Template Argument Deduction

对于函数模板，编译器可以通过调用的实参来确定 (deduce，推断) 模板类型，这一过程称为**模板实参推断 (template argument deduction)** 。

Conversions and Template Type Parameters

函数模板的形参有类型转换限制。只有很少的情况下，编译器会自动将实参转换为形参。然而，编译器通常不会对实参进行类型转换，而是生成一个新的模板实例。

适用情况：

- `const`转换：将non`const`对象的引用 (指针) 传递给一个`const`的引用 (指针) 形参。
- 数组或函数指针转换：数组实参转换为一个指向其首元素的指针。

不适用的例子：

- 算术转换
- 派生类向基类的转换
- 用户自定义的转换

Example:

```
template<typename T> T fobj(T,T);   // arguments are copied
template<typename T> T fref(const T &,const T &);   // references
string s1("a value");
const string s2("another value");
fobj(s1,s2);// calls fobj(string,string); const is ignored
fref(s1,s2);// calls fref(const string &,const string &);
            // uses premissble conversion to const on s1

int a[10],b[42];
fobj(a,b);// calls fobj(int *,int *);
fref(a,b);// error: array types don't match
```

因为`fobj`的形参是拷贝形式，因此实参是否是`const`，都无所谓。实参是`const`，形参是独立于实参的，因此形参的改变不会影响实参的内容。在`fref`的调用中，参数类型是`const`的引用。对于引用实参而言，转换为`const`是允许的。经测试
`template <typename T> T fref(T&,T&);`
调用``fref(s1,s2)，T也会被转换为const string`。

后两个调用，数组`a,b`被转换为`int *`类型。最后一个调用中，形参是引用，数组不会转换为`int *`。

Function Parameters That Use the Same Template Parameter Type

对于`template<typename T> int cmp(const T &,const T &)`而言，调用必须提供相同类型的实参。或者修改`cmp`为

```
template<typename T>
int cmp(const T &a,const V&b){
    if(a<b)return -1;
    if(a>b)return 1;
    return 0;
}
```

这样就可以对比不同类型的值了，但是T类型和V类型之间的值是可比较的。也就是定义了相关的<,>等操作符。例如cmp(3.2,1024);。

Normal Conversions Apply for Ordinary Arguments

函数模板中的非模板参数，遵循普通转换规则。

Function-Template Explicit Arguments

某些情况下，编译器无法推断出模板实参的类型。或者我们希望用户控制模板实例化。

Specifying an Explicit Template Argument

Example:

```
// T1 cannot be deduced: it doesn't appear in the function parameter list
template<typename T1,typename T2,typename T3>
T1 sum(T2,T3);
```

这种情况下，没有任何函数实参可以用来推断T1的类型。每次调用sum函数时，调用者必须为T1提供一个**显示模板实参（explicit template argument）**。

```
// T1 is explicitly specified;
// T2 and T3 are inferred from the argument types
auto val3=sum<long long> (i,lng);// long long sum(int,long);
```

显示模板实参按从左至右的顺序与对应的模板参数相匹配。只有尾部（最右）的形参的显示模板实参可以省略，而且前提是他们可以从函数实参中推断出来。如果sum改成下面，

```
// very poor design: users must explicitly specify all three template parameters
template<typename T1,typename T2,typename T3>
T3 alternative_sum(T2,T1);
```

我们先来，看一下正常调用的代码生成：

```
auto val3=foo<long long ,int ,int > (1,lng);
```

其中,`T1,T2,T3`被分别实例化为`long long ,int ,int`.
因此,错误的用法`auto val=alternative_sum<long long> (1,lng);`是由于编译器无法推断出`T3`的类型。`T1`将会被显示实例化为`long long`,`T2`由实参类型推断为`int`,而返回类型`T3`无法被推断出来。

Normal Conversions Apply for Explicitly Specified Arguments

对于函数模板,
`template<typename T> int cmp(const T &a,const T &b){}`
我们不可`cmp(lng,1024)`,这样`cmp`不会被实例化,两个实参提供的类型不同,这样编译器无法推断出`T`的类型。因为根据第一个参数推断出`T`为`long`类型,而根据第二个参数推断出`T`为`int`类型,因此出现错误。

而我们显示指定形参的类型时,即`cmp<long> (lng,1024)`是可以的,因为函数模板将会显示指定实例化为`int cmp(const long &a,const long &b)`,实例化后的函数(此时可以看成普通函数),其形参可以被转换(规则和普通函数一样)。调用`cmp<long> (lng,1024)`,第二个参数将会由`int`转换为`long`。

