<stl\_iterator\_bash.h>

## 迭代器设计思维 – STL关键所在

* 1. 设计模式中对迭代器的定义：

提供一种方法，使之能够依序巡访某个聚合物（容器）所含的各个元素，而又无需暴露该聚合物的内部表述方式；

* 1. STL的中心思想在于：将数据容器（containers）和算法 (algorithms) 分开，彼此独立设计，最后再以一粘胶着剂（ Iterator ）将他们撮合在一起；
  2. 容器和算法的泛型化：通过C++的 class templates（类模板） 和 function templates（函数模板） 实现；
  3. 实现容器迭代器,需要对容器的实现的细节有丰富的了解。为了不暴露容器内部实现细节，故STL提供了容器专属迭代器；

## 迭代器是一种smart point

* 1. 迭代器是一种行为类似指针的对象；
  2. 指针最常见的操作：解引用和成员访问；

因此，迭代器最重要的编程操作：operator \* 和 operator -> 进行重载；

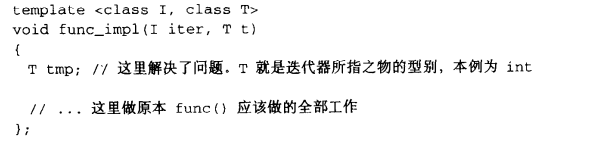
## 迭代器相应的型别（ associated types ）

### 迭代器相应型别由来

算法中很可能用到“迭代器相应的型别 -- 迭代器所指对象的型别”；此时如何是好？

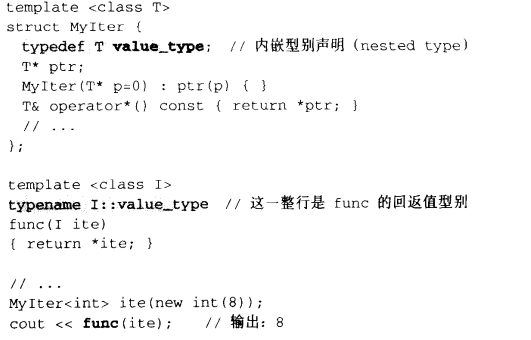
C++只支持sizeof() , 并未支持typeof() ,即便使用RTTI性质的typeid() , 获得的也只是型别名称；

解决办法：利用function template 的参数推导（argument deducation）机制。



弊端：T 类型做不成返回值；“function template 的参数推导”，推导的只是参数类型，无法推导函数的返回值类型；

解决返回值为T类型：声明内嵌型别



Ps：func() 返回值必须加上关键词typename , 因为T是一个template参数，具现化之前，编译器对T一无所知（不知道 MyIter<T>::value\_type , 还是 data member ,还是 member function）,typename 告诉编译器 T为一个型别;

声明内嵌型别的隐晦陷阱：并不是“所有迭代器”都是class type ，原生指针就不是？ 原生指针无法定义内嵌型别，所以以上的func()会出问题；但STL（及整个泛型思维）绝对必须接受原生指针作为一种迭代器；

原始指针无法定义内嵌型别？

使用“ template partial specialization（偏特化） ”作特化处理；

文本, 信件

描述已自动生成

### C++ 偏特化

定义：针对（任何）template参数更进一步的条件限制所设计出来的一个特化版本；

特化的由来：对于编译器来说，对某个特定类型，谁有更好的实现就听谁的；（也就是说，会“优先使用”特化版本）

全特化 ：模板参数全部明确

template<>

class A<int>

{

};

偏特化：模板参数未全部明确

template< typename T1 >

class A<T1 , int>

{

};

特殊的偏特化：指针、引用

template< typename T1 >

class A<T1\*>

{};

template< typename T1 >

class A<T1&>

{};

### 迭代器相应的型别分类

* + 迭代器所指对象的型别 (value\_type)；
  + 迭代器之间的距离（difference\_type ）

默认使用C++内建的ptrdiff\_t作为difference\_type；

* + 被迭代类型的引用（reference 引用）
  + 被迭代类型的指针（pointer 指针）
  + 迭代器分类（ iterator\_category ）

图示

描述已自动生成

Ps：五种迭代器并非C++继承，而是强化；

Random Access Iterator 本质上也是一个Forward Iterator；以此类推；

文本

描述已自动生成

执行时期决定使用哪个迭代器会影响效率，如何将其改成编译时使用哪个迭代器？

文本

描述已自动生成

解决方案：函数重载；

文本, 信件

描述已自动生成

消除“单纯的传递调用”

如：\_\_advance(………

, forward\_iterator\_tar )

重载多个函数封装上层接口；

文本

描述已自动生成

## Traits编程技法 – STL源代码门钥

Traits编程技法：iterator\_traits专门用来“萃取”迭代器的相应型别；通过traits获取迭代器相应value\_type 得到了完全统一；

* 萃取迭代器特性：

文本, 信件

描述已自动生成

typename Iterator\_traits<I>::value\_type ;

