# Effective Modern C++

Scott Meyers

June 23, 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。

## 1.3 条款 3：理解 decltype

decltype 是一个古怪的东西。给定一个名称或者表达式，decltype 能告诉你它们的类型。通常用来告诉你它们的类型是不是你想要的。然而有的时候，它也会让你百思不得其解，转而向在线的 Q%A 网站求助。

我们将从典型的案例开始，它们的结果通常在你的意料之中。与模板类型推导和 auto 类型推导不同，decltype 会返回你给出的名称和表达式准确的类型：

```
const int i = 0; //decltype(i)返回const int

bool f(const Widget& w); //decltype(w)返回const Widget&
                         //decltype(f)返回bool(const Widget &)

struct Point {
    int x, y;
};
//decltype(Point::x)返回int
//decltype(Point::y)返回int

Widget w; //decltype(w)返回Widget

if (f(w)) ... //decltype(f(w))返回bool

template<typename T>          // std::vector的简单版本
class vector {
public:
    ...
    T& operator[](std::size_t index);
    ...
};

vector<int> v;                // decltype(v)返回vector<int>
...
if (v[0] == 0) ...            // decltype(v[i])返回int&
```

看到了吗？并没有什么令人惊讶的。

在 C++11 中，可能 decltype 的主要用处是声明函数模板，当其的返回类型取决于参数类型时。举个例子，假定我们要写一个函数，它的参数是一个支持 [ ] 下标访问的容器，函数首先对使用者进行严验证，然后返回下标操作的结果。函数返回值的类型应该与下标操作返回值的类型相同。

对一个对象类型为 T 的容器使用 [ ] 运算符应当返回一个 T& 类型的对象，std::deque 就是这样。std::vector 几乎也是这样，但只有一个例外，对于 std::vector<bool>，[ ] 运算符并不返回一个 bool& 类型的对象，而是返回一个全新的对象，条款 6 会解释这样的原因。但是重要的是，作用在容器上的 [ ] 运算符的返回类型取决于这个容器本身。

decltype 让这件事变得简单。下面是我们写的第一个版本，显示了使用 decltype 推导返回类型的方法，这个模板还可以更精简一些，但是我们先暂时不考虑这个：

```
template<typename Container, typename Index> //可以工作
auto authAndAccess(Container& c, Index i)    //但是能再精简一些
    -> decltype(c[i])
{
    authenticateUser();
    return c[i];
}
```

在函数名前使用 auto 不会进行任何的类型推导，它暗示了 C++11 的返回类型追踪（trailing return type）语意正在使用。例如：函数的返回类型将在参数列表后声明（在-> 后面）。追踪返回类型的好处是函数的参数能够在能在返回类型的声明中使用。例如在 authAndAccess 中，我们使用 c 和 i 表明函数的返回类型。如果我们想要将返回类型声明在函数名的前面，但是此时 c 和 i 是不可用的，因为它们此时还没有声明。

使用例子中的声明方法，authAndAccess 能够返回 [ ] 运算符所返回的类型，如我们想要的一样。

C++11 允许推导单一 lambda 语句的返回类型，C++14 扩展了这个功能，使得所有的 lambda 和函数表达式都能够使用，包括含有多条语句的函数。这意味着，在 C++14 中，我们可以不需要返回类型追踪，只需要使用一个 auto。在这种形式的声明中，auto 确实代表着这里应当表达的类型。这意味着编译器将依据函数的内容来推导函数的返回值类型。

```
template<typename Container, typename Index> //c++14支持
auto authAndAccess(Container& c, Index i)    //但不是十分正确
{
    authenticateUser();
    return c[i]; //由c[i]推断返回类型
}
```

条款 2 描述了 auto 如何推导函数的返回值类型：编译器使用模板类型推导的规则。在这个例子中，就出现了问题。如我们前面所讨论的，[ ] 运算符为大多数 T 类型的容器返回一个 T 的引用，但是条款 1 中又说了：在模板类型推导的过程中，引用性在表达式初始化过程中会被忽略。思考一下下面的代码：

```
std::deque<int> d;
...
authAndAccess(d, 5) = 10; //函数返回d[5]
                          //并为其赋值10
                          //但是不会通过编译!
```

