



STL 源码剖析

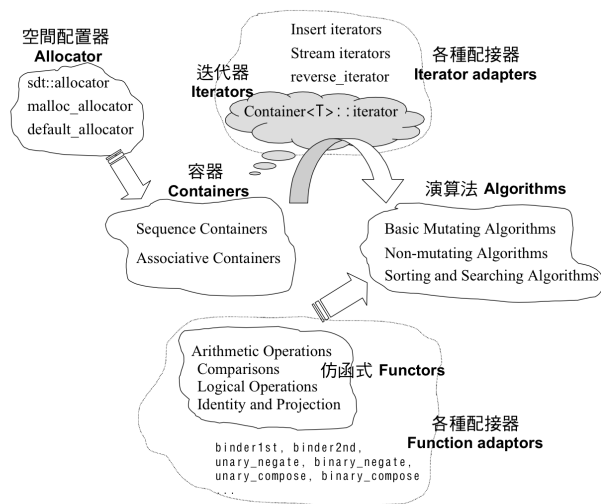


圖 1-1 STL 六大組件的交互關係：Container 透過 Allocator 取得資料儲存空間，Algorithm 透過 Iterator 存取 Container 內容，Functor 可以協助 Algorithm 完成不同的策略變化，Adapter 可以修飾或套接 Functor。

組態：__STL_NULL_TMPL_ARGS (bound friend template friend)

<stl_config.h> 定義 __STL_NULL_TMPL_ARGS 如下：

```
# ifdef __STL_EXPLICIT_FUNCTION_TMPL_ARGS
#   define __STL_NULL_TMPL_ARGS <>
# else
#   define __STL_NULL_TMPL_ARGS
# endif
```

条款 6:

```
// 前缀形式: 增加然后取回值
UPInt& UPInt::operator++()
{
    *this += 1; // 增加
    return *this; // 取回值
}

// postfix form: fetch and increment
const UPInt UPInt::operator++(int)
{
    UPInt oldValue = *this; // 取回值
    ++(*this); // 增加
    return oldValue; // 返回被取回的值
}
```

话说一般临时产生的变量不是被限制为 const。
后缀的 increment 返回 oldValue 的 const

为什么是 const? 假设不是 const 对象, 下面的代码就是正确的:

```
UPInt i;
```

```
i++++;
```

等价于 `i.operator++(0).operator++(0)`

第 1, `int` 类不允许这样做 `int i; i++++;` // 错误

第 2, 根本没有任何改变, `i++ return i; i++++;`

Return 仍是 i ; 如果是 Const ^{則↓} 禁止这样做。

allocator :

allocator::rebind

一個巢狀的 (nested) class template。class **rebind<U>** 擁有唯一成員 **other**，
那是一個 typedef，代表 **allocator<U>**。

allocator::allocator()

default constructor。

allocator::allocator(const allocator&)

copy constructor。

template <class U>allocator::allocator(const allocator<U>&)

泛化的 copy constructor。

allocator::~~allocator()

default constructor。

pointer allocator::address(reference x) const

傳回某個物件的位址。算式 **a.address(x)** 等同於 **&x**。

const_pointer allocator::address(const_reference x) const

傳回某個 **const** 物件的位址。算式 **a.address(x)** 等同於 **&x**。

pointer allocator::allocate(size_type n, const void* = 0)

配置空間，足以儲存 **n** 個 **T** 物件。第二引數是個提示。實作上可能會利用它來
增進區域性 (locality)，或完全忽略之。

void allocator::deallocate(pointer p, size_type n)

歸還先前配置的空間。

size_type allocator::max_size() const

傳回可成功配置的最大量。

void allocator::construct(pointer p, const T& x)

等同於 **new(const void*) p T(x)**。

void allocator::destroy(pointer p)

等同於 **p->~T()**。

construct and destroy

STL 規定
配置器 (allocator)
定義於此

`<memory>`

`<stl_construct.h>`

`<stl_alloc.h>`

`<stl_uninitialized.h>`

這裡定義有全域函式
construct() 和 **destroy()**，
負責物件的建構和解構。
它們隸屬於 STL 標準規範。

這裡定義有一、二級配置器，
彼此合作。配置器名為 **alloc**。

這裡定義有一些全域函式，用來充填 (fill)
或複製 (copy) 大塊記憶體內容，它們也都
隸屬於 STL 標準規範：

un_initialized_copy()
un_initialized_fill()
un_initialized_fill_n()

