Lesson09--STL进阶之STL总结

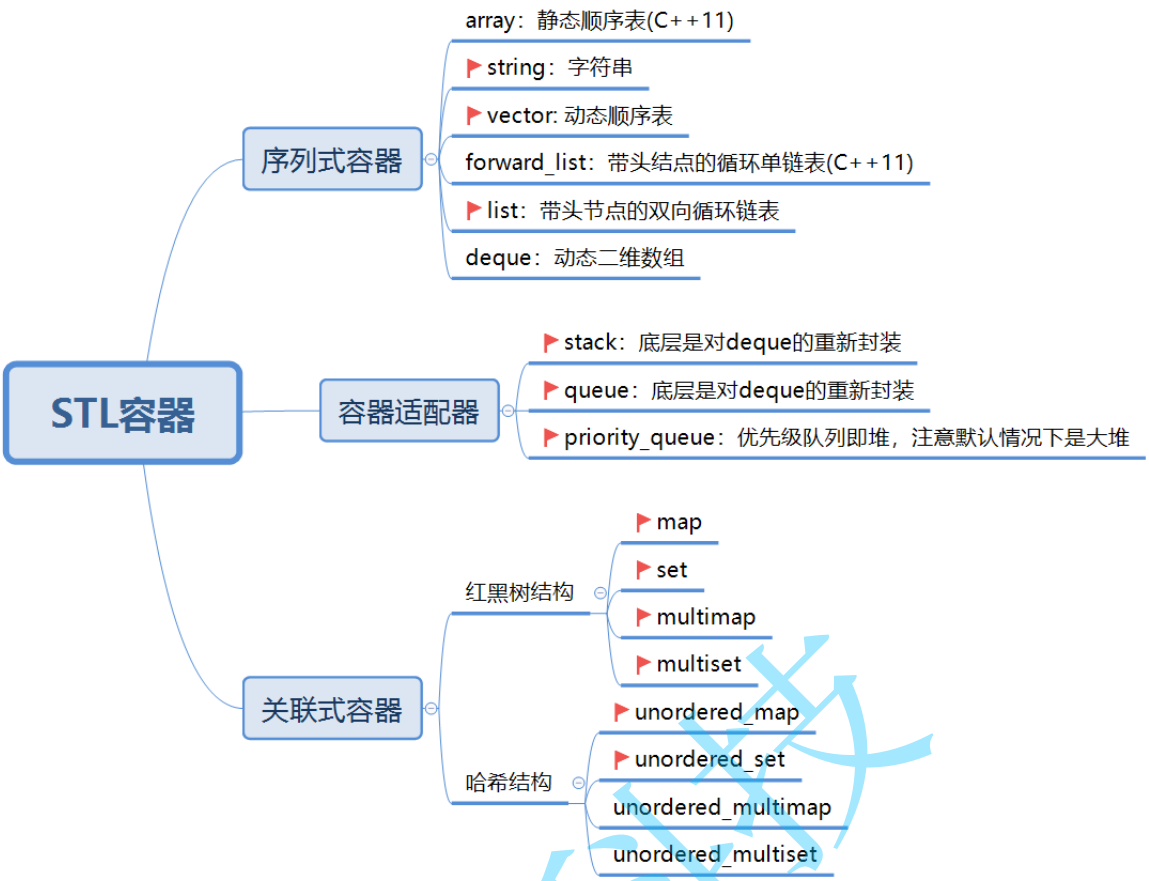
1. STL的本质

> 通过前面的学习以及使用，我们对STL已经有了一定的认识。**通俗说：STL是Standard Template Library(标准模板库)，是高效的C++程序库，其采用泛型编程思想对常见数据结构(顺序表，链表，栈和队列，堆，二叉树，哈希)和算法(查找、排序、集合、数值运算...)等进行封装，里面处处体现着泛型编程程序设计思想以及设计模式，已被集成到C++标准程序库中。**具体说：**STL中包含了容器、适配器、算法、迭代器、仿函数以及空间配置器。**STL设计理念：追求代码高复用性以及运行速度的高效率，在实现时使用了许多技术，因此熟悉STL不仅对我们正常使用有很大帮助，而且对自己的知识也有一定的提高。

