先序文章请看C++模板元编程详细教程（之一）

模板参数自动推导

在上一章我们讲解了C++模板的基本用法，并且强调了模板并非直接可用的代码，需要经历「实例化」，而实例化的过程其实就是指定参数的过程。

但如果模板只能通过显式指定参数来进行实例化的话，那C++模板的「威力」也就止步于此了，进而也就不会出现复杂的模板元编程体系这样的东西。所以C++模板还有一个非常重要的特性，就是模板参数的自动推导。

模板参数的自动推导主要分为3种：

1. 根据函数参数自动推导（模板函数）
2. 根据构造参数自动推导（模板类）
3. 自定义构造推导（模板类）

需要强调的一点是，自动推导只能推导类型参数，而整数和整数派生参数是没法推导的，只能靠显式传入。

下面就来逐一介绍。

根据函数参数自动推导

本条针对的是模板函数。再次强调，模板参数和函数参数是不同的东西，用于实例化模板的参数是模板参数，用于调用函数的参数是函数参数。直观上来说，尖括号里的是模板参数，圆括号里的是函数参数。

我们先来看一个例子：

```
template <typename T>
void show(T t) {
  std::cout << t << std::endl;
}

void Demo() {
  int a = 5;
  show(a); // [1]
  show(5); // [2]
}
```

在上述例程中，show是一个模板函数，在下面两处标记的位置我们是直接调用show的，而没有指定模板参数，那么这时，就会触发模板参数的自动推导。

先来看一下[1]位置的调用，由于我们传入了参数a，编译器就会根据a的类型来推导模板参数T。在模板函数show的声明处，参数列表是(T t)，所以T会根据a的类型来推导。那么问题来了，这里到底会推导出int还是int &还是const int还是const int &？

答案也很简单，**模板参数的自动推导是完全按照auto的推导原则进行的**。（如果读者对这部分还不是很清楚的话可以参考《C++的缺陷和思考（三）》中「auto推导策略」章节的内容。）

也就是说，这里相当于auto t = a;，会推导出int，因此T会推导出int。

同理，针对于[2]位置的调用，我们传入的是一个常量5，照理说int、const int、const int &、int &&、const int &&都可以匹配，但根据auto的推导原则，仅仅保留「最简类型」，所以仍然会推导出int。

那么[1]和[2]的位置其实相当于：

```
show<int>(a); // [1]
show<int>(5); // [2]
```

再次强调，**推导的原则与auto相同**。那么同样地，也就支持和*、&、&&、const的组合，下面给出几个实例：

```
template <typename T>
void f1(T &t) {}

template <typename T>
void f2(const T &t) {}

template <typename T>
void f3(T *p) {}

void Demo() {
  f1(5); // 会报错，因为会推导出f1<int>，从而t的类型是int &，不能绑定常量
  int a = 1;
  f1(a); // f1<int>，t的类型是int &
  f2(a); // f2<int>，t的类型是const int &
  f3(a); // 会报错，因为会推导出f3<int>，此时t的类型是int *，int不能隐式转换为int *
  f3(&a); // f3<int>， t的类型是int *
}
```

这里需要注意的是，T是按照auto法则来推导的，但由于我们加上了修饰符，所以实际的函数参数t的类型是会带上这种描述符的，详情可以看上面例程的注释。

既然模板参数类型推导是按照auto法则，那就不得不提到一个特殊的推导，它就是auto &&。我们知道auto &&会根据绑定对象的左右性来推导出左值引用或是右值引用。（如果读者对这部分不清楚的话，可以参考《C++的缺陷和思考（二）》中「引用折叠」章节的内容。）同理对于用&&修饰的参数，在自动推导时也会拥有这样的特性。

换句话说，T &&也可以绑定可变值，此时会推导出左值引用。请看下面例程：

```
template <typename T>
void f4(T &&t) {}

void Demo() {
  int a = 5;
  const int b = 10;

  f4(1); // f4<int>，t的类型是int &&
  f4(a); // f4<int &>，t的类型是int &
  f4(b); // f4<const int &>，t的类型是const int &
  f4(std::move(a)); // f4<int>，t的类型是int &&
}
```

因此，这里总结出2条规律：

1. 当T &&匹配到可变值（也就是C++11里定义的「左值」）的时候，T会推导出左值引用，再根据引用折叠原则，最终实例化为左值引用
2. 当T &&匹配到不可变值（也就是C++11里定义的「右值」）的时候，T会推导出基本类型，最终实例化为右值引用

对于auto &&来说，我们只关心最终推导出的类型，并不会关心auto本身到底代表了什么。但对于模板的类型推导则不同，我们既要关心「模板参数推导出了什么类型」，又要关心「模板实例化后的函数参数是什么类型」。换做上面的例子来说就是，我们既要关系T推导出了什么，又要关心当T确定以后，t会变成什么类型。

