Item 4:Know how to view deduced types

条款四:学会查看类型推导结果

选择使用工具查看类型推导取决于软件开发过程中你想在哪个阶段显示类型推导信息，我们探究三种方案：在你编辑代码的时候获得类型推导的结果，在编译期间获得结果，在运行时获得结果

IDE编辑器

在IDE中的代码编辑器通常可以显示程序代码中变量，函数，参数的类型，你只需要简单的把鼠标移到它们的上面，举个例子，有这样的代码中：

```
const int theAnswer = 42;

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

一个IDE编辑器可以直接显示x推导的结果为**int**，y推导的结果为**const int***

为此，你的代码必须或多或少的处于可编译状态，因为IDE之所以能提供这些信息是因为一个C++编译器（或者至少是前端中的一个部分）运行于IDE中。如果这个编译器对你的代码不能做出有意义的分析或者推导，它就不会显示推导的结果。

对于像int这样简单的推导，IDE产生的信息通常令人很满意。正如我们将看到的，如果更复杂的类型出现时，IDE提供的信息就几乎没有什么用了。

编译器诊断

另一个获得推导结果的方法是使用编译器出错时提供的错误消息。这些错误消息无形的提到了造成我们编译错误的类型是什么。
举个例子，假如我们想看到之前那段代码中x和y的类型，我们可以首先声明一个类模板但不定义。就像这样：

```
template<typename T>     //只对TD进行声明
class TD;            //TD == "Type Displayer"
```

如果尝试实例化这个类模板就会引出一个错误消息，因为这里没有用来实例化的类模板定义。为了查看x和y的类型，只需要使用它们的类型去实例化TD：

```
TD<decltype(x)> xType;  //引出错误消息
TD<decltype(y)> yType;  //x和y的类型
```

我使用**variableNameType**的结构来命名变量，因为这样它们产生的错误消息可以有助于我们查找。对于上面的代码，我的编译器产生了这样的错误信息，我取一部分贴到下面：

```
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
```

另一个编译器也产生了一样的错误，只是格式稍微改变了一下：

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

除了格式不同外，几乎所有我测试过的编译器都产生了这样有用的错误消息。

运行时输出

使用**printf**的方法使类型信息只有在运行时才会显示出来（尽管我不是非常建议你使用printf），但是它提供了一种格式化输出的方法。现在唯一的问题是只需对于你关心的变量使用一种优雅的文本表示。“这有什么难的“，你这样想”这正是typeid和std::type_info::name的价值所在”。为了实现我们我们想要查看x和y的类型的需求，你可能会这样写：

```
std::cout<<typeid(x).name()<<"\n";  //显示x和y的类型
std::cout<<typeid(y).name()<<"\n";
```

这种方法对一个对象如x或y调用**typeid**产生一个**std::type_info**的对象，然后**std::type_info**里面的成员函数**name()**来产生一个C风格的字符串表示变量的名字。

调用**std::type_info::name**不保证返回任何有意义的东西，但是库的实现者尝试尽量使它们返回的结果有用。实现者们对于“有用”有不同的理解。举个例子，GNU和Clang环境下x会显示为**i**，y会显示为**PKi**，这样的输出你必须要问问编译器实现者们才能知道他们的意义：i表示int，PK表示**const to konst**（**const**）。Microsoft的编译器输出得更直白一些：对于x输出“int“对于y输出”int const*“

因为对于x和y来说这样的结果是正确的，你可能认为问题已经接近了，别急，考虑一个更复杂的例子：

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

std::vector<Widget> createVec();

const auto vw = createVec();

if(!vw.empty()){
    f(&vw[0]);
    ...
}
```

在这段代码中包含了一个用户定义的类型Widget，一个STL容器和一个auto变量vw，这个更现实的情况是你可能在会遇到的并且想获得他们类型推导的结果，比如模板类型参数T，比如函数参数param。