2. STL的六大组件

2.1 容器

> 容器，置物之所也。**STL中的容器，可以划分为两大类：序列式容器和关联式容器。**



必备技能：

1. 熟悉每个容器的常用接口以及帮助文档查阅，并能熟练使用，建议再刷题以及写项目时多多应用，熟能生巧。
2. 熟悉每个容易的底层结构、实现原理以及应用场景，比如：红黑树、哈希桶
3. 熟悉容器之间的区别：比如vector和list区别？map和set区别？map和unordered_map的区别？

2.2 算法

算法：问题的求解步骤，以有限的步骤，解决数学或逻辑中的问题。STL中的算法主要分为两大类：与数据结构相关算法(容器中的成员函数)和通用算法(与数据结构不相干)。**STL中通用算法总共有70多个，主要包含：排序，查找，排列组合，数据移动，拷贝，删除，比较组合，运算等。**以下只列出了部分常用的算法：

算法名称	算法功能
accumulate	**元素统计**
binary_search	二分查找
copy	拷贝
copy_backward	逆向拷贝
copy_n	拷贝n个元素
count	计数
count_if	在特定条件下计数
equal	判断两个区间相等与否
fill	填充元素
fill_n	填充元素n次
find	循环查找
find_if	循环查找符合特定条件元素
find_end	查找某个子序列的最后一次出现点
find_first_of	查找某个元素首次出现点
for_each	对区间内的每隔一元素实行某种操作
is_heap	判断某区间是否为一个heap
is_sorted	判断某区间是否已排序
lexicographical_compare	以字典顺序进行比较
max	获取最大值
max_element	最大值所在位置
merge	合并两个序列
min	获取最小值
min_element	最小值所在位置
next_permutation	获取下一个排列组合
pre_permutation	获取前一个排列组合
partial_sort	局部排序
partial_sum	局部求和

算法名称	算法功能
partition	分割
remove	删除某类元素
remove_copy	删除某类元素并将结果拷贝到另一个容器中
remove_if	有条件的删除某类元素
replace	替换某类元素
replace_if	有条件的替换
reverse	反转序列
sort	排序(不稳定)
stable_partition	分割并保持元素的相对次序
stable_sort	分割并保持相等元素的相对位置(稳定排序算法)
unique	取出重复性元素
make_heap	创建堆
push_heap	堆插入
pop_heap	堆删除
sort_heap	堆排序

STL算法总结

必备技能：

1. 熟悉常用算法的作用，并熟练使用
2. 熟悉房间算法时间复杂&空间复杂度求解方式

2.3 迭代器

2.3.1 什么是迭代器

迭代器是一种设计模式，**让用户通过特定的接口访问容器的数据，不需要了解容器内部的底层数据结构。**
C++中迭代器本质：是一个指针，让该指针按照具体的结构去操作容器中的数据。

2.3.1 什么需要迭代器

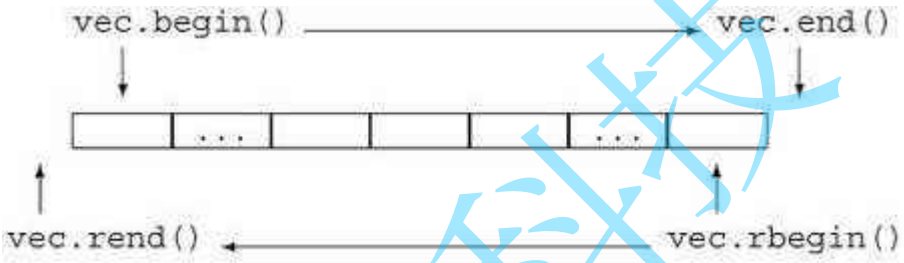
通过前面算法的学习了解到：STL中算法分为容器相关联与通用算法。所谓通用算法，即与具体的数据结构无关，比如：

```
template<class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last, const T& value )
{
    for ( ;first!=last; first++)
        if ( *first==value )
            break;

    return first;
}
```

