GeekBand 极客班

互联网人才 + 油站!

STL与泛型编程

www.geekband.com

GeekBand **极客班** 互联网人才+油站：

极客班携手网易云课堂，针对热门IT互联网岗位，联合业内专家大牛，紧贴企业实际需求，量身打造精品实战课程。

专业课程 + 项目碾压 + 习题&辅导

- **顶尖大牛亲授**
- **贴合企业实际需求**
- **找对重点深挖学习**
- **紧贴课程内容**
- **全程实战操练**
- **作品就是最好的PASS卡**
- **学前导读**
- **周末直播答疑**
- **定期作业点评**
- **多项专题辅导**

www.geekband.com

Understanding the C++ Standard Template Library

—— 张文杰

课程内容

Part 5 泛型算法(Generic Algorithm)

Part 6 内存分配器(Memory Allocator)

Part 5 泛型算法

- 非变易算法(Non-mutating Algorithms)
- 变易算法(Mutating Algorithms)
- 排序(Sorting)
- 泛型数值算法(Generalized Numeric Algorithms)

非变易算法概述

- **非变易算法**是一系列模板函数，在**不改变**操作对象的前提下对元素进行处理，诸如：查找、子序列搜索、统计、匹配等等
- 具体包括:
  - for_each
  - find
  - find_if
  - adjacent_find
  - find_first_of
  - count
  - count_if
  - mismatch
  - equal
  - search

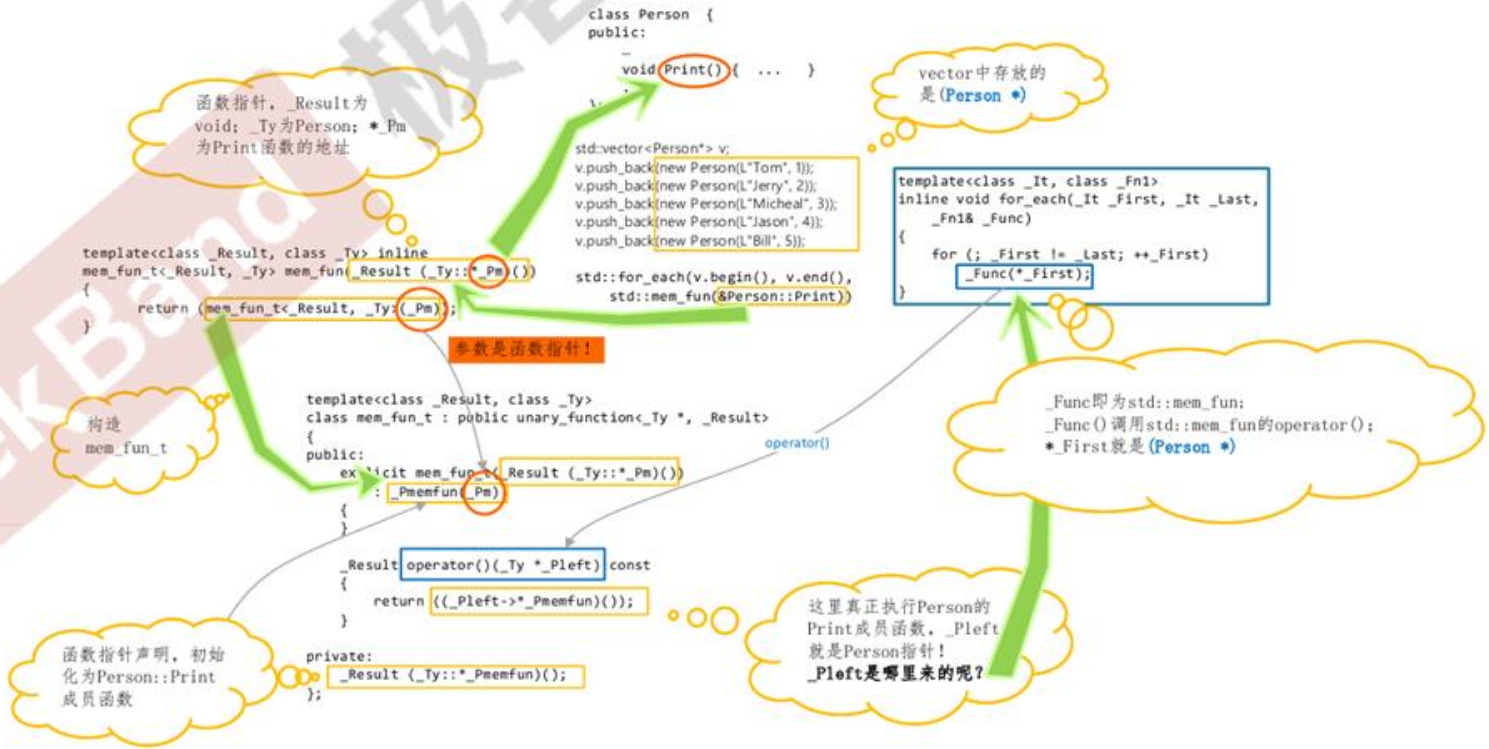
非变易算法 - for_each

- for_each：

```
template<class _InIt, class _Fn1> inline
_Fn1 for_each(_InIt _First, _InIt _Last, _Fn1 _Func)
```

- 在区间[_First, _Last)上对每一个元素应用_Func函数

- 参考“仿函数适配器(mem_fun/mem_fun_ref)”部分的详细说明



非变易算法 - find

- find：

```
template<class _InIt, class _Ty> inline
_InIt find(_InIt _First, _InIt _Last, const _Ty& _Val)
```

  - 对于it ∈ [_First, _Last)，如果*it == Value则返回该it；如果没找则返回_Last
  - 举例：

```
int elements[] = { 1, 2, 3, 23, 432, 596 };
std::vector<int> v(elements, elements + 6);
std::vector<int>::iterator it = std::find(v.begin(), v.end(), 23);
if (it != v.end())
{
    // Found!
}
```

非变易算法 – find_if

- find_if：

```
template<class _InIt, class _Pr> inline
_InIt find_if(_InIt _First, _InIt _Last, _Pr _Pred)
```

  - 对于it ∈ [_First, _Last)，如果_Pred(*it) == true则返回该it；如果没找则返回_Last
  - 举例：

```
int elements[] = { 1, 2, 3, 23, 432, 596 };
std::vector<int> v(elements, elements + 6);
std::vector<int>::iterator it = std::find_if(v.begin(), v.end(),
    std::bind2nd(std::greater<int>(), 100));
```

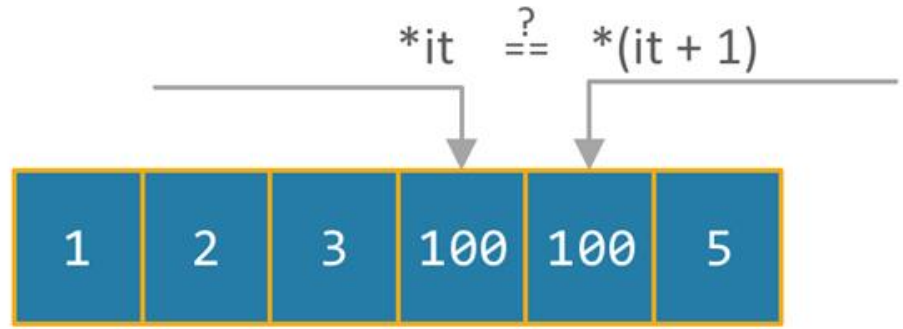
非变易算法 – adjacent_find(1)

- adjacent_find(1)：

```
template<class _FwdIt> inline
_FwdIt adjacent_find(_FwdIt _First, _FwdIt _Last)
```

  - 对于it ∈ [_First, _Last)，如果*it== *(it+ 1)则返回该it；如果没找则返回_Last
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
int n = sizeof(elements) / sizeof(int);
std::vector<int> v(elements, elements + n);
it = std::adjacent_find(v.begin(), v.end());
```



非变易算法 – adjacent_find(2)

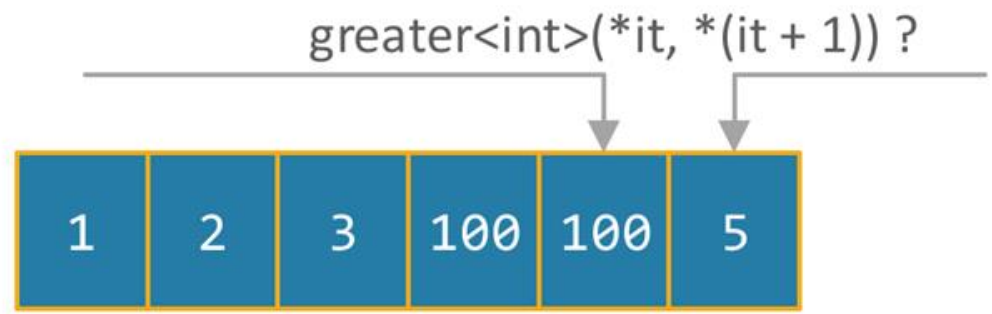
- adjacent_find(2)：

```
template<class _FwdIt, class _Pr> inline
_FwdIt adjacent_find(_FwdIt _First, _FwdIt _Last, _Pr _Pred)
```

  - 对于it ∈ [_First, _Last)，如果_Pred(*it, *(it + 1)) == true则返回该it；如果没找则返回_Last
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
it = std::adjacent_find(v.begin(), v.end(),
    std::greater<int>());

// output: 100
```



非变易算法 – find_first_of

- find_first_of (1)：

```
template<class _FwdIt1, class _FwdIt2> inline
_FwdIt1 find_first_of(_FwdIt1 _First1, _FwdIt1 _Last1, _FwdIt2 _First2, _FwdIt2 _Last2)
```

- 返回在区间[_First1, _Last1)中的it1，使得对于区间[_First2, _Last2)中某个it2，满足：*it1 == *it2；如果没找则返回_Last

- find first of (2)：

```
template<class _FwdIt1, class _FwdIt2, class _Pr> inline
_FwdIt1 find_first_of(_FwdIt1 _First1, _FwdIt1 _Last1, _FwdIt2 _First2, _FwdIt2 _Last2,
_Pr _Pred)
```

- 返回在区间[_First1, _Last1)中的it1，使得对于区间[_First2, _Last2)中某个it2，满足：_Pred(*it1, *it2) == true；如果没找则返回_Last

非变易算法 – count

- count:

```
template<class _InIt, class _Ty> inline
typename iterator_traits<_InIt>::difference_type
count(_InIt _First, _InIt _Last, const _Ty& _Val)
```

  - 返回在区间[_First1, _Last1)中满足(*it== _Val)的迭代器的个数
  - 举例:

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
int n = std::count(v.begin(), v.end(), 100);

// output: 2
```

非变易算法 – count_if

- count_if:

```
template<class _InIt, class _Ty, class _Pr> inline
typename iterator_traits<_InIt>::difference_type
count_if (_InIt _First, _InIt _Last, _Pr _Pred)
```

  - 返回在区间[_First1, _Last1)中满足_Pred(*it) == true的迭代器的个数
  - 举例:

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
int n = std::count_if(v.begin(), v.end(),
std::bind2nd(std::less<int>(), 100));

// output: 4
```