這些函式雖不屬於配置器的範疇，但與物件初值
設定有關，對於容器的大規模元素初值設定很有
幫助。這些函式對於效率都有面面俱到的考量，
最差情況下會呼叫 **construct()**，
最佳情況則使用 C 標準函式 **memmove()** 直接進行
記憶體內容搬移。

The Annotated STL Sources

trivial destructor

如果用户不定义析构函数，而是用系统自带的，则说明，析构函数基本没有什么用（但默认会被调用）我们称之为trivial destructor。反之，如果特定定义了析构函数，则说明需要在释放空间之前做一些事情，则这个析构函数称为non-trivial destructor。如果某个类中只有基本类型的话是没有必要调用析构函数的，delete p的时候基本不会产生析构代码。

在C++的类中如果只有基本的数据类型，也就不需要写显式的析构函数，即用默认析构函数就够用了，但是如果类中有个指向其他类的指针，并且在构造时候分配了新的空间，则在析构函数中必须显式释放这块空间，否则会产生内存泄露，

在STL中空间配置时候destory () 函数会判断要释放的迭代器的指向的对象有没有 trivial destructor (STL中有一个 has_trivial_destructor函数，很容易实现检测) 放，如果有trivial destructor则什么都不做，如果没有即需要执行一些操作，则执行真正的destory函数。destory () 有两个版本，第一个版本接受一个指针，准备将该指针所指之物析构掉，第二个版本接受first和last两个迭代器，准备将[first, last]范围内的所有对象析构掉。我们不知道这个范围有多大，万一很大，而每个对象的析构函数都无关痛痒，那么一次次调用这些析构函数，对效率是一种伤害，因此这里首先利用value_type() 获得迭代器所指对象的类别，再利用 type_traits<T>判断该型别的析构函数是否无关痛痒，若是 (true_type)，则什么也不做就结束，若否 (false_type)，这才以循环的方式遍历整个范围，并在循环中每经历一个对象就调用第一个版本的destory()。

placement new 是一个全局版本不可重载

set new handler 是设定 new handler 的函数，而 new handler 是在 operator new 中当内存失败的时候，则调用处理函数 (new-handler)

set_new_handler进行设置

这两个函数声明如下：

其中，`new_handler`是个typedef，定义一个函数指针，该函数没有参数，也没有返回值；

`set_new_handler`用于设置处理函数，设置p为当前处理函数，并返回之前的new_handler

`(* (void(*)())0) ()` 等同 `((void(*)())0) ()` ----原因函数是一种function-to-pointer的方式, `&fun`, `fun`, `*fun`, `**fun`都是一样

硬件地址跳到0处

```
(* (void(*)())0) ();
```

预备知识

```
float (*h) ();
```

表示h是一个指向返回值float类型的函数的指针

```
(float(*)())
```

表示一个"指向返回值float类型的函数的指针"的类型转换符

假设fp是一个函数指针, 那么如何调用fp所指向的函数, 调用方法如下:

```
(*fp) ();
```

按照人们的惯性思维, 那么我们可以这样写

```
(*0) ();
```

上式不能生效, 因为运算符*必须要一个指针来做操作数, 而且这个指针还必须是个函数指针。所以我们要把0强制转换成一个函数指针(指向返回值为void类型的函数的指针)

假设fp是个float指针, 声明如下

```
float * fp;
```

强转0的收益在于方便计算偏移

把0强制转换成一个float指针(把变量fp去掉就可以了)

```
(float *) 0;
```

类似:

假设fp是函数指针为void类型的函数的指针), 声明如下:

```
void (*fp) ();
```

把0强制转换成该函数指针(变量fp去掉就可以了)

```
(void(*)())0
```

最后用`(void(*)())0`代替fp, 从而得到调用的用法

```
(* (void(*)())0) ();
```

可用typedef简化函数指针

例如:

```
typedef char * string;
```

```
string test="hello";
```

类似

```
typedef void(*func) (); //这样func就表示一个函数指针的类型
```

```
(* (func)0) ();
```