在这里，d[5] 应当返回一个 int&，但是 auto 推导的返回类型会忽略掉引用，因此这里的返回值类型是 int。这里的 int 作为一个函数的返回值，是一个右值表达式，而上面的代码尝试将 10 赋值给一个右值表达式。这在 C++ 中是禁止的，所以这段代码不能通过编译。

为了让 authAndAccess 能像我们想要的方式工作，我们需要使用 decltype 为返回值作类型推导，例如：令 authAndAccess 的返回值类型正好是 c[i] 表达式所返回的。C++ 标准的制定者预料到了在某种情况下，类型推导需要使用 decltype。所以在 C++14 中出现了 decltype(auto) 说明符。刚遇到这种情况时，似乎有一些矛盾。但事实上这是合情合理的，auto 指明了了类型需要被推导，而 decltype 则指示了在推导中所需要使用的规则。因此我们可以这样改写 autoAndAccess：

```
template<typename Container, typename Index> //c++14支持
decltype(auto)                               //但还能够改进
authAndAccess(Container& c, Index i)
{
    authenticateUser();
    return c[i];
}
```

现在 authAndAccess 的返回值类型将会和 c[i] 的的返回值完全一致。当 c[i] 返回一个 T& 时，autoAndAccess 也会返回一个 T&，而当 c[i] 返回一个对象时，antuAndAccess 也会返回一个对象。

decltype(auto) 的用途并不限于函数的返回值类型。当你想要使用 decltype 类型推导初始化表达式时，它们也能很方便声明变量：

```
Widget w;
const Widget& cw = w;
auto myWidget1 = cw;                // auto推导出的:
                                    // myWidget1类型是Widget
decltype(auto) myWidget2 = cw;      // decltype推导出的:
                                    // myWidget2类型是
                                    // const Widget&
```

但是我知道这里有两个问题正困扰着你。一个是我之前提到的 authAndAccess 的改进，让我们现在来解决这个问题。

再看一下 C++14 版的 authAndAccess 的声明：

```
template<typename Container, typename Index>
decltype(auto) anthAndAccess(Container& c, Index i);
```

其中容器形参是通过非 const 的左值引用传递，因为对一个容器的引用允许我们修改容器其中的元素。但这也意味着不能够向这个函数传递一个右值容器。右值不能够绑定在一个左值引用上（除非是一个 const 的左值引用，但本例中不是这样的）。

诚然，向 authAndAccess 传递一个右值容器是一个特殊情况。一个右值引用一般是一个临时对象，会在调用 authAndAccess 的函数的语句后摧毁，这也意味着对该容器的某一个元素的引用将会在调用语句的结束时（一般是 authAndAccess 返回时）悬空。但是，向 anthAndAccess 传递一个临时变量仍然是有意义的。一个客户可能只是想要拷贝这个临时变量中的一个元素：

```
std::deque<std::string> makeStringDeque(); // 工厂函数
                                           //从makeStringDeque的函数
                                               值中拷贝
                                           //容器的第五个元素
auto s = authAndAccess(makeStringDeque(), 5);
```

支持这种用法意味着我们需要修改 authAndAccess 的声明，让其既可以接受左值也可以接受右值。这里可以使用重载（一个重载函数声明一个左值引用形参，另一个重载函数声明一个右值引用形参），但是这样我们就要维护两个函数。一种避免这种情况的方法是令 authAndAccess 使用一个能绑定左值和右值的引用形参，条款 24 中阐述了这也正好是通用引用所能做的。因此 authAndAccess 能像这样声明：

```
template<typename Container, typename Index>
decltype(auto) anthAndAccess(Container&& c, Index i); //c是一个通用引
    用
```

在这个模板里，我们并不知道操作的容器类型，这也意味着我们一样不知道下标所对应对象的类型。对一个为止类型的对象使用传值方法往往会因为不必要的拷贝开销而影响性能，对象分割问题（见条款 17）和来自同事的嘲笑。但是根据标准库中的例子 (例如 std::string, std::vector 和 std::deque)，这种情况下看起来也是合理的，所以我们坚持按值传递。

