的长度成正比。如果使用了在第 24.2.3 节介绍的、用到了包展开的 PushBack 的话（只需要一个 PushBack 实例），那么 Reverse 的复杂度和最终的列表长度也成正比。而如果是使用同样在这一节介绍的使用了常规递归方法的 PushBack，由于 PushBack 本身的复杂度和列表的长度也是正比关系，这样就会导致最终的 Reverse 和列表的长度成平方关系。

也可以基于索引值从一个已有列表中选择一些元素，并生成新的列表。Select 元函数接受一个类型列表和一个存储索引值的 Valuelist 作为参数，并最终生成一个包含了被索引元素的新的类型列表：

```
template<typename Types, typename Indices>
class SelectT;

template<typename Types, unsigned… Indices>
class SelectT<Types, Valuelist<unsigned, Indices…>>
{
    public:
        using Type = Typelist<NthElement<Types, Indices>…>;
};

template<typename Types, typename Indices>
using Select = typename SelectT<Types, Indices>::Type;
```

索引值被捕获进参数包 Indices 中，它被扩展成一串指向已有类型列表的 NthElement 类型，并生成一个新的类型列表。下面的代码展示了一种通过 Select 反转类型列表的方法：

```
using SignedIntegralTypes = Typelist<signed char, short, int, long, long
long>;

using ReversedSignedIntegralTypes = Select<SignedIntegralTypes,
Valuelist<unsigned, 4, 3, 2, 1, 0>>;
// produces Typelist<long long, long, int, short, signed char>
```

一个包含了指向另一个列表的索引的非类型类型列表，通常被称为索引列表（index list，或者索引序列，index sequence），可以通过它来简化甚至省略掉递归计算。在第 25.3.4 节会对索引列表进行详细介绍。

24.5 Cons-style Typelists（不完美的类型列表？）

在引入变参模板之前，类型列表通常参照 LISP 的 cons 单元的实现方式，用递归数据结构实现。每一个 cons 单元包含一个值（列表的 head）和一个嵌套列表，这个嵌套列表可以是另一个 cons 单元或者一个空的列表 nil。这一思路可以直接在 C++中按照如下方式实现：

```
class Nil { };

template<typename HeadT, typename TailT = Nil>
```

```
class Cons {
    public:
        using Head = HeadT;
        using Tail = TailT;
};
```

一个空的类型列表被记作 Nil，一个包含唯一元素 int 的类型列表则被记作 Cons<int, Nil>，也可以更简洁的记作 Cons<int>。比较长的列表则需要用到嵌套：

```
using TwoShort = Cons<short, Cons<unsigned short>>;
```

任意长度的类型列表则需要用比较深的递归嵌套来实现，虽然手写这么长的一个列表显得很不明智：

```
using SignedIntegralTypes = Cons<signed char, Cons<short, Cons<int,
Cons<long, Cons<long long, Nil>>>>>;
```

要从这样一个 cons-style 的列表中提取第一个元素，只需直接访问其头部元素：

```
template<typename List>
class FrontT {
    public:
        using Type = typename List::Head;
};

template<typename List>
using Front = typename FrontT<List>::Type;
```

向其头部追加以一个元素只需在当前类型列表外面包上一层 Cons 即可：

```
template<typename List, typename Element>
class PushFrontT {
    public:
        using Type = Cons<Element, List>;
};

template<typename List, typename Element>
using PushFront = typename PushFrontT<List, Element>::Type;
```

而如果要删除首元素的话，只需要提取出当前列表的 Tail 即可：

```
template<typename List>
class PopFrontT {
    public:
        using Type = typename List::Tail;
};

template<typename List>
using PopFront = typename PopFrontT<List>::Type;
```

至于 IsEmpty 的实现，只需要对 Nil 进行下特例化：

```
template<typename List>
struct IsEmpty {
    static constexpr bool value = false;
};

template<>
struct IsEmpty<Nil> {
    static constexpr bool value = true;
};
```

有了这些操作，就可以使用在第 24.2.7 节中介绍的 InsertionSort 算法了，只是这次是将它用于 cons-style list（不完美列表）：

```
template<typename T, typename U>
struct SmallerThanT {
    static constexpr bool value = sizeof(T) < sizeof(U);
};

void conslisttest()
{
    using ConsList = Cons<int, Cons<char, Cons<short, Cons<double>>>>;
    using SortedTypes = InsertionSort<ConsList, SmallerThanT>;using
Expected = Cons<char, Cons<short, Cons<int, Cons<double>>>>;
    std::cout << std::is_same<SortedTypes, Expected>::value <<' \n' ;
}
```

正如在 InsertionSort 算法中所见的那样，用 Cons-style 类型列表，可以实现本章中介绍的所有适用于变参类型列表的算法。事实上，其中一些算法的实现方式和我们操作 cons-style 类型列表的风格完全一样。但是 cons-style 类型列表的一些缺点还是促使我们更倾向于变参的版本：首先，嵌套的使用使得长的 cons-style 类型列表在源代码和编译器诊断信息方面即难以编写又难以阅读。其次，对于变参类型列表，一些算法（包含 PushBack 和 Transform）可以通过偏特化变的更高效（按照实例的数目计算）。最后，使用变参模板的类型列表能够很好的适应使用了变参模板的异质容器（比如第 25 章介绍的 tuple 和第 26 章介绍的可识别联合）。

24.6 后记

类型列表是在 1998 年 C++标准发布之后很快就出现的概念。在[CzarneckiEiseneckerGenProg]中 Krysztof Czarnecki 和 Ulrich Eisenecker 介绍了受 LISP 启发的、cons-style 的整型常数列表，虽然没能在此基础上实现向常规类型列表的跨越。

Alexandrescu 在其颇具影响力的图书 Modern C++ Design 中使得类型列表变得流行起来。除

此之外，Alexandrescu 还展示了使用类型列表和模板元编程解决一些很有意思的设计问题的可能，从而让 C++程序员能够比较容易的使用这一技能。

在[AbrahamsGurtovoyMeta]中，Abrahams 和 Gurtovoy 则提供了为元编程所急需的结构，介绍了类型列表的抽象，类型列表算法，和一些与 C++标准库中名称相似的相关元素：序列，迭代，算法，和元函数。相关的 Boost.MPL 库，被广泛用来操作类型列表。

