CHAPTER 1 Deducing Types

C++98有一套用于模板类型推导的规则，C++11修改了其中的一些规则并为**auto**和**decltype**添加了新的规则。类型推导的广泛应用让我们不必再输入那些明显多余的类型，它让C++程序更具适应性，因为在源代码某处修改类型会通过类型推导自动传播到其它地方。但是类型推导也会让代码更复杂，因为由编译器进行的类型推导并不总是如我们期望的那样进行。

如果对于类型推导操作没有一个扎实的理解，要想写出有现代感的C++程序是不可能的。类型推导随处可见：在函数模板调用中，在**auto**出现的地方，在**decltype**表达式出现的地方，以及C++14的**decltype(auto)** 中。

这一章的内容是每个C++程序员都应该掌握的知识。它解释了模板类型推导是如何工作的，**auto**是如何依赖模板类型推导的，以及**decltype**是如何按照它自己那套独特的规则工作的。它甚至解释了你该如何强制编译器产生他进行类型推导的结果，这能让你确认编译器的类型推导是否按照你期望的那样进行。

Item1 :Understand template type deduction

条款一:理解模板类型推导

对于一个复杂系统的用户来说很多时候他们最关心的是它做了什么而不是它怎么做的。在这一点上C++中的模板类型推导表现得非常出色。数百万的程序员只需要向模板函数传递实参就能通过编译器的类型推导获得令人满意的结果，尽管他们中的大多数对于传递给函数的那些实参是如何引导编译器进行类型推导的只能给出非常模糊的描述，而且还是在被逼无奈的情况。

如果那些人中包括你，我有一个好消息和一个坏消息。好消息是现在C++最重要最吸引人的特性**auto**是建立在模板类型推导的基础上的，如果你熟悉C++98的模板类型推导，那么你不必害怕C++11的**auto**。坏消息是虽然**auto**是建立在模板类型推导的基础上，但是在某些情况下**auto**不如模板类型推导那么直观容易理解。这个条款便包含了你需要知道的关于模板类型推导的全部内容：

如果你不介意浏览少许伪代码，考虑这样一个函数模板：

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

它的调用看起来像这样

```
f(expr);    //使用表达式调用f
```

在编译期间，编译器使用expr进行两个类型推导：一个是针对T的，另一个是针对ParamType的。这两个类型通常是不同的，因为ParamType包括了const和引用的修饰。举个例子，如果模板这样声明：

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

然后这样进行调用

```
int x = 0;
f(x);   //用一个int类型的变量调用f
```

T被推导为**int**，ParamType却被推导为**const int&**

我们可能很自然的期望T和传递进函数的参数是相同的类型，在上面的例子中，事实就是那样，**x**是**int**，T是**expr**的类型即**int**。但有时情况并非总是如此，T的推导不仅取决于**expr**的类型，也取决于**ParamType**的类型。这里有三种情况：

- ParamType是一个指针或引用，但不是通用引用（关于通用引用请参见Item24。在这里你只需要知道它存在，而且不同于左值引用和右值引用）
- ParamType一个通用引用
- ParamType既不是指针也不是引用

我们下面将分成三个情景来讨论这三种情况，每个情景的都基于我们之前给出的模板：

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

f(expr);    //从expr中推导T和ParamType
```

情景一：ParamType是一个指针或引用但不是通用引用

最简单的情况是**ParamType**是一个指针或者引用但非通用引用，也就是我们这个情景讨论的内容。在这种情况下，类型推导会这样进行：

1. 如果expr的类型是一个引用，忽略引用部分
2. 然后剩下的部分决定T，然后T与形参匹配得出最终ParamType

举个例子，如果这是我们的模板

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

我们声明这些变量：

```
int x=27;       //x是int
const int cx=x;     //cx是const int
const int & rx=cx;  //rx是指向const int的引用
```

当把这些变量传递给f时类型推导会这样

```
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，注意因为cx和rx被指定为**const**值，所以T被推导为**const int**，从而产生了**const int&**类型的**param**。这对于调用者来说很重要，当他们传递一个**const**对象给一个引用类型的参数时，他们传递的对象保留了常量性。这也是为什么向**T&**类型的参数传递**const**对象是安全的：对象T的常量性会被保留为T的一部分。