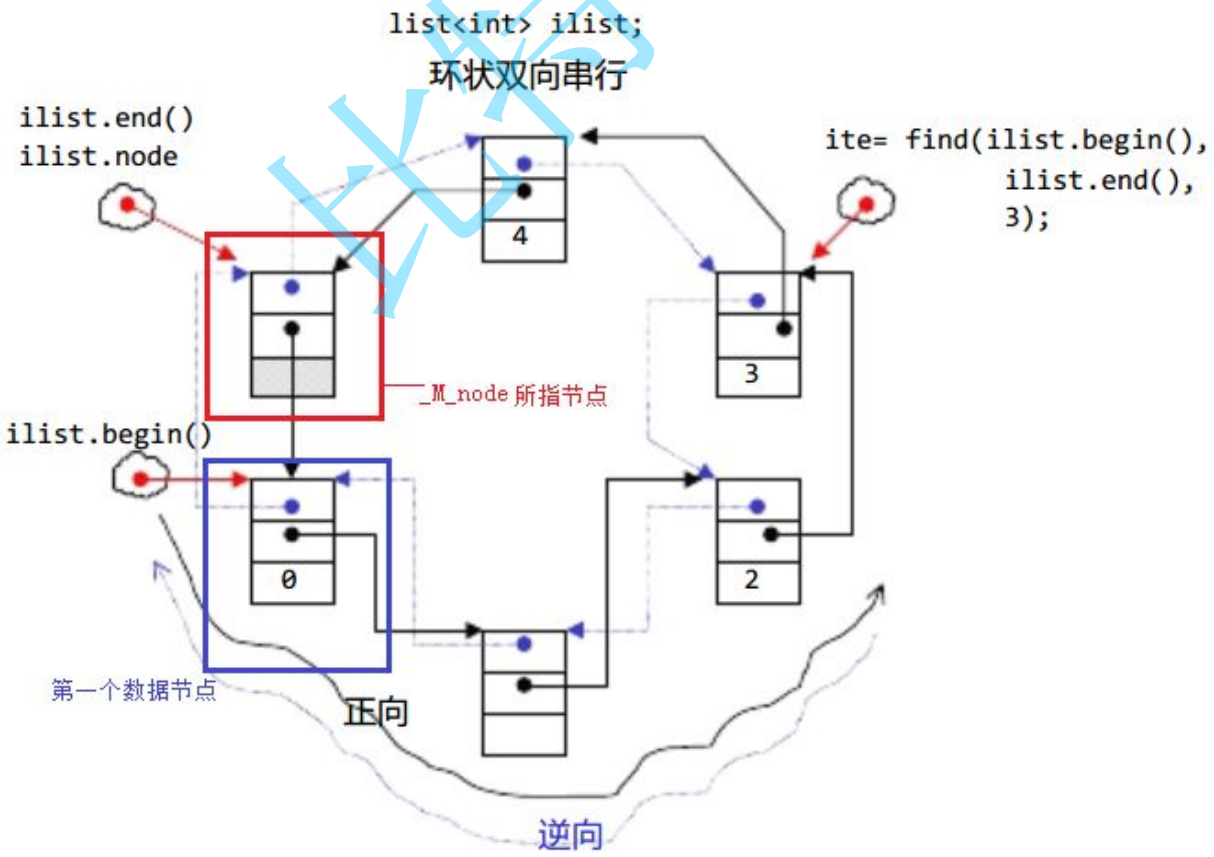
find算法在查找时，与具体的数据结构无关，只要给出待查找数据集合的范围，find就可在该范围中查找，找到返回该元素在区间中的位置，否则返回end。

问题：对于vector、list、deque、map、unordered_set等容器，其底层数据结构均不相同，那find算法是怎么统一向后遍历呢?