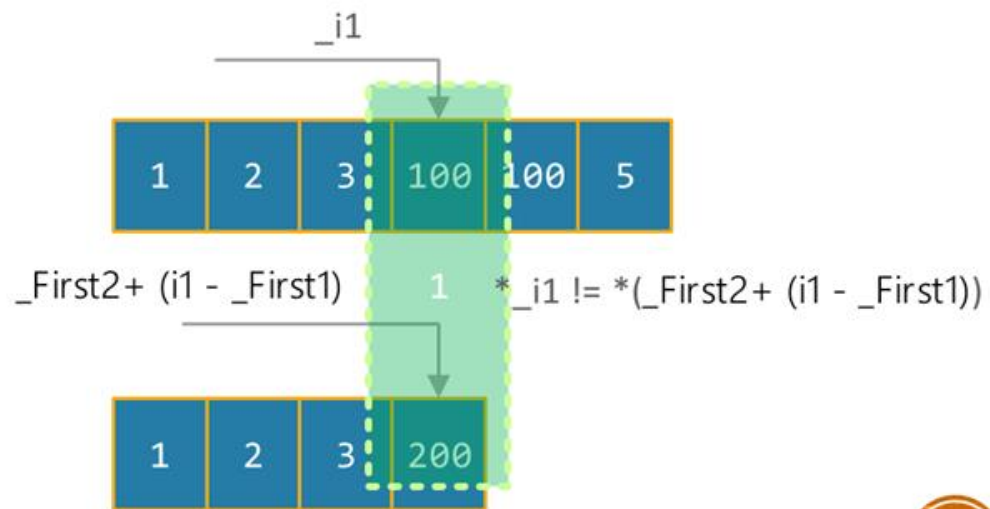
非变易算法 – mismatch(1)

- mismatch (1)：

```
template<class _InIt1, class _InIt2> inline
pair<_InIt1, _InIt2> mismatch (_InIt1 _First1, _InIt1 _Last1,
    _InIt2 _First2, _InIt2 _Last2)
```

  - 查找在区间[_First1, _Last1)中的i1，使得：*i1 != *(_First2 + (i1 - _First1))，返回pair<i1, _First2+ (i1 - _First1)>；如果找不到则返回pair<_Last1, _First2 + (_Last1 - _First1)>
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
int elements2[] = { 1, 2, 3, 200 };
std::vector<int> v(elements, elements + 6);
std::vector<int> v2(elements2, elements2 + 4);
typedef std::vector<int>::iterator IntIteartor;
std::pair<IntIteartor, IntIteartor> p =
   std::mismatch(v.begin(), v.end(),
       v2.begin(), v2.end());
```



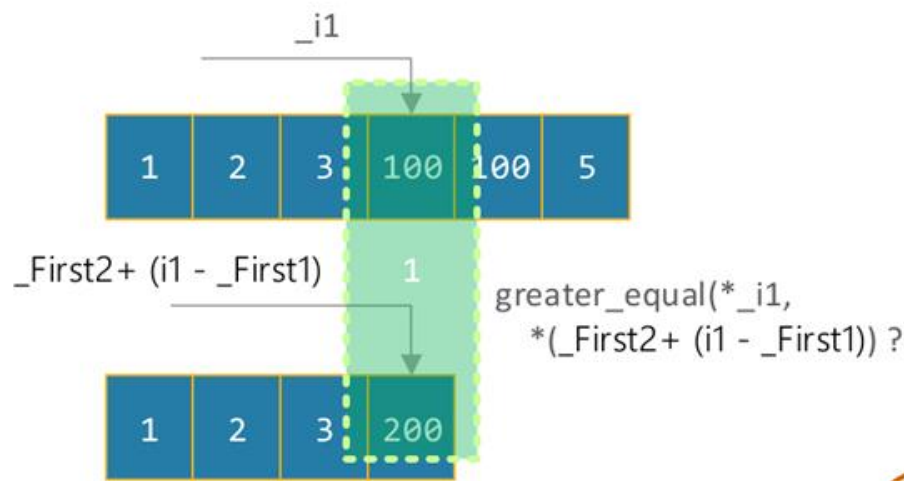
非变易算法 – mismatch(2)

- mismatch (2)：

```
template<class _InIt1, class _InIt2, class _Pr> inline
pair<_InIt1, _InIt2> mismatch (_InIt1 _First1, _InIt1 _Last1,
    _InIt2 _First2, _InIt2 _Last2, _Pr _Pred)
```

- 查找在区间[_First1, _Last1)中的i1，使得：_Pred(*i1 !, *(_First2 + (i1 - _First1))) == false，返回pair<i1, _First2+ (i1 - _First1)>；如果找不到则返回pair<_Last1, _First2 + (_Last1 - _First1)>
- 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
int elements2[] = { 1, 2, 3, 200 };
std::vector<int> v(elements, elements + 6);
std::vector<int> v2(elements2, elements2 + 4);
typedef std::vector<int>::iterator IntIteartor;
std::pair<IntIteartor, IntIteartor> p =
   std::mismatch(v.begin(), v.end(),
      v2.begin(), v2.end(),
      std::greater_equal<int>());
```



非变易算法 – equal

- equal(1):

```
template<class _InIt1, class _InIt2> inline
bool equal (_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _InIt2 _Last2)
```

  - 对于it ∈ [_First1, _Last1)，如果：*i1 == *(_First2 + (i1 - _First1))，则返回true；如果找不到则返回false.

- equal(2):

```
template<class _InIt1, class _InIt2, class _Pr> inline
bool equal (_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _InIt2 _Last2,
_Pr _Pred)
```

  - 对于it ∈ [_First1, _Last1)，如果：_Pred(*i1, *(_First2 + (i1 - _First1))) == true，则返回true；如果找不到则返回false.

非变易算法 – search(1)

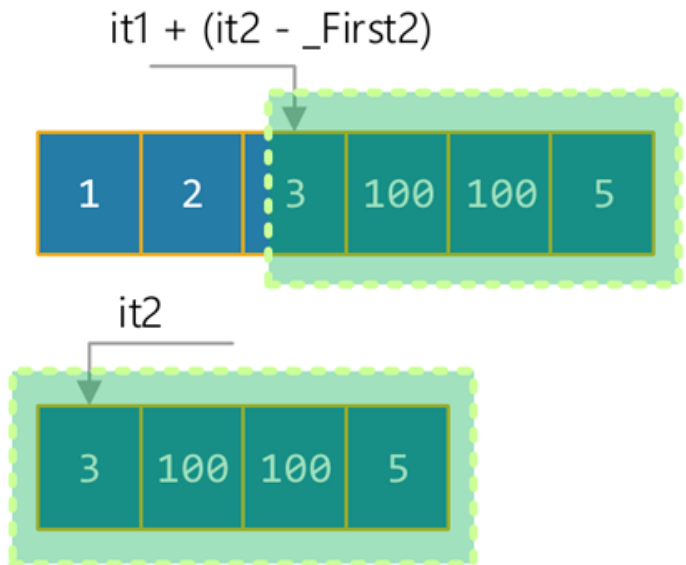
- search (1)：

```
template<class _FwdIt1,class _FwdIt2> inline
_FwdIt1 search(_FwdIt1 _First1, _FwdIt1 _Last1, _FwdIt2 _First2, _FwdIt2
_Last2)
```

  - 查找在区间[_First1, _Last1)中的it1，使得对于每一个在区间[_First2, _Last2)中的it2，满足：*(it1 + (it2 - _First2)) == *it2. 返回it1；如果找不到则返回_Last1
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
int elements2[] = { 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::vector<int> v2(elements2, elements2 + 4);
it = std::search(v.begin(), v.end(),
    v2.begin(), v2.end());

// *it is “3”
```



非变易算法 – search(2)

- search (2)：

```
template<class _FwdIt1,class _FwdIt2, class _Pr> inline
_FwdIt1 search(_FwdIt1 _First1, _FwdIt1 _Last1, _FwdIt2 _First2, _FwdIt2 _Last2,
_Pr _Pred)
```

  - 查找在区间[_First1, _Last1)中的it1，使得对于每一个在区间[_First2, _Last2)中的it2，满足：_Pred(*(it1 + (it2 - _First2)), *it2) == true. 返回it1；如果找不到则返回_Last1
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
int elements2[] = { 6, 200, 200, 10 };
std::vector<int> v(elements, elements + 6);
std::vector<int> v2(elements2, elements2 + 4);

it = std::search(v.begin(), v.end(),
    v2.begin(), v2.end(),
    SearchDouble<int>());
```

```
template <typename T>
struct SearchDouble :
    public std::binary_function<T, T, bool> {
  bool operator()(const T& left, const T& right)
    const
  {
        return (right == (left * 2));
  }
};
```

Part 5 泛型算法

- 非变易算法(Non-mutating Algorithms)
- 变易算法(Mutating Algorithms)
- 排序(Sorting)
- 泛型数值算法(Generalized Numeric Algorithms)

变易算法概述

- **变易算法**是指那些改变容器中对象的操作
- 具体包括:
  - copy
  - swap
  - transform
  - replace
  - fill
  - generate
  - remove
  - unique
  - reserve
  - rotate
  - random_shuffle
  - partition

变易算法 – copy(1)

- copy：

```
template<class _InIt, class _OutIt> inline
_OutIt copy(_InIt _First, _InIt _Last, _OutIt _Dest)
```

  - 将对象从 [_First, _Last)拷贝至[_Dest, _DestLast)，其中：_DestLast = _Dest + (_Last - _First)。此算法是按照顺序拷贝的: _First, _First + 1, …, _Last – 1
  - copy可以实现将容器中的对象左移(Left shifting)
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::vector<int> v2(v.size()); // 注意要预留空间
std::copy(elements, elements + 6, v2.begin()); // v拷贝至v2
std::copy(v2.begin() + 2, v2.end(), v2.begin()); // v2向左移2个元素
```

变易算法 – copy(2)

- copy_n:

```
template<class _InIt, class _Diff, class _OutIt> inline
_OutIt copy_n(_InIt _First, _Diff _Count, _OutIt _Dest)
```

  - 将对象从 [_First, _First + _Count)拷贝至[_Dest, _Dest + _Count)
  - 举例:

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::vector<int> v2(v.size()); // 注意要预留空间
std::copy_n(elements, 4, v2.begin());
```