第 25 章 元组（Tuples）

在本书中，我们经常使用“同质容器”（元素类型相同的），或者类似于数组的类型来说明模板的功能。这些同质的结构扩展了 C/C++中数组的概念，并且常见于大多数应用之中。C++（以及 C）也有“异质”的组件：class 或者 struct。本章将会讨论 tuples，它采用了类似于 class 和 struct 的方式来组织数据。比如，一个包含 int，double 和 std::string 的 tuple，和一个包含 int，double 以及 std::string 类型的成员的 struct 类似，只不过 tuple 中的元素是用位置信息（比如 0，1，2）索引的，而不是通过名字。元组的位置接口，以及能够容易地从 typelist 构建 tuple 的特性，使得其相比于 struct 更适用于模板元编程技术。

另一种观点是将元组看作在可执行程序中，类型列表的一种表现。比如，类型列表 Typelist<int, double, std::string>，描述了一串包含了 int，double 和 std::string 的、可以在编译期间操作的类型，而 Tuple<int,double, std::string>则描述了可以在运行期间操作的、对 int，double 和 std::string 的存储。比如下面的程序就创建了这样一个 tuple 的实例：

```
template<typename… Types>
class Tuple {
    … // implementation discussed below
};

Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

通常会使用模板元编程和 typelist 来创建用于存储数据的 tuple。比如，虽然在上面的程序中随意地选择了 int，double 和 std::string 作为元素类型，我们也可以用元程序创建一组可被 tuple 存储的类型。

在本章剩余的部分，我们会探讨 Tuple 类模板的相关实现和操作，可以将其看作是 std::tuple 的简化版本。

25.1 基本的元组设计

25.1.1 存储（Storage）

元组包含了对模板参数列表中每一个类型的存储。这部分存储可以通过函数模板 get 进行访问，对于元组 t，其用法为 get<I>(t)。比如，对于之前例子中的 t，get<0>(t)会返回指向 int 17 的引用，而 get<1>(t)返回的则是指向 double 3.14 的引用。

元组存储的递归实现是基于这样一个思路：一个包含了 N > 0 个元素的元组可以被存储为一个单独的元素（元组的第一个元素，Head）和一个包含了剩余 N-1 个元素（Tail）的元组，对于元素为空的元组，只需当作特例处理即可。因此一个包含了三个元素的元组 Tuple<int,

double, std::string> 可以被存储为一个 int 和一个 Tuple<double, std::string>。这个包含两个元素的元组又可以被存储为一个 double 和一个 Tuple<std::string>，这个只包含一个元素的元组又可以被存储为一个 std::string 和一个空的元组 Tuple<>。事实上，在类型列表算法的泛型版本中也使用了相同的递归分解过程，而且实际递归元组的存储实现也以类似的方式展开：

```
template<typename… Types>
class Tuple;

// recursive case:
template<typename Head, typename… Tail>
class Tuple<Head, Tail…>
{
    private:
        Head head;
        Tuple<Tail…> tail;
    public:
        // constructors:
        Tuple() {
        }

        Tuple(Head const& head, Tuple<Tail…> const& tail): head(head),
tail(tail) {
        }
        …

        Head& getHead() { return head; }
        Head const& getHead() const { return head; }
        Tuple<Tail…>& getTail() { return tail; }
        Tuple<Tail…> const& getTail() const { return tail; }
};

// basis case:
template<>
class Tuple<> {
    // no storage required
};
```

在递归情况下，Tuple 的实例包含一个存储了列表首元素的 head，以及一个存储了列表剩余元素的 tail。基本情况则是一个没有存储内容的简单的空元组。

而函数模板 get 则会通过遍历这个递归的结构来提取所需要的元素：

```
// recursive case:
template<unsigned N>
struct TupleGet {
```

```
    template<typename Head, typename… Tail>
    static auto apply(Tuple<Head, Tail…> const& t) {
        return TupleGet<N-1>::apply(t.getTail());
    }
};

// basis case:
template<>
struct TupleGet<0> {
    template<typename Head, typename… Tail>
    static Head const& apply(Tuple<Head, Tail…> const& t) {
        return t.getHead();
    }
};

template<unsigned N, typename… Types>
auto get(Tuple<Types…> const& t) {
    return TupleGet<N>::apply(t);
}
```

注意，这里的函数模板 get 只是封装了一个简单的对 TupleGet 的静态成员函数调用。在不能对函数模板进行部分特例化的情况下（参见 17.3 节），这是一个有效的变通方法，在这里针对非类型模板参数 N 进行了特例化。在 N > 0 的递归情况下，静态成员函数 apply()会提取出当前 tuple 的 tail，递减 N，然后继续递归地在 tail 中查找所需元素。对于 N=0 的基本情况，apply()会返回当前 tuple 的 head，并结束递归。

25.1.2 构造

除了前面已经定义的构造函数：

```
Tuple() {
}

Tuple(Head const& head, Tuple<Tail…> const& tail)
: head(head), tail(tail)
{
}
```

为了让元组的使用更方便，还应该允许用一组相互独立的值（每一个值对应元组中的一个元素）或者另一个元组来构造一个新的元组。从一组独立的值去拷贝构造一个元组，会用第一个数值去初始化元组的 head，而将剩余的值传递给 tail：

```
Tuple(Head const& head, Tail const&… tail)
: head(head), tail(tail…)
{
```

```
}
```

这样就可以像下面这样初始化一个元组了：

```
Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

不过这并不是最通用的接口：用户可能会希望用移动构造（move-construct）来初始化元组的一些（可能不是全部）元素，或者用一个类型不相同的数值来初始化元组的某个元素。因此我们需要用完美转发（参见 15.6.3 节）来初始化元组：

```
template<typename VHead, typename… VTail>
Tuple(VHead&& vhead, VTail&&… vtail)
  : head(std::forward<VHead>(vhead)), tail(std::forward<VTail>(vtail)…)
{
}
```

下面的这个实现则允许用一个元组去构建另一个元组：

```
template<typename VHead, typename… VTail>
Tuple(Tuple<VHead, VTail…> const& other)
: head(other.getHead()), tail(other.getTail())
{ }
```

但是这个构造函数不适用于类型转换：给定上文中的 t，试图用它去创建一个元素之间类型兼容的元组会遇到错误：

```
// ERROR: no conversion from Tuple<int, double, string> to long
Tuple<long int, long double, std::string> t2(t);
```