Trailing Return Types and Type Transformation

尾置返回类型(trailing return type),即
`auto fun=[]()->ReturnType{}`。

我们来一下什么时候需要尾置返回类型在模板函数中的作用。我们有如下函数,

```
template<typename It>
??? &fuc(It beg,It end){
    // process the sequence
    return *beg; // return an reference to an element from the range
}
```

这里,我们并不知道返回结果的准确类型,我们可以这样调用

```
vector<int> v={1,2,3};
Blob<string> ca={"hi","bye"};
auto &i=fuc(vi.begin(),vi.end());// fuc returns int &
auto &s=fun(ca.begin(),ca.end());// fuc returns string &
```

我们可以通过`decltype(*beg)`来获取返回类型。但是,再编译器遇到函数的参数列表之前,`beg`是不存在的。为了更好的定义这样的函数,我们**必须**采用尾置返回类型(因为尾置返回出现在参数列表之后,它可以使用函数的参数):

```
// trailling return allows us declara the return type after the parameter list is
seen
template <typename It>
auto fun(It beg,It end)->decltype(*beg){
    //...
    return *beg;
}
```

解运算返回一个左值。因此，通过`decltype`推断的类型为`beg`表示的元素的类型的引用（这部分和C++左值的概念和`decltype`的用法相关）。

The type Transformation Library Template Classes

对于迭代器的操作，不会生成元素，只能生成元素的引用。为了获取元素类型，可以使用**type transformation**模板，定义在`type_traits`头文件中。通过`remove_reference`，可以获取其元素类型。
`remove_reference<decltype(*beg)>::type`
将会去除`*beg`的引用属性，即获取`beg`引用的元素的类型本身。组合使用`remove_reference`、尾置返回和`decltype`，就可以获取元素值的拷贝：

```
template<typename It>
auto fuc2(It beg,It end)->
    typename remove_reference<decltype(*beg)>::type{
        //...
        return *beg;// return a copy of *beg
    }
```

其中，应当注意`type`是一个模板类的成员（类型成员），因此必须在声明前加上`typename`来告知编译器这是一个类型。

Function Pointers and Argument Deduction

当使用一个函数模板初始化一个函数指针或为一个函数指针赋值时，编译器使用指针的类型来推断模板实参类型。例如，

```
template<typename T> int cmp(const T &, const T &);
// pf1 points to the instantiation, int cmp(const int &,const int &)
int (*pf1)(const int &,const int &)=cmp;
```

另一个有趣的例子是，

```
// overloaded versions of func, each takes one distinct function pointer type
void func(int (*)(const string &,const string &));
void func(int (*)(const int &,const int &));
func(cmp);
```

该问题在于，通过`func`的参数无法确定模板实参的唯一类型，因而出错。我们可以，

```
// ok, explicitly specify which version of cmp to instantiate
func(cmp<int>);// passing cmp(const int &,const int &);
```

Template Argument Deduction and References

对于模板实参类型推断，我们需要注意当形参为**引用**时的情况，
`template<typename T> void f(T &p);`

对于形参为引用的情形，满足一般的引用绑定规则。

Type Deduction from Lvalue Reference Function Parameters

当函数形参是模板类型参数的普通引用（左值引用）时，即`T&`，规则规定只能绑定一个**左值**。实参可以是`const`类型，也可以不是。如果是，`T`被推断为`const`类型：

```
template<typename T> void f1(T &);// argument must be an lvalue
// calls to f1 use the referred-to type of the argument as the template parameter
type
f1(i);  // i is int; template parameter T is int
f1(ci); // ci is const int; T is const int
f1(5);  // error, argument to a & parameter must be an lvalue, 5 is not an lvalue
```

如果函数形参是`const T &`类型时，按照正常的绑定规则：

- 可以绑定`nonconst`对象
- 可以绑定`const`对象
- 可以绑定临时对象
- 字面值常量（不可修改的左值）
  也就是说，任意`nonconst` or `const`的左值或右值都可以绑定到`const T &`形参上。这种情况时，`T`不会被推断为`const`类型，因为`const`已经是函数形参类型的一部分。因此，**它不会也是模板参数的一部分**：

```
template<typename T> void f2(const T &); // rvalue is permitted
// parameter in f2 is const &; const in the argument is irrelevant
// in each of these three calls, f2's function parameter is inferred as const int
&
f2(i);  // i is int, T is int
f2(ci); // ci is const int, T is int
f2(5);  // a const & parameter can be bound to an rvalue; T is int
```

Type Deduction from Rvalue Reference Function Parameter

当函数形参是右值引用的时候，即`T&&`，一般绑定规则允许我们传递右值绑定到该形参。

```
tmeplate<typename T> void f3(T&&);
f3(42); // ok, argument is an rvalue of int; T is an int
```

