Traits :特性化

模板特性化主要针对问题：该模板类或函数需要针对不同的模板参数类型进行不同的算法处理。

模板的部分特性化（重载）

定义数据结构

typdef struct \_Boy{

int iPrice;

}Boy;

typdef struct \_Girl{

int iPrice;

}

定义一个通用模板

Template <typename T>

class HandleHair

{

void SetPrice(T persion)

{}

}

定义一个Boy特性化模板

Template <>

class HandleHair<Boy>

{

void SetPrice(Boy persion){}

}

定义一个Girl特性化模板

Template<>

class HandleHair <Girl>

{

void SetPrice (Girl persion){}

}

该模板主要是理发，针对男性和女性特殊类型的模板进行特殊处理，其他使用通用。

使用中：

HandleHair <int> hair; //调用第一个通用模板实例化

hair.SetPrice();

HandleHair <Boy> hair; //调用第二个Boy特性化模板实例化

hair.SetPrice();

HandleHair <Girl> hair; //调用第三个Girl特性化模板实例化

hair.SetPrice();

类模板部分特化必须有它自己对成员函数、静态数据成员和嵌套类的定义

还是一个模板类Test：

**[cpp]** [view plain](http://blog.csdn.net/my_business/article/details/7891687) [copy](http://blog.csdn.net/my_business/article/details/7891687)

1. **template** <**typename** T>
2. **class** Test {
3. **public**:
4. **int** Compute(**int** d);
5. **private**:
6. T mData;
7. };

它有一个Compute方法来做一些计算，具有int型的参数并返回int型的值。

现在需求变了，需要在T为int类型时，Compute方法的参数为int，返回类型也为int，当T为float时，Compute方法的参数为float，返回类型为int，而当T为其他类型，Compute方法的参数为T，返回类型也为T，怎么做呢？还是用traits的方式思考下。

**[cpp]** [view plain](http://blog.csdn.net/my_business/article/details/7891687) [copy](http://blog.csdn.net/my_business/article/details/7891687)

1. **template** <**typename** T>
2. **struct** TraitsHelper {
3. **typedef** T ret\_type;
4. **typedef** T par\_type;
5. };
6. **template** <>
7. **struct** TraitsHelper<**int**> {
8. **typedef** **int** ret\_type;
9. **typedef** **int** par\_type;
10. };
11. **template** <>
12. **struct** TraitsHelper<**float**> {
13. **typedef** **float** ret\_type;
14. **typedef** **int** par\_type;
15. };

然后我们再把Test类也更新下：

**[cpp]** [view plain](http://blog.csdn.net/my_business/article/details/7891687) [copy](http://blog.csdn.net/my_business/article/details/7891687)

1. **template** <**typename** T>
2. **class** Test {
3. **public**:
4. TraitsHelper<T>::ret\_type Compute(TraitsHelper<T>::par\_type d);
5. **private**:
6. T mData;
7. };