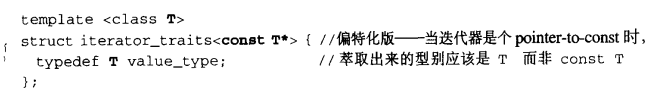
* 萃取原生指针特性：

文本

中度可信度描述已自动生成

typename Iterator\_traits< int\* >::value\_type ;

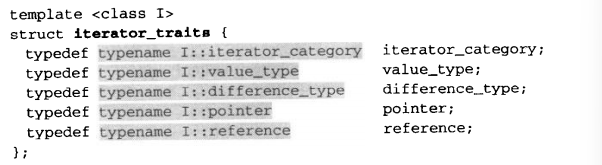
* 萃取const指针的特性：



typename Iterator\_traits<const int\* >::value\_type ;

现在，不论面对的是Iterator（迭代器） , 或 int\*（原生指针），或const int\* (指向常数对象的指针)，都可以通过traits取出正确的value type；

* 如果要使traits所扮演的“特性萃取机”角色有效运作，每一个迭代器必须遵循约定：自行以内嵌型别定义的方式定义出相应型别；谁不遵守这个约定，谁就不能兼容于STL；



## std::iterator的保证

任何迭代器都应该提供五个内嵌型别，为了保证统一：STL提供了iterator基类

表格

中度可信度描述已自动生成

总结：设计“相应的型别”是迭代器的责任，设计“适当的迭代器”是容器的责任；

文本, 信件

描述已自动生成

## SGI STL私房菜 ：\_type\_traits

* 头文件<type\_traits.h>
* Iterator\_traits : 负责萃取迭代器特性； \_\_type\_traits：负责萃取型别特性；
* 此处所关注的型别特性指：

1. 这个类型是否具备默认构造函数？
2. 这个类型是否具备拷贝构造？
3. 这个类型是否具备赋值操作？
4. 这个类型是否具备析构函数？

如果答案是否定的，我们对该类型变量，进行构造、析构、拷贝、赋值等操作时，就可以采用最有效率的措施（采用内存直接处理操作,malloc , memcpy ），获得最高效率；（当频繁操作容器时）

* \_\_type\_traits 提供了一种机制，允许针对不同的型别属性，在编译时期完成函数派送决定；（提前知道元素是否有以上列举的型别特性）
* \_\_type\_traits 相应型别分类

文本, 信件

描述已自动生成

Ps ： 是否可采用最有效率的方式（memcpy等）进行拷贝，赋值等操作？

\_\_true\_type：可以采用；

\_\_false\_type：不可采用；

* \_\_type\_traits对各种型别的保守定义

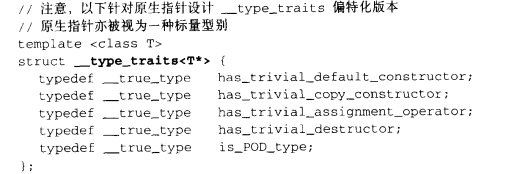
文本, 信件

描述已自动生成

* 对所有C++基本类型提供对应的\_\_type\_traits特化声明

报纸上的文字

描述已自动生成



* 一个classs什么是否该有自己的 non-trivial xxx

判断原则：

class内含指针成员，并且对它进行动态配置，那么这个class 就需要实现出自己的non-trivial-xxx ；

* POD类型详解：

1. POD 全称：Plain Old Data ；
2. Plain：代表一个对象是一个普通类型；
3. Old：代表一个对象可以与C兼容；
4. POD类型：一个对象即使普通类型（trivial type ）又是标准布局类型(standard-layout type)

普通类型：

* 没有虚函数，虚基类；
* 由C++编译器提供默认的特殊成员函数（默认构造、默认拷贝构造 &、默认移动构造 && 、赋值运算符 、移动赋值运算符、和析构函数）；
* 数据成语同样满足条件（1）和（2）；

std::is\_trivial<T>::value -- 数据类型是否为普通类型；

标准布布局类型：

* 没有虚函数，虚基类；
* 所有非静态数据成员都具有相同的访问说明符（private / public / protected ）；
* 在继承体系中，最多只有一个类中有非静态数据成员；
* 子类中的第一个非静态成员的类型与其基类不同（基类也会占用空间）；此规则是因为 C++ 允许优化不包含成员基类而产生的。在 C++ 标准中，如果基类没有任何数据成员，基类应不占用空间，为了体现这一点，C++ 标准允许派生类的第一个成员与基类共享同一地址空间。但是如果派生类的第一个非静态成员的类型和基类相同，由于 C++ 标准要求相同类型的对象的地址必须不同，编译器就会为基类分派一个字节的地址空间。比如下面的代码：

class B1{};

class B2{};

class D1: public B1

{

B1 b;

int i ;

};

class D2: public B1

{

B2 b ;

int i ;

}