Effective Modern C++

Scott Meyers

June 14, 2015

Contents

Chapter 1

类型推导

C++98 有一套单一的类型推导规则：函数模板，C++11 对此做了少量的修改，并加入了 2 种新的方法：一种是 `auto`，另一种是 `decltype`。C++14 则进一步扩展了 `auto` 和 `decltype` 能够使用的语境。类型推断的广泛应用将程序员从繁重的类型拼写工作中解放出来。在源码中某一处的修改能够自动的通过类型推导应用到其他的地方，提升了 C++ 开发的弹性。然而，类型推导也会使得程序更加难以理解，因为编译器做出的类型推导很有可能与你希望的方式不同。

因此，如果不深入的理解类型推导的原理，高效地使用现代 C++ 编程是不可能的。因为有太多的场景会用到它：调用函数模板，大多数 `auto` 出现的地方，`decltype` 表达式中和 C++14 中神秘的的 `decltype(auto)` 构造应用的地方，

本章涵盖了每个 C++ 开发者都应了解的类型推导知识，阐述了模板类型推导是如何工作、`auto` 在此基础上如何建立规则和 `decltype` 如何按照自己独立的规则工作。其中甚至描述了如何强制令编译器推导的类型结果可见，使你能够确保编译器推导的结果是你所希望的。

1.1 条款 1：理解模板类型推导

大多数复杂系统的用户都只关心系统所带来的功效，却并不关心它的工作原理。从这个层面来看，C++ 的模板推导是成功的。尽管大多数的程序员对类型推导的工作方式并不了解，但他们使用函数模版传递参数都获得了满意的结果。

如果你也是这个群体的一部分，我将告诉你一个好消息和一个坏消息。好消息是模板的类型推导是现代 C++ 中最引人注目功能之一：`auto` 的基石。如果你对于 C++98 中的 `auto` 很熟悉，那么 C++11 中的 `auto` 对你来说也会很熟悉。坏消息是当模板类型推断在有 `auto` 的上下文中使用时，相对于其在其他情况下（简单模板类型中）的应用变得更加令人疑惑了。因此，理解 `auto` 所依赖的模板类型推导的原理是非常重要的。本条款涵盖了该方面你所需要了解的所有知识。

如果你愿意忽略少量的伪代码，我们可以思考下面一个函数模板的例子：

```
template<typename T>
void f(ParamType param);
```

函数的调用形如：

```
f(expr); //使用表达式调用f函数
```

在编译过程中，编译器使用 *expr* 推导 2 个类型：一个是 `T` 另一个是 *ParamType*。这 2 个类型往往是不同的，因为 *ParamType* 通常都会包含修饰符，如 `const` 或者是引用限定符。例如，如果模板是这样声明的：

```
template<typename T>
void f(const T& param); //ParamType是const T&类型
```

并且我们这样调用：

```
int x = 0;

f(x); //调用f，实参是int类型
```

T 被推导为 int 类型，但 *ParamType* 被推导为 const int&。

人们通常希望 T 的类型推导和传入函数的参数类型是一致的，例如 T 的类型与 *expr* 是相同的。在上述的例子中，就是这种情况：x 是 int 类型，T 被推导为 int 类型。但并不总是这样，T 的类型推导不仅仅和 *expr* 的类型有关，也和 *ParamType* 的类型相关。下面有 3 个例子：

- *ParamType* 是一个指针或者引用，但不是一个通用引用 (Universal Reference)。（通用引用在条款 24 中描述，在这里，你只需要知道它存在，并且和左值引用、右值引用不同。）
- *ParamType* 是一个通用引用。
- *ParamType* 既不是指针，也不是引用。

接下来我们会介绍 3 种类型推导的情形，每一个调用都会以我们的通用模板为基础：

```
template<typename T>
void f(ParamType param);

f(expr); //从expr中推导T和ParamType的类型
```

