模板特例化

有时候，对于一些特殊情况，通用的模板并不是最适合的，甚至有可能是不正确的。

例子如下，第二个版本是一个模板重载，以处理字符串字面常量和字符数组。

```
// 可以比较任意两个类型
template<typename T> int compare(const T &,const T &);
// 第二个版本，处理字符串字面常量
template<size_t N,size_t M>
int compare(const char (&)[N],const char (&)[M]);
```

对于如下调用，即传递给compare一个字符串字面常量或一个数组时，才会调用第二个版本，如果我们传递给它字符指针，就会调用第一个版本；

```
const char *p1="hi",*p2="mom";
compare(p1,p2); // 调用第一个版本
compare("hi","mom"); // 调用有两个非类型参数的版本
```

这是因为，我们无法将指针转换为一个数组的引用，因此当参数是两个字符指针是，第二个版本是不可行的。为了解决这个问题，我们可以为第一个模板版本定义个**模板特例化**版本。一个特例化版本就是一个独立的定义，在其中一个或多个模板参数被指定为特定的类型。

定义函数模板特例化

当我们特例化一个函数模板的时候，必须为原模板中的每个模板参数都提供实参。为了指明我们正在实例化一个模板，使用<>。空尖括号指出我们将为原模板的所有模板参数提供实参：

```
// compare的特例化版本
template <>
int compare(const char * const &p1,const char *const &p2){
    return strcmp(p1,p2);
}
```

当我们定义一个特例化版本时，函数参数类型必须与一个先前声明的模板中对应的类型匹配。本例中，我们特例化：

```
template<typename T> int compare(const T &,const T&);
```

函数重载和模板特例化

当定义函数模板的特例化版本时，我们本质上接管了编译器的工作。也就是，我们为原模板的一个特殊实例提供了定义。重要区别就是，模板特例化版本本质上是一个实例，而非函数名的一个重载版本。

一个模板的特例化版本和一个独立的非模板函数，会影响到函数匹配。例如，我们定义的各个版本的compare函数模板（前面两个版本，一个接受数组引用参数，一个接受const T &）。以及第三个特例化版本。当我们调用

```
compare("hi","mom");
```

两个函数模板都是可以的，且提供同样好的匹配。但是，接受字符数组参数的版本更加specialized，因此编译器会选择这个。

如果，我们将接受字符指针的compare版本定义为一个普通的非模板函数（即，不是模板的一个特例化版本），此调用的解析就会不同。此情况下，将会有三个可行的函数。两个模板和非模板的字符指针版本。所有的三个函数都提供同样好的匹配，此时，编译器优先选择非模板版本。

类模板特例化

除了特例化函数模板，也可以特例化类模板。这里给标准库hash模板定义一个特例化版本，可以用它来将Sales_data1对象保存在无序容器中。默认情况下，无序容器使用hash<key_type>来组织容器。为了让我们自己的数据类型也能使用这种默认组织方式，必须定义hash模板的一个特例化版本。一个特例化hash类必须定义：

- 重载的调用运算符，接受一个容器关键字类型的对象，返回一个size_t
- 两个类型成员，result_type和argument_type，分别是调用运算符的返回类型和参数类型。
- 默认构造函数和拷贝赋值运算符

在定义特例化版本的hash时，唯一复杂的地方，必须在原模板定义所在的命名空间中特化它。我们可以向命名空间std中添加成员。因此，先打开命名空间，

```
// 打开std命名空间
namespace std{

}// 关闭std命名空间；注意，这里没有分号
```

花括号括之间的任何定义都能成为命名空间std的一部分。现定义能处理Sales_data的特例化版本：

```
// open namespace std
namespace std{
    template<>
    struct hash<Sales_data>{
        // 用来hash的一个无序容器的类型必须定义以下类型
        typedef size_t result_type;
        typedef Sales_data argument_type;
        size_t operator()(const Sales_data &s)const;
        // 合成的拷贝构造函数和默认构造函数
    };

    size_t hash<Sales_data>::operator()(const Sales_data &s)const{
        return hash<string>()(s.bookNo)^
               hash<unsigned>()(s.units_sold)^
               hash<double>()(s.revenue);
```

```
    }
}// close namespace std;
```