那么按照上面的规律总结可以知道，即便我们传入的本身是一个右值引用（比如上面的std::move(a)），T依然会推导为int而并不是int &&。只不过实例化后的函数参数t的类型会变成int &&。

我相信有的读者在这里一定会产生疑问，既然「T推导出int」跟「T推导出int &&」的结果都是「t的类型是int &&」，那何必还要在此纠结呢？照目前的情况来说f4<int>和f4<int &&>可能确实看不出太大区别，但它会影响到模板的特化。如果我们定义了该种类型的特化，则会出现完全不同的行为。有关模板特化的内容将在后续章节详细讲解，请读者在此时记住，**利用函数参数自动推导出的模板参数永远不会推导出右值引用**，只可能推导出左值引用或者基本类型，这一点对于后续模板元编程是一个很重要的基础概念。

除了与引用、指针等组合外，还可以跟其他模板进行嵌套组合，编译器同样可以推导出正确的类型，请看例程：

```
template <typename T>
struct Test {};

template <typename T>
void f(const Test<T> &t) {}

void Demo() {
  Test<int> t1;
  Test<char> t2;

  f(t1); // 推导出f<int>，t的类型是const Test<int> &
  f(t2); // 推导出f<char>，t的类型是const Test<char> &
}
```

这种技巧非常适用于各种模板库（比如说STL），例如我们希望把一个vector的内容连续地放入一个buffer中，就可以这样来写：

```
#include <vector>
#include <cstdlib>
#include <cstddef>

template <typename T>
void CopyVec2Buf(const std::vector<T> &ve, void *buf, size_t buf_size) {
  if (buf_size < ve.size() * sizeof(T)) {
    return;
  }
  std::memcpy(buf, ve.data(), ve.size() * sizeof(T));
}

void Demo() {
  std::vector<int> ve{1, 2, 3, 4};
  std::byte buf[64];
  // 把ve的内容连续地复制到buf中
  CopyVec2Buf(ve, buf, 64); // 这里会推导出CopyVec2Buf<int>
}
```

模板函数之间还会存在重载问题，并且可能会引发二义性冲突，这部分内容将在后续章节详细介绍。

根据构造参数自动推导

模板函数可以通过函数参数来自动推导，那么模板类呢？当然就是通过构造参数来推导了。

先来看一个简单的例子：

```
template <typename T1, typename T2>
class Pair {
 public:
  Pair(const T1 &t1, const T2 &t2);
  void show() const;
 private:
  T1 t1_;
  T2 t2_;
};

template <typename T1, typename T2>
Pair<T1, T2>::Pair(const T1 &t1, const T2 &t2) : t1_(t1), t2_(t2) {}

template <typename T1, typename T2>
void Pair<T1, T2>::show() const {
  std::cout << "(" << t1_ << ", " << t2_ << ")" << std::endl;
}

void Demo() {
  Pair pair1{'a', 3.5}; // Pair<char, double>
  pair1.show();

  int a = 5;
  std::string str = "abc";

  Pair pair2{a, str}; // Pair<int, std::string>
  pair2.show();
}
```


上面的例子很好理解，就不过多赘述。但C++总是一种很「缺德」的语言，有的时候它的行为就是跟我们的直觉是相差甚远的，比如说：

```
template <typename T>
struct Test {
  Test(const T &t) {}
};


void Demo() {
  Test t{"abc"}; // 推导出Test<char[4]>类型
}
```

是不是很无厘头？这里传入的字面量"abc"，所以符合我们直觉的应当是，字符串字面量会处理为一个全局区的数据，然后在局部用其地址代替，那么这里应当是Test<const char *>才对，但结果并不是我们想的这样的。

究其原因，我们还是要对C++的语法定义进行一些深入的研究。对于字符串字面量来说，编译器会把它识别为一个全局的字符数组，对于代码中出现的相同字符串字面量，都会用同一个全局字符数组来代替。举例来说：

```
void Demo() {
  auto s1 = "abc";
  const char *s2 = "abc";
  std::string s3 = "abc";
}
```

当编译期发现有一个字符串字面量"abc"的时候，就会把它提取成一个全局的字符数组。所以上面的例子等价于：

```
const char g_str1[] {'a', 'b', 'c'};
void Demo() {
  auto s1 = g_str1; // auto推导出const char *类型
  const char *s2 = g_str1; // 数组类型隐式转换为首元素的指针类型
  std::string s3{g_str1}; // 调用了string(const char *)构造函数
}
```

而当字符串字面量直接初始化字符数组的时候，并不会生成全局的字符数组，而是直接成为了初始化字符数组的语法糖：

```
char str[] = "abc";
// 等价于
char str[] {'a', 'b', 'c', 0}; // 此时不会有g_str1之类的全局数组出现
```