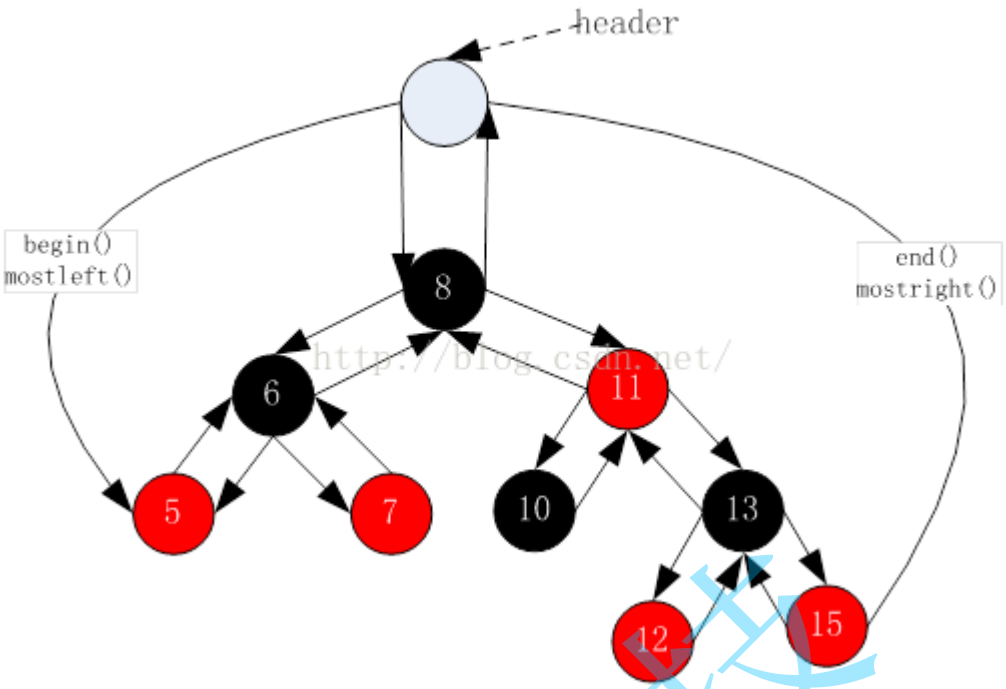
vector的底层结构：



list的底层结构：



map的底层结构：



2.3.2 迭代器应该由谁负责提供

每个容器的底层结构都不同，为了降低算法使用容器时的复杂度，底层结构应该对于算法透明，迭代器就充当了算法与容器之间的转接层，因此：**每个容器的迭代器应该由容器设计者负责提供，然后容器按照约定给出统一的接口即可。**

比如：

```
// vector中:
typedef T* iterator;
iterator begin();
iterator end();
find(v.beign(), v.end(), 5);

// list中
typedef list_iterator<T, T&, T*> iterator;
iterator begin();
iterator end();
find(l.begin(), l.end(), 5);
```

2.3.3 迭代器实现原理

容器底层结构不同，导致其实现原理不同，**容器迭代器的设计，必须结合具体容器的底层数据结构。**比如：

1. vector

   因为vector底层结构为一段连续空间，迭代器前后移动时比较容易实现，因此**vector的迭代器实际是对原生态指针的封装，即：typedef T* iterator。**

2. list

list底层结构为**带头结点的双向循环链表，迭代器在移动时，只能按照链表的结构前后依次移动，因此链表的迭代器需要对原生态的指针进行封装，因为当对迭代器++时，应该通过节点中的next指针域找到下一个节点。**

```
template<class T, class Ref, class Ptr>
struct __list_iterator
{
  typedef __list_iterator<T, T&, T*>             iterator;
  typedef __list_iterator<T, const T&, const T*> const_iterator;
  typedef __list_iterator<T, Ref, Ptr>           self;

  typedef bidirectional_iterator_tag iterator_category;
  typedef T value_type;
  typedef Ptr pointer;
  typedef Ref reference;
  typedef __list_node<T>* link_type;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;

  link_type node;

  __list_iterator(link_type x)
      : node(x)
  {}

  __list_iterator()
  {}

  __list_iterator(const iterator& x)
      : node(x.node)
  {}

  bool operator==(const self& x) const
  {
      return node == x.node;
  }

  bool operator!=(const self& x) const
  {
      return node != x.node;
  }

  reference operator*() const
  {
      return (*node).data;
  }

  pointer operator->() const
  {
      return &(operator*());
  }

  self& operator++()
```

```
    {
      node = (link_type)((*node).next);
      return *this;
    }

    self operator++(int)
    {
      self tmp = *this;
      ++*this;
      return tmp;
    }

    self& operator--()
    {
      node = (link_type)((*node).prev);
      return *this;
    }

    self operator--(int)
    {
      self tmp = *this;
      --*this;
      return tmp;
    }
};
```

如果迭代器不能直接使用原生态指针操作底层数据时，必须要对指针进行封装，在封装时需要提供一下方法：

1. 迭代器能够像指针一样方式进行使用：**重载pointer operator*() / reference operator->()**
2. 能够让迭代器移动

   向后移动：**self& operator++() / self operator++(int)**

   向前移动：**self& operator--() / self operator--(int)** (注意：有些容器不能向前移动，比如forward_list)
3. 支持比较-因为在遍历时需要知道是否移动到区间的末尾

   bool operator!=(const self& it)const / bool operator==(const self& it)const

2.3.4 迭代器与类的融合

1. 定义迭代器类
2. 在容器类中统一迭代器名字

```
// 比如list:
template <class T, class Alloc = alloc>
class list
{
    // ...
    typedef __list_iterator<T, T&, T*>             iterator;

    // ...
};
```