这是因为上面这个调用，会更匹配用一组数值去初始化一个元组的构造函数模板，而不是用一个元组去初始化另一个元组的构造函数模板。为了解决这一问题，就需要用到 6.3 节介绍的 std::enable_if<>，在 tail 的长度与预期不同的时候就禁用相关模板：

```
template<typename VHead, typename… VTail, typename = std::enable_if_t<sizeof…
(VTail)==sizeof… (Tail)>>
Tuple(VHead&& vhead, VTail&&… vtail)
: head(std::forward<VHead>(vhead)), tail(std::forward<VTail>(vtail)…)
{ }

template<typename VHead, typename… VTail, typename = std::enable_if_t<sizeof…
(VTail)==sizeof… (Tail)>>
Tuple(Tuple<VHead, VTail…> const& other)
: head(other.getHead()), tail(other.getTail()) { }
```

你可以在 tuples/tuple.h 中找到所有的构造函数声明。

函数模板 makeTuple()会通过类型推断来决定所生成元组中元素的类型，这使得用一组数值创建一个元组变得更加简单：

```
template<typename… Types>
```

```
auto makeTuple(Types&&… elems)
{
  return Tuple<std::decay_t<Types>…>(std::forward<Types> (elems)…);
}
```

这里再一次将 std::decay<>和完美转发一起使用，这会将字符串常量和裸数组转换成指针，并去除元素的 const 和引用属性。比如：

```
makeTuple(17, 3.14, "Hello, World!")
```

生成的元组的类型是：

```
Tuple<int, double, char const*>
```

25.2 基础元组操作

25.2.1 比较

元组是包含了其它数值的结构化类型。为了比较两个元组，就需要比较它们的元素。因此可以像下面这样，定义一种能够逐个比较两个元组中元素的 operator==：

```
// basis case:
bool operator==(Tuple<> const&, Tuple<> const&)
{
  // empty tuples are always equivalentreturn true;
}

// recursive case:
template<typename Head1, typename… Tail1,
       typename Head2, typename… Tail2,
       typename = std::enable_if_t<sizeof…(Tail1)==sizeof…(Tail2)>>
bool operator==(Tuple<Head1, Tail1…> const& lhs, Tuple<Head2, Tail2…> const& rhs)
{
  return lhs.getHead() == rhs.getHead() &&
         lhs.getTail() == rhs.getTail();
}
```

（应该还需要定义一版 sizeof…(Tail1) != sizeof…(Tail2)的 operator==）

和其它适用于类型列表和元组的算法类似，逐元素的比较两个元组，会先比较首元素，然后递归地比较剩余的元素，最终会调用 operator 的基本情况结束递归。运算符!=，<，>，以及>=的实现方式都与之类似。

25.2.2 输出

贯穿本章始终，我们一直都在创建新的元组类型，因此最好能够在执行程序的时候看到这些元组。下面的 operator<<运算符会打印那些元素类型可以被打印的元组：

```
#include <iostream>
void printTuple(std::ostream& strm, Tuple<> const&, bool isFirst = true)
{
  strm << ( isFirst ? ' (' : ' )' );
}

template<typename Head, typename… Tail>
void printTuple(std::ostream& strm, Tuple<Head, Tail…> const& t, bool isFirst =
true)
{
  strm << ( isFirst ? "(" : ", " );
  strm << t.getHead();
  printTuple(strm, t.getTail(), false);
}

template<typename … Types>
std::ostream& operator<<(std::ostream& strm, Tuple<Types…> const& t)
{
  printTuple(strm, t);
  return strm;
}
```

这样就可以很容易地创建并打印元组了。比如：

```
std::cout << makeTuple(1, 2.5, std::string("hello")) << ' \n' ;
```

会打印出：

```
(1, 2.5, hello)
```

25.3 元组的算法

元组是一种提供了以下各种功能的容器：可以访问并修改其元素的能力（通过 get<>），创建新元组的能力（直接创建或者通过使用 makeTuple<>创建），以及将元组分割成 head 和 tail 的能力（通过使用 getHead()和 getTail()）。使用这些功能足以创建各种各样的元组算法，比如添加或者删除元组中的元素，重新排序元组中的元素，或者选取元组中元素的某些子集。

元组很有意思的一点是它既需要用到编译期计算也需要用到运行期计算。和第 24 章介绍的类型列表算法类似，将某种算法作用与元组之后可能会得到一个类型迥异的元组，这就需要用到编译期计算。比如反转元组 Tuple<int, double, string>会得到 Tuple<string, double, int>。

但是和同质容器的算法类似（比如作用域 std::vector 的 std::reverse()），元组算法是需要在运行期间执行代码的，因此我们需要留意被产生出来的代码的效率问题。

25.3.1 将元组用作类型列表

如果我们忽略掉 Tuple 模板在运行期间的相关部分，可以发现它在结构上和第 24 章介绍的 Typelist 完全一样：都接受任意数量的模板类型参数。事实上，通过使用一些部分特例化，可以将 Tuple 变成一个功能完整的 Typelist：

```
// determine whether the tuple is empty:
template<>
struct IsEmpty<Tuple<>> {
  static constexpr bool value = true;
};

// extract front element:
template<typename Head, typename… Tail>
class FrontT<Tuple<Head, Tail…>> {
  public:
      using Type = Head;
};

// remove front element:
template<typename Head, typename… Tail>
class PopFrontT<Tuple<Head, Tail…>> {
  public:
      using Type = Tuple<Tail…>;
};

// add element to the front:
template<typename… Types, typename Element>
class PushFrontT<Tuple<Types…>, Element> {
  public:
      using Type = Tuple<Element, Types…>;
};

// add element to the back:
template<typename… Types, typename Element>
class PushBackT<Tuple<Types…>, Element> {
  public:
      using Type = Tuple<Types…, Element>;
};
```

现在，所有在第 24 章开发的 typlist 算法都既适用于 Tuple 也适用于 Typelist，这样就可以很

方便的处理元组的类型了。比如：

```
Tuple<int, double, std::string> t1(17, 3.14, "Hello, World!");
using T2 = PopFront<PushBack<decltype(t1), bool>>;
T2 t2(get<1>(t1), get<2>(t1), true);
std::cout << t2;
```