Reference Collapsing and Rvalue Reference Parameters

引用折叠（reference collapsing）和右值引用形参类型的关系。一般而言，`f3(i)`这样的调用是不合法的，因为`i`是一个左值，我们**通常**（一些情况可以）不能将一个右值引用绑定到一个左值上。但是有些情况是被允许的，这也是`move`正确工作的基础。

1. 第一个规则是，我们将左值传给函数的右值引用形参，且该右值引用指向模板类型参数（如`T&&`）时，编译器推断模板类型形参为实参的左值引用类型。也就是说，当我们调用`f3(i)`时，`T`被推断为`int &`，而非`int`。

此时，f3的函数形参类型是int& &&?似乎是一个int &类型的右值引用。通常，我们不能定义一个引用的引用，但是，通过类型别名或通过模板类型参数间接定义是可以的。

2. 这种情况下，第二个绑定规则：如果我们间接创建了一个引用的引用，则这些引用会形成“折叠”。在所有情况下（除了一个例外），引用会折叠成一个普通的左值引用类型。具体如下：

- X & &, X& &&, and X&& &将会被折叠成X &
- X && &&会被折叠成X&&

> Notes:引用折叠只能应用于间接创建的引用的引用，如类型别名或模板参数。

此时，左值传递给f3时，T会被推断为左值引用类型：

```
f3(i);// argument is an lvalue; T is an int &
f3(ci);// argument is an lvalue; T is a const int &
```

折叠规则告诉我们函数形参T &&折叠为一个左值类型，f3(i)的实例化结果可能像下面这样：

```
invalid code, just for demostration
void f3<int> (int & &&);// parameter is int & &&, if T is int &
```

折叠规则，会导致：

- 函数形参T&&，指向模板参数的右值引用，可以被绑定到一个左值；
- 如果实参是一个左值，则推断出的模板实参类型T将会是一个左值引用，且函数形参类型将会被实例化为一个（普通）左值引用参数（T&）；

❗ 因此，我们可以将任意类型的实参传递给T&&类型的函数形参。

Writing Template Functions with Rvalue Reference Parameters

模板参数T可以被推断为引用类型，会有哪些影响？

```
template<typename T> void f3(T &&val){
    T t=val;// copy or reference binding?
    t=fcn(t);// does the assignment change only t or val and t?
    if(val==t){/*...*/}// if T is a reference type, then always true
}
```

当我们对一个右值调用f3时，如字面常量42，T为int，此时，局部变量t的类型是int，并且通过拷贝参数val的值被初始化。当我们对t赋值的时候，val保持不变。

另一方面，当我们对一个左值i调用f3，则T为int &，此时，局部变量的t的类型为int &，对其初始化将会绑定到val，也就是说，此时的t是一个绑定到val的左值引用，改变t，val也会相应改变。

值得注意的是，对右值引用的函数模板通常用如下方式进行重载：

```
template<typename T> void f(T &&);// bind to nonconst rvalue
template<typename T> void f(const T &);// lvalue and const rvalue
```

和普通函数一样，第一个版本将其绑定到**可修改的**右值，而第二个版本将绑定到左值或const右值。

Understanding `std::move`

`std::move`函数是一个使用右值引用的模板。

通常，我们不能将一个右值引用绑定到一个左值上，但是可以用`move`获得一个绑定到左值上的右值引用。

How `std::move` Is Defined

```
template<typename T>
typename remove_reference<T>::type &&move(T&&t){
    return static_cast<typename remove_reference<T>::type &&> (t);
}
```

首先，`move`函数形参是一个`T&&`，通过引用折叠，该参数可以和任何类型的实参匹配。

```
string s1("hi"),s2;
s2=std::move(string("bye"));// ok, moving from an rvalue
s2=std::move(s1); // ok, but after the assignment s1 has indeterminate value
```

How `std::move` Works

在`s2=std::move(string("bye"));`中，实参是一个右值。此时，

- `T`的类型被推断为`string`
- `remove_reference`用`string`实例化
- `remove_reference<string>`的`type`成员是`string`
- `move`的返回类型因此是`string &&`
- `move`的函数参数`t`的类型是`string &&`
  因此，这个调用实例化`move<string>`，即函数
  `string &&move(string &&t)`
  函数体返回语句`static_cast<string &&> (t)`。`t`的类型是`string &&`，于是类型转换什么都不做。因此，此调用的结果就是它接受的右值引用。