3. 在容器类中添加获取迭代器范围的接口：

```
template <class T, class Alloc = alloc>
class list
{
    // ...
    iterator begin()
    {
        return (link_type)((*node).next);
    }

    iterator end()
    {
        return node;
    }

    // ...
};
```

2.3.5 反向迭代器

反向迭代器：正向迭代器的适配器，即正向迭代器++往end方向移动，--往begin方向移动，而反向迭代器++则往begin方向移动，--则向end方向移动。

```
template <class Iterator>
class reverse_iterator
{
public:
  // ...
  typedef typename iterator_traits<Iterator>::pointer pointer;
  typedef typename iterator_traits<Iterator>::reference reference;
  typedef Iterator iterator_type;
  typedef reverse_iterator<Iterator> self;

public:
  reverse_iterator()
  {}

  explicit reverse_iterator(iterator_type x)
      : current(x)
  {}
```

```
  reverse_iterator(const self& x)
      : current(x.current)
  {}

  reference operator*() const
  {
    Iterator tmp = current;
    return *--tmp;
  }

  pointer operator->() const
  {
      return &(operator*());
  }

  self& operator++()
  {
    --current;
    return *this;
  }

  self operator++(int)
  {
    self tmp = *this;
    --current;
    return tmp;
  }

  self& operator--()
  {
    ++current;
    return *this;
  }

  self operator--(int)
  {
    self tmp = *this;
    ++current;
    return tmp;
  }

    // ...
protected:
  Iterator current;
};
```


2.3.4 迭代器萃取(了解)

> 该部分内容请同学们参考：《STL源码剖析》
>
> 迭代器萃取

2.4 适配器

> 适配器：**又接着配接器，是一种设计模式**，简单的说：需要的东西就在眼前，但是由于接口不对而无法使用，需要对其接口进行转化以方便使用。即：**将一个类的接口转换成用户希望的另一个类的接口，使原本接口不兼容的类可以一起工作。**

STL中适配器总共有三种类型：

- **容器适配器-stack和queue**

  stack的特性是后进先出，queue的特性为先进先出，该种特性deque的接口完全满足，因此stack和queue在底层将deque容器中的接口进行了封装。

```
template < class T, class Container = deque<T> >;
class stack
{
    // ...
};

template < class T, class Container = deque<T> >
class queue
{
    // ...
}
```

- **迭代器适配器-反向迭代器**

  反向迭代器++和--操作刚好和正向迭代器相反，因此：反向迭代器只需将正向迭代器进行重新封装即可。

- **函数适配器**

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;

class IsOdd
{
public:
    bool operator()(int x)
    {
```

```
        return 1 == x % 2;
    }
};

int main()
{
    vector<int> v{1, 2, 3, 4, 5};

    // 计算区间中奇数的个数
    cout << count_if(v.begin(), v.end(), IsOdd()) << endl;

    //计算奇数元素的个数
    // 这里的bind2nd将二元函数对象modulus转换为一元函数对象。
    //bind2nd(op, value) (param)相当于op(param, value)
    cout << count_if(v.begin(), v.end(),
                     bind2nd(modulus<int>(), 2)) << endl;


    //bind1st(op, value)(param)相当于op(value, param);
    cout << count_if(v.begin(), v.end(),
                     bind1st(less<int>(), 4)) << endl;

    return 0;
}
```

函数适配器

2.5 仿函数

仿函数：一种具有函数特征的对象，调用者可以像函数一样使用该对象，**为了能够“行为类似函数”，该对象所在类必须自定义函数调用运算符operator()，重载该运算符后，就可在仿函数对象的后面加上一对小括号，以此调用仿函数所定义的operator()操作**，就其行为而言，“仿函数”一次更切贴。

仿函数一般配合算法，作用就是：提高算法的灵活性。

```
#include <vector>
#include <algorithm>

class Mul2
{
public:
    void operator()(int& data)
    {
        data <<= 1;
    }
};

class Mod3
{
public:
    bool operator()(int data)
    {
        return 0 == data % 3;
```

```
    }
};

int main()
{
    // 给容器中每个元素乘2
    vector<int> v{1,2,3,4,5,6,7,8,9,0};
    for_each(v.begin(), v.end(), Mul2());
    for (auto e : v)
        cout << e << " ";
    cout << endl;

