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。