情形 1：*ParamType* 是一个指针或者引用，但不是一个通用引用

在这种情况下，类型推导是这样工作的：

1. 如果 *expr* 的类型是一个引用，忽略引用的符号。

2. 然后模式匹配 expr 的类型来决定 ParamType 的类型从而决定 T 的类型

例如，如果这是我们的模板：

```
template<typename T>
void f(T& param); //param是一个引用
```

然后我们有如下变量声明：

```
int x = 27; //x是int类型
const int cx = x; //cx是const int类型
const int& rx = x; //rx是x的常量引用
```

则 param 和 T 的类型推导如下：

```
f(x); //T是int，param的类型是int &

f(cx); //T是const int，param的类型是const int&

f(rx); //T是const int，param的类型是const int &
```

在第二个和第三个调用中，注意由于 cx 和 rx 是常量类型，T 被推导为 const int，因此产生了 const int& 类型的形参。这对于调用者来说非常重要。当他们传递一个 const 对象给一个引用形参时，他们希望这个对象仍是不可修改的，例如：形参是对常量的引用。这也是为什么传递一个 const 对象给一个使用 T& 作为形参的模板是安全的：对象的常量性（constness）被推导为了 T 类型的一部分。

在第三个例子中，注意尽管 rx 的类型是一个引用，T 仍被推导为了一个非引用类型。这是因为 rx 的引用性（reference-ness）在类型推导的过程中被忽略了。

这些例子中使用的都是左值引用形参，类型推导同样以相同的方式能够作用于右值引用形参。当然，只有右值的实参可能会被传递给右值引用的形参，但是这对类型推断没有什么影响。

如果我们把 f 的形参由 T& 改为 const T&，就会发生一些变化，但也并不令人惊讶。cx 和 rx 的常量性依然被保留，但是因为我们已经假定 param 的是一个对常量的引用，所以 const 不再是 T 类型的一部分了。

```
template<typename T>
void f(const T& param); //param现在是一个常量引用

int x = 27;
const int cx = x;
const int &rx = x;

f(x); //T是int，param的类型是const T&

f(cx); //T是int，param的类型是const T&

f(rx); //T是int，param的类型是const T&
```

同样，rx 的引用性在类型推导的过程中被忽略。

如果 param 是一个指针（或是指向 const 的指针）而不是一个引用，规则并没有本质上改变：

```
template<typename T>
void f(T* param); //param现在是一个指针

int x = 27;
const int *px = &x; //px是一个将x视作const int的指针

f(&x); //T是int，param的类型是int *
f(px); //T是const int，param的类型是const int *
```

现在，你可能发现你自己在不断的打哈欠和点头，因为 C++ 的类型推导在引用和指针形参上工作得如此自然。每件事情都是如此清晰，与你希望的类型推导系统完全一样。

情形 2：*ParamType* 是一个通用引用

当使用了通用引用后，事情就变得不是那么清楚了。形参似乎被声明为一个一个右值引用（例如：函数模板中的形参被声明为 T&&），但是它们在传入一个左值实参后的表现却完全不同。完整的故事会在条款 24 中描述，这里有一个概要的版本。

- 如果 expr 是一个左值表达式，那么 T 和 ParamType 都被会推导为左值引用。这是非常不同寻常的。首先这是唯一一种情形：T 被推导为一个引用。第二，尽管 ParamType 被声明为一个右值引用，但是其推导的类型却是一个左值引用。
- 如果 expr 是一个右值表达式，则按照情形 1 的方式处理。

例如：

```
template<typename T>
void f(T&& param); //param现在是一个通用引用

int x = 27; //x是int类型
const int cx = x; //cx是const int类型
const int& rx = x; //rx是x的常量引用

f(x); //x是一个左值表达式，因此T是一个int&，param也是int&
f(px); //px是一个左值表达式，因此T是const int，param是const int &
```

```
f(rx); //rx是一个左值表达式，因此T是const int，param是const int &
f(27); //27是一个右值表达式，因此T是一个int，param是int &&
```

条款 24 清楚的解释了这些情形发生的原因。关键点在于通用引用的类型推导规则取决于形参是左值引用还是右值引用。当使用通用引用时，类型推导规则会区分左值实参和右值实参，而这从来都不会发生在非通用引用上。

