Lesson11--STL进阶之STL总结

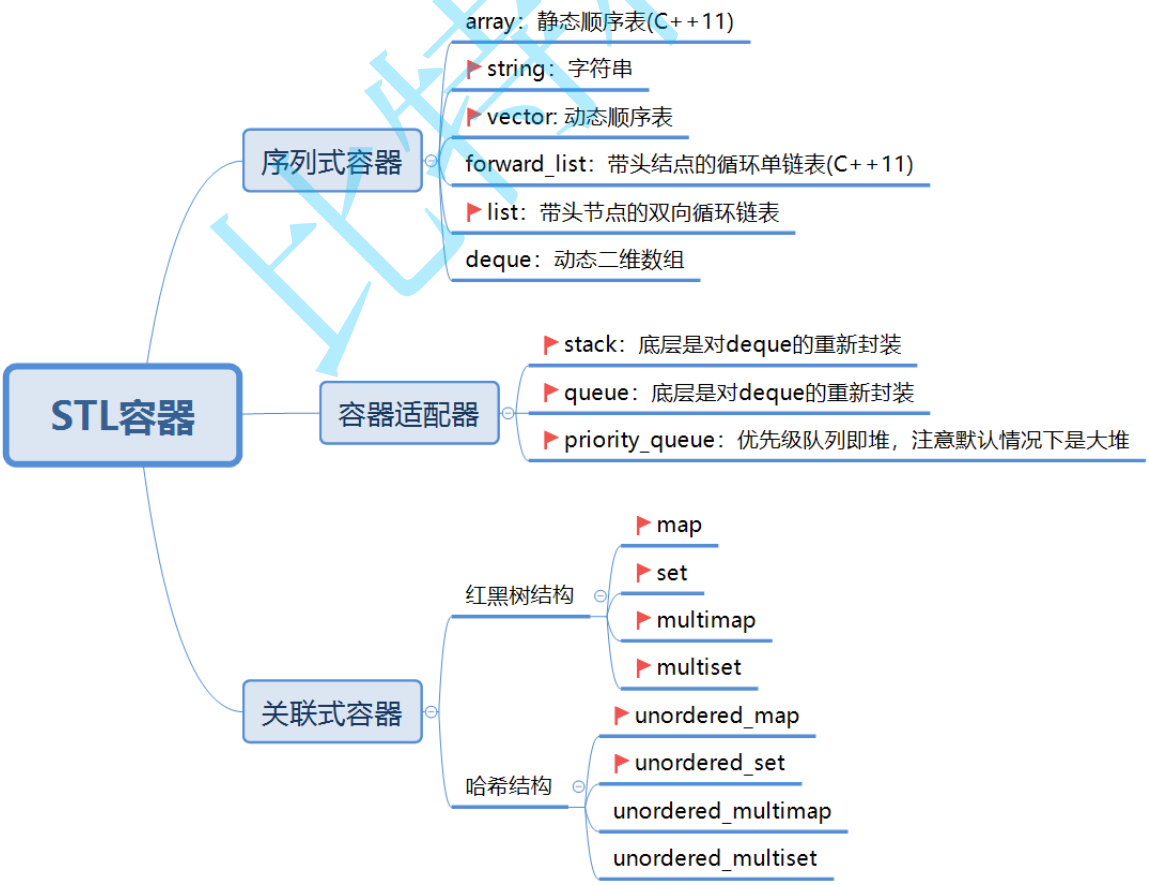
1. STL的本质

> 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：

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. 熟悉常见算法时间复杂&空间复杂度求解方式

常用算法举例：

1. **accumulate**

   该算法作用是对区间中的元素进行累加。有以下两个版本：

```
// 对[first, last)区间中元素在init的基础上进行累加
template <class InputIterator, class T>
T accumulate ( InputIterator first, InputIterator last, T init );

// 对[fist, last)区间中元素在init的基础上按照binary_op指定的操作进行累加
template <class InputIterator, class T, class BinaryOperation>
T accumulate ( InputIterator first, InputIterator last, T init,
                  BinaryOperation binary_op );
```

   注意：使用时必须添加头文件。

```
#include <numeric>
#include <vector>

struct Mul2
{
    int operator()(int x, int y) { return x + 2 * y; }
};

int main()
{
    // 对区间中的元素进行累加
    vector<int> v{ 10, 20, 30 };
    cout << accumulate(v.begin(), v.end(), 0)<<endl;

    // 对区间中的每个元素乘2，然后累加
    cout << accumulate(v.begin(), v.end(), 0, Mul2()) << endl;
    return 0;
}
```