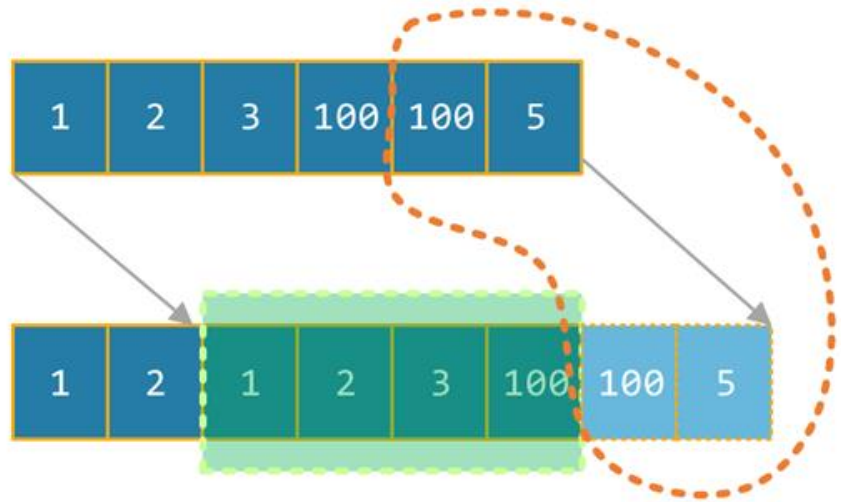
变易算法 – copy(3)

- copy_backward：

```
template<class _BidIt1, class _BidIt2> inline
_BidIt2 copy_backward(_BidIt1 _First, _BidIt1 _Last, _BidIt2 _Dest)
```

  - 将对象从 [_First, _Last)拷贝至[_DestFirst, _Dest)，其中_DestFirst = _Dest - (_Last - _First)
  - copy_backward可实现将容器中的对象右移(Right shifting)
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v2(elements, elements + 6);
std::copy_backward(v2.begin(), v2.end() - 2,
    v2.end());
```



变易算法 – copy(4)

- copy_if:

```
template<class _InIt, class _OutIt, class _Pr> inline
_OutIt copy_if(_InIt _First, _InIt _Last, _OutIt _Dest, _Pr _Pred)
```

  - 对于it ∈ [_First, _Last), 将满足_Pred(*it) == true的对象从 [_First, _Last)拷贝至[_Dest, _Dest + (_Last - _First))
  - 举例:

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::vector<int> v2(v.size()); // 注意要预留空间
std::copy_if(v.begin(), v.end(), v2.begin(),
std::bind2nd(std::less<int>(), 100));

// v2: 1, 2, 3, 5
```

变易算法 – swap(1)

- swap:

```
template<class _Ty> inline
void swap(_Ty& _Left, _Ty& _Right)
```

  - 交换对象_Left和_Right
  - 举例:

```
int elements1[] = { 1, 2, 3, 100, 100, 5 };
int elements2[] = { 6, 200, 200, 10 };
std::vector<int> v1(elements1, elements1 + 6);
std::vector<int> v2(elements2, elements2 + 4);
std::swap(v1, v2);
```

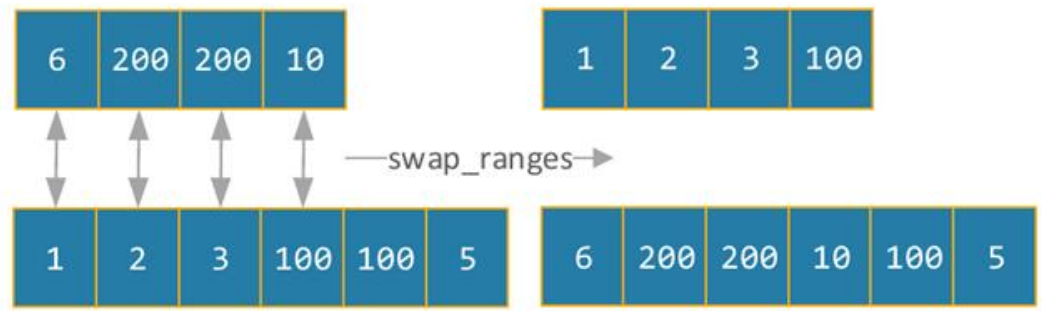
变易算法 – swap(2)

- swap_ranges:

```
template<class _FwdIt1, class _FwdIt2> inline
_FwdIt2 swap_ranges(_FwdIt1 _First1, _FwdIt1 _Last1, _FwdIt2 _Dest)
```

  - 对于0 ≤ i < (_Last1-_First1)，交换对象*(_First + i)和*(_Dest + i)
  - [_First1, _Last1)必须包含在区间[_Dest, _Dest + (_Last1 - _First1))中
  - 举例:

```
int elements1[] = { 1, 2, 3, 100, 100, 5 };
int elements2[] = { 6, 200, 200, 10 };
std::vector<int> v1(elements1, elements1 + 6);
std::vector<int> v2(elements2, elements2 + 4);
std::swap_ranges(v2.begin(), v2.end(),
v1.begin());
```



变易算法 – transform(1)

- transform(1)：

```
template<class _InIt, class _OutIt, class _Fn1> inline
_OutIt transform(_InIt _First, _InIt _Last, _OutIt _Dest, _Fn1 _Func)
```

  - 对于[_First1, _Last)中每个it，应用_Func(*it)，并且将_Func执行的结果放入_Dest指定的区间，即：*(_Dest + i) = _Func(*(_First + i))，其中0 ≤ i < _Last1 - _First1
  - 举例：

```
int elements1[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v1(elements1, elements1 + 6);
std::vector<int> v2(v1.size());
std::transform(v2.begin(), v2.end(), v2.begin(), std::negate<int>());

// output: -1, -2, -3, -100, -100, -5
```

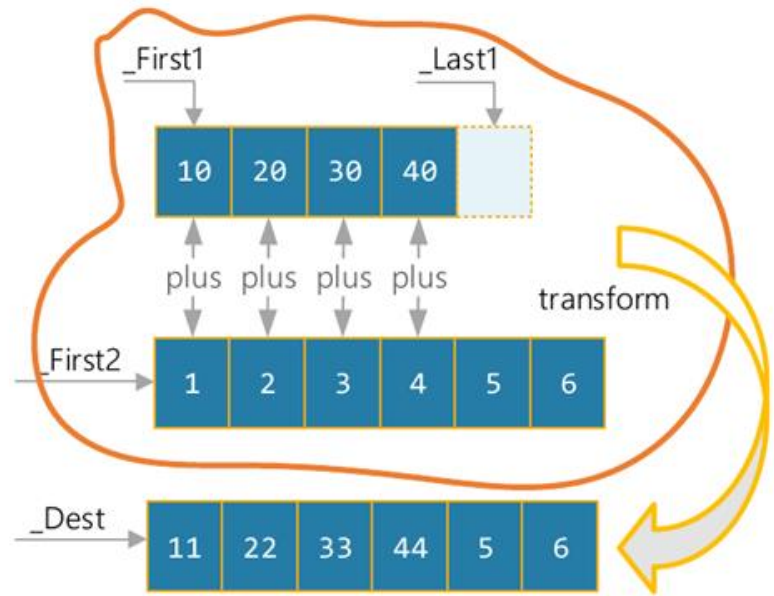
变易算法 – transform(2)

- transform(2)：

```
template<class _InIt1, class _InIt2, class _OutIt, class _Fn2> inline
_OutIt transform(_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _OutIt _Dest, _Fn2 _Func)
```

  - 对于[_First1, _Last1)中每个it1、[_First2, _First2 + (_Last1 - _First1))中的每个it2，应用_Func(*it, *it2)，并且将_Func执行的结果放入_Dest指定的区间，即：*(_Dest + i) = _Func(*(_First1 + i), *(_First2 + i))，其中0 ≤ i < _Last1 - _First1
  - 举例：

```
int elements1[] = { 1, 2, 3, 4, 5, 6 };
int elements2[] = { 10, 20, 30, 40 };
std::vector<int> v2(elements1, elements1 + 6);
std::vector<int> v1(elements2, elements2 + 4);
std::transform(v1.begin(), v1.end(),
v2.begin(),
    v2.begin(), std::plus<int>());
```



变易算法 – replace (1)

- replace(1)：

```
template<class _FwdIt, class _Ty> inline
void replace(_FwdIt _First, _FwdIt _Last, const _Ty& _Oldval, const _Ty& _Newval)
```

  - 对于区间[_First, _Last)中的每个迭代器it，如果满足：*it == _Oldval，则执行：*it = _Newval
  - 举例：

```
int elements1[] = { 1, 2, 3, 100, 200, 5 };
std::vector<int> v1(elements1, elements1 + 6);
std::replace(v1.begin(), v1.end(), 100, 10);

// output: 1, 2, 3, 10, 200, 5
```

变易算法 – replace (2)

- replace_if：

```
template<class _FwdIt, class _Pr, class _Ty> inline
void replace_if(_FwdIt _First, _FwdIt _Last, _Pr _Pred, const _Ty& _Val)
```

  - 对于区间[_First, _Last)中的每个迭代器it，如果满足：_Pred(*it) == true，则执行：*it = _Val
  - 举例：

```
int elements1[] = { 1, 2, 3, 100, 200, 5 };
std::vector<int> v1(elements1, elements1 + 6);
std::replace(v1.begin(), v1.end(),
    std::bind2nd(std::greater_equal<int>(), 100), 0);

// 将v1中所有大于等于100的元素替换为0
// output: 1, 2, 3, 0, 0, 5
```

变易算法 – replace (3)

- replace_copy:

```
template<class _InIt, class _OutIt, class _Ty> inline
_OutIt replace_copy(_InIt _First, _InIt _Last, _OutIt _Dest,
    const _Ty& _Oldval, const _Ty& _Newval)
```

  - 将元素从[_First, _Last)拷贝至[_Dest, Dest + (_Last - _First))，并且将[_First, _Last)中满足*it == _Oldval的元素替换为_Newval
  - 举例:

```
int elements1[] = { 1, 2, 3, 100, 200, 5 };
std::vector<int> v1(elements1, elements1 + 6);
std::vector<int> v2(v1.size());
std::replace_copy(v1.begin(), v1.end(), v2.begin(), 100, 0);

// v2: 1, 2, 3, 0, 200, 5
```

变易算法 – replace (4)

- replace_copy_if：

```
template<class _InIt, class _OutIt, class _Pr, class _Ty> inline
_OutIt replace_copy_if(_InIt _First, _InIt _Last, _OutIt _Dest,
    _Pr _Pred, const _Ty& _Val)
```

  - 将元素从[_First, _Last)拷贝至[_Dest, Dest + (_Last - _First))，并且将[_First, _Last)中满足_Pred(*it, _Oldval) == true的元素替换为_Newval
  - 举例：

```
int elements1[] = { 1, 2, 3, 100, 200, 5 };
std::vector<int> v1(elements1, elements1 + 6);
std::vector<int> v2(v1.size());
std::replace_copy_if(v1.begin(), v1.end(), v2.begin(),
    std::bind2nd(std::greater_equal<int>(), 100), 0);
// 将v1中所有大于等于100的元素替换为0并拷贝至v2
// v2: 1, 2, 3, 0, 0, 5
```

变易算法-fill

- fill:

```
template<class _FwdIt, class _Ty> inline
void fill(_FwdIt _First, _FwdIt _Last, const _Ty& _Val)
```

  - 将_Val赋值给[_First, _Last)中的每个元素
  - 举例：

```
std::vector<int> v(5);
std::fill(v.begin(), v.end(), 10);
```

