CHAPTER 2 auto

从概念上来说，auto要多简单有多简单，但是它比看起来要微妙一些。使用它可以存储类型，当然，它也会犯一些错误，而且比之手动声明一些复杂类型也会存在一些性能问题。此外，从程序员的角度来说，如果按照符合规定的流程走，那auto类型推导的一些结果是错误的。当这些情况发生时，对我们来说引导auto产生正确的结果是很重要的，因为严格按照说明书上面的类型写声明虽然可行但是最好避免。

本章简单的覆盖了auto的里里外外。

Item 5:Prefer auto to explicit type declarations

条款五:优先考虑auto而非显式类型声明

哈，开心一下：

```
int x;
```

等等，该死！我忘记了初始化x，所以x的值是不确定的。它可能会被初始化为0，这得取决于工作环境。哎。

别介意，让我们转换一个话题， 对一个局部变量使用解引用迭代器的方式初始化：

```
template<typename It>
void dwim(It b, It e)
{
    while(b!=e){
        typename std::iterator_traits<It>::value_type
        currValue = *b;
    }
}
```

嘿！`typename std::iterator_traits<It>::value_type`是想表达迭代器指向的元素的值的类型吗？我无论如何都说不出它是多么有趣这样的话，该死！等等，我早就说过了吗？

好吧，声明一个局部变量，变量的类型只有编译后知道，这里必须使用'typename'指定，该死!

该死该死该死，C++编程不应该是这样不愉快的体验。

别担心，它只在过去是这样，到了C++11所有的这些问题都消失了，这都多亏了auto。auto变量从初始化表达式中推导出类型，所以我们必须初始化。这意味着当你在现代化C++的高速公路上飞奔的同时你不得不对只声明不初始化变量的老旧方法说拜拜：

```
int x1;             //潜在的未初始化的变量

auto x2;            //错误！必须要初始化

auto x3=0;          //没问题，x已经定义了
```

而且即使初始化表达式使用解引用迭代器也不会对你的高速驾驶有任何影响

```
template<typename It>
void dwim(It b,It e)
{
    while(b!=e){
        auto currValue = *b;
        ...
    }
}
```

因为auto使用Item2所述的auto类型推导技术，它甚至能表示一些只有编译器才知道的类型：

```
auto derefUPLess = [](const std::unique_ptr<Widget> &p1,    //专用于Widget类型的比较函数
const std::unique_ptr<Widget> &p2){return *p1<*p2;};
```

很酷对吧，如果使用C++14，将会变得更酷，因为lambda表达式中的形参也可以使用auto：

```
auto derefUPLess = [](const auto& p1,const auto& p2){return *p1<*p2;};
```

尽管这很酷，但是你可能会想我们完全不需要使用auto声明局部变量来保存一个闭包，因为我们可以使用`std::function`对象。
没错，我们的确可以那么做，但是事情可能不是完全如你想的那样。当然现在你可能会问：
`std::function`对象到底是什么，让我来给你解释一下：

`std::function`是一个C++11标准模板库中的一个模板，它泛化了函数指针的概念。与函数指针只能指向函数不同，`std::function`可以指向任何可调用对象，也就是那些像函数一样能进行调用的东西。当你声明函数指针时你必须指定函数类型（即函数签名），同样当你创建`std::function`对象时你也需要提供函数签名，由于它是一个模板所以你需要在它的模板参数里面提供。举个例子，假设你想声明一个`std::function`对象func使他指向一个可调用对象，比如一个具有这样函数签名的函数：

```
bool(const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2);
```

你就得这么写：

```
std::function<bool(const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2)> func;
```

因为lambda表达式能产生一个可调用对象，所以我们现在可以把闭包存放到`std::function`对象中。这意味着我们可以不使用auto写出C++11版的**dereUPLess**：

```
std::function<bool(const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2)>
dereUPLess = [](const std::unique_ptr<Widget> &p1,
const std::unique_ptr<Widget> &p2){return *p1<*p2;};
```

语法冗长不说，还需要重复写很多形参类型，使用`std::function`还不如使用auto。用auto声明的变量保存一个闭包这个变量将会得到和闭包一样的类型。

实例化`std::function`并声明一个对象这个对象将会有固定的大小。当使用这个对象保存一个闭包时它可能大小不足不能存储，这个时候`std::function`的构造函数将会在堆上面分配内存来存储，这就造成了使用`std::function`比auto会消耗更多的内存。并且通过具体实现我们得知通过std::function调用一个闭包几乎无疑比auto声明的对象调用要慢。
换句话说，std::function方法比auto方法要更耗空间且更慢，并且比起写一大堆类型使用auto要方便得多。在这场存储闭包的比赛中，auto无疑取得了胜利（也可以使用std::bind来生成一个闭包，但在Item34我会尽我最大努力说服你使用lambda表达式代替std::bind)