对于`s2=std::move(s1)`，`s1`是一个左值，此时

- `T`的类型推断为`string &`（特殊的推断规则一，前面有提到）
- `remove_reference`用`string &`进行实例化
- `remove_reference<string &>`的`type`成员是（去除了引用属性）`string`
- `move`的返回类型是`string &&`
- `move`函数的形参`t`的类型为`string & &&`，会被折叠为`string &`
  因此，这个调用实例化`move<string &>`，即函数
  `string && move(string &t)`
  这个实例的函数返回`static_cast<string &&>(t)`。`cast`将`t`的类型转换为`string &&`。

`static_cast`from an Lvalue to an Rvalue References Is Permitted

不能隐式地将一个左值转换为右值引用，但是我们可以用`static_cast`显示地将一个左值转换为一个右值引用。

> A question:

```

```

```
vector<int> vec={1,2,3};
auto val=std::move(vec[0]);
```

这里val将会被一个右值赋值为1，但是，vec[0]似乎没有被move，依然是1。不过可以将vec作为一个整体move。

对于操作右值引用的代码而言，将一个右值引用绑定到一个左值的特性允许它们截断左值（clobber the lvalue，也就是说，该左值的资源不在归该左值管理）。

为了更安全的截断左值，我们应该强制使用static_cast，最好直接使用std::move。

Forwading

某些函数需要将其一个或多个实参连同类型不变地**转发**给其他函数。这个转发指的是我们需要保持被转发实参的所有性质，包括实参类型是否是const以及实参是否是左值还是右值。

```
template<typename F,typename T1,typename T2>
void flip1(F f,T1 t1,T2 t2){
    f(t2,t1);
}
```

其中，flip1接受一个可调用对象f，和两个参数。对于带有引用类型的参数时会出现问题：

```
void f(int v1,int &v2){
    cout<<v1<<" "<<++v2<<endl;
}
```

f(42,i)会改变实参i的值，但是如果通过flip1调用f，
flip1(f,j,42)
此时的j的值不会被改变。很容易理解，因为j被转递给t1，而t1是一个普通的、非引用的int。因此，flip1调用会被实例化为
void flip1(void(*fcn)(int,int &),int t1,int t2);

Defining Function Parameters That Retain Type Information

当然我们可以通过将t1改变一下，保持其实参的“左值性”。更进一步的，我们希望可以保持参数的const属性。

通过将一个函数参数定义为一个指向模板类型参数的右值引用，我们可以保持其对应实参的所有l类型信息。引用参数（左值或右值）可以保持const属性，因为引用类型中的const是底层的。我们将函数参数类型定义为右值引用，通过引用折叠，可以保持实参的左值和右值属性，

```
template<typename F,typename T1,typename T2>
void flip2(F f, T1&&t1, T2&&t2){
    f(t2,t1);
}
```

此时，传递左值给t1,t2，其类型会被推断为左值引用类型（包含引用折叠规则）。但是，当传递的参数是右值引用的时候，

```
void g(int &&i,int &j){
    cout<<i<<" "<<j<<endl;
}
```

如果我们通过flip2调用g，则参数t2将被传递给g的右值引用参数，即使我们转递一个右值给flip2：
flip2(g,i,42)
也会产生错误，该调用试图实例化
void flip2(void (*fcn)(int &&,int &),int &t1,int &&t2);

> ! 这里应当注意，下面中的val是一个左值，更具体点说，val是一个右值引用类型的左值。

```
int &&val=get();// get() returns an rvalue
```

因此上面的t2是右值引用类型的左值，我们不能用t2绑定g的第一个参数（即不能将一个右值引用参数绑定一个左值）。

Using sta::forward to Preserve Type Information in a Call

通过以上的例子可以看出，完整的保持原有实参的类型信息，有点麻烦。C++11标准可以使用std::forward来完整转发原始实参的类型，定义在utility中。

- forward必须通过显示模板实参来调用。
- forward返回该显示实参类型的右值引用，即forward<T>的返回类型是T&&。

通常情况，我们使用forward传递那些定义为模板类型参数的右值引用的函数参数。通过其返回类型上的引用折叠，forward可以保持给定实参的左值、右值属性：用法如下，

```
template<typename Type> intermediary(Type &&arg){
    finalFcn(std::forward<Type>(arg));
    // ...
}
```

本例中，Type的类型从arg推断出来。由于arg是一个模板类型形参的右值引用，Type将表示传递给arg的实参的所有类型信息。即，

- 实参是右值，Type是一个普通类型（非引用），forward<Type>将返回Type&&，右值引用类型
- 实参是左值，Type将会被推断为左值引用类型（引用折叠）。此时返回类型为Type & &&，再次引用折叠，返回一个左值引用类型

> Notes: 当用于一个指向模板参数类型的右值引用函数T&&时，forward会保持实参类型的所有细节。