可见，我们把因类型不同而引起的变化隔离在了Test类以外，对用户而言完全不需要去关心这些逻辑，他们甚至不需要知道我们是否使用了traits来解决了这个问题。

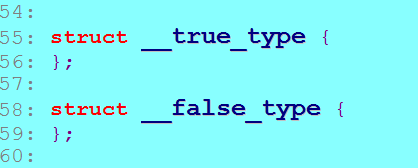
到这里，再让我们回过来取品味下开始我说的那句话：

**当函数，类或者一些封装的通用算法中的某些部分会因为数据类型不同而导致处理或逻辑不同时，traits会是一种很好的解决方案。**

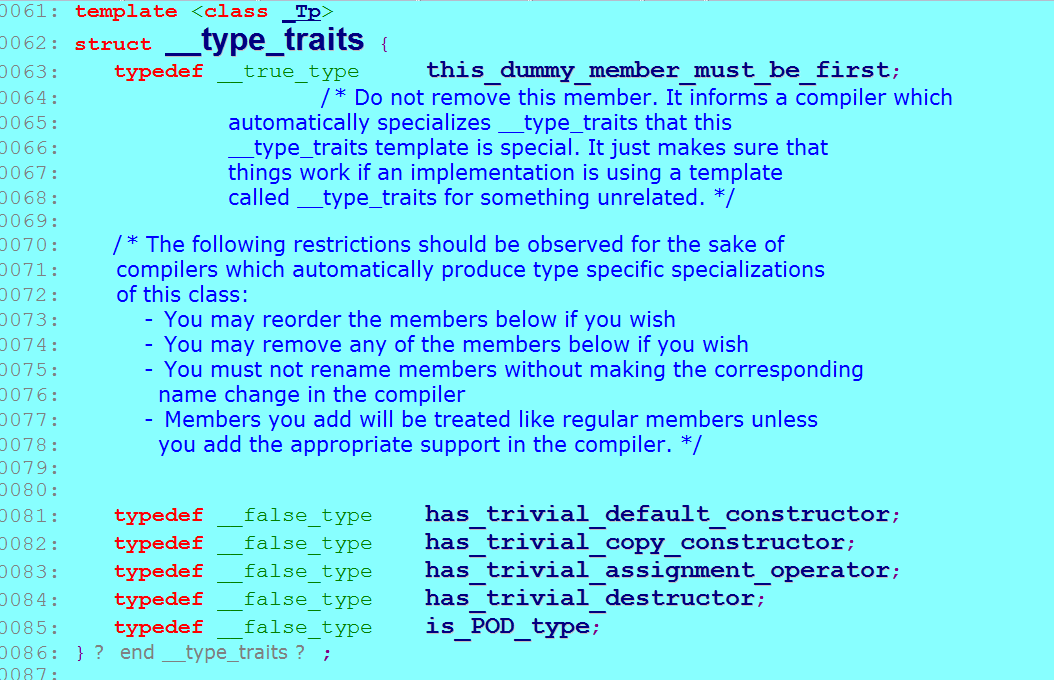
STL 源码中 模板部分特性化应用

Type\_traits.h

首先定义特性化的数据结构

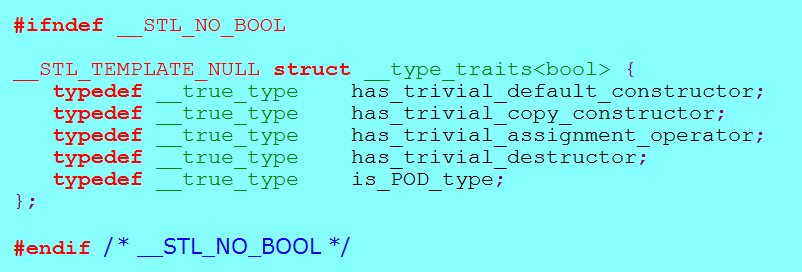


定义通用模板



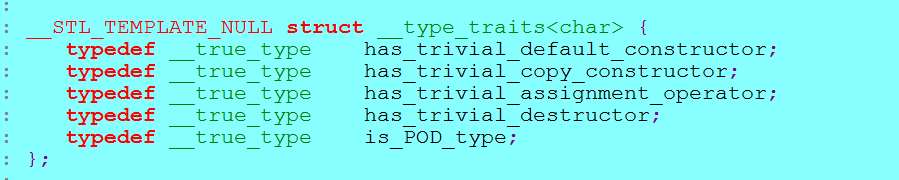
通用模板中定义了特定的数据类型都为\_false\_type

定义Bool特性化模板



Bool特性模板中的类型都定义为\_true\_type

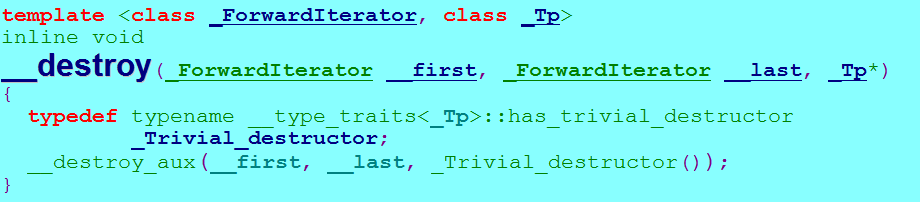
定义char特性化模板



char特性模板中的类型都定义为\_true\_type

。。。。。。。。。。。。。。。。。。。。。。。

使用部分特性化模板



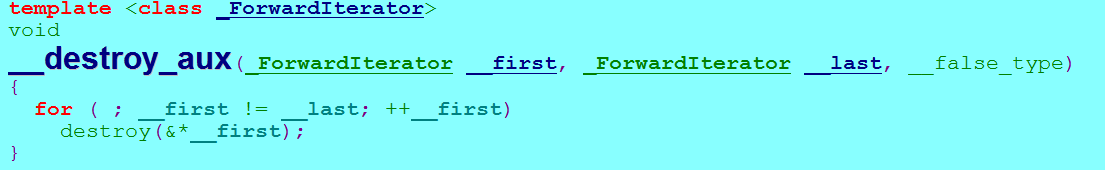
定义部分特性化模板

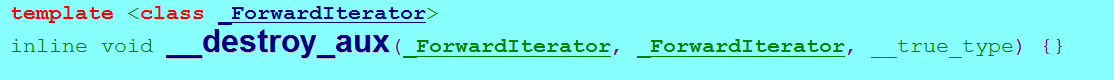
typedef typename \_\_type\_traits<\_Tp>::has\_trivial\_destructor

\_Trivial\_destructor;

typename \_\_type\_traits<\_Tp> 为前面定义的模板，若\_Tp为Bool类型，则部分化特性模板，调用struct \_\_type\_traits<bool> 模板类，则成员类型has\_trivial\_destructor为\_true\_type 类型。所以\_Trivial\_destructor为\_true\_type 结构体类型(typedef定义了这种类型)。\_Trivial\_destructor()为构造这个类型的对象。

最后作为形参传递给函数\_destroy\_aux()，重载类型形参函数





这里有趣的是对于函数的重载，其中不同的参数可以使用一个类型，调用时传递类型()

int Add (int n1,int n2,int)

{

return n1+n2;

}

int Add (int n1,int n2,double)

{

return n1-n2;

}

CSon::CSon():CMother(),CFather()

{

printf("I am son \n");

printf("testFalse:%d \n",Add(1,1,int()));

printf("testTrue:%d \n",Add(1,1,double()));

}