    // 删除容器中3的倍数
    auto pos = remove_if(v.begin(), v.end(), Mod3());
    v.erase(pos, v.end());

    // 将容器中的元素打印出来
    // 注意：对于功能简单的操作，可以使用C++11提供的lambda表达式来代替
    // lambda表达式实现简单，其在底层与仿函数的原理相同，编译器会将lambda表达式转换为仿函数
    for_each(v.begin(), v.end(), [](int data){cout << data << " "; });
    cout << endl;
    return 0;
}
```

2.6 空间配置器

> 请参考：Lesson08--STL总结之空间配置器

3. STL框架

仿函数(函数对象):可以灵活配置的小部件，使得算法和容器更加灵活

实现原理：在类中实现operator(参数列表)
即可向使用对象来使用函数

通用算法：<algorithm>
数值算法
set相关算法
heap相关算法
.....

```
template<class Iterator, class Compare>
void srot(Iterator first, Iterator last, Compare = Less<T>)
```

```
InputIterator find(InputIterator first, InputIterator last, const T& value)
```

函数适配器

迭代器榨汁机----->用相应的迭代器实例化iterator_traits<容器自定义迭代器类型>
<原生态指针>

迭代器适配器
revserve_iterator

```
template <class Iterator>
struct iterator_traits
{
  typedef typename Iterator::iterator_category iterator_category;
  typedef typename Iterator::value_type        value_type;
  typedef typename Iterator::difference_type   difference_type;
  typedef typename Iterator::pointer           pointer;
  typedef typename Iterator::reference         reference;
};
```

```
template <class T>
struct iterator_traits<T*>
{
  typedef random_access_iterator_tag
  typedef T                  iterator_category;
  typedef ptrdiff_t          value_type;
  typedef T*                 difference_type;
  typedef T&                 pointer;
                             reference;
};
```

适配器

T* list_iterator deque_iterator rb_tree_iterator hash_iterator

C++11

vector list foward_list(C++11) deque map/multimap--->(key,value) unordered_map/unordered_multimap
set/multiset--->(value,value) unordered_set/unordered_multiset

顺序表 双向循环链表 单链表 顺序表+链表=杂交体 红黑树 哈希表

容器适配器
stack------>deque
queue------>deque
priority_queue----->heap

simple_alloc<T, alloc>

stl_alloc.h

空间配置器：配置、管理和释放空间

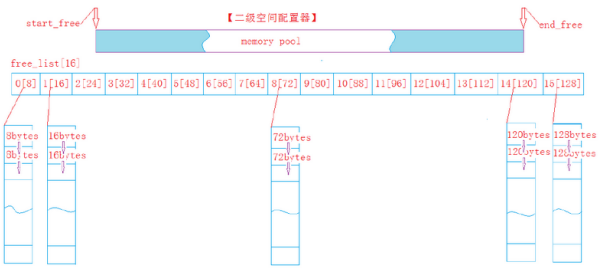
```
template<class T, class Alloc>
class simple_alloc
{
.....
    void* allocate(void);          // 申请单个对象
    void* allocate(size_t n);      // 申请n个对象

    void deallocate(T* p);         // 释放单个对象
    void deallocate(T* p, size_t n) // 释放n个对象
}
```

一级空间配置器 _malloc_alloc_template--------->alloc

处理大块内存，封装malloc和free
并模拟实现c++ set_new_handle机制

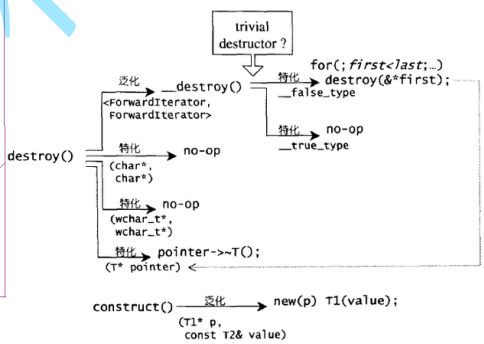
二级空间配置器 __default_alloc_template-------->alloc

为解决频繁向系统申请小块内存所引起的内存碎片、效率和额外开销问题
采用内存池的实现技术



stl_construct.h

空间的构造销毁--->完成 T类型 对象的初始化和空间销毁



construct() 泛化 new(p) T1(value);
(T1* p, const T2& value)