在第三个例子中，注意即使rx的类型是一个引用，T也会被推导为一个非引用，这是因为如上面提到的如果**expr**的类型是一个引用，将忽略引用部分。

这些例子只展示了左值引用，但是类型推导会如左值引用一样对待右值引用。通常，右值只能传递给右值引用，但是在模板类型推导中这种限制将不复存在。

情景二：ParamType一个通用引用

如果**ParamType**是一个通用引用那事情就比情景一更复杂了。如果**ParamType**被声明为通用引用（在函数模板中假设有一个模板参数**T**,那么通用引用就是**T&&**),它们的行为和**T&**大不相同，完整的叙述请参见Item24，在这有些最必要的你还是需要知道：

- 如果**expr**是左值，T和ParamType都会被推导为左值引用。这非常不寻常，第一，这是模板类型推导中唯一一种T和ParamType都被推导为引用的情况。第二，虽然ParamType被声明为右值引用类型，但是最后推导的结果它是左值引用。
- 如果**expr**是右值，就使用**情景**一的推导规则

举个例子：

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

int x=27;                //如之前一样
const int cx=x;          //如之前一样
const int & rx=cx;       //如之前一样

f(x);                //x是左值，所以T是int&
                     //param类型也是int&

f(cx);               //cx是左值，所以T是const int &
                     //param类型也是const int&

f(rx);               //rx是左值，所以T是const int &
                     //param类型也是const int&

f(27);               //27是右值，所以T是int
                     //param类型就是int&&
```

Item24详细解释了为什么这些例子要这样做。这里关键在于类型推导对于通用引用是不同于普通的左值或者右值引用。比如，当通用引用被使用时，类型推导会区分左值实参和右值实参，但是**情况**一就不会。

情景三：ParamType既不是指针也不是引用

当**ParamType**既不是指针也不是引用时，我们通过传值（pass-by-value）的方式处理：

```
template<typename T>
void f(T param);             //以传值的方式处理param
```

这意味着无论传递什么param都会成为它的一份拷贝——一个完整的新对象。事实上param成为一个新对象这一行为会影响T如何从expr中推导出结果。

1. 和之前一样，如果expr的类型是一个引用，忽略这个引用部分
2. 如果忽略引用之后expr是一个const，那就再忽略const。如果它是volatile，也会被忽略（volatile不常见，它通常用于驱动程序的开发中。关于volatile的细节请参见Item40)

因此

```
int x=27;               //如之前一样
const int cx=x;         //如之前一样
const int & rx=cx;      //如之前一样

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

注意即使**cx**和**rx**表示const值，param也不是**const**。这是有意义的。param是一个拷贝自**cx**和**rx**且现在独立的完整对象。具有常量性的**cx**和**rx**不可修改并不代表param也是一样。这就是为什么expr的常量性或易变性（volatileness)在类型推导时会被忽略：因为**expr**不可修改并不意味着他的拷贝也不能被修改。

认识到只有在传值给形参时才会忽略常量性和易变性这一点很重要，正如我们看到的，对于形参来说指向const的指针或者指向const的引用在类型推导时const都会被保留。但是考虑这样的情况，expr是一个const指针，指向const对象，expr通过传值传递给**param**：

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

const char* const ptr = //ptr是一个常量指针，指向常量对象
" Fun with pointers";
```

在这里，解引用符号（*）的右边的const表示ptr本身是一个const：ptr不能被修改为指向其它地址，也不能被设置为null（解引用符号左边的const表示ptr指向一个字符串，这个字符串是const，因此字符串不能被修改）。当ptr作为实参传给f，像这种情况，ptr自身会传值给形参，根据类型推导的第三条规则，**ptr**自身的常量性将会被省略，所以param是**const char*** 。也就是说一个常量指针指向const字符串，在类型推导中这个指针指向的数据的常量性将会被保留，但是指针自身的常量性将会被忽略。

数组实参

上面的内容几乎覆盖了模板类型推导的大部分内容，但这里还有一些小细节值得注意，比如在模板类型推导中指针不同于数组，虽然它们两个有时候是完全等价的。关于这个等价最常见的例子是在很多上下文中数组会退化为指向它的第一个元素的指针，比如下面就是允许的做法：

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

const char * ptrToName = name;      //数组退化为指针
```