然而，我们需要更新模板的实现方式，依据条款 25 的警告，将 std::forward 应用到通用引用上：

```
template<typename Container, typename Index>
decltype(auto)
anthAndAccess(Container&& c, Index i); //C++14的最终版本
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

这样就满足了我们所需要的所有要求，但是这段代码需要 C++14 的编译器。如果你还没有的话，你需要将其改成 C++11 的版本。这和 C++14 版本相似，除了你需要自己标注出返回的类型。

```
template<typename Container, typename Index> //C++11的
auto                                         // 的最终
authAndAccess(Container&& c, Index i)        // 版本
-> decltype(std::forward< Container>(c)[i])
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

另一个值得对你唠叨的问题我已经标注在了这一条款的开始处了，decltype 获得的结果几乎和你期待的一样，这并不奇怪。老实说，你几乎不太可能遇到这个规则的例外情况，除非你需要实现一个任务非常繁重的代码库。

为了完全理解 decltype 的行为，你需要让你自己熟悉一些特殊的情况，大多数在这本书里证明讨论起来会非常的晦涩，但是其中一条能让我们更加理解 decltype 的使用。

对一个变量名使用 decltype 产生声明这个变量时的类型。有名字的是左值表达式，但这并不影响 decltype 的行为。因为对于比变量名更复杂的左值表达式，decltype 确保推导出的类型总是一个左值的引用，这意味着如果一个左值表达式不同于变量名的类型 T，decltype 推导出的类型将会是 T&，这几乎不会照成什么影响，因为大多数左值表达式的类型内部通常包含了一个左值引用的限定符，例如，返回左值的函数总是返回一个左值引用。

这里有一个值得注意的地方：

```
int x = 0;
```

x 是一个变量的名字，所以 decltype(x) 的结果是 int，但是将名字 x 用括号包裹起来，”(x)”产生了一个比名字更复杂的表达式，作为一个变量名，x 是一个左值，C++ 同时定义了 (x) 也是一个左值，因此 decltype((x)) 结果是 int&，将一个变量用括号包裹起来改变了 decltype 最初的结果！

在 C++11 中，这仅仅会会让人有些奇怪，但是结合 C++14 中对 decltype(auto) 的支持后，你对返回语句的一些简单的修改会影响到函数最终推导出的结果。

```
decltype( auto) f1()
{
    int x = 0;
    ...
    return x;    // decltype(x) 是 int, 所以f1返回int
}
decltype(auto) f2()
{
    int x = 0;
    ...
    return (x); // decltype((x)) 是int&, 所以f2返回int&
}
```

注意到 f2 和 f1 不仅仅是返回类型上的不同，f2 返回的是一个局部变量的引用！这种代码的结果是未定义的，你当然不希望发生这种情况。

你需要记住的是当你使用 decltype(auto) 的时候，需要格外注意。一些看起来无关紧要的细节会影响到 decltype(auto) 推导出的结果，为了确保被推导出的类型是你期待的，可以使用条款 4 中描述的技术。

但同时不要失去对大局的注意，decltype(无论是独立使用还是和 auto 一起使用）推导的结果可能偶尔让人惊讶，但是这并不会经常发生。通常，decltype 的结果和你所期待的类型一样，尤其是当 decltype 应用在变量名的时候，因为在这种情况下，decltype 做的就是提供变量的声明类型。

请记住：

- decltype 几乎总是返回变量名或是表达式的类型而不会进行任何的修改。
- 对于不同于变量名的左值表达式，decltype 的结果总是 T&。
- C++14 提供了 decltype(auto) 的支持，比如 auto，从它的初始化式中推导类型，但使用 decltype 的推导规则。

## 1.4 条款 4：掌握如何查看推导类型

如何选择合适的、用于查看类型推导结果的工具，取决于你在软件开发过程中所处于的阶段。我们将探讨其中的三种可能性：第一种是在编辑代码时获取类型推导信息，第二种是编译代码时获取，最后一种是运行时过程中获取。