会打印出：

```
(3.14, Hello, World!, 1)
```

很快就会看到，将 typelist 算法用于 tuple，通常是为了确定 tuple 算法返回值的类型。

25.3.2 添加以及删除元素

对于 Tuple，能否向其头部或者尾部添加元素，对开发相关的高阶算法而言是很重要的。和 typelist 的情况一样，向头部插入一个元素要远比向尾部插入一个元素要简单，因此我们从 pushFront 开始：

```
template<typename… Types, typename V>
PushFront<Tuple<Types…>, V>
pushFront(Tuple<Types…> const& tuple, V const& value)
{
  return PushFront<Tuple<Types…>, V>(value, tuple);
}
```

将一个新元素（称之为 value）添加到一个已有元组的头部，需要生成一个新的、以 value 为 head、以已有 tuple 为 tail 的元组。返回结过的类型是 Tuple<V, Types…>。不过这里我们选择使用 typelist 的算法 PushFront 来获得返回类型，这样做可以体现出 tuple 算法中编译期部分和运行期部分之间的紧密耦合关系：编译期的 PushFront 计算出了我们应该生成的运行期结果的类型。

将一个新元素添加到一个已有元组的末尾则会复杂得多，因为这需要遍历一个元组。注意下面的代码中 pushBack()的实现方式，是如何参考了第 24.2.3 节中类型列表的 PushBack()的递归实现方式的：

```
// basis case
template<typename V>
Tuple<V> pushBack(Tuple<> const&, V const& value)
{
  return Tuple<V>(value);
}

// recursive case
template<typename Head, typename… Tail, typename V>
Tuple<Head, Tail…, V>
pushBack(Tuple<Head, Tail…> const& tuple, V const& value)
```

```
{
  return Tuple<Head, Tail…, V>(tuple.getHead(),
                               pushBack(tuple.getTail(), value));
}
```

对于基本情况，和预期的一样，会将值追加到一个长度为零的元组的后面。对于递归情况，则将元组分为 head 和 tail 两部分，然后将首元素以及将新元素追加到 tail 的后面得到结果组装成最终的结果。虽然这里我们使用的返回值类型是 Tuple<Head, Tail…, V> ，但是它和编译期的 PushBack<Tuple<Hrad, Tail...>, V>是一样的。

同样地，popFront()也很容易实现：

```
template<typename… Types>
PopFront<Tuple<Types…>> popFront(Tuple<Types…> const& tuple)
{
  return tuple.getTail();
}
```

现在我们可以像下面这样编写第 25.3.1 节的例子：

```
Tuple<int, double, std::string> t1(17, 3.14, "Hello, World!");
auto t2 = popFront(pushBack(t1, true));
std::cout << std::boolalpha << t2 << ' \n' ;
```

打印结果为：

```
(3.14, Hello, World!, true)
```

25.3.3 元组的反转

元组的反转可以采用另一种递归的、类似在第 24.2.4 节介绍的、类型列表的反转方式实现：

```
// basis case
Tuple<> reverse(Tuple<> const& t)
{
  return t;
}

// recursive case
template<typename Head, typename… Tail>
Reverse<Tuple<Head, Tail…>> reverse(Tuple<Head, Tail…> const& t)
{
  return pushBack(reverse(t.getTail()), t.getHead());
}
```

基本情况比较简单，而递归情况则是递归地将 head 追加到反转之后的 tail 的后面。也就是说：

```
reverse(makeTuple(1, 2.5, std::string("hello")))
```

会生成一个包含了 string(“hello”)，2.5，和 1 的类型为 Tuple<string, double, int>的元组。

和类型列表类似，现在就可以简单地通过先反转元组，然后调用 popFront()，然后再次反转元组实现 popBack():

```
template<typename… Types>
PopBack<Tuple<Types…>> popBack(Tuple<Types…> const& tuple){
  return reverse(popFront(reverse(tuple)));
}
```

25.3.4 索引列表

虽然上文中反转元组用到的递归方式是正确的，但是它在运行期间的效率却非常低。为了展现这一问题，引入下面这个可以计算其实例被 copy 次数的类：

```
template<int N>
struct CopyCounter
{
  inline static unsigned numCopies = 0;
  CopyCounter()
  {
  }
  CopyCounter(CopyCounter const&) {
      ++numCopies;
  }
};
```

然后创建并反转一个包含了 CopyCounter 实例的元组：

```
void copycountertest()
{
  Tuple<CopyCounter<0>, CopyCounter<1>, CopyCounter<2>, CopyCounter<3>,
CopyCounter<4>> copies;
  auto reversed = reverse(copies);
  std::cout << "0: " << CopyCounter<0>::numCopies << " copies\n";
  std::cout << "1: " << CopyCounter<1>::numCopies << " copies\n";
  std::cout << "2: " << CopyCounter<2>::numCopies << " copies\n";
  std::cout << "3: " << CopyCounter<3>::numCopies << " copies\n";
  std::cout << "4: " << CopyCounter<4>::numCopies << " copies\n";
}
```

这个程序会打印出：

```
0: 5 copies
1: 8 copies
```

```
2: 9 copies
3: 8 copies
4: 5 copies
```

这确实进行了很多次 copy！在理想的实现中，反转一个元组时，每一个元素只应该被 copy 一次：从其初始位置直接被 copy 到目的位置。我们可以通过使用引用来达到这一目的，包括对中间变量的类型使用引用，但是这样做会使实现变得很复杂。

在反转元组时，为了避免不必要的 copy，考虑一下我们该如何实现一个一次性的算法，来反转一个简单的、长度已知的元组（比如包含 5 个元素）。可以像下面这样只是简单地使用 makeTuple()和 get():

```
auto reversed = makeTuple(get<4>(copies), get<3>(copies), get<2>(copies),
                          get<1>(copies), get<0>(copies));
```

这个程序会按照我们预期的那样进行，对每个元素只进行一次 copy：

```
0: 1 copies
1: 1 copies
2: 1 copies
3: 1 copies
4: 1 copies
```

索引列表（亦称索引序列，参见第 24.4 节）通过将一组元组的索引捕获进一个参数包，推广了上述概念，本例中的索引列表是 4，3，2，1，0，这样就可以通过包展开进行一组 get 函数的调用。采用这种方法可以将索引列表的计算（可以采用任意复杂度的模板源程序）和使用（更关注运行期的性能）分离开。在 C++14 中引入的标准类型 std::integer_sequence，通常被用来表示索引列表。