2. **count与count_if**

该算法的作用是统计区间中某个元素出现的次数。

```
// 统计value在区间[first, last)中出现的次数
template <class InputIterator, class T>
ptrdiff_t count ( InputIterator first, InputIterator last, const T& value )
{
    ptrdiff_t ret=0;
    while (first != last) if (*first++ == value) ++ret;
    return ret;
}

// 统计满足pred条件的元素在[first, last)中的个数
template <class InputIterator, class Predicate>
ptrdiff_t count_if ( InputIterator first, InputIterator last, Predicate pred )
{
    ptrdiff_t ret=0;
    while (first != last) if (pred(*first++)) ++ret;
    return ret;
}
```

注意：使用时必须添加头文件。

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

bool IsOdd(int i)
{ return ((i % 2) == 1); }

int main()
{
    // 统计10在v1中出现的次数
    vector<int> v1{ 10, 20, 30, 30, 20, 10, 10, 20 };
    cout << count(v1.begin(), v1.end(), 10) << endl;

    // 统计v2中有多少个偶数
    vector<int> v2{0,1,2,3,4,5,6,7,8,9};
    cout << count_if(v2.begin(), v2.end(), IsOdd) << endl;
    return 0;
}
```

3. **find、find_if**

该算法的作用是**找元素在区间中第一次出现的位置**

```
// 在[first, last)中查找value第一次出现的位置，找到返回该元素的位置，否则返回last
// 时间复杂度O(N)
template<class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last, const T& value )
{
    for ( ;first!=last; first++) if ( *first==value ) break;
```

```
    return first;
}

// 在[first, last)中查找满足pred条件的元素第一次出现的位置，找到返回该位置，否则返回last
// 时间复杂度O(N)
template<class InputIterator, class Predicate>
InputIterator find_if ( InputIterator first, InputIterator last, Predicate pred )
{
    for ( ; first!=last ; first++ ) if ( pred(*first) ) break;
    return first;
}
```

注意：使用时必须包含头文件

4. **max和min**

   max返回两个元素中较大值，min返回两个元素中较小值。

```
template <class T>
const T& max(const T& a, const T& b)
{
  return (a<b)?b:a;
}

template <class T>
const T& min(const T& a, const T& b)
{
  return !(b<a)?a:b;
}
```

注意：使用时必须包含头文件

5. **merge**

   该算法作用将两个有序序列合并成一个有序序列，使用时必须包含头文件。

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge ( InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       OutputIterator result )
{
    while (true)
    {
      *result++ = (*first2<*first1)? *first2++ : *first1++;
       if (first1==last1) return copy(first2,last2,result);
       if (first2==last2) return copy(first1,last1,result);
  }
}

#include <algorithm>
#include <vector>
#include <list>

int main()
```

```
{
    vector<int> v{ 2, 6, 5, 8 };
    list<int> L{ 9, 3, 0, 5, 7 };

    sort(v.begin(), v.end());
    L.sort();

    vector<int> vRet(v.size() + L.size());
    merge(v.begin(), v.end(), L.begin(), L.end(), vRet.begin());

    for (auto e : vRet)
        cout << e << " ";
    cout << endl;
    return 0;
}
```

注意：

- **使用时必须保证区间有序**
- **时间复杂度为O(M+N)**

6. **partial_sort**

该算法的作用是：找TOPK

```
// 在区间[first, last)中找前middle-first个最小的元素，并存储在[first, middle)中
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);

// 在[first, last)中找前middle-first个最大或者最小的元素，并存储在[first, middle)中
template <class RandomAccessIterator, class Compare>
  void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                    RandomAccessIterator last, Compare comp);
```

partial_sort的实现原理是：**对原始容器内区间为[first, middle)的元素执行make_heap()操作构造一个最大堆，然后拿[middle, last)中的每个元素和first进行比较，first内的元素为堆内的最大值。如果小于该最大值，则互换元素位置，并对[first, middle)内的元素进行调整，使其保持最大堆序。比较完之后在对[first, middle)内的元素做一次对排序sort_heap()操作，使其按增序排列。**注意，堆序和增序是不同的。因此该算法的功能实际就是：TOP-K

注意：使用时必须包含头文件

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

int main()
{
    // 找该区间中前4个最小的元素，元素最终存储在v1的前4个位置
    vector<int> v1{ 4, 1, 8, 0, 5, 9, 3, 7, 2, 6 };
    partial_sort(v1.begin(), v1.begin() + 4, v1.end());

```