变易算法-generate

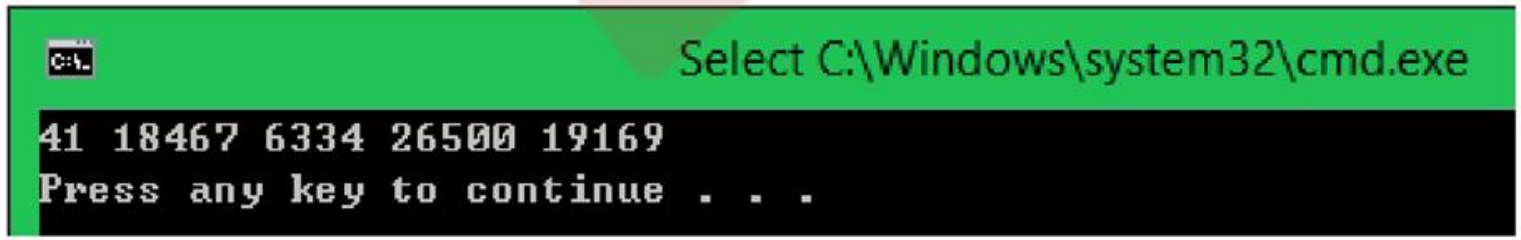
- generate:

```
template<class _FwdIt, class _Fn0> inline
void generate(_FwdIt _First, _FwdIt _Last, _Fn0 _Func)
```

  - 将调用_Func的结果赋值给[_First, _Last)中的每个元素
  - 举例：

```
std::vector<int> v(5);
std::fill(v.begin(), v.end(), rand);

// 产生5个随机数依次赋值给v中的元素
// Output:
```



变易算法-remove(1)

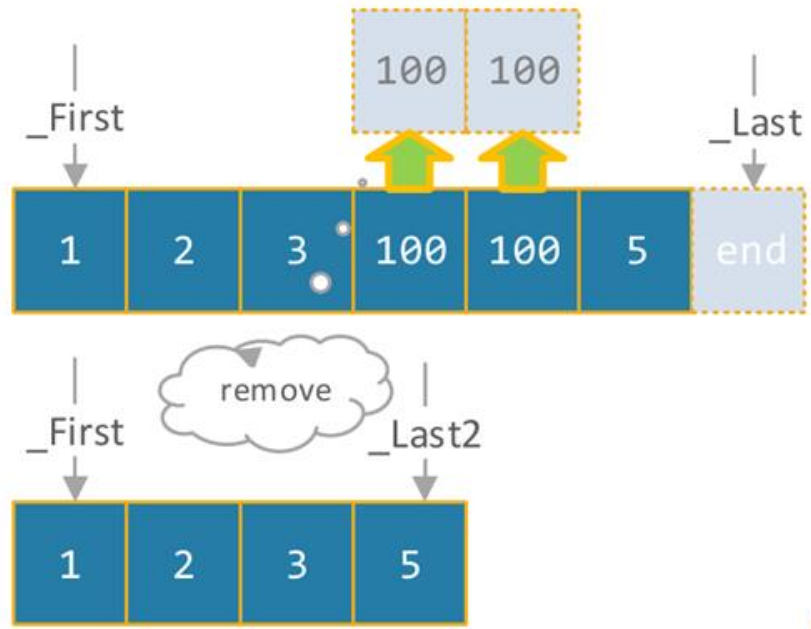
- remove:

```
template<class _FwdIt, class _Ty> inline
_FwdIt remove(_FwdIt _First, _FwdIt _Last, const _Ty& _Val)
```

  - 将[_First, _Last)中的所有等于_Val的元素全部删除。也就是说，remove返回一个迭代器_Last2(_Last2 ≤ _Last)，使得[_First, _Last2)中没有与_Val相等的元素
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::vector<int>::iterator rit =
    std::remove(v.begin(), v.end(), 100);

//此时的v变为: 1, 2, 3, 5
```



变易算法-remove(2)

- remove_if:

```
template<class _FwdIt, class _Pr> inline
_FwdIt remove_if(_FwdIt _First, _FwdIt _Last, _Pr _Pred)
```

  - 对于[_First, _Last)中的it，将所有满足：_Pred(*it) == true的元素全部删除。也就是说，remove返回一个迭代器_Last2(_Last2 ≤ _Last)，使得[_First, _Last2)中没有满足_Pred(*it) == true的元素
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::vector<int>::iterator rit = std::remove(v.begin(), v.end(),
     std::bind2nd(std::greater<int>(), 4));
// 将v中大于4的元素全部删除
// 此时的v变为: 1, 2, 3
```

变易算法-remove(3)

- remove_copy:

```
template<class _InIt, class _OutIt, class _Ty> inline
_OutIt remove_copy(_InIt _First, _InIt _Last, _OutIt _Dest, const _Ty& _Val)
```

  - 从[_First, _Last)中拷贝不等于_Val的元素至_Dest
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
int elements2[10] = { 0 };
int* ptr = std::remove_copy(v.begin(), v.end(), elements2, 100);

// 此时elements2为: 1, 2, 3, 5
```

变易算法-unique

- unique:

```
template<class _FwdIt> inline
_FwdIt unique(_FwdIt _First, _FwdIt _Last)
```

  - 从[_First, _Last)去除重复的元素（只保留一份），返回一个新的迭代器_Last2，使得[_First, _Last2)中没有重复出现的元素
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::vector<int>::iterator rit = std::unique(v.begin(), v.end());

// v变为: 1, 2, 3, 100, 5
```

变易算法-reverse

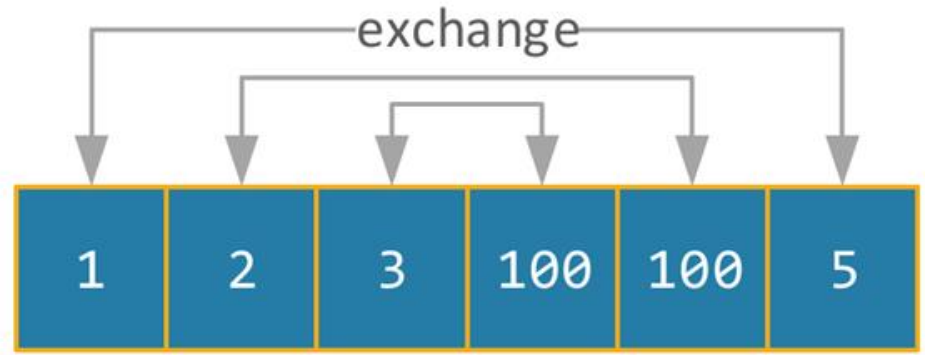
- reverse:

```
template<class _BidIt> inline
void reverse(_BidIt _First, _BidIt _Last)
```

  - 对于0 ≤ it ≤ (_Last - _First) / 2中的每个元素，做如下交换:
    *(_First + it) ←→ *(_Last - (it + 1))
  - 举例:

```
int elements[] = { 1, 2, 3, 100, 100, 5 };
std::vector<int> v(elements, elements + 6);
std::reverse(v.begin(), v.end());

// v变为: 5, 100, 100, 3, 2, 1
```



变易算法-rotate

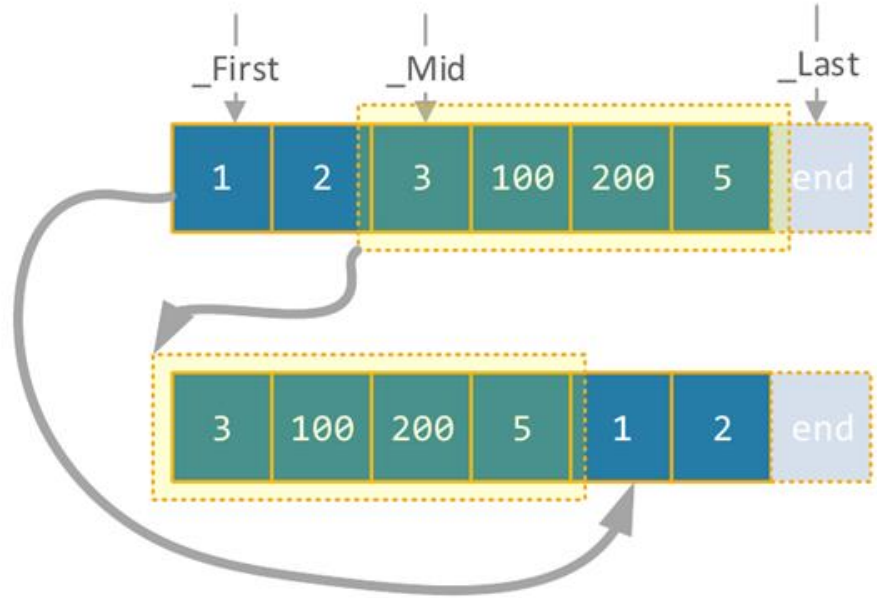
- reverse:

```
template<class _FwdIt> inline
_FwdIt rotate(_FwdIt _First, _FwdIt _Mid, _FwdIt _Last)
```

  - 交换区间：[_First, _Mid) ←→ [_Mid, _Last)
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 200, 5 };
std::vector<int> v(elements, elements + 6);
std::rotate(v.begin(), v.begin() + 2,
v.end());

// v变为: 3, 100, 200, 5, 1, 2
```



变易算法-random_shuffle

- random_shuffle:

```
template<class _RanIt> inline
void random_shuffle(_RanIt _First, _RanIt _Last) // 采用rand

template<class _RanIt, class _Fn1> inline
void random_shuffle(_RanIt _First, _RanIt _Last, _Fn1&& _Func) // 自定义
```

- 对区间[_First, _Last)中的元素进行洗牌。如果令N = _Last - _First，那么该算法即从N!的可能排列中随机选择一种
- 默认采用rand函数作为随机函数，也可自定义随机数生成函数
- 举例：

```
int elements[] = { 1, 2, 3, 100, 200, 5 };
std::vector<int> v(elements, elements + 6);
std::random_shuffle(v.begin(), v.end());
```

变易算法-partition

- partition:

```
template<class _FwdIt, class _Pr> inline
_FwdIt partition(_FwdIt _First, _FwdIt _Last, _Pr _Pred)
```

  - 基于_Pred对容器中的的元素进行划分，将区间[_First, _Last)划分成两部分：[_First, _Mid)以及[_Mid, _Last)，使得：对于任意的it1 ∈ [_First, _Mid)，_Pred(*it1) == true，而对于任意的it2 ∈ [_Mide, _Last)，_Pred(*it2) == false
  - 举例：

```
int elements[] = { -1, 2, 3, -100, 200, 5 };
std::vector<int> v(elements, elements + 6);
std::partition(v.begin(), v.end(), std::bind2nd(std::less<int>(), 0));

// v变为：-1, -100, 2, 3, 200, 5
```

Part 5 泛型算法

- 非变易算法(Non-mutating Algorithms)
- 变易算法(Mutating Algorithms)
- 排序(Sorting)
- 泛型数值算法(Generalized Numeric Algorithms)

排序-sort(1)

- sort:

```
template<class _RanIt> inline
void sort(_RanIt _First, _RanIt _Last)
```

  - 对[_First, _Last)中的元素进行排序，使得对于it1、it2 ∈ [_First, _Last)且it1 < it2，满足(*it1) < (*it2)
  - 对于被排序的元素，需要提供operator<;
  - 举例：