(译者注：以上对左值引用、右值引用、通用引用不太明白的部分可以看 http://www.cnblogs.com/qicosmos/p/3369940.html 学习)

情形 3：*ParamType* 既不是指针，也不是引用

当 ParamType 既不是一个指针，也不是一个引用时，我们使用值传递的方式：

```
template<typename T>
void f(T param); //param通过值传递
```

这意味着 param 将会成为传递进来的实参的一份拷贝—一个全新的对象。事实上，param 成为一个全新的对象指导 T 如何从 expr 中推导。

1. 与之前一样，如果 expr 是一个引用，引用的部分会被忽略。2. 在忽略了 expr 的引用部分后，如果 expr 是一个 const，同样也会被忽略。如果它是 volatile，也会被忽略。（volatile 对象并不常用，它们通常仅用于设备驱动程序的开发。详细内容可见条款 40。）

因此：

```
int x = 27; //x是int类型
const int cx = x; //cx是const int类型
const int& rx = x; //rx是x的常量引用

f(x); //T和paramdoiu是int
f(cx); //T和param都是int
f(rx); //T和param都是int
```

注意尽管 cx 和 rx 都是静态变量，但 param 不是。这是讲得通的，因为 param 是一个完全独立于 cx 和 rx 的对象—它们之一的一份拷贝。事实上，cx 和 rx 不可被修改不会对 param 产生任何影响。这也是为什么说 expr 的静态性在类型推导时被忽略：因为 expr 不可被修改并不意味着它的拷贝也不可被修改。

意识到当使用值传递方式时，const（volatile）会被忽略是很重要的。我们看到，对于形参是静态量引用或指针，在类型推导时 expr 的静态性被保留。但是当 expr 是一个指向 const 对象的 const 指针，而且 expr 是通过值传递的方式时：

```
template<typename T>
void f(T param); //param仍通过值传递

const char* const ptr = "Fun with pointers"; //ptr是一个指向const对象
    的const指针

f(ptr); //实参类型是const char *const
```

在这里，最右边的 const 表明了指针是静态的：意味着 ptr 不能再次指向别的位置，亦或是空。（左边的 const 表明 ptr 所指向的字符串不可能被修改。）当 ptr 传递给 f 时，指针按位拷贝到 param 中。同样的，指针自身也是通过值传递。按照使用值传递方式的类型推导规则，ptr 的静态性将被忽略，param 的类型推导为 const char*，一个指向不可被修改的字符串的指针。在类型推导的过程中，ptr 所指向内容的静态性被保留，但是 ptr 自身的静态性在拷贝并创建新指针的过程中丢失了。

数组实参

上面的内容已经涵盖了模板类型推导的主流部分，但还有一些少见的情况值得我们了解。数组类型何指针类型是不同的，尽管一些时候它们是可以交换的。这个谬误主要来源于：在很多语境中，一个数组会退化成其中第一个元素的指针。这种退化允许下面的代码能够通过编译：

```
const char name[] = "J.P. Briggs"; //name的类型是const char[13]

const char *prtToName = name; //数组退化成指针
```

在这里，const char* 的指针 ptrToName 使用 name 初始化，这 2 个类型（const char * 和 const char[13]）是不同的，但是因为数组-指针退化规则，这段代码可以通过编译。

但当一个数组使用值传递方式传给一个模板呢？那会发生什么？

```
template<typename T>
void f(T param); //param通过值传递

f(name); //此时T和param会被推导为什么类型?
```

一开始我们观察到，函数没有使用数组作为一个形参的，但是下面的语法确实合法的：

```
void myFunc(int param[]);
```

但是在这里数组声明被视作一个指针的声明，一位置 myFunc 的声明实际上等价于：

```
void myFunc(int *param); //与上面相同的函数
```