详见:

<http://m.blog.csdn.net/zyboy2000/article/details/4202349>

例子

方法一：

```
typedef void (*pfunction)(void);
```

```
void FMI_Jump(void)
{
    pfunction jump;
    jump=(pfunction) (0x80000);
    jump();
}
```

方法二：

```
((void(code *) (void)) 0xF400) ();
```

POD类型

啥是POD类型？

POD全称Plain Old Data。通俗的讲，一个类或结构体通过二进制拷贝后还能保持其数据不变，那么它就是一个POD类型。

平凡的定义

1. 有平凡的构造函数
2. 有平凡的拷贝构造函数
3. 有平凡的移动构造函数
4. 有平凡的拷贝赋值运算符
5. 有平凡的移动赋值运算符
6. 有平凡的析构函数
7. 不能包含虚函数
8. 不能包含虚基类

POD类型可以进行二进制拷贝

traits:

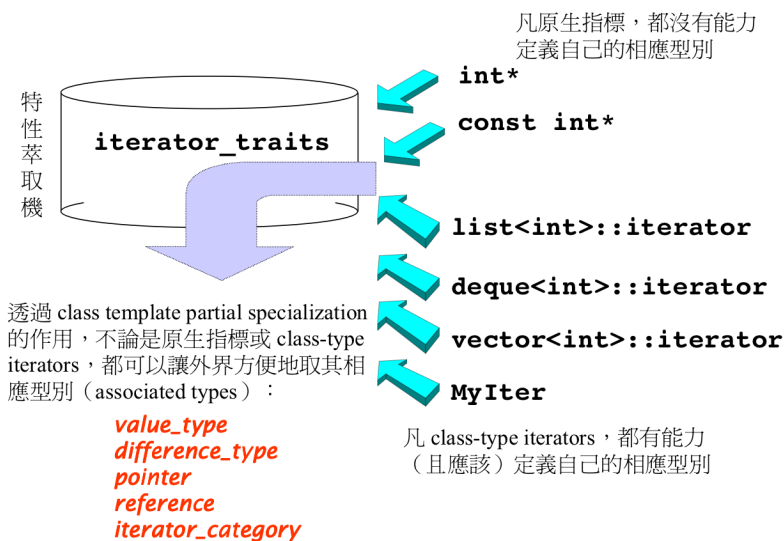


圖 3-1 traits 就像一台「特性萃取機」，擷取各個迭代器的特性（相應型別）。

C++的一些语法让人看着费解，其中就有：

```
typedef typename std::vector<T>::size_type size_type;  
1
```

详见《C++ Primer》（第五版）P584

有些不懂的语法有时候虽然知道大概是什么意思，忽略一下就过了其实，不过心里老是膈应，就刨根问底一次吧。

```
vector::size_type
```

明白上述语法，首先要先看清vector::size_type的意思。参考《STL源码剖析》不难发现，其实：

```
template <class T, class Alloc=alloc>  
class vector{  
public:  
    //...  
    typedef size_t size_type;  
    //...  
};
```

这样就看得很清晰了，vector::size_type是vector的嵌套类型定义，其实际等价于size_t类型。
也就是说：

```
vector<int>::size_type ssize;  
//就等价于  
size_t ssize;
```

为什么使用typename关键字

那么问题来了，为什么要加上typename关键字？

```
typedef std::vector<T>::size_type size_type; //why not?  
1
```

实际上，模板类型在实例化之前，编译器并不知道vector<T>::size_type是什么东西，事实上一共有三种可能：

静态数据成员

静态成员函数

嵌套类型

那么此时typename的作用就在此时体现出来了——定义就不再模棱两可。

总结

所以根据上述两条分析,

```
typedef typename std::vector<T>::size_type size_type;
```

语句的真是面目是：

typedef创建了存在类型的别名，而typename告诉编译器std::vector<T>::size_type是一个类型而不是一个成员。