```
int elements[] = { -1, 2, 3, -100, 200, 5 };
std::vector<int> v(elements, elements + 6);
std::sort(v.begin(), v.end());

// v变为：-100, -1, 2, 3, 5, 200
```

排序-partial_sort

- partial_sort:

```
template<class _RanIt> inline
void partial_sort(_RanIt _First, _RanIt _Mid, _RanIt _Last)
```

  - 对[_First, _Last)中的部分元素进行排序，使得[_First, _Mid)中的元素是有序的，而[_Mid, _Last)中的元素的顺序则未定义
  - 举例：

```
int elements[] = { -1, 2, 3, -100, 200, 5, 9, -3, 12, -10, 8, -4 };
std::vector<int> v(elements, elements + 12);
std::partial_sort(v.begin(), v.begin() + 6, v.end()); // 对前6个元素排序

// v变为: -100, -10, -4, -3, -1, 2, 200, 9, 12, 5, 8, 3
```

排序-binary_search

- binary_search:

```
template<class _FwdIt, class _Ty> inline
bool binary_search(_FwdIt _First, _FwdIt _Last, const _Ty& _Val)
```

  - 在[_First, _Last)中查找等于_Val的元素
  - 容器中的元素首先要排序
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 200, 5 };
std::vector<int> v(elements, elements + 6);
std::sort(v.begin(), v.end());
for (int i = 1; i < 10; i++) {
  std::wcout << L"Searching for " << i << ": "
    << (std::binary_search(v.begin(), v.end(),
i) ? L“Found" : L“Not found")
    << std::endl;
}
```

```
Searching for 1: Found
Searching for 2: Found
Searching for 3: Found
Searching for 4: Not found
Searching for 5: Not found
Searching for 6: Not found
Searching for 7: Not found
Searching for 8: Not found
Searching for 9: Not found
```

```
Searching for 1: Found
Searching for 2: Found
Searching for 3: Found
Searching for 4: Not found
Searching for 5: Found
Searching for 6: Not found
Searching for 7: Not found
Searching for 8: Not found
Searching for 9: Not found
```

未排序，最后一个元素未找到

排序-merge

- merge:

```
template<class _InIt1, class _InIt2, class _OutIt> inline
_OutIt merge(_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _InIt2 _Last2,
    _OutIt _Dest)
```

  - 将**排好序的**[_First1, _Last1)与[_First2, _Last2)合并到_Dest
  - 举例：

```
int elements[] = { 1, 2, 3, 100, 200, 5 };
int elements2[] = { 7, 900, -10, 33, 8 };
const int N = sizeof(elements) / sizeof(int);
const int N2 = sizeof(elements2) / sizeof(int);
std::vector<int> v(elements, elements + N);
std::vector<int> v2(elements2, elements2 + N2);
std::vector<int> result(N + N2);
```

```
std::sort(v.begin(), v.end());
std::sort(v2.begin(), v2.end());

std::vector<int>::iteratorrit =
std::merge(v.begin(), v.end(),
    v2.begin(), v2.end(),
    result.begin());
```

```
// result: -10, 1, 2, 3, 5, 7, 8, 33, 100, 200, 900
```

基于排序集合的一些算法 (Set operations on sorted ranges)

- includes
- set_union
- set_intersection
- set_difference

集合算法-includes

- includes:

```
template<class _InIt1, class _InIt2> inline
bool includes(_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _InIt2 _Last2)
```

  - 判别[_First1, _Last1)是否包含[_First2, _Last2)
  - [_First1, _Last1)和[_First2, _Last2)都是排好序的
  - 举例：

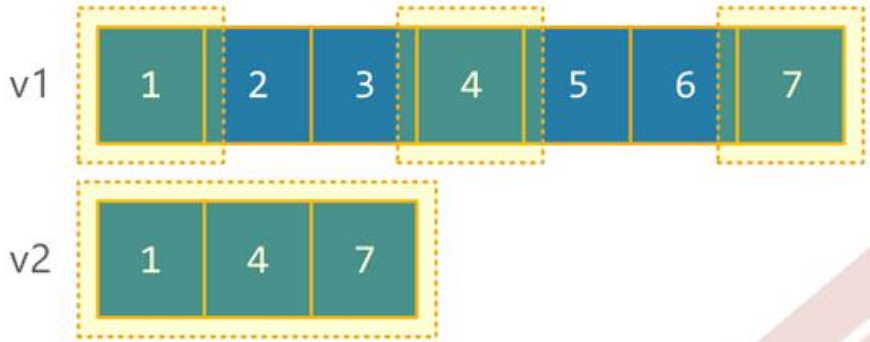
```
int elements1[] = { 1, 2, 3, 4, 5, 6, 7 };
int elements2[] = { 1, 4, 7 };
int elements3[] = { 2, 7, 9 };
int elements4[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
int elements5[] = { 1, 2, 13, 13 };
int elements6[] = { 1, 1, 3, 21 };
```

```
std::vector<int> v1(elements1, elements1 + 7);
std::vector<int> v2(element2, elements2 + 3);
std::vector<int> v3(element3, elements3 + 3);
std::vector<int> v4(element4, elements4 + 8);
std::vector<int> v5(element5, elements5 + 4);
std::vector<int> v6(element6, elements6 + 4);
```

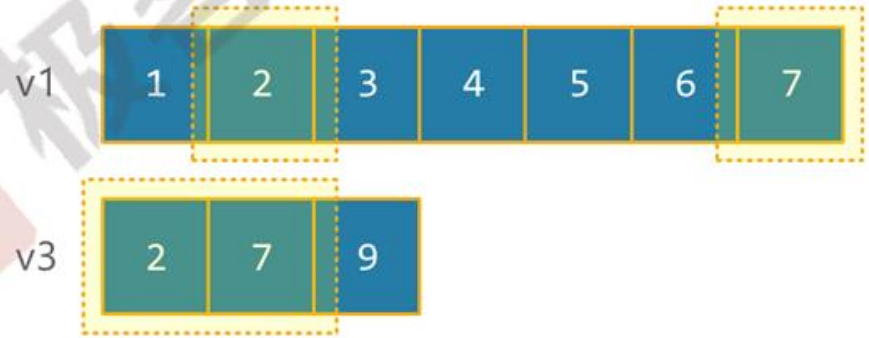
集合算法-includes(举例)

- 举例（续）：

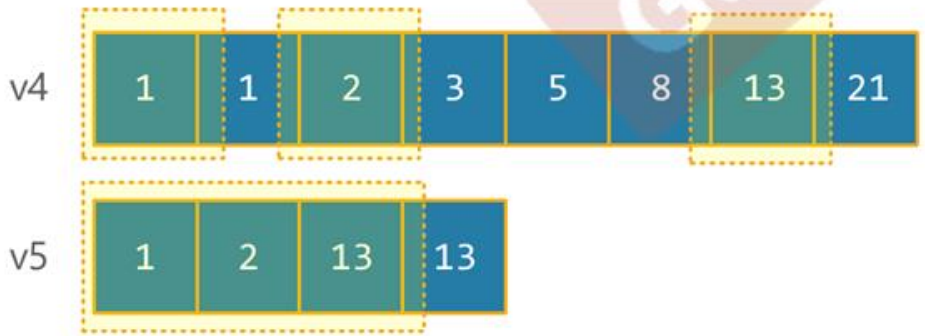
```
std::includes(v1.begin(), v1.end(),
    v2.begin(), v2.end()); // true
```



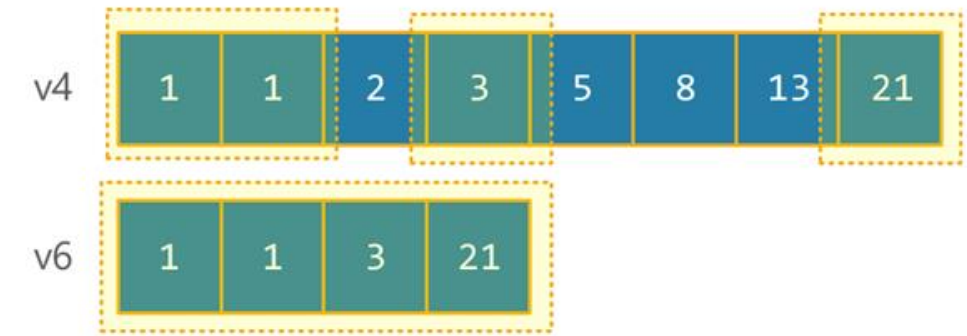

```
std::includes(v1.begin(), v1.end(),
    v3.begin(), v3.end()); // false
```




```
std::includes(v4.begin(), v4.end(),
    v5.begin(), v5.end()); // false
```




```
std::includes(v4.begin(), v4.end(),
    v6.begin(), v6.end()); // true
```



集合算法-set operations

- Set相关算法:
  - set_union
  - set_intersection
  - set_difference
- 参见“STL容器”部分Set相关内容

基于堆的算法(Heap operations)

- make_heap
- push_heap
- pop_heap
- sort_heap

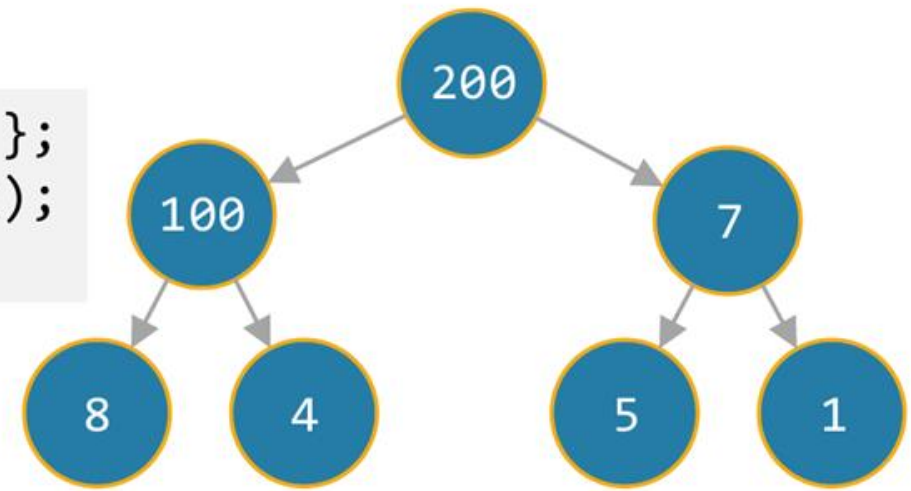
Heap算法-make_heap

- make_heap:

```
template<class _RanIt> inline
void make_heap(_RanIt _First, _RanIt _Last)
```