25.3.5 通过索引列表进行反转

为了将索引列表用于元组反转，我们首先要找到一种能够表达索引列表的方式。索引列表是一种包含了数值的类型列表，这些数值被用作指向另一个类型列表或者异质容器（参见 25.4 节）的索引。此处我们将第 24.3 节介绍的 Valuelist 用作类型列表。上文例子中反转元组时用到的索引列表可以被写成：

```
Valuelist<unsigned, 4, 3, 2, 1, 0>
```

那么该如何生成一个索引列表呢？一种方式是使用下面的这个简单的模板元函数 MakeIndexList，它从 0 到 N-1（N 是元组长度）逐步生成索引列表：

```
// recursive case
template<unsigned N, typename Result = Valuelist<unsigned>>
struct MakeIndexListT
: MakeIndexListT<N-1, PushFront<Result, CTValue<unsigned, N-1>>>
{};
```

```
// basis case
template<typename Result>
struct MakeIndexListT<0, Result>
{
  using Type = Result;
};

template<unsigned N>
using MakeIndexList = typename MakeIndexListT<N>::Type;
```

现在就可以结合 MakeIndexList 和在第 24.2.4 节介绍的类型列表的 Reverse 算法，生成所需的索引列表：

```
using MyIndexList = Reverse<MakeIndexList<5>>;
// equivalent to Valuelist<unsigned, 4, 3, 2,1, 0>
```

为了真正实现反转，需要将索引列表中的索引捕获进一个非类型参数包。这可以通过将 reverse()分成两部分来实现：

```
template<typename… Elements, unsigned… Indices>
auto reverseImpl(Tuple<Elements…> const& t, Valuelist<unsigned, Indices…>)
{
  return makeTuple(get<Indices>(t)…);
}

template<typename… Elements>
auto reverse(Tuple<Elements…> const& t)
{
  return reverseImpl(t, Reverse<MakeIndexList<sizeof…(Elements)>>());
}
```

在 C++11 中相应的返回类型要通过尾置返回类型声明：

```
-> decltype(makeTuple(get<Indices>(t)…))
```

和：

```
-> decltype(reverseImpl(t, Reverse<MakeIndexList<sizeof… (Elements)>>()))
```

其中函数模板 reverseImpl()从其参数 Valuelist 中捕获相应的索引信息，并将之存储进参数包 Indices 中。然后以 get<Indices>(t)…为参数调用 makeTuple()，并生成返回结果。

而 reverse()所做的只是生成合适的索引组，然后以之为参数调用 reverseImpl。这里用模板元程序操作索引列表，因此不会生成任何运行期间的代码。唯一的运行期代码是 reverseImpl，它通过调用 makeTuple()，只用一步就生成了最终的结果，而且只对元组中的元素进行了一次 copy。

25.3.6 洗牌和选择（Shuffle and Select ）

事实上，上一节中为了反转元组而用到的函数模板 reverseImpl()，并不是仅适用于 reverse()。它所做的只是从一个已有元组中选出一组特定的值，并用它们生成一个新的元组。虽然 reverse()提供的是一组反序的索引，但是其它一些算法可以通过提供一组自己的索引来使用下面的 select()算法：

```
template<typename… Elements, unsigned… Indices>
auto select(Tuple<Elements…> const& t, Valuelist<unsigned, Indices…>)
{
  return makeTuple(get<Indices>(t)…);
}
```

一个使用了 select()的简单算法是“splat”，它从元组中选出一个元素，将之重复若干次之后组成一个新的元组。比如：

```
Tuple<int, double, std::string> t1(42, 7.7, "hello");
auto a = splat<1, 4>(t);
std::cout << a << ' \n' ;
```

它会生成一个 Tuple<double, double, double, double>类型的元组，其每一个值都是 get<1>(t)的一份 copy，因此最终打印的结果是：

```
(7.7, 7.7, 7.7, 7.7)
```

在提供了一个能够生成一组重复索引（N 个 I）的元程序后，就可以直接用 select()实现 splat()：

```
template<unsigned I, unsigned N, typename IndexList = Valuelist<unsigned>>
class ReplicatedIndexListT;

template<unsigned I, unsigned N, unsigned… Indices>
class ReplicatedIndexListT<I, N, Valuelist<unsigned, Indices…>>
: public ReplicatedIndexListT<I, N-1, Valuelist<unsigned, Indices…, I>>
{ };

template<unsigned I, unsigned… Indices>
class ReplicatedIndexListT<I, 0, Valuelist<unsigned, Indices…>> {
  public:
      using Type = Valuelist<unsigned, Indices…>;
};

template<unsigned I, unsigned N>
using ReplicatedIndexList = typename ReplicatedIndexListT<I, N>::Type;

template<unsigned I, unsigned N, typename… Elements>
auto splat(Tuple<Elements…> const& t)
{
```

```
    return select(t, ReplicatedIndexList<I, N>());
}
```

即使是更复杂的元组算法，也可以通过使用 select()函数和一个操作索引列表的模板元函数实现。比如，可以用在第 24.2.7 节开发的插入排序算法，基于元素类型的大小对元组进行排序。假设有这样一个 sort()函数，它接受一个用来比较元组元素类型的模板元函数作为参数，就可以按照下面的方式对元组进行排序：

```
#include <complex>
template<typename T, typename U>
class SmallerThanT
{
  public:
      static constexpr bool value = sizeof(T) < sizeof(U);
};

void testTupleSort()
{
  auto T1 = makeTuple(17LL, std::complex<double>(42,77), 'c', 42, 7.7);
  std::cout << t1 << '\n';
  auto T2 = sort<SmallerThanT>(t1); // t2 is Tuple<int, long,
std::string>
  std::cout << "sorted by size: " << t2 << '\n';
}
```