### 1.4.1 IDE 编辑器

当你将你的光标放置在实体的附件时，IDE 中的编辑器往往能够标注出代码中实体的类型（例如：变量，形参，函数等）。以下面的代码为例；

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAndwer;
```

一个 IDE 的编辑器将会显示 x 的推导类型是 int，而 y 的推导类型是 const int*。

此时此刻，你的代码实际上也处于某种已编译的状态，因为这样 IDE 才能够通过编译器获取这些信息。如果编译器不能够在此时分析得到足够的类型推导信息，那么 IDE 也不能够告诉你类型推导的结果。

对于像 int 这样的简单类型，IDE 通常能够获得正确的结果。然而，我们很快就会发现，当使用了复杂类型时，IDE 所显示的结果可能就不会有什么作用了。

### 1.4.2 编译器诊断

利用某种导致编译错误的手段，是一个高效的使用编译器获取类型推导结果的方法。问题的错误提示信息实际上就会暗示是什么类型所导致的。

举个例子，我们想要推导前面例子中 x 和 y 的类型。首先声明一个我们并没有定义的模板类，就像这样：

```
template<typename T> //仅为TD声明
class TD;                    //TD的意思是类型显示器
```

当尝试实例化这个模板时，编译器会引发一个错误。因为现在并没有用于实例化的模板定义。为了查看 x 和 y 的类型，于是使用它们的类型实例化 TD：

```
TD<decltype(x)> xType; //显示的错误将会包括x和y的类型
TD<decltype(y)> yType;
```

我使用了形如 vaiableNameType 这样的变量名，因为这样所引出的错误消息将引导我找到想要获取的的信息。对于上面的代码，我的一种编译器生成了如下的诊断信息：

```
error: aggregate 'TD<int> xType' has incomplete type and cannot be
    defined
error: aggregate 'TD<const int *> yType' has incomplete type and
    cannot be defined
```

另一种编译器使用另一种形式显示了同样的信息:

```
error: 'xType' use undefined class 'TD<int>'
error: 'yType' use undefined class 'TD<const␣int␣*>'
```

仅仅只是不同的格式。通过这种技术，我所测试的所有编译器都产生我所需要的类型信息。

### 1.4.3 运行时输出

# Chapter 2

# auto

接下来的这一章深入浅出的探讨了 auto。

## 2.1 条款 6：当 auto 无法推导出正确的类型时，使用明确的类型初始化句式

条款 5 解释了使用 auto 声明变量类型相比明确声明变量类型所带来的一些技术优势，但有些时候 auto 的类型推导并不总是如你所愿。举个例子，假设我有一个函数接收 Widget 作为参数然后返回 std::vector<bool>，其中每一个 bool 表示 Widget 是否拥有某种特性：

```
std::vector<bool> features(const Widget& w);
```

进一步假设容器中的第 5 个元素表示 Widget 是否拥有高优先级。因此我们可以写出这样的代码：

```
Widget w;
...

bool highPriority = features(w)[5]; // 是否拥有高优先级?
...

processWidget(w, highPriority);      // 根据优先级级别来处理Widget w
```

这段代码并没有任何错误，可以正常工作。但如果我们做出一个看似无害的修改，把 highPriority 的类型声明从 bool 改成 auto：

```
auto highPriority = features(w)[5]; // 是否拥有高优先级?
```

那么情况就完全变了。所有的代码能够继续通过编译，但是它的行为将变得不可预测：

```
processWidget(w, highPriority);      // 未定义行为!
```

正如注释指出的那样，现在调用 processWidget 将造成未定义行为。为什么会这样?答案可能会让你有些惊讶：在使用 auto 的代码中，highPriority 的类型不再是 bool 了。尽管 std::vector<bool> 在概念上拥有 bool 变量，但是 std::vector<bool> 的 operator[] 并不返回这个容器中元素的引用（除了 bool，其他的 std::vector::operator[] 都返回元素的引用）。取而代之的是返回一个 std::vector<bool>::reference 对象。（它是 std::vector<bool> 的嵌套类）

之所以存在 std::vector<bool>::reference，是因为 std::vector<bool> 需要指定它作为 bool 的打包形式。如果改用一个 bit 表示 bool，std::vector<bool> 的 operator[] 将存在一个明显的问题：对于任意 std::vector<T>，operator[] 理应返回一个 T&，但 C++ 并不支持对 bit 的引用。既然不能返回一个 bool&，于是 std::vector<bool> 的 operator[] 便返回一个看起来像 bool 的对

象。为了实现这个目的，std::vector<bool>::reference 必须能够支持 bool& 能够出现的每个上下文。因此在 std::vector<bool>::reference 的特性中，包含了对 bool 的隐式转换来完成这项工作。(是 bool，而不是 bool&。如果要解释 std::vec tor<bool>::reference 模拟 bool& 的行为用到的所有技术将超出了本书的范围，所以我只是简单的指出隐式转换是是其中的一部分)

记住这个特性后，我们再来看看原始代码的这部分：

