C++ 模板（第二版）

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感念的 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++7）
- 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：

```
//basics/max1.hpp
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 sts::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 ing
```

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）：

```
basics/maxauto.hpp
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：

```
basics/maxdecltype.hpp
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：

```
basics/maxdecltypedecay.hpp
#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：

```
basics/maxcommon.hpp
#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：

```
basics/maxdefault1.hpp
#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：

```
basics/maxdefault3.hpp
#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：

```
basics/max2.cpp
// 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：

```
basics/maxdefault4.hpptemplate<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：

```
basics/max3val.cpp
#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：

```
basics/max3ref.cpp
#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：

basics/max4.cpp

```
#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：

```
basics/maxconstexpr.hpp
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：

basics/stack1.hpp

```
#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 Stack<T>::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，它返回但是不删除首元素：

```
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();
}
```

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
        }
    }
};
```

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 {
    …
    void printOn() (std::ostream& strm) const {
    …
    }
  friend std::ostream& operator<< (std::ostream& strm, Stack<T> const& s) {
        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：

basics/stackpartspec.hpp

```
#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<>:

```
basics/stack3.hpp
#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：

basics/stack3test.cpp

```
#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     //
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）

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：

```
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）

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

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，除了对 std::string，不能用字符串常量去拷贝初始化一个别的对象。因此必须使用下面的方法初始化这个 Stack:

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

如果对此存疑，由于类模板参数推导和下面的情况类似（Note that, if in doubt, class template argument deduction copies），可以参考着理解。在声明了 stringStack 的类型为 Stack<std::string>之后，下面的初始化语句也声明了相同的类型，但是并不是通过字符串 stacks 的元素去初始化另一个 stack，而是通过其拷贝构造函数：

```
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：

basics/stacknontype.hpp

```
#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：

```
basics/stacknontype.cpp
#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：

```
basics/addvalue.hpp
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：

basics/stackauto.hpp

```
#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：

```
basics/stackauto.cpp
#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：

```
basics/message.cpp
#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：

```
basics/varprint1.hpp
#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：

basics/varprint2.hpp

```
#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)	(((pack1 op pack2) op pack3) … op packN)
(pack op …)	(pack1 op (… (packN-1 op packN)))
(init op … op pack)	(((init op pack1) op pack2) … op packN)
(pack op … op init)	(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：

```
basics/foldtraverse.cpp
// 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：

basics/addspace.hpp

Click here to view code image

```
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：

```
basics/varusing.cpp#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。