输出结果如下：

```
(17, (42,77), c, 42, 7.7)
sorted by size: (c, 42, 7.7, 17, (42,77))
```

sort()的具体实现使用了 InsertionSort 和 select()：

```
// metafunction wrapper that compares the elements in a tuple:
template<typename List, template<typename T, typename U> class F>
class MetafunOfNthElementT {
  public:
      template<typename T, typename U>
      class Apply;

      template<unsigned N, unsigned M>
      class Apply<CTValue<unsigned, M>, CTValue<unsigned, N>>
          : public F<NthElement<List, M>, NthElement<List, N>>
      { };
};

// sort a tuple based on comparing the element types:
template<template<typename T, typename U> class Compare, typename…
```

```
Elements>
auto sort(Tuple<Elements…> const& t)
{
  return select(t, InsertionSort<MakeIndexList<sizeof…(Elements)>,
                                 MetafunOfNthElementT<Tuple<Elements…>,
Compare>::template Apply>());
}
```

注意 InsertionSort 的使用：真正被排序的类型列表是一组指向类型列表的索引，该索引通过 MakeIndexList<>构造。因此插入排序的结果是一组指向元组的索引，并被传递给 selete()使用。不过由于 InsertionSort 被用来操作索引，它所期望的比较操作自然也是比较两个索引。考虑一下对一个 std::vector 的索引进行排序的情况，就很容易理解背后的相关原理了，比如下面的这个（非元编程）例子：

```
#include <vector>
#include <algorithm>
#include <string>
int main()
{
  std::vector<std::string> strings = {"banana", "apple", "cherry"};
  std::vector<unsigned> indices = { 0, 1, 2 };
  std::sort(indices.begin(), indices.end(),
           [&strings](unsigned i, unsigned j) {
             return strings[i] < strings[j];
           }
          );
}
```

这里变量 indices 包含的是指向变量 strings 的索引。sort()函数对索引进行排序，它用到了一个接受两个 unsigned 类型的数值作为参数的 lambda 比较函数。但是由于 lambda 函数的主体将 unsigned 的数值当作 strings 变量的索引处理，因此真正被排序的还是 strings 的内容。在排序的最后，变量 indices 包含的依然是指向 strings 的索引，只是这个索引是按照 strings 的值进行排序之后的索引。

我们在代码中将 InsertionSort 用于元组的 sort()函数，情况和上面的例子是一样的。在适配模板 MetafuncOfNthElementT 中提供了一个接受两个索引作为参数的模板元函数（Apply()），而它又会使用 NthElement 从其 Typelist 参数中提取相应的元素。在某种意义上，成员模板 Apply 捕获了提供给其外层模板（MetafunOfNthElementT）的类型列表参数，这和 lambda 函数捕获其外层作用域中的 strings vector 的情况类似。然后 Apply 将其提取的元素类型转发给底层的元函数 F，并结束适配。

注意上文中所有排序相关的计算都发生在编译期间，作为结果的元素也是直接生成的，不会用到运行期间的拷贝。

25.4 元组的展开

在需要将一组相关的数值存储到一个变量中时（不管这些相关数值的数量是多少、类型是什么），元组会很有用。在某些情况下，可能会需要展开一个元组（比如在需要将其元素作为独立参数传递给某个函数的时候）。作为一个简单的例子，可能需要将一个元组的元素传递给在第 12.4 节介绍的变参 print()：

```
Tuple<std::string, char const*, int, char> t("Pi", "is roughly", 3, ' \n' );
print(t…); //ERROR: cannot expand a tuple; it isn' t a parameter pack
```

正如例子中注释部分所讲的，这个“明显”需要展开一个元组的操作会失败，因为它不是一个参数包。不过我们可以使用索引列表实现这一功能。下面的函数模板 apply()接受一个函数和一个元组作为参数，然后以展开后的元组元素为参数，去调用这个函数：

```
template<typename F, typename… Elements, unsigned… Indices>
auto applyImpl(F f, Tuple<Elements…> const& t,
Valuelist<unsigned, Indices…>) ->decltype(f(get<Indices>(t)…))
{
  return f(get<Indices>(t)…);
}

template<typename F, typename… Elements, unsigned N = sizeof…(Elements)>
auto apply(F f, Tuple<Elements…> const& t) ->decltype(applyImpl(f, t,
MakeIndexList<N>()))
{
  return applyImpl(f, t, MakeIndexList<N>());
}
```

函数模板 applyImpl()会接受一个索引列表作为参数，并用其将元组中的元素展开成一个适用于函数对象 f 的参数列表。而供用户直接使用的 apply()则只是负责构建初始的索引列表。这样就可以将一个元组扩展成 print()的参数了：

```
Tuple<std::string, char const*, int, char> t("Pi", "is roughly",
3, ' \n' );
apply(print, t); //OK: prints Pi is roughly 3
```

在 C++17 中，则提供了一个功能类似的、适用于任意和元组相近的类型的函数。

25.5 元组的优化

元组是一种基础的、潜在用途广泛的异质容器。因此有必要考虑下该怎么在运行期（存储和执行时间）和编译期（实例化的数量）对其进行优化。本节将介绍一些适用于上文中实现的元组的特定优化方案。

25.5.1 元组和 EBCO

我们实现的元组，其存储方式所需要的存储空间，要比其严格意义上所需要的存储空间多。其中一个问题是，tail 成员最终会是一个空的数值（因为所有非空的元组都会以一个空的元组作为结束），而任意数据成员又总会至少占用一个字节的内存（参见 21.1 节）。

为了提高元组的存储效率，可以使用第 21.1 节介绍的空基类优化（EBCO，empty base class optimization），让元组继承自一个尾元组（tail tuple），而不是将尾元组作为一个成员。比如：

```
// recursive case:
template<typename Head, typename… Tail>
class Tuple<Head, Tail…> : private Tuple<Tail…>
{
  private:
      Head head;
  public:
      Head& getHead() { return head; }
      Head const& getHead() const { return head; }
      Tuple<Tail…>& getTail() { return *this; }
      Tuple<Tail…> const& getTail() const { return *this; }
};
```

这和第 21.1.2 节中的 BaseMemberPair 使用的优化方式一致。不幸的是，这种方式有其副作用，就是颠倒了元组元素在构造函数中被初始化的顺序。在之前的实现中，head 成员在 tail 成员前面，因此 head 总是会先被初始化。在新的实现方式中，tail 则是以基类的形式存在，因此它会在 head 成员之前被初始化。

这一问题可以通过将 head 成员放入其自身的基类中，并让这个基类在基类列表中排在 tail 的前面来解决。该方案的一个直接实现方式是，引入一个用来封装各种元素类型的 TupleElt 模板，并让 Tuple 继承自它：