这种数组与指针形参的等价从 C 中传递到了 C++，产生了数组和指针是相同的这种错觉。

因为数组形参声明被视作指针形参，当数组以值传递方式入一个函数模板时，其被推导为指针类型。这一位置当调用模板 f 时，它的类型形参 T 会被推导为 const char *：

```
f(name); //name是一个数组，但是T推导为const char*
```

这样现在就出现了一个诡计（curve ball）。尽管函数不能真正的声明其参数是数组类型的，但是它们能够声明参数是数组的引用！这样如果我们修改模板 f，令其通过引用接收实参，

```
template<typename T>
void f(T& param); //param通过引用传递
```

然后我们传入一个数组，

```
f(name); //传递数组给f
```

T 的实际类型被推导为一个数组！这种类型将会包括数组的大小，在这个例子中，T 的类型是 const char[13]，而且 f 的形参类型是 const char (&)[13]。是的，这个语法看起来是有毒的，但是知道这个会让你获得其他人所得不到的罕见分数（一些英文幽默）。

有趣的是，声明一个数组的引用能够创建一个能推导出数组元素个数的模板：

```
//在编译时期间返回数组的大小
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept
{
    return N;
}
```

如条款 15 所描述的，声明一个函数是 constexpr 的将使其结果在编译期间可用。这让这种用法变得可能：声明一个与之前创建的数组一样大的数组。

```
int keyVals[] = {1, 3, 7, 9, 11, 22, 35} //keyVals包括7个元素

int mappedVals[arraySize(coevals)]; //mappedVals也有7个元素。
```

当然，作为一个现代 C++ 的开发人员，你当然可以用 std::array 构造一个数组：

```
std::array<int, arraySize(keyVals)> mappedVals; //mappedVals的大小是7
```

至于被声明为 noexcept 的函数 arraySize，这将帮助编译器生成更好的代码，详见条款 14。

函数实参

数组类型不是 C++ 中唯一能被退化为指针的类型，函数类型也能够退化为指针类型，我们讨论的任何一个关于类型推导的规则和对数组相关的情形对于函数的类型推导也适用，函数类型会退化为函数的指针，因此：

```
void someFunc(int, double) //一个类型是void(int, double)的函数

template<typename T>
void f1(T param); //值传递方式

template<typename T>
void f2(T& param); //引用传递方式

f1(someFunc); //param被推导为一个函数指针，类型是void(*)(int, double)

f1(someFunc); //param被推导为一个函数引用，类型是void(&)(int, double)
```

在实际情况中很难有什么区别，但是当你了解到了数组-指针的退化时，你也同样应该了解到函数-指针的退化。

所以你现在明白了：auto 相关的模板类型推导规则。如我所说，在一开始它们大多都非常简单直接。唯一需要特殊处理的只有在使用通用引用时，左值表达式会有特别的处理。然而数组和函数的退化规则使得规则变得更加混乱。有时，你可能只是简单的抓住你的编译器，”告诉我，你推导出的类型是什么“，这时候，你可以看看条款 4，因为条款 4 就是讲述如何劝诱你的编译器这么做的。

请记住：

- 在模板类型推导时，对引用实参会被视作非引用类型的。
- 当推导通用引用的形参，左值实参需要特殊的处理。
- 当推导使用值传递方式的形参时，const 和 volatile 类型的实参会被忽略掉这两个属性。
- 在模板类型推导时，数组和函数实参中会退化为指针类型，除非它们被实例化为引用。

1.2 条款 2：理解 auto 类型推导

如果你已经阅读了条款 1，那么你几乎已经掌握了关于 auto 类型推导的全部知识，因为除了一个例外之外，auto 类型推导几乎就是模板类型推导。但是怎么会呢？模板类型推导包括了模板、函数和形参，但是 auto 并不处理它们中的任一个。