在这里**const char*** 指针**ptrToName**会由name初始化，而name的类型为**const char[13]**，这两种类型(**const char *** 和**const char[13]**)是不一样的，但是由于数组退化为指针的规则，编译器允许这样的代码。

但要是一个数组传值给一个模板会怎样？会发生什么？

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

f(name);        //对于T和param会产生什么样的类型
```

我们从一个简单的例子开始，这里有一个函数的形参是数组，是的，这样的语法是合法的：

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

但是数组声明会被视作指针声明，这意味着myFunc的声明和下面声明是等价的：

```
void myFunc(int *param);    //同上
```

这样的等价是C语言的产物，C++又是建立在C语言的基础上，它让人产生了一种数组和指针是等价的的错觉。

因为数组形参会视作指针形参，所以传递给模板的一个数组类型会被推导为一个指针类型。这意味着在模板函数f的调用中，它的模板类型参数T会被推导为**const char***：

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

但是现在难题来了，虽然函数不能接受真正的数组，但是可以接受指向数组的引用！所以我们修改f为传引用：

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

我们这样进行调用

```
f(name);    //传数组
```

T被推导为了真正的数组！这个类型包括了数组的大小，在这个例子中T被推导为**const char[13]**，param则被推导为**const char(&)[13]**。是的，这种语法看起来简直有毒，但是知道它将会让你在关心这些问题的人的提问中获得大神的称号。

有趣的是，对模板函数声明为一个指向数组的引用使得我们可以在模板函数中推导出数组的大小：

```
//在编译期间返回一个数组大小的常量值（
//数组形参没有名字，因为我们只关心数组
//的大小）
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept
{
    return N;
}
```

在Item15提到将一个函数声明为constexpr使得结果在编译期间可用。这使得我们可以用一个花括号声明一个数组，然后第二个数组可以使用第一个数组的大小作为它的大小，就像这样：

```
int keyVals[] = {1,3,5,7,9,11,22,25};   //keyVals有七个元素

int mappedVals[arraySize(keyVals)]; //mappedVals也有七个
```

当然作为一个现代C++程序员，你自然应该想到使用**std::array**而不是内置的数组：

```
std::array<int,arraySize(keyVals)> mappedVals;      //mappedVals的size为7
```

至于**arraySize**被声明为**noexcept**，会使得编译器生成更好的代码，具体的细节请参见Item14。

函数实参

在C++中不止是数组会退化为指针，函数类型也会退化为一个函数指针，我们对于数组的全部讨论都可以应用到函数来：

```
void someFunc(int, double); //someFunc是一个函数，类型是void(int,double)

template<typename T>
void f1(T param);        //传值

template<typename T>
void f2(T & param);      //传引用

f1(someFunc);            //param被推导为指向函数的指针，类型是void(*)(int, double)
f2(someFunc);            //param被推导为指向函数的引用，类型为void(&)(int, bouel)
```

这个实际上没有什么不同，但是如果你知道数组退化为指针，你也会知道函数退化为指针。

这里你需要知道：**auto**依赖于模板类型推导，正如我在开始谈论的，在大多数情况下它们的行为很直接。在通用引用中对于左值的特殊处理使得本来很直接的行为变得有些污点，然而，数组和函数退化为指针把这团水搅得更浑浊。有时你只需要编译器告诉你推导出的类型是什么。这种情况下，翻到**item4**，它会告诉你如何让编译器这么做。

记住：

- 在模板类型推导时，有引用的实参会被视为无引用，他们的引用会被忽略
- 对于通用引用的推导，左值实参会被特殊对待
- 对于传值类型推导，实参如果具有常量性和易变性会被忽略
- 在模板类型推导时，数组或者函数实参会退化为指针，除非它们被用于初始化引用