从这里中我们不难看出typeid的问题所在。我们添加一些代码来显示类型：

```
template<typename T>
void f(const T& param){
    using std::cout;
    cout<<"T=       "<<typeid(T).name()<<"\n";
    cout<<"param =  "<<typeid(param).name()<<"\n";
    ...
}
```

GNU和Clang执行这段代码将会输出这样的结果

```
T=      PK6Widget
param=  PK6Widget
```

我们早就知道在这些编译器中PK表示“指向常量”，所以只有数字6对我们来说是神奇的。其实数字6是类名称的字符串长度，所以这些编译器高数我们T和param都是**const Widget***。

Microsoft的编译器也同意上述言论：

```
T=      class Widget const *
param=  class Widget const *
```

这三个独立的编译器产生了相同的信息而且非常准确，当然看起来不是那么准确。在模板f中，param的类型是**const T&**。难道你们不觉得T和param相同类型很奇怪吗？比如T是int，param的类型应该是**const int&**而不是相同类型才对吧。

遗憾的是，事实就是这样，**std::type_info::name**的结果并不总是可信的，就像上面一样三个编译器都犯了相同的错误。因为**std::type_info::name**被批准犯这样的错。正如Item1提到的如果传递的是一个引用，那么引用部分将被忽略，如果忽略后还具有常量性或者易变性，那么常量性或者易变性也会被忽略。那就是为什么**const Widget *const &**类型会输出**const Widget ***，首先引用被忽略，然后这个指针自身的常量性被忽略，剩下的就是指针指向一个常量对象。

同样遗憾的是，IDE编辑器显示的类型信息也不总是可靠的，或者说不总是有用的。还是一样的例子，一个IDE编辑器可能会把T的类型显示为

```
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget>>::_Alloc>::value_type>::value_type *
```

同样把param的类型显示为

```
const std::_Simple_types<...>::value_type *const&
```

这个比起T来说要简单一些，但是如果你不知道<...>表示编译器忽略T的类型那么可能你还是会产生困惑。如果你运气好点你的IDE可能表现得比这个要好一些。

比起运气如果你更倾向于依赖库，那么你乐意被告知**std::type_info::name**不怎么好，Boost TypeIndex Library（通常写作Boost.TypeIndex）是更好的选择。这个库不是标准C++的一部分，也不时IDE或者TD这样的模板。Boost TypeIndex是跨平台，开源，有良好的开源协议的库，这意味着使用Boost和STL一样具有高度可移植性。

这里是如何使用Boost.TypeIndex得到f的类型的代码

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param){
    using std::cout;
    using boost::type_index::type_id_with_cvr;

    //显示T
    cout<<"T=    "
            <<type_id_with_cvr<T>().pretty_name()
            <<"\n";
    //显示param类型
    cout<<"param=    "
            <<type_id_with_cvr<decltype(param)>().pretty_name()
            <<"\n";
}
```

boost::type_index::type_id_with_cvr获取一个类型实参，它不消除实参的常量性，易变性和引用修饰符，然后**pretty_name**成员函数输出一个我们能看懂的友好内容。
基于这个f的实现版本，再次考虑那个产生错误类型信息的调用：

```
std::vetor<Widget> createVec();
const auto vw = createVec();
if(!vw.empty()){
    f(&vw[0]);
    ...
}
```

在GNU和Clang的编译器环境下，使用Boost.TypeIndex版本的f最后会产生下面的输出：

```
T=      Widget const *
param=  Widget const * const&
```

在Microsoft的编译器环境下，结果也是极其相似：

```
T=      class Widget const *
param=  class Widget const * const&
```

这样近乎一致的结果是很不错的，但是请记住IDE，编译器错误诊断或者Boost.TypeIndex只是用来帮助你理解编译器推导的类型是什么。它们是有用的，但是作为本章结束语我想说它们根本不能让你不用理解Item1-3提到的。

记住

- 类型推断可以从IDE看出，从编译器报错看出，从一些库的使用看出
- 这些工具可能既不准确也无帮助，所以理解C++类型推导规则才是最重要的