```
    // 找该区间中前4个最大的元素，元素最终存储在v1的前4个位置
    vector<int> v2{ 4, 1, 8, 0, 5, 9, 3, 7, 2, 6 };
    partial_sort(v2.begin(), v2.begin() + 4, v2.end(), greater<int>());
    return 0;
}
```

7. partition

该算法的作用是按照条件对区间中的元素进行划分，使用时必须包含头文件。

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last, Predicate pred)
{
     while (true)
     {
        while (first!=last && pred(*first)) ++first;
        if (first==last--) break;
        while (first!=last && !pred(*last)) --last;
        if (first==last) break;
        swap (*first++,*last);
      }
      return first;
}
```

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

bool IsOdd(int i)
{ return (i % 2) == 1; }

int main()
{
    vector<int> v{0,1,2,3,4,5,6,7,8,9};
    // 将区间中元素分割成奇数和偶数两部分
    auto div = partition(v.begin(), v.end(), IsOdd);

    // 打印[begin, div)的元素
    for (auto it = v.begin(); it != div; ++it)
        cout << *it <<" ";
    cout << endl;

    // 打印[div, end)的元素
    for (auto it = div; it != v.end(); ++it)
        cout << " " << *it;
    cout << endl;

    return 0;
}
```

8. **reverse**

该算法的作用是对区间中的元素进行逆置，使用时必须包含头文件。

```
template <class BidirectionalIterator>
void reverse ( BidirectionalIterator first, BidirectionalIterator last)
{
  while ((first!=last)&&(first!=--last))
    swap (*first++,*last);
}
```

9. **sort(重要)**

排序在实际应用中需要经常用到，而在目前的排序中，**快排平均情况下是性能最好的一种排序**，但是快排也有其自身的短板，比如说：**元素接近有序、元素量比较大的情况下**，直接使用快排时，堪称一场灾难。因此**STL中sort算法并没有直接使用快排，而是针对各种情况进行了综合考虑。**下面关于sort函数分点进行说明：

1. sort函数提供了两个版本
   - sort(first, last)：默认按照小于方式排序，排序结果为升序，一般用排内置类型数据
   - sort(first, last, comp)：可以通过comp更改元素比较方式，即可以指定排序的结果为升序或者降序，一般以仿函数对象和函数指针的方式提供
2. sort并不是一种排序算法，而是将多个排序算法混合而成
3. 当元素个数少于__stl_threshold阈值时(16)，使用直接插入排序处理
4. 当元素个数超过__stl_threshold时，考虑是否能用快排的方式排序，因为当元素数量达到一定程度，递归式的快排可能会导致栈溢出而崩，因此：
   - 通过__lg函数判断递归的深度

     ```
     template <class Size>
     inline Size __lg(Size n)
     {
         Size k;
         for (k = 0; n > 1; n >>= 1) ++k;
         return k;
     }
     ```

   - 如果递归的深度没有超过2*$log_2 N$时，则使用快排方式排序，注意：快排时使用到了三数取中法预防分割后一边没有数据的极端情况
   - 如果递归深度超过2*$log2_N$时，说明数据量大，递归层次太深，可能会导致栈溢出，此时使用堆排序处理。

   STL排序算法sort的详细说明

   sort面试题

10. **unique**

该函数的作用是删除区间中相邻的重复性元素，确保元素唯一性，注意在使用前要保证区间中的元素是有序的，才能达到真正的去重。

```
// 元素不相等时，用后一个将前一个元素覆盖掉
template <class ForwardIterator>
```

```
ForwardIterator unique ( ForwardIterator first, ForwardIterator last );

// 如果元素不满足pred条件时，用后一个将前一个覆盖掉
template <class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique ( ForwardIterator first, ForwardIterator last,
                           BinaryPredicate pred );

template <class ForwardIterator>
ForwardIterator unique ( ForwardIterator first, ForwardIterator last )
{
    ForwardIterator result=first;
    while (++first != last)
    {
      if (!(*result == *first))  // or: if (!pred(*result,*first)) for the pred version
        *(++result)=*first;
    }
    return ++result;
}
```

注意：

1. 该函数并不是将重复性的元素删除掉，而是用后面不重复的元素将前面重复的元素覆盖掉了。
2. 返回值是一个迭代器，指向的是去重后容器中不重复最后一个元素的下一个位置。
3. 该函数需要配合erase才能真正的将元素删除掉

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

int main()
{
    vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
    auto it = unique(v.begin(), v.end());

    for (auto e : v)
        cout << e << " ";
    cout << endl;

    /*
    从打印的结果可以看出：
    1. unique并没有将所有重复的元素删除掉，而值删除了一个9，因为unique删除的是相邻的重复元素，而上述元素中只有一个9重复相邻
    2. unique删除时只是用后面元素将前面重复位置覆盖掉了，并没有达到真正删除，若要真正删除，还需要erase配合
    */
    v.erase(it, v.end());

    // 如果想将区间中所有重复性的元素删除掉，可以先对区间中的元素进行排序
    for (auto e : v)
        cout << e << " ";
    cout << endl;

    // 先对区间中的元素进行排序，另重复的元素放在相邻位置
```

```
    sort(v.begin(), v.end());
    for (auto e : v)
        cout << e << " ";
    cout << endl;

    // 使用unique将重复的元素覆盖掉
    it = unique(v.begin(), v.end());

    // 将后面无效的元素移除
    v.erase(it, v.end());
    for (auto e : v)
        cout << e << " ";
    cout << endl;
    return 0;
}
```