```
bool highPriority = features(w)[5]; // 明确定义了highPriority的类型
```

在这里，函数 features 返回了一个 std::vector<bool> 对象，然后再对它调用 operator[]。而 operator[] 返回的 std::vector<bool>::reference 通过隐式转换为 bool 来初始化 highPriority。所以正如我们的预期，highPriority 存储了 features 返回的 std::vector<bool> 中的第五个元素。

与之相反，如果使用 auto 来声明 highPriority：

```
auto highPriority = features(w)[5]; // 编译器推导highPriority的类型
```

同样的，函数 features 将返回一个 std::vector<bool> 对象，然后再对它调用 operator[]。但发生变化的是，highPriority 将会被编译器推导成一个 std::vector<bool>::reference 对象，因此 highPriority 的值根本就不是 std::vector<bool> 中的第五个元素。

highPriority 的值取决于 std::vector<bool>::reference 的具体实现。一种实现方式是包括一个指向机器字的指针，加上一个偏移位。结合机器字和偏移位进行偏移就能得到判别值。如果 std::vector<bool>::reference 正如这样实现，那么我们来考虑 highPriority 的初始化到底意味着什么。

对函数 features 的调用将返回一个临时的 std::vector<bool> 对象。为了讨论方便，我们把这个没有名字的对象称为 temp。在 temp 上调用 operator[]，将会返回一个拥有机器字指针和偏移位的 std::vector<bool>::reference 对象。而 highPriority 正是这个对象的拷贝，并且指针指向了与 temp[5] 相同的机器字。当这个语句执行完后，临时变量 temp 将会被销毁。因此 highPriority 将包含一个悬空指针，这是导致函数 processWidget 的产生未定义行为的根本原因：

```
processWidget(w, highPriority);      // 未定义行为!
                                     // highPriority包含悬空指针
```

std::vector<bool>::reference 是一个代理类的例子：代理类是一种为了模仿和增强其他类的行为而创造的类。代理类有多种用途，std::vector<bool>::reference 的作用就是把 std::vector<bool> 的 operator[] 返回值模拟成 bool。再例如，标准库中的智能指针就是在指针的基础上扩充了资源管理的代理类。事实上，代理类的应用几乎无处不在，而“代理”也是设计模式领域经久不衰的模式之一。

有些代理类被设计成用户透明，例如 std::shared_ptr 和 std::unique_ptr。其他的代理类则被设计成或多或少程度上的不可见，std::vector<bool>::reference 就是一个不可见代理的例子，同样的还有 std::bitset::reference。

在 C++ 库中还有一部分代理类使用了被称为表达式模板的技术，这些库的目的通常是为了提升数值运算的效率。例如，假如有一个类 Matrix 和几个 Matrix 对象 m1、m2、m3、m4，那么表达式：

```
Matrix sum = m1 + m2 + m3 + m4;
```

在 operator+ 中使用代理类相比直接返回运算结果能够提升极大的效率，因为 operator+ 返回类似于 Sum<Matrix, Matrix> 的代理类相比直接返回 Matrix 对象的开销更小。正如 std::vector<bool>::reference 和 bool 的关系，Sum<Matrix, Matrix> 能够通过隐式转换成为 Matrix，因此表达式中的等号右边能够给 sum 准确的赋值。(这个赋值的对象通常会对整个运算式编码，最终成为 Sum<Sum<Sum<Matrix, Matrix>,Matrix>, Matrix> 的形式，这个结果也应该对用户不可见)

作为基本规则，不可见的代理类通常会造成 auto 类型推导上的问题。这些类一般情况下会在语句执行完后当场销毁，因此定义变量来存储和使用这样的代理类对象违反了基本的库设计前提。这也是为什么 std::vector<bool>::reference 会造成未定义行为的根本原因。

因而你肯定特别想避免这类语句：