使用auto除了使用未初始化的无效变量，省略冗长的声明类型，直接保存闭包外，它还有一个好处是可以避免一个问题，我称之为依赖类型快捷方式的问题。你将看到这样的代码——甚至你会这么写：

```
std::vector<int> v;
unsigned sz = v.size();
```

`v.size()`的标准返回类型是`std::vector<int>::size_type`,但是很多程序猿都知道`std::vector<int>::size_type`实际上被指定为无符号整型，所以很多人都认为用`unsigned`比写那一长串的标准返回类型方便。这会造成一些有趣的结果。
举个例子，在**Windows 32-bit**上`std::vector<int>::size_type`和`unsigned int`都是一样的类型，但是在**Windows 64-bit**上`std::vector<int>::size_type`是64位，`unsigned int`是32位。这意味着这段代码在Windows 32-bit上正常工作，但是当把应用程序移植到Windows 64-bit上时就可能会出现一些问题。
谁愿意花时间处理这些细枝末节的问题呢?

所以使用auto可以确保你的不需要浪费时间：

```
auto sz =v.size();
```

你还是不相信使用auto是多么明智的选择？考虑下面的代码：

```
std::unordered_map<std::string,int> m;
...
for(const std::pair<std::string,int>& p : m)
{
    ...
}
```

看起来好像很合情合理的表达，但是这里有一个问题，你看到了吗?

要想看到错误你就得知道`std::unordered_map`的key是一个常量，所以`std::pair`的类型不是`std::pair<std::string,int>`而是`std::pair<const std::string,int>`。编译器会努力的找到一种方法把前者转换为后者。它会成功的，因为它会创建一个临时对象，这个临时对象的类型是p想绑定到的对象的类型，即m中元素的类型，然后把p的引用绑定到这个临时对象上。在每个循环迭代结束时，临时对象将会销毁，如果你写了这样的一个循环，你可能会对它的一些行为感到非常惊讶，因为你确信你只是让成为p指向m中各个元素的引用而已。

使用auto可以避免这些很难被意识到的类型不匹配的错误：

```
for(const auto & p : m)
{
    ...
}
```

这样无疑更具效率，且更容易书写。而且，这个代码有一个非常吸引人的特性，如果你把p换成是指向m中各个元素的指针，在没有auto的版本中p会指向一个临时变量，这个临时变量在每次迭代完成时会被销毁!

后面这两个例子说明了显式的指定类型可能会导致你不像看到的类型转换。如果你使用auto声明目标变量你就不必担心这个问题。

基于这些原因我建议你优先考虑auto而非显式类型声明。然而auto也不是完美的。每个auto变量都从初始化表达式中推导类型，有一些表达式的类型和我们期望的大相径庭。关于在哪些情况下会发生这些问题，以及你可以怎么解决这些问题我们在Item2和6讨论，所以这里我不再赘述。我想把注意力放到你可能关心的另一点：使用auto代替传统类型声明对源码可读性的影响。

首先，深呼吸，放松，auto是**可选项**，不是**命令**，在某些情况下如果你的专业判断告诉你使用显式类型声明比auto要更清晰更易维护，那你就不必再坚持使用auto。牢记C++没有在其他众所周知的语言所拥有的类型接口上开辟新土地。

其他静态类型的过程式语言（如C#,D,Sacla,Visual Basic等)或多或少的都有那些非静态类型的函数式语言（如ML,Haskell,OCaml.F#等）的特性。在某种程度上，几乎没有显式类型使得动态类型语言Perl,Python,Ruby等取得了成功，软件开发社区对于类型接口有丰富的经验，他们展示了在维护大型工业强度的代码上使用这种技术没有任何争议。

一些开发者也担心使用auto就不能瞥一眼源代码便知道对象的类型，然而，IDE扛起了部分担子，在很多情况下，少量显示一个对象的类型对于知道对象的确切类型是有帮助的，这通常已经足够了。举个例子，要想知道一个对象是容器还是计数器还是智能指针，不需要知道它的确切类型，一个适当的变量名称就能告诉我们大量的抽象类型信息。

真正的问题是显式指定类型可以避免一些微妙的错误，以及更具效率和正确性，而且，如果初始化表达式改变变量的类型也会改变，这意味着使用auto可以帮助我们完成一些重构工作。举个例子，如果一个函数返回类型被声明为int，但是后来你认为将它声明为long会更好，调用它作为初始化表达式的变量会自动改变类型，但是如果你不使用auto你就不得不在源代码中挨个找到调用地点然后修改它们。

记住

- auto变量必须初始化，通常它可以避免一些移植性和效率性的问题，也使得重构更方便，还能让你少打几个字。
- 正如Item2和6讨论的，auto类型的变量可能会踩到一些陷阱。