11. **next_permutation和pre_permutation(重要)**

    next_permutation是获取一个排序的下一个排列，可以遍历全排列，prev_permutation刚好相反，获取一个排列的前一个排列，使用时必须包含头文件

    对序列 {a, b, c}，每一个元素都比后面的小，按照字典序列，固定a之后，a比bc都小，c比b大，它的下一个序列即为{a, c, b}，而{a, c, b}的上一个序列即为{a, b, c}，同理可以推出所有的六个序列为：{a, b, c}、{a, c, b}、{b, a, c}、{b, c, a}、{c, a, b}、{c, b, a}，其中{a, b, c}没有上一个元素，{c, b, a}没有下一个元素。

    注意：使用时，必须保证序列是有序的。

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

int main()
{
    // 因为next_permutation函数是按照大于字典序获取下一个排列组合的
    // 因此在排序时必须保证序列是升序的
    vector<int> v = {4, 1, 2, 3 };
    sort(v.begin(), v.end());
    do
    {
        cout << v[0] << " " << v[1] << " " << v[2] <<" " <<v[3]<< endl;
    } while (next_permutation(v.begin(), v.end()));
    cout << endl;

    // 因为prev_permutation函数是按照小于字典序获取下一个排列组合的
    // 因此在排序时必须保证序列是降序的
    //sort(v.begin(), v.end());
    sort(v.begin(), v.end(), greater<int>());
    do
    {
        cout << v[0] << " " << v[1] << " " << v[2] << " " << v[3] << endl;
    } while (prev_permutation(v.begin(), v.end()));
    return 0;
}
```

2.3 迭代器

2.3.1 什么是迭代器

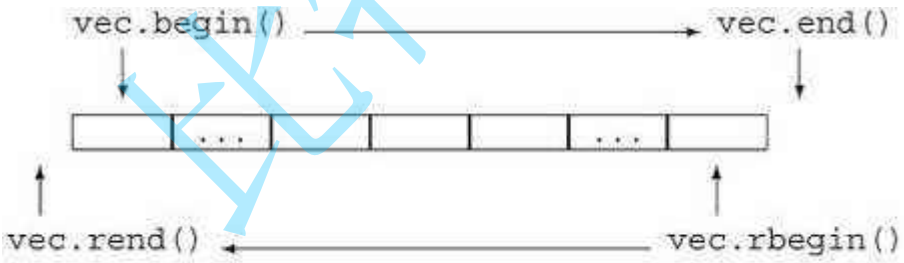
> 迭代器是一种设计模式，**让用户通过特定的接口访问容器的数据，不需要了解容器内部的底层数据结构。**
> C++中迭代器本质：是一个指针，让该指针按照具体的结构去操作容器中的数据。

2.3.1 为什么需要迭代器

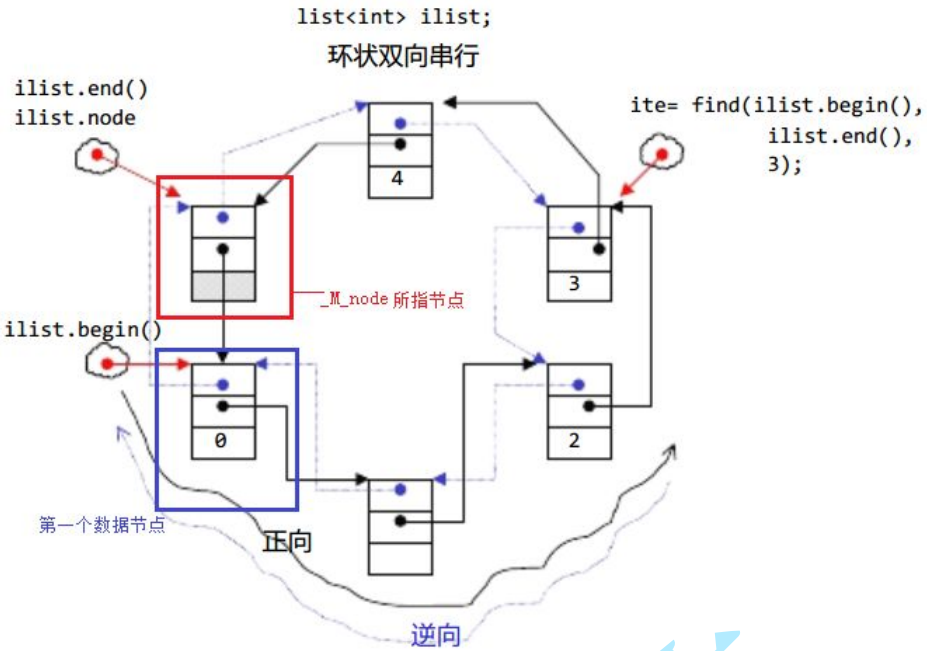
> 通过前面算法的学习了解到：STL中算法分为容器相关联与通用算法。所谓通用算法，即与具体的数据结构无关，比如：

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

    return first;
}
```