  - 将区间[_First, _Last)转换成一个堆
  - 堆结构采用max_heap，维持平衡二叉树
  - 举例：

```
int elements[] = { 1, 4, 200, 8, 100, 5, 7 };
const int n = sizeof(elements) / sizeof(int);
std::make_heap(elements, elements + n);
```



make_heap后的节点分布

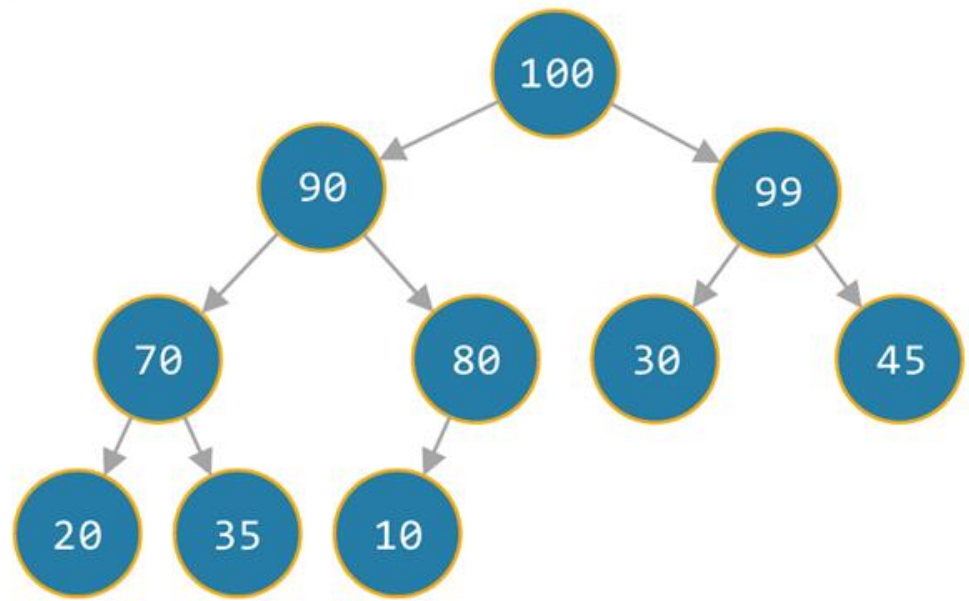
Heap算法-push_heap

- push_heap:

```
template<class _RanIt> inline
void push_heap(_RanIt _First, _RanIt _Last)
```

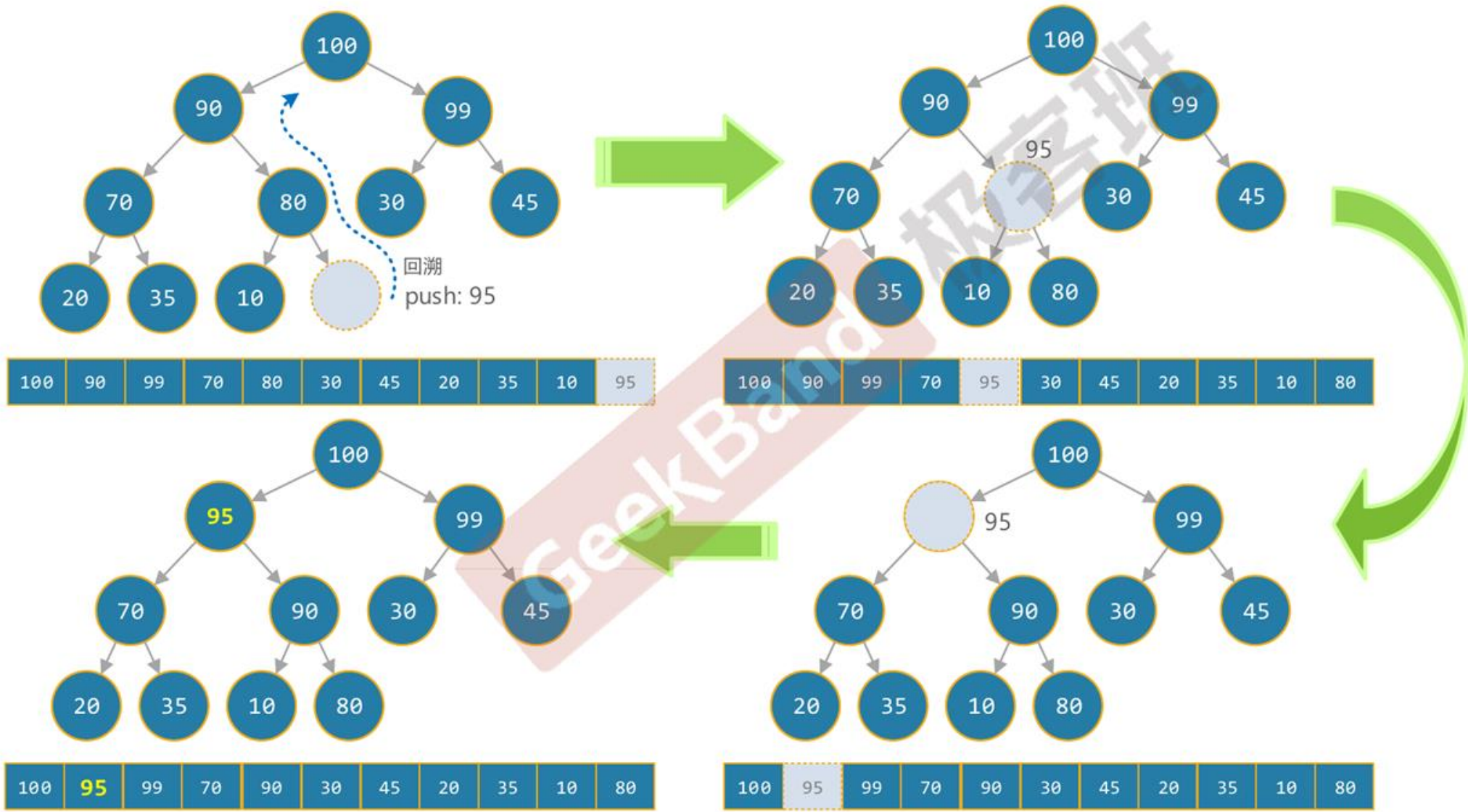
  - 向堆中添加一个元素，该算法的前提是假设[_First, _Last - 1)已经是个堆，被添加到堆的元素为*(_Last - 1)
  - 堆结构采用max_heap，维持平衡二叉树
  - 举例：

```
int elements[] = {
  100, 90, 99, 70, 80, 30, 45, 20, 35, 10, 95
};
const int n = sizeof(elements) / sizeof(int);
std::make_heap(elements, elements + (n - 1));
std::push_heap(elements, elements + n);
```



make_heap后的节点分布

Heap算法-push_heap的过程



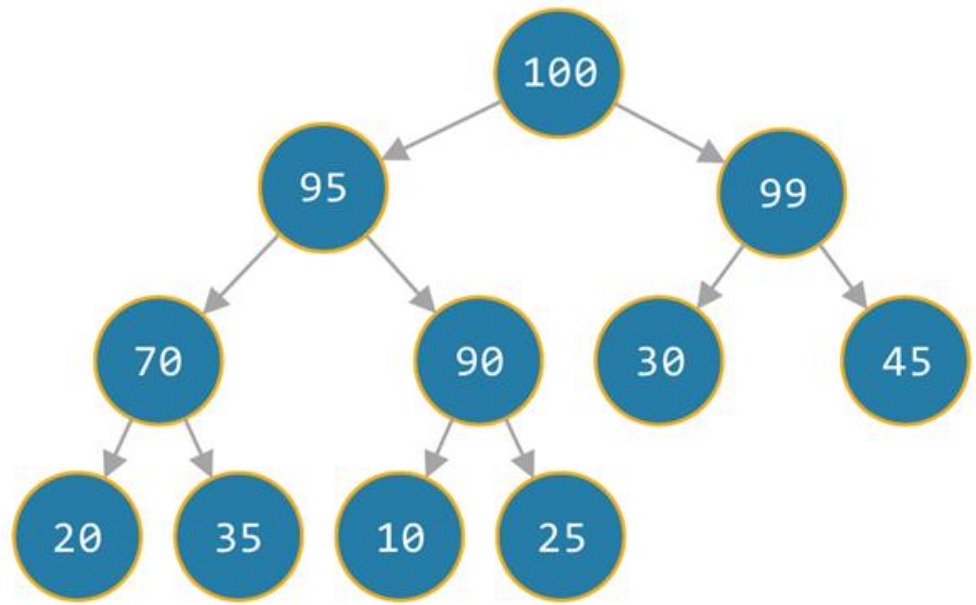
Heap算法-pop_heap

- pop_heap:

```
template<class _RanIt> inline
void pop_heap(_RanIt _First, _RanIt _Last)
```

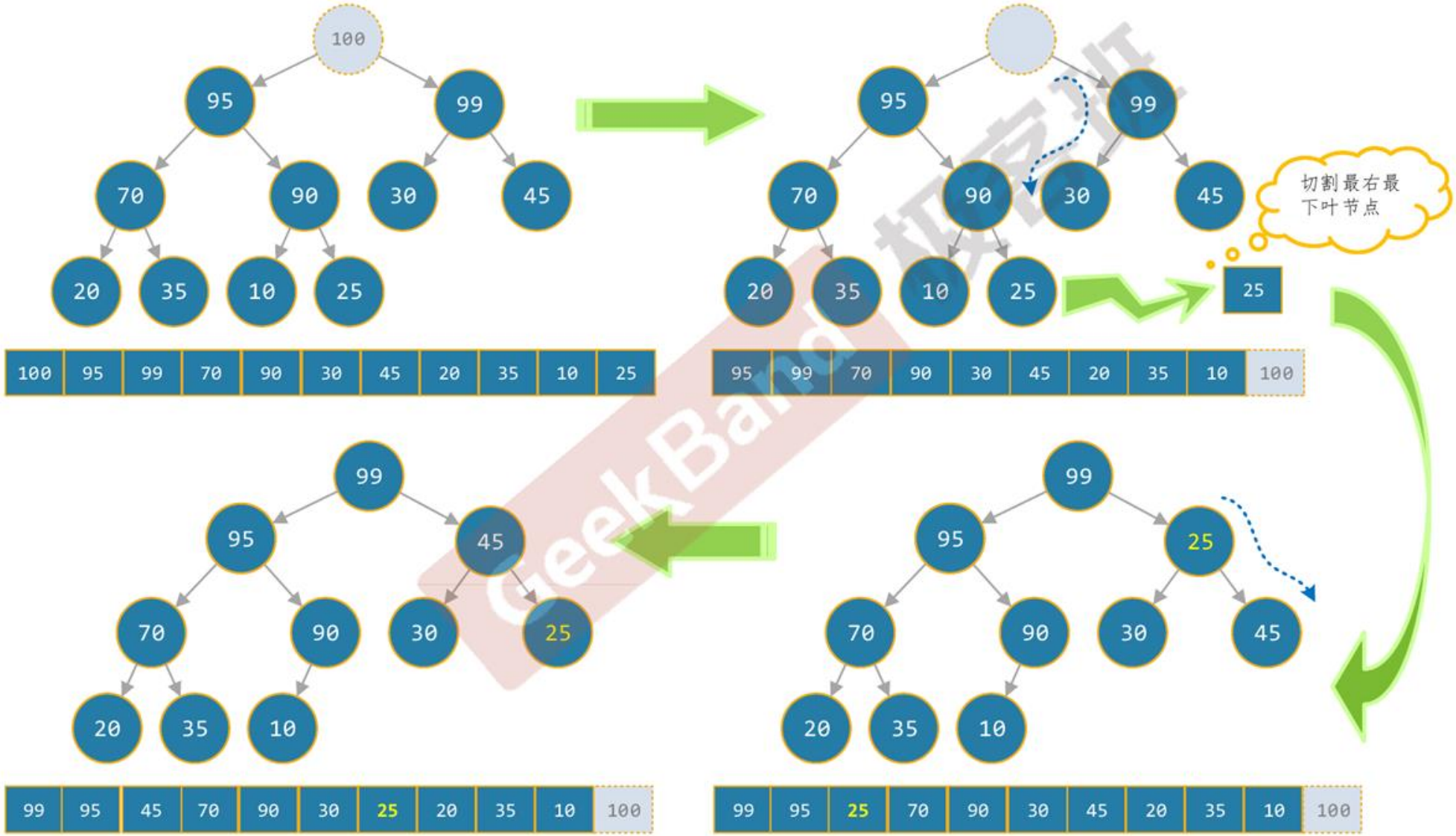
  - 从堆中弹出一个元素，该算法的前提是假设[_First, _Last)已经是个堆，被弹出堆的元素为根顶元素
  - 堆结构采用max_heap，维持平衡二叉树
  - 举例：

```
int elements[] = {
  100, 90, 99, 70, 25, 30, 45, 20, 35, 10, 95
};
const int n = sizeof(elements) / sizeof(int);
std::make_heap(elements, elements + n);
```