如前所述，template<>指出，我们正在特例化一个模板。且正在特例化的模板名为hash，特例化版本为hash<Sales_data>。

假设特例化的版本在作用域中，当将Sales_data作为容器的关键字类型时，编译器会自动使用此特例化版本：

```
// 使用hash<Sales_data>和Sales_data的operator==
unordered_multiset<Sales_data> SDset;
```

并且hash<Sales_data>使用Sales_data的私有成员，我们必须将它声明为Sales_data的友元：

```
template<class T> class std::hash;// 友元声明所需要的
class Sales_data{
    friend class std::hash<Sales_data>;
};
```

这里，应当注意friend声明中应使用std::hash。以及友元声明之前只能声明std::hash类模板。

类模板部分特例化

与函数模板不同，类模板的特例化不必为所有的模板参数提供实参。我们可以只指定一部分而非所有的模板参数，或是参数的一部分而非全部特性。一个类模板的**部分特例化**本身是一个模板，使用的时候必须为那些在特例化版本中未指定的模板参数提供实参。

部分特例化可以是全部特例化，也可以是部分。

- 模板特例化（全部特例化），指明所有模板参数

```
template<class T1,class T2,class T3>class Foo{};
template<> class Foo<int,int,int>{};
```

- 部分特例化，指明部分模板参数（也可以是全部）

```
template<class T1,class T2,class T3>class Foo{};
template<class T1>class Foo<T1,int,int>{};
template<>class Foo<int,int,int>{};
template<class T1,class T2,class T3>class Foo<T1&&,T2&,T3>{};
```

> ❗ 只可以部分特例化类模板，而不能部分特例化函数模板。

对于标准库中的remove_reference类型，该模板就是通过一系列的特例化版本完成其功能：

```
// 原始的、最通用的版本
template<class T> struct remove_referece{
    typedef T type;// using type = T;
};
// 部分特例化版本,将用于左值引用和右值引用
template<class T> struct remove_reference<T &>// 左值引用
{
    typedef T type;
};
template<class T> struct remove_reference<T &&>{// 右值引用
    typedef T type;
};
```

部分特例化版本的模板参数列表是原始模板的一个子集或者一个特例化版本。本例中,特例化版本的参数数目与原始模板相同,但是类型不同。

第一个版本可以用任意类型实例化,它将模板实参作为type类型。

```
int i;
// decltype(42) 为int,使用原始模板
remove_reference<decltype(42)>::type a;
// decltype(i)为int &,使用第一个(T &)部分特例化版本
remove_reference<decltype(i)>::type b;
// decltype(std::move(i))为int &&
remove_reference<decltype(std::move(i))>::type c;
```

三个变量a,b,c都是int类型。

特例化成员而不是类

这是一个很有趣的情况,定义类Foo:

```
template<typename T> struct Foo{
    //...
    void Bar(){/*...*/}
};

template<>  // 指明正在特例化一个模板
void Foo<int>::Bar(){   // 指明正在特例化Foo<int>的成员Bar
    // 进行应用于int的特殊化处理
}
```

这里我们只特例化了Foo<int>的一个成员,其他成员将由Foo模板提供:

```
Foo<string> fs; // 实例化Foo<string>::Foo()
fs.Bar();       // 实例化Foo<string>::Bar()
```

```
Foo<int> fi;    // 实例化Foo<int>::Foo()
fi.Bar();       // 实例化我们特例化版本的Foo<int>::Bar()
```

当使用int之外的任何类型实例化Foo的时候，其成员像往常一样实例化（用的时候才实例化）。当我们用int实例化Foo的时候，Bar之外的成员如往常一样实例化，如果使用Foo<int>的成员Bar，则使用我们定义的特例化版本。