事实确实如此，但是也并没有关系。模板类型推导和 auto 类型推导之间存在一个映射。通过一种逐字逐句的算法进行互相转换。

在条款 1 中，模板类型推导使用了如下的函数模板：

```
template<typename T>
void f(ParamType param);
```

和如下的函数调用：

```
f(expr); //使用表达式调用f
```

在 f 的函数调用中，编译器使用 expr 去推导 T 和 ParamType 的类型。当一个变量使用 auto 声明时，auto 扮演了模板中 T 的角色，变量的类型说明符（specifier）则扮演了 ParamType 的角色。这里用以下的例子能够更好的描述这个情形：

```
auto x = 27;
```

这里 x 的类型说明符就是 auto 本身。从另一方面来讲，在这个声明中，

```
const auto cx = x;
```

的类型说明符是 const auto。并且在这里，

```
const auto& rx = x;
```

的类型描述符是 const auto&。为了推导例子中 x，cx 和 rx 的类型，编译器会认为每个声明都是一个模板，并且按照模板的方式来初始化表达式：

```
template<typename T>
void func_for_x(T param); //推导x类型的概念模板（conceptual template
    ）

func_for_x(27); //概念调用：param的推导类型是x的类型

template<typename T>
void func_for_x(const T param); //推导cx类型的概念模板

func_for_cx(x); //概念调用：param的推导类型是cx的类型

template<typename T>
void func_for_rx(const T& param); //推导rx类型的概念模板

func_for_rx(x); //概念调用：param的推导类型是rx的类型
```

如我所说，除了一个例外外，auto 的类型推导与模板的类型推导规则一样。

条款 1 基于 ParamType 的特点、函数模板中 param 的类型描述符，将模板类型的推导分为 3 种情形。在使用 auto 的变量推导中，类型描述符取代了 ParamType 的位置，因此这里同样也有 3 种情形：

- 情形 1：类型描述符是一个指针或引用，但不是一个通用引用。
- 情形 2：类型描述符是一个通用引用。
- 情形 3：类型描述符既不是一个指针，也不是一个引用。

我们已经看到了情形 1 和情形 3：

```
auto x = 27; //情形3，x不是指针，也不是引用。

const auto cx = x; //情形3，cx也不是。

const auto& rx = x; //情形1，rx是一个非通用引用。
```

情形 2 与你期望的一样：

```
auto&& uref1 = x; //x是一个int类型的左值表达式，所以uref1是int&

auto&& uref2 = cx; //cx是一个const int类型的左值表达式，所以uref2是
    const int &

auto&& uref3 = 27; //27是一个int类型的右值表达式，所以uref3是int&&
```

条款 1 总结了数组和函数如何退化为非引用类型指针的类型描述符。这种情形也在 auto 类型推断中发生了：

```
const char name[] = "R. N. Briggs"; //name的类型是const char[13]

auto arr1 = name; //arr1的类型是const char *

auto& arr2 = name; //arr2的类型是const char &[13]

void someFunc(int, double); //someFunc是一个void(int, double)类型的函
    数

auto func1 = someFunc; //func1的类型是void(*)(int, double)

auto& func2 = someFunc; //func2的类型是void(&)(int, double)
```

如你所见，auto 类型推导与模板类型推导非常相似。它们就像是硬币的两边。

除了一个例外。我们将会通过一个例子开始：你希望声明一个初值为 27 的 int 对象，C++98 给了 2 种方法：

```
int x1 = 27;
int x2(27);
```

通过 C++11 中的同意初始化（Uniform Initialzation），加入了这些方法：

```
int x3 = {27};
int x4{27};
```

总共使用了 4 种语法，获得了同样的结果：一个初值为 27 的 int 对象。

但如条款 5 中描述的那样，使用 auto 类型相较于使用固定类型声明变量有许多好处。因此使用 auto 替换上述例子中的 int 将会非常愉快的。简单的文本替换后，变成了如下代码：

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

这些声明都能够通过编译，但是它们却并不代表相同的含义。前两个声明确实使用 27 声明了 1 个 int 类型的变量。然而后两个却是声明了一个拥有一个元素 27 的 std::initializer_list<int> 类型的变量！

```
auto x1 = 27; //类型是int，值是27
auto x2(27); //同上
auto x3 = { 27 }; //类型是std::initializer_list<int>，值是{27}
auto x4{ 27 }; //同上
```