make_heap后的节点分布

Heap算法-pop_heap的过程



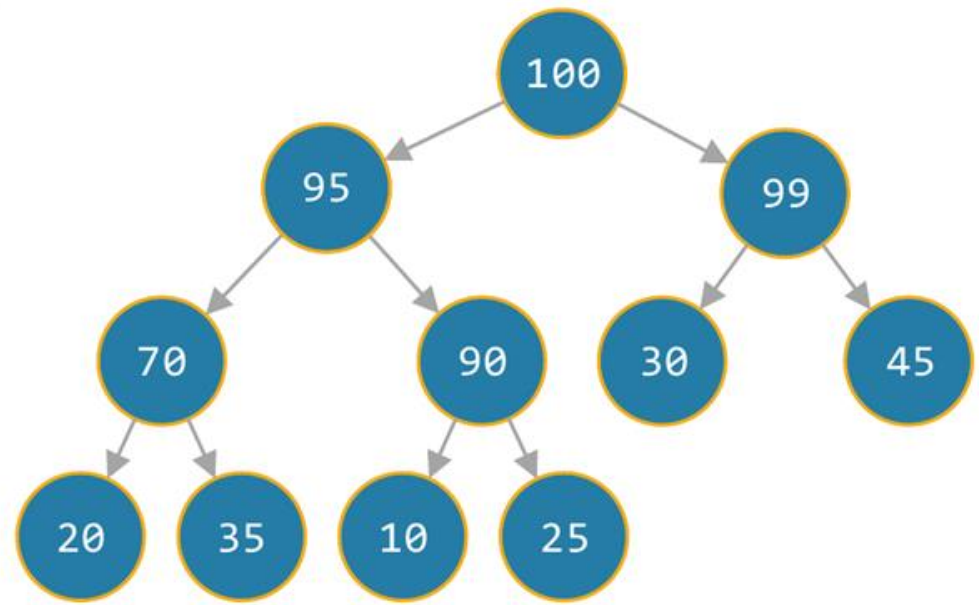
Heap算法-sort_heap

- sort_heap:

```
template<class _RanIt> inline
void sort_heap(_RanIt _First, _RanIt _Last)
```

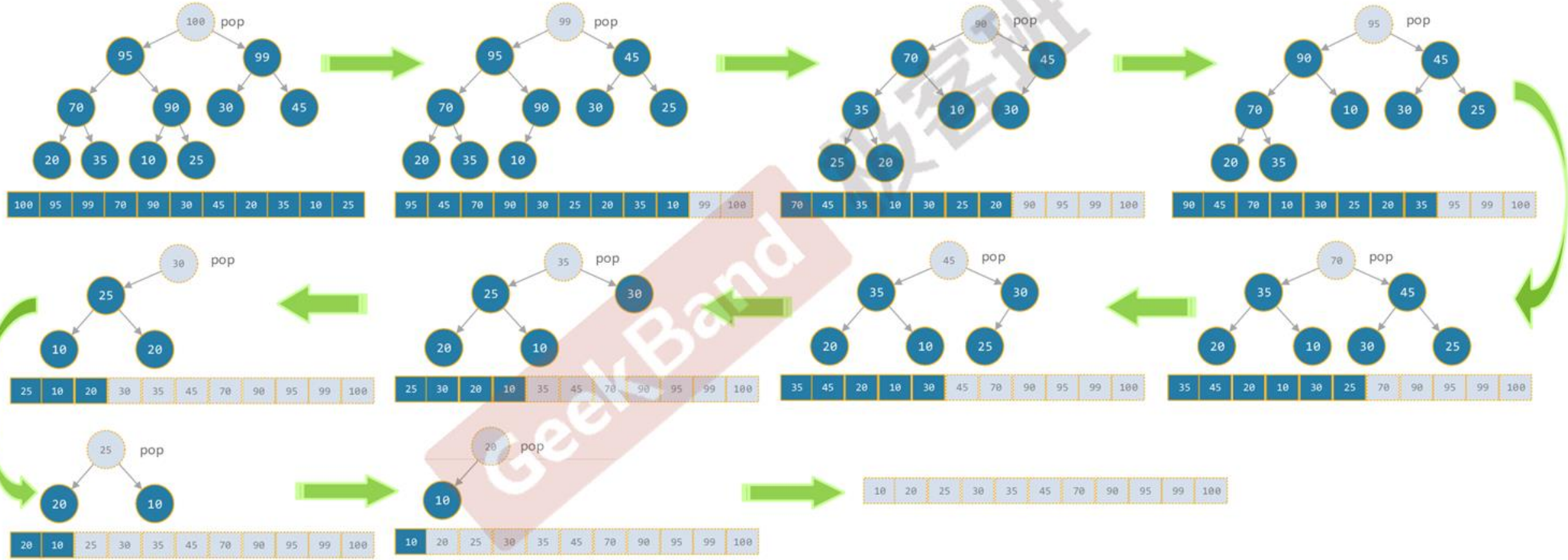
  - 将堆[_First, _Last)中的元素进行排序
  - 堆结构采用max_heap，维持平衡二叉树
  - Sort的做法是不断pop_heap，因为每次pop都可以获得堆中的最大元素
  - 举例：

```
int elements[] = {
  100, 90, 99, 70, 25, 30, 45, 20, 35, 10, 95
};
const int n = sizeof(elements) / sizeof(int);
std::make_heap(elements, elements + n);
std::sort_heap(elements, elements + n);
```



make_heap后的节点分布

Heap算法-sort_heap的过程



Part 5 泛型算法

- 非变易算法(Non-mutating Algorithms)
- 变易算法(Mutating Algorithms)
- 排序(Sorting)
- 泛型数值算法(Generalized Numeric Algorithms)

泛型数值算法

- **泛型数值算法**包含在<numeric>头文件中
- 包括:
  - accumulate
  - inner_product
  - partial_sum
  - adjacent_difference

泛型数值算法- accumulate(1)

- accumulate(1):

```
template<class _InIt, class _Ty> inline
_Ty accumulate(_InIt _First, _InIt _Last, _Ty _Val)
```

  - 对[_First, _Last)中的每个元素进行累加，具体操作为：设result = _Val，对于每个it ∈ [_Furst, _Last)，result += *it
  - 举例：

```
std::vector<int> v(100);
std::iota(v.begin(), v.end(), 1);
int sum = std::accumulate(v.begin(), v.end(), 0);

// 小学奥数题: 1 + 2 + … + 100 = 5050 ☺
```

泛型数值算法- accumulate(2)

- accumulate(2):

```
template<class _InIt, class _Ty, class _Fn2> inline
_Ty accumulate(_InIt _First, _InIt _Last, _Ty _Val, _Fn2 _Func)
```

  - 对[_First, _Last)中的每个元素进行累加，具体操作为：设result = _Val，对于每个it ∈ [_Furst, _Last)，result = _Func(*it, result)
  - 举例：

```
std::vector<int> v(10);
std::iota(v.begin(), v.end(), 1);
int sum = std::accumulate(v.begin(), v.end(), 1,
std::multiplies<int>());

// 小学奥数题：1 × 2 × … × 10 = 10! = 3628800 ☺
```

泛型数值算法- inner_product(1)

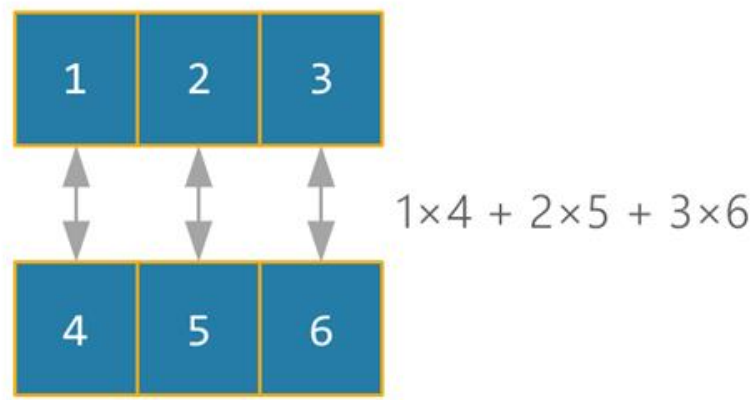
- inner_product(1):

```
template<class _InIt1, class _InIt2, class _Ty> inline
_Ty inner_product(_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _Ty _Val)
```

  - 对[_First1, _Last1)和[_First2, _Last2)进行如下操作:设result = _Val, 对于每个it1 ∈ [_First1, _Last1), result = result + (*it1) ×*(_First2 + (it1 - _First1))
  - 举例:

```
int a[] = { 1, 2, 3 };
int b[] = { 4, 5, 6 };
inner_product(a, a + 3, b, 0)

// 计算结果为: 32
```



泛型数值算法- inner_product(2)

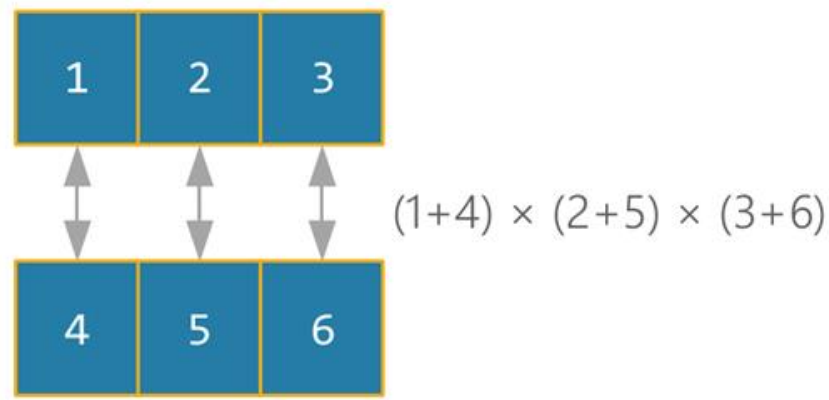
- inner_product(2):

```
template<class _InIt1, class _InIt2, class _Ty, class _Fn21, class _Fn22> inline
_Ty inner_product(_InIt1 _First1, _InIt1 _Last1, _InIt2 _First2, _Ty _Val,
    _Fn21 _Func1, _Fn22 _Func2)
```