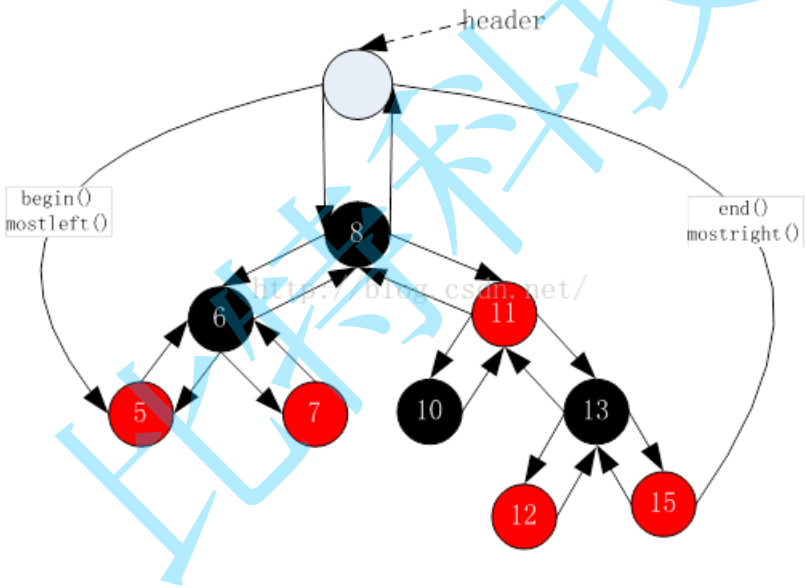
> find算法在查找时，与具体的数据结构无关，只要给出待查找数据集合的范围，find就可在该范围中查找，找到返回该元素在区间中的位置，否则返回end。
>
> 问题：对于vector、list、deque、map、unordered_set等容器，其底层数据结构均不相同，那find算法是怎么统一向后遍历呢?
>
> vector的底层结构：



> 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。**

   同学们参考下在C++初阶中：list迭代器的模拟实现。

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，在封装时需要提供以下方法：

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。

请同学们参考：C++初阶中list的反向迭代器模拟实现

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(了解)**

  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

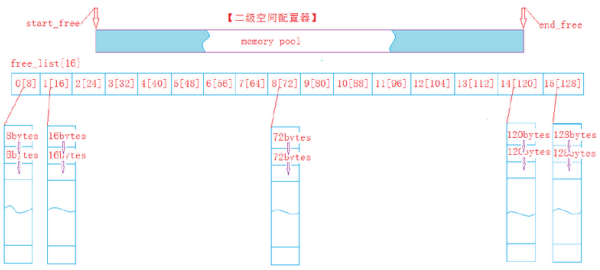
```
template<class T, class Alloc>
class simple_alloc
{
.....
    void* allocate(void);          // 申请单个对象
    void* allocate(size_t n);      // 申请n个对象

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

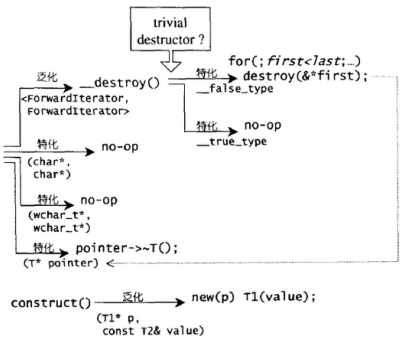
一级空间配置器 _malloc_alloc_template--------->alloc

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





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