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

template<typename T>
class TupleElt
{
  T value;
  public:
      TupleElt() = default;
      template<typename U>
      TupleElt(U&& other) : value(std::forward<U>(other)) { }
      T& get() { return value; }
      T const& get() const { return value; }
};
```

```
// recursive case:
template<typename Head, typename... Tail>
class Tuple<Head, Tail...>
: private TupleElt<Head>, private Tuple<Tail...>
{
  public:
      Head& getHead() {
          // potentially ambiguous
          return static_cast<TupleElt<Head> *>(this)->get();
      }
      Head const& getHead() const {
          // potentially ambiguous
          return static_cast<TupleElt<Head> const*>(this)->get();
      }
      Tuple<Tail...>& getTail() { return *this; }
      Tuple<Tail...> const& getTail() const { return *this; }
};

// basis case:
template<>
class Tuple<> {
    // no storage required
};
```

虽然这一方式解决了元素初始化顺序的问题，但是却引入了一个更糟糕的问题：如果一个元组包含两个类型相同的元素（比如 Tuple<int, int>），我们将不再能够从中提取元素，因为此时从 Tuple<int, int>向 TupleElt<int>的转换（自派生类向基类的转换）不是唯一的（有歧义）。

为了打破歧义，需要保证在给定的 Tuple 中每一个 TupleElt 基类都是唯一的。一个方式是将这个值的“高度”信息（也就是 tail 元组的长度信息）编码进元组中。元组最后一个元素的高度会被存储生 0，倒数第一个元素的长度会被存储成 1，以此类推：

```
template<unsigned Height, typename T>
class TupleElt {
  T value;
  public:
      TupleElt() = default;
      template<typename U>
      TupleElt(U&& other) : value(std::forward<U>(other)) { }
      T& get() { return value; }
      T const& get() const { return value; }
};
```

通过这一方式，就能够实现一个即使用了 EBCO 优化，又能保持元素的初始化顺序，并支持

包含相同类型元素的元组：

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

// recursive case:
template<typename Head, typename... Tail>
class Tuple<Head, Tail...>
: private TupleElt<sizeof...(Tail), Head>, private Tuple<Tail...>
{
  using HeadElt = TupleElt<sizeof...(Tail), Head>;
  public:
      Head& getHead() {
          return static_cast<HeadElt *>(this)->get();
      }

      Head const& getHead() const {
          return static_cast<HeadElt const*>(this)->get();
      }
      Tuple<Tail...>& getTail() { return *this; }
      Tuple<Tail...> const& getTail() const { return *this; }
};

// basis case:
template<>
class Tuple<> {
  // no storage required
};
```

基于这一实现，下面的程序：

```
#include <algorithm>
#include "tupleelt1.hpp"
#include "tuplestorage3.hpp"
#include <iostream>

struct A {
  A() {
      std::cout << "A()" << ' \n' ;
  }
};

struct B {
  B() {
      std::cout << "B()" << ' \n' ;
  }
```

```
};

int main()
{
  Tuple<A, char, A, char, B> t1;
  std::cout << sizeof(t1) << " bytes" << ' \n' ;
}
```

会打印出：

```
A()
A()
B()
5 bytes
```

从中可以看出，EBCO 使得内存占用减少了一个字节（减少的内容是空元组 Tuple<>）。但是请注意 A 和 B 都是空的类，这暗示了进一步用 EBCO 进行优化的可能。如果能够安全的从其元素类型继承的话，那么就让 TupleElt 继承自其元素类型（这一优化不需要更改 Tuple 的定义）：

```
#include <type_traits>
template<unsigned Height, typename T,
         bool = std::is_class<T>::value && !std::is_final<T>::value>
class TupleElt;

template<unsigned Height, typename T>
class TupleElt<Height, T, false>
{
  T value;
  public:
      TupleElt() = default;

      template<typename U>
      TupleElt(U&& other) : value(std::forward<U>(other)) { }

      T& get() { return value; }

      T const& get() const { return value; }
};

template<unsigned Height, typename T>
class TupleElt<Height, T, true> : private T
{
  public:
      TupleElt() = default;
```

```
        template<typename U>
        TupleElt(U&& other) : T(std::forward<U>(other)) { }

        T& get() { return *this; }

        T const& get() const { return *this; }
};
```

当提供给 TupleElt 的模板参数是一个可以被继承的类的时候，它会从该模板参数做 private 继承，从而也可以将 EBCO 用于被存储的值。有了这些变化，之前的程序会打印出：

```
A()
A()
B()
2 bytes
```

25.5.2 常数时间的 get()

在使用元组的时候，get()操作的使用是非常常见的，但是其递归的实现方式需要用到线性次数的模板实例化，这会影响编译所需要的时间。幸运的是，基于在之前章节中介绍的 EBCO，可以实现一种更高效的 get，我们接下来会对其进行讨论。

主要的思路是，当用一个（基类类型的）参数去适配一个（派生类类型的）参数时，模板参数推导（参见第 15 章）会为基类推断出模板参数的类型。因此，如果我们能够计算出目标元素的高度 H，就可以不用遍历所有的索引，也能够基于从 Tuple 的特化结果向 TupleElt<H,T>（T 的类型由推断得到）的转化提取出相应的元素：

```
template<unsigned H, typename T>
T& getHeight(TupleElt<H,T>& te)
{
  return te.get();
}

template<typename... Types>
class Tuple;

template<unsigned I, typename... Elements>
auto get(Tuple<Elements...>& t) ->
decltype(getHeight<sizeof...(Elements)-I-1>(t))
{
  return getHeight<sizeof...(Elements)-I-1>(t);
}
```

由于 get<I>(t)接收目标元素（从元组头部开始计算）的索引 I 作为参数，而元组的实际存储是以高度 H 来衡量的（从元组的末尾开始计算），因此需要用 H 来计算 I。真正的查找工

作是由调用 getHeight()时的参数推导执行的：由于 H 是在函数调用时显示指定的，因此它的值是确定的，这样就只会有一个 TupleElt 会被匹配到，其模板参数 T 则是通过推断得到的。这里必须要将 getHeight()声明伟 Tuple 的 friend，否则将无法执行从派生类向 private 父类的转换。比如：

```
// inside the recursive case for class template Tuple:
template<unsigned I, typename… Elements>
friend auto get(Tuple<Elements…>& t)
  -> decltype(getHeight<sizeof…(Elements)-I-1>(t));
```