这种情形是源于 auto 的特殊类型推导规则。当一个 auto 声明的变量的初始化器（Initializer）放置在一对大括号中时，这个变量的推导类型是的 std::initializer_list。如果这个类型不能被推导（例如大括号中的值都是不同类型的），这段代码将会无法通过编译。

```
auto x5 = {1, 2, 3.0}; //错误！不能为std::initializer_list<T>推导T
```

如注释所指出的那样，在这种情形中，类型推导失败了。但是理解在这个地方实际上发生了两种类型的类型推导是很重要的。一种源于 auto 的使用：x5 的类型必须被推导。因为 x5 的初始化器在大括号中，所以 x5 必须被推断为 std::initializer_list。第二种则是，因为 std::initializer_list 是一个模板，std::initializer_list<T> 的为某种类型的 T 实例化，这也意味着 T 的类型必须被推导。上述的类型推导是因为第二种：模板类型推导而失败的，因为大括号中的初始化器中的元素有多种类型。

对于大括号初始化器的不同的处理方式是 auto 类型推导和模板类型推导唯一不同的地方。当 auto 声明的变量被大括号初始化器初始化时，推导出的类型是 std::initializer_list。但是如果相应的模板被传入一个相同的初始化器时，类型推导会失败，代码无法通过编译。

```
auto x = {11, 23, 9} //x的类型是std::initializer_list<int>

template<typename T>
void f(T param); //与x声明等价的模板形参声明

f({11, 23, 9}); //错误，无法为T推导类型
```

然而，当你指定形参的类型是 std::initializer_list<T> 时，模板类型推导规则就会成功推导出 T 的类型：

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

f({11, 23, 9});
//T被推导为int,initList的类型是std::initializer_list<int>
```

所以 auto 和模板类型推导的唯一差别就是：auto 假定大括号初始化器代表着 std::initializer_list，而模板类型则不同。

你可能希望知道为什么 auto 类型推导对于大括号初始化器使用了特别的规则，但是模板类型推导没有。我也想知道，但是不幸的是，我没有找到一个方便的解释。因为规则就是规则，这意味着你在使用 auto 声明一个变量，并使用一个大括号初始化器时必须记住：推导出的类型一定是 std::initializer_list。如果你想更深入的使用统一的集合初始化时，你就更要牢记这一点。C++11 中一个最经典的错误就是程序员意外的声明了一个 std::initializer_list 类型的变量，但他们的本意却是想声明一个其他类型的变量。错误造成的主要原因是一些程序员只有当必要的时候，才使用大括号初始化器进行初始化。（将会在条款 7 中详细讨论。）

对于 C++11 来说，这已经是一个完整的故事了，但是对于 C++14，故事还没有结束，C++14 允许 auto 来表示一个函数的返回值的类型（见条款 3），并且 C++14 的 lambda 表达式可以在参数的声明时使用 auto。不管怎样，这些 auto 的使用，采用的都是模板类型推导的规则，而不是 auto 类型推导规则，这意味着，大括号的初始化式会造成类型推导的失败，所以一个带有 auto 返回类型的函数如果返回一个大括号的初始化式将不会通过编译。

```
auto createInitList()
{
    return {1, 2, 3}; //错误，不能够推导类型f{1, 2, 3}
}
```

同样，规则也适用于当 auto 用于 C++14 的 lambda 的参数类型说明符时：

```
std::vector<int> v;
...

auto resetV =
    [&V](const auto& newValue) { v = newValue; }; //C++14
...
```

```
resetV( {1, 2, 3} ); //错误，不能够推导类型f{1, 2, 3}
```

请记住：

- auto 类型推导规则通常与模板类型推导相同，但是 auto 类型推导假定 1 个大括号初始化器代表着 std::initializer_list，而模板类型推导不然。
- 当 auto 是一个函数的返回值类型或是一个 lambda 传递形参类型时，使用模板类型推导规则，而不是 auto 类型推导规则。