  - 对[_First1, _Last1)和[_First2, _Last2)进行如下操作:设result = _Val，对于每个it1 ∈ [_First1, _Last1), result = _Func1(result, _Func2(*it1, *(_First2 + (it1 - _First1))))
  - 举例:

```
int a[] = { 1, 2, 3 };
int b[] = { 4, 5, 6 };
inner_product(a, a + 3, b, 1,
    std::multiplies<int>(),
    std::plus<int>())

// 计算结果为: 315
```



泛型数值算法- partial_sum(1)

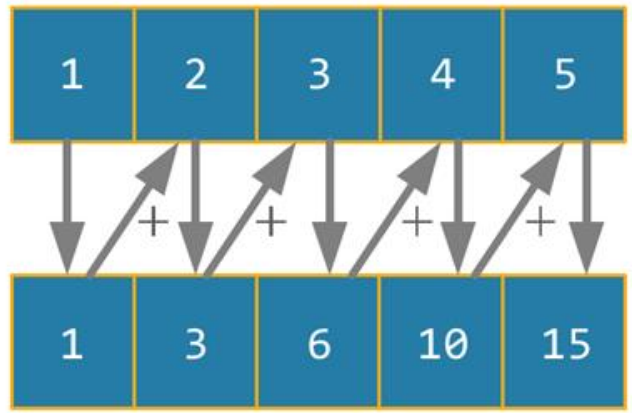
- partial_sum(1):

```
template<class _InIt, class _OutIt> inline
_OutIt partial_sum(_InIt _First, _InIt _Last, _OutIt _Dest)
```

  - 设it = _First, *it = *_First, *(it + 1) = *_First + *(_First + 1), ..., 以此类推
  - 举例：

```
int elements[] = { 1, 2, 3, 4, 5 };
const int n = sizeof(elements) / sizeof(int);
std::vector<int> v(elements, elements + n);
std::partial_sum(v.begin(), v.end(), v.begin());

// v: 1, 3, 6, 10, 15
```



泛型数值算法- partial_sum(2)

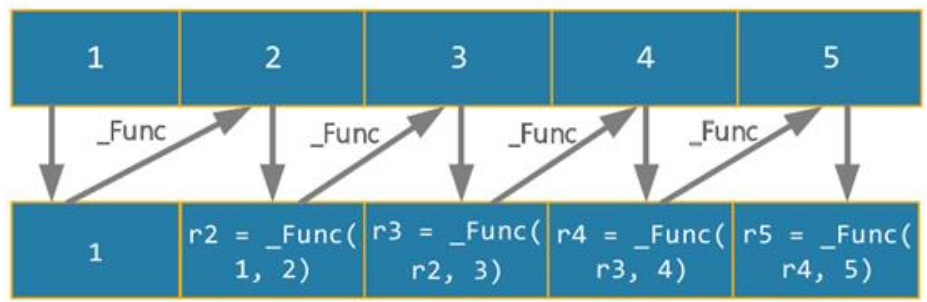
- partial_sum(2):

```
template<class _InIt, class _OutIt, class _Fn2 > inline
_OutIt partial_sum(_InIt _First, _InIt _Last, _OutIt _Dest, _Fn2 _Func)
```

  - 设it = _First, *it = *_First, 对于每个it ∈ [_First + 1, _Last), _Func(*it, *(it - 1))的运算结果赋值给*(result + (it - _First))
  - 举例：

```
int elements[] = { 1, 2, 3, 4, 5 };
const int n = sizeof(elements) / sizeof(int);
std::vector<int> v(elements, elements + n);
std::partial_sum(v.begin(), v.end(), v.begin(),
    std::minus<int>());

// v: 1, -1, -4, -8, -13
```



泛型数值算法- adjacent_difference (1)

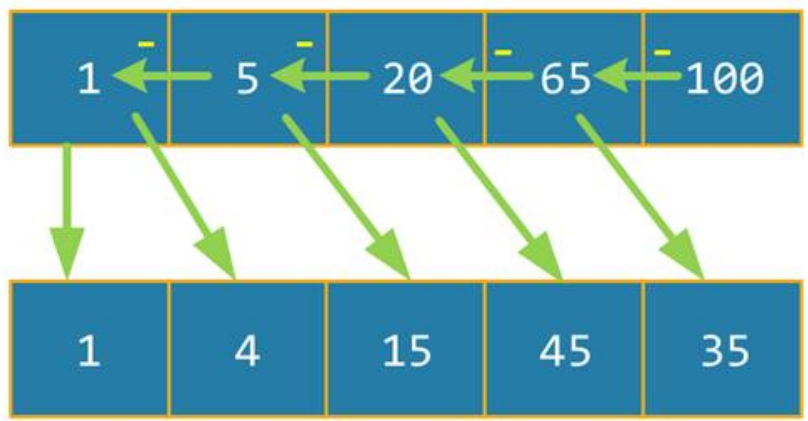
- adjacent_difference(1):

```
template<class _InIt, class _OutIt> inline
_OutIt adjacent_difference(_InIt _First, _InIt _Last, _OutIt _Dest)
```

  - 设*result = *_First, 对于每个it ∈ [_First + 1, _Last), *it与*(it - 1)的差赋值给*(result + (it - _First))
  - 举例：

```
int elements[] = { 1, 5, 20, 65, 100 };
const int n = sizeof(elements) / sizeof(int);
std::vector<int> v(elements, elements + n);
std::adjacent_difference(v.begin(), v.end(),
    v.begin());

// v: 1, 4, 15, 45, 35
```



泛型数值算法- adjacent_difference (2)

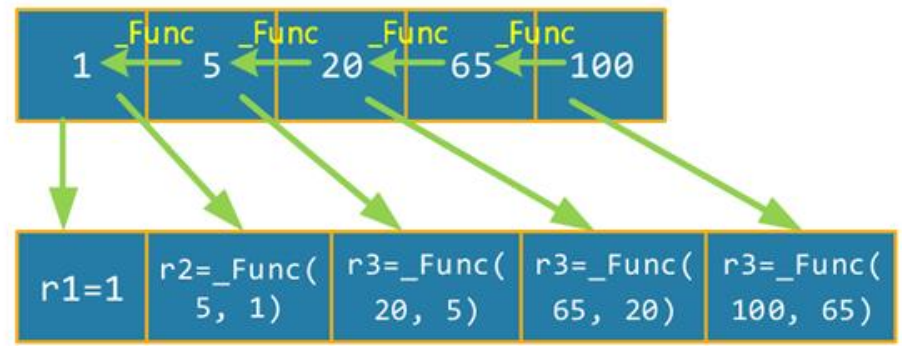
- adjacent_difference(2):

```
template<class _InIt, class _OutIt, class _Fn2> inline
_OutIt adjacent_difference(_InIt _First, _InIt _Last, _OutIt _Dest,
    _Fn2 _Func)
```

  - 设*result = *_First, 对于每个it ∈ [_First + 1, _Last), 将_Func(*it, *(it - 1))的运算结果赋值给*(result + (it - _First))
  - 举例：

```
int elements[] = { 1, 5, 20, 65, 100 };
const int n = sizeof(elements) / sizeof(int);
std::vector<int> v(elements, elements + n);
std::adjacent_difference(v.begin(), v.end(),
    v.begin(), std::multiplies<int>());

// v: 1, 5, 100, 1300, 6500
```



Part 6 内存分配器

内存分配器的标准接口(1)

- 如果需要自己写一个allocator，则需满足以下接口：
  - 一组typedef:
    - allocator::value_type
    - allocator::pointer
    - allocator::const_pointer
    - allocator::reference
    - allocator::const_reference
    - allocator::size_type
    - allocator::difference_type
  - allocator::rebind：allocator的内嵌模板，需要定义other成员
  - allocator::allocator()：构造函数

内存分配器的标准接口(2)

- allocator::allocator (const allocator&)：拷贝构造函数
- template <typename T> allocator::allocator (const allocator<T>&)：泛化的拷贝构造函数
- allocator::~allocator()：析构函数
- pointer allocator::address(reference x) const: 返回对象地址，allocator.address(x)相当于&x
- pointer allocator::allocate (size_type n, const void* = 0)：分配可以容纳n个对象的空间，对象型别是T
- void allocator::deallocator (pointer p, size_type n)：释放空间
- size_type allocator::max_size() const: 可以分配的最大空间
- void allocator::construct(pointer p, const T& x)：相当于new (const void*) p) T(x)
- void allocator::destroy(pointer p)：相当于p->~T()

一个简单的自定义内存分配器 (1)

- 在满足以上标准接口的基础上，实现简单的自定义内存分配器：
  - 定义MyAllocator为一个模板，为清楚期间，将自定义的内存分配器放置在wj命名空间里：

```
namespace wj {
    template <typename T> struct MyAllocator { ... };
}
```

  - 实现标准的一组typedef：

```
typedef T           value_type;
typedef T*          pointer;
typedef const T*    const_pointer;
typedef T&          reference;
typedef const T&    const_reference;
typedef size_t      size_type;
typedef int         difference_type;
```


一个简单的自定义内存分配器 (2)

- 定义内嵌的rebind模板：

```
template <typename U> struct rebind {
    typedef MyAllocator<U> other;
};
```

- 构造函数：

```
MyAllocator() { }
MyAllocator(MyAllocator<T> const&) { }

template<typename U>
MyAllocator(MyAllocator<U> const&) { }
```


一个简单的自定义内存分配器 (3)

- 实现allocate函数：

```
pointer allocate (size_type n, const void* p = 0) {
    T* buffer = (T*) malloc((size_t)(n * sizeof(T)));
    if (buffer == NULL) { // error handling
        ...
    }

    return buffer;
};
```

- 实现deallocate函数：

```
void deallocate (pointer p, size_type n) {
    if (p != NULL)
        free(p);
}
```


一个简单的自定义内存分配器 (4)

- 实现construct函数：

```
void construct (pointer p, const T& value) {
    new(p) T(value); // placement new, 在地址p处构造T, 并将T赋值为value
}
```

- 实现destroy函数：

```
void destroy (pointer p, size_type n) {
    p->~T(); // 调用T的析构函数以销毁之
}
```

- 实现max_size函数：

```
size_type max_size() const {
    return size_type(UINT_MAX / sizeof(T));
}
```


一个简单的自定义内存分配器 (5)

- 实现address和const_address函数：

```
pointer address (reference x)
{
    return (pointer)&x;
}

const_pointer const_address (const_reference x)
{
    return (const_pointer)&x;
}
```


测试自定