因此，很多人印象中，「字符串字面量就是const char *类型」这个观念，其实是有一点点小问题的。因为它会被编译期理解为数组，而「数组」除了包含了首地址，还包含了长度的概念。只不过在大多数情况下，数组会“退化”成指针类型罢了。为了验证这个说法，我们做一个非常简单的小实验就好了：

```
#include <iostream>

int main() {
  std::cout << sizeof("1234") << std::endl; // 5
  std::cout << sizeof(const char *) << std::endl; // 8
  return 0;
}
```

通过静态sizeof运算就可以验证上面的说法，因此"1234"其实是const char [5]类型，而不是const char *。

所以回到一开始的例子，用一个字符串字面量去实例化模板类的时候，编译器会识别为数组语法，因此把T推导为char [4]类型，而构造参数t就成为了const char (&)[4]类型，这是一个数组的引用类型。

```
template <typename T>
struct Test {
  Test(const T &t) {}
};


void Demo() {
  Test t{"abc"}; // 推导出Test<char[4]>类型
}
```

但假如我们的模板类里有一些成员操作的话，就会导致这里报错，比如说:

```
template <typename T>
struct Test {
  Test(const T &t): mem_(t) {}
  T mem_;
};


void Demo() {
  Test t{"abc"}; // 报错
}
```

实例化时会报错，原因就在于，构造函数中我们用t去初始化mem_，而此时mem_是char [4]类型，t是const char (&)[4]类型（本质上就是char [4]类型的常引用），两个数组类型是不能直接复制的，所以会报错。

但如果这时，我们显式实例化的话：

```
template <typename T>
struct Test {
  Test(const T &t): mem_(t) {}
  T mem_;
};


void Demo() {
```

```
  Test<const char *> t{"abc"}; // OK
}
```

由于Test被强制实例化为Test<const char *>，那么此时t的类型就变成了const char *const类型，然后再接收字符串字面量的时候，数组类型会转化为数组首元素指针类型，也就是const char [4]转化为了const char *。而此时的mem_是const char *类型，自然就可以进行初始化的操作了（两个指针当然可以直接赋值）。

那这种情况要怎么办呢？我们能不能想办法让它不出现这种数组类型的实例化呢？答案是肯定的，那就要用到「模板参数类型推导指南」了，详见下节「自定义构造推导」。

自定义构造推导

为了「促使」模板类能够按照我们希望的方式来进行类型推导并实例化，当我们发现自动的类型推导不满足需求的时候，就可以考虑添加一种自定义的构造推导，这个语法称之为「推导指南(Deduction Guide)」。

当定义了推导指南后，编译期会优先根据推导指南来进行实例化，如果没有合适的推导指南，才会根据构造函数参数来进行实例化。我们用推导指南来解决上一节实例化字符数组的问题：

```
template <typename T>
struct Test {
  Test(const T &t): mem_(t) {}
  T mem_;
};

// Deduction Guide
template <typename T>
Test(T)->Test<T>;

void Demo() {
  Test t{"abc"}; // Test<const char *>
}
```

具体解释一下推导指南的语法，就是说对于类型Test，当构造参数是T的时候，我们要实例化为Test<T>。

相信有的读者看到这里会想，这不是废话嘛……本来不也是按这种方式推导的呀？但其实并不是！因为**推导指南会按照函数调用法则来识别**，也就是说，这里的Test(T)应当看做一个函数，当我们把const char [4]类型的参数传进函数参数的时候，就会转换为const char *。所以拥有了推导指南后，T会识别为const char *，再根据指南，实例化的结果就是Test<const char *>了。

当然了，遇到一些特殊需求（比如说你希望它推出对应的指针类型）也是可以方便地用推导指南来实现的：

```
template <typename T>
struct Test {};

template <typename T>
Test(T)->Test<T *>;

void Demo() {
  int a = 0;
  Test t{a}; // Test<int *>
}
```

甚至可以做一些更定制化的组合，比如说：

```
template <typename T>
struct Test {
  Test(T t) {}
};

template <typename T>
Test(T)->Test<T>;

// 如果传2个参数，按第二个走
template <typename T1, typename T2>
Test(T1, T2)->Test<T2>;

void Demo() {
  int a = 0;
  double b = 2.5;

  Test t1{a}; // Test<int>
  Test t2{a, b}; // Test<double>，注意此时调用的是Test<double>::Test(int, double)构造函数，又因为这个构造函数不存在所以会报错，可以通过偏特化解决
}
```

所以推导指南的用法远不止展示的这么简单，大家可以根据需要来发挥，另外它在后面重点介绍的模板元编程中也起了相当大的作用，希望读者可以理解渗透，打好基础。

小结

这一篇我们主要介绍了模板的自动类型推导，这是模板元编程很重要的理论基础，希望读者可以熟练掌握。

离核心内容又进一步了，大家稍安勿躁，磨刀不误砍柴工，下一篇将会重点介绍模板的特化。

C++模板元编程详细教程（之三）