```
auto someVar = expression of "invisible" proxy class type;
                                        // 生成不可见代理类对象的表达式
```

但是你又怎能轻易识别出代码是否使用了这样的代理类呢？在软件开发中使用代理类的目的并不是向程序员宣扬它们的存在，因为它们至少在概念上应该是不可见的。但是如果你一旦发现了这样的代理类，你真的想要扔掉 auto，以及我们在条款 5 中展示的那些优势吗？

首先让我们来看看你是怎样发现代理类的。尽管不可见的代理类被设计成不会被程序员在日常使用中发现，但是库代码通常会在文档中标记出它们的存在。如果你对库的基本设计原则越熟悉，那么你就越不可能被库中的代理类所蒙蔽。

另一方面，头文件也填补了文档的缺失。源代码中的代理类不可能被完全封装，它们总是会暴露在函数调用的返回值上，所以函数签名通常能够反映它们的存在。这是 std::vector<bool>::operator[] 的一个实现例子：

```
namespace std {                         // 来自C++标准库
    template <class Allocator>
    class vector<bool, Allocator> {
    public:
        ...
        class reference { ... };
        reference operator[](size_type n);
        ...
    };
}
```

如果你知道 std::vector<T> 的 operator[] 通常会返回 T&，那么这个不常见的返回值就暴露了它正在使用代理类，留意你正在使用的接口通常能够揭露这些代理类的存在。

实际上，很多开发者发现代理类的存在，基本上是由于试图追踪神秘的编译错误或者调试单元测试的意外结果。不管你是怎么发现它们的，一旦 auto 注定会将类型错误的推导成代理类，而非这个对象本身，那么你就应该考虑放弃使用 auto。在这个问题上，auto 并不是罪魁祸首，真正错误的是 auto 没有推导出我们想要的类型。所以解决方案应该是强制性的进行正确的类型推导，这种方法我称之为明确的类型初始化句式。

明确的类型初始化句式指的是用 auto 声明一个变量，然后把初始化表达式转换成你想要 auto 推倒出的类型。举例来说，用它给 highPriority 赋值成 bool:

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

这里 features(w)[5] 依然返回一个 std::vector<bool>::reference 对象，但是类型转换将表达式转型成了 bool，因此 auto 能够将 highPriority 的类型推导成 bool。在运行过程中，std::vector<bool>::operator[] 将返回的 std::vector<bool>::reference 对象转换成了支持的类型 bool。而在这个转换过程中，对 std::vector<bool>::reference 的解引用是合法的，这样就避免了我们先前的未定义行为。最终 highPriority 就被正确的赋值为数组中的第 5 个元素，类型为 bool。

至于 Matrix 这个例子，明确的类型初始化句式应该是这样：

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

明确的类型初始化句式的应用并不仅仅局限于代理类，它也适用于去强调声明变量的类型与初始化表达式的返回值类型不同。假如你有一个计算误差值的函数：

```
double calcEpsilon();                   // 返回误差值
```

函数 calcEpsilon 的返回值类型为 double。假如你的程序能够接受精度损失，并且使用 float 相比 double 是更好的选择，那么你可能会用 float 来存储 calcEpsilon 的结果：

```
float ep = calcEpsilon();               // float隐式转换成double
```

但是这样的代码很难说明程序员的做法是故意丢弃了函数返回值的精度。不同的是，明确的类型初始化句式可以这样写：

```
auto ep = static_cast<float>(calcEpsilon());
```

在相同的情况下，这种写法个更能够说明是故意丢弃精度的做法。假如你需要计算随机访问容器（例如 std::vector、std::deque 或 std::array）中某个元素的序号，并且给定了一个 0.0 到 1.0 之间的 double 值来表示这个元素在容器中的位置比例。(0.5 表示在容器正中间）进一步假设计算的结果会被赋值给一个 int，来表示元素的序号。如果容器的名字为 c，double 值为 d，那么计算序号的语句可以这样写：

```
int index = d * c.size();
```

但是这种写法隐藏了你想要把 double 转型成 int 的本意，然而明确的类型初始化句式让整件事变得明朗：

```
auto index = static_cast<int>(d * c.size());
```

请记住：

- 不可见的代理类会导致 auto 给初始化表达式推导出错误的类型。
- 明确的类型初始化句式能够让 auto 推导出你想要的类型。