由于我们已经将繁杂的索引匹配工作转移到了编译器的模板推断那里，因此这一实现方式只需要常数数量的模板实例化。

25.6 元组下标

理论上也可以通过定义 operator[]来访问元组中的元素，这和在 std::vector 中定义 operator[]的情况类似。不过和 std::vector 不同的是，元组中元素的类型可以不同，因此元组的 operator[]必须是一个模板，其返回类型也需要随着索引的不同而不同。这反过来也就要求每一个索引都要有不同的类型，因为需要根据索引的类型来决定元素的类型。

使用在第 24.3 节介绍的类模板 CTValue，可以将数值索引编码进一个类型中。将其用于 Tuple 下标运算符定义的代码如下：

```
template<typename T, T Index>
auto& operator[](CTValue<T, Index>) {
  return get<Index>(*this);
}
```

然后就可以基于被传递的 CTValue 类型的参数，用其中的索引信息去执行相关的 get<>()调用。

上述代码的用法如下：

```
auto t = makeTuple(0, '1', 2.2f, std::string{"hello"});
auto a = t[CTValue<unsigned, 2>{}];
auto b = t[CTValue<unsigned, 3>{}];
```

变量 a 和 b 分别会被 Tuple t 中的第三个和第四个参数初始化成相应的类型和数值。

为了让常量索引的使用变得更方便，我们可以用 constexpr 实现一种字面常量运算符，专门用来直接从以_c 结尾的常规字面常量，计算出所需的编译期数值字面常量：

```
#include "ctvalue.hpp"
#include <cassert>
#include <cstddef>
// convert single char to corresponding int value at compile time:
constexpr int toInt(char c) {
  // hexadecimal letters:
```

```
  if (c >= 'A' && c <= 'F') {
  return static_cast<int>(c) - static_cast<int>('A') + 10;
  }
  if (c >= 'a' && c <= 'f') {
  return static_cast<int>(c) - static_cast<int>('a') + 10;
  }
  // other (disable '.' for floating-point literals):
  assert(c >= '0' && c <= '9');
  return static_cast<int>(c) - static_cast<int>('0');
}

// parse array of chars to corresponding int value at compile time:
template<std::size_t N>
constexpr int parseInt(char const (&arr)[N]) {
  int base = 10; // to handle base (default: decimal)
  int offset = 0; // to skip prefixes like 0x
  if (N > 2 && arr[0] == '0') {
      switch (arr[1]) {
          case 'x': //prefix 0x or 0X, so hexadecimal
          case 'X':
              base = 16;
              offset = 2;
          break;
          case 'b': //prefix 0b or 0B (since C++14), so binary
          case 'B':
              base = 2;offset = 2;
          break;
          default: //prefix 0, so octal
              base = 8;
              offset = 1;
          break;
      }
  }
  // iterate over all digits and compute resulting value:
  int value = 0;
  int multiplier = 1;
  for (std::size_t i = 0; i < N - offset; ++i) {
      if (arr[N-1-i] != '\'') { //ignore separating single quotes (e.g. in 1'
000)
          value += toInt(arr[N-1-i]) * multiplier;
          multiplier *= base;
      }
  }
  return value;
```

```
}

// literal operator: parse integral literals with suffix _c as sequence of chars:
template<char… cs>
constexpr auto operator"" _c() {
  return CTValue<int, parseInt<sizeof…(cs)>({cs…})>{};
}
```

此处我们用到了这样一个事实，对于数值字面常量，可以用字面常量运算符推导出该字面常量的每一个字符，并将其用作字面常量运算符模板的参数（参见第 15.5.1 节）。然后将这些字符传递给一个 constexpr 类型的辅助函数 parseInt()（它可以计算出字符串序列的值，并将其按照 CTValue 类型返回）。比如：

- 42_c 生成 CTValue<int,42>
- 0x815_c 生成 CTValue<int,2069>
- 0b1111'1111_c 生成 CTValue<int,255>

注意该程序不会处理浮点型字面值常量。对这种情况，相应的 assert 语句会触发编译期错误，因为这是一个运行期的特性，不能用在编译期上下文中。

基于以上内容，可以像下面这样使用元组：

```
auto t = makeTuple(0, '1', 2.2f, std::string{"hello"});
auto c = t[2_c];
auto d = t[3_c];
```

这一方式同样被 Boost.Hana 采用，它是一个适用于类型和数值计算的元编程库。

25.7 后记

元组的构造，是诸多由不同程序员独立尝试的模板应用中的一个。Boost.Tuple 库是 C++中最流行的一种元组的实现方式，并最终发展成 C++11 中的 std::tuple。

在 C++之前，很多元组的实现方式是基于递归的 pair 结构；在本书的第一版中通过其“recursive duos”展示了这样一种实现方式。另一种有趣的实现方式是由 Andrei Alexandrescu 在[AlexandrescuDesign]中开发的。他将元组中的类型列表和数据列表明确地分离开，并将 typelist（第 24 章）的概念用作元组的实现基础。

在 C++11 的实现中则使用了变参模板，这样可以通过参数包明确的为元组捕获类型列表，从而不需要使用递归的 pair。包展开以及索引列表概念的引入[GregorJarviPowellVariadicTemplates]，使得递归模板实例化变得简单且高效，从而也使元组变得更实用。索引列表对元组和类型列表的算法是如此的关键，以至于编译器为之包含了一个内置的模板别名__make_integer_seq<S,T,N>，它会在不需要额外的模板实例化的情况下展开成 S<T,0,1,..., N>，从而促进了 std::make_index_sequence 和 make_integer_sequence 的应用。

元组是最广泛使用的异质容器，但是并不是唯一一个。Boost.Fusion 提供了其它一些与常规容器对应的异质容器，比如异质的 list，deque，set 和 map。最重要的是，它还提供了一个与 C++标准库使用了相同的抽象和术语（比如 iterators，sequences 和 containers）的框架，使用该框架可以为异质集合编写算法。

Boost.Hana 采用了一些 Boost.MPL 和 Boost.Fusion 中的理念，它们都在 C++11 迈向成熟之前就已被设计和实现，并在之后被用 C++11（以及 C++14）的新特性重新实现。最终就产生出了一个简洁的、提供了强大的异质计算能力的库。