Template Type Deduction

C++模板类型推导

对于如下的C++模板类型推导

```
template<typename T>
void f(ParamType param);
f(expr);// 根据expr推导T和ParamType
```

要分三钟情况：

Case 1 ParamType是引用或指针，且不是Universal Reference

- 首先，如果expr是引用类型，则C++模板类型推导时会忽略参数expr的reference-ness。
- 然后，对ParamType的推导决定对T的推导

正常情况下，expr的constness会被保留，但是下面代码中函数f(const T& param)参数型中已经有const关键字时，则T只就不再含有constness。

```
#include <iostream>
using namespace std;
template<typename T>
void f(const T& param) // 此时T被推导为int, param的类型为const int &
{
    cout << param << endl;
    T tmp = 2019; // 正常运行
    tmp += 1;
    cout << tmp << endl;
}

void g(T& param) // 此时T被推导为const int, param的类型为const int &
{
    cout << param << endl;
    T tmp = 2019; // error, 无法编译
    tmp += 1;
    cout << tmp << endl;
}

int main()
{
    int x = 27; // as before
    const int& rx = x; // as before
```

```
    f(rx); // T is int
    g(rx); // T is const int
    return 0;
}
```

指针型的推导类似。

Case 2 ParamType是Universal Reference

Universal Reference翻译成通用引用？

这种情况，没那么明显直观了。模板类型的声明有点像右值引用(rvalue references, T&&)。但是当expr是左值(lvaue)时，推导行为有点“怪异”。

- 如果expr是一个左值(lvalue)，则T和ParamType都被推导为lvaue references。这很不常见。首先，在模板类型推导中，这是唯一一个把T推导为引用的场景。其次，尽管ParamType是用rvalue reference的语法来声明的，但是其推导出来的类型是lvalue reference。
- 如果expr是一个rvalue，则应用普通的推导规则(如，Case 1的规则)

举例：

```
template<typename T>
void f(T&& param); // param is now a universal
reference
int x = 27; // as before
const int cx = x; // as before
const int& rx = x; // as before
f(x); // x is lvalue, so T is int&,
// param's type is also int&
f(cx); // cx is lvalue, so T is const int&,
// param's type is also const int&
f(rx); // rx is lvalue, so T is const int&,
// param's type is also const int&
f(27); // 27 is rvalue, so T is int,
// param's type is therefore int&&, 右值(rvalue)
```

Case 3 ParamType既不是指针也不是引用

当ParamType即不是指针也不是引用时，则按值传递(pass-by-value):

```
template<typename T>
void f(T param); // param is now passed by value
```

这意味着无论传进来的是什么，param都是一份copy，一个全新的object。

- 和之前一样，如果expr是引用类型，则忽略referenceness
- 如果忽略过referenceness后，expr是const的，则也忽略constness。如果expr是volatile的，则也忽略。

因此：

```
int x = 27; // as before
const int cx = x; // as before
const int& rx = x; // as before
f(x); // T's and param's types are both int
f(cx); // T's and param's types are again both int
f(rx); // T's and param's types are still both int
```

注意尽管上面的cx和rx是const的，但param却不是const的。expr是const的，即不能修改，并不意味着它的copy也是不能修改的。
只有在按值传递时const(和volatile)才会被忽略。对于引用或指针类的parameters，constness是要保留的。

```
template<typename T>
void f(T param); // param is still passed by value
const char* const ptr = // ptr is const pointer to
const object
"Fun with pointers";
f(ptr);
```

注意上面的代码中ptr是按值传递的，param的类型是const char*。

总结：

- 在模板类型推导时，忽略reference-ness
- 在推导universal references类型的paramterer时，lvalue类型的arguments要特殊对待
- 当推导按值传递的parameters时，const和volatile类型的arguments被当作non-const和non-volatile处理
- 在模板类型推导时，数组或函数作为arguments时，它们会退化为pointers。(除非它们用于初始化引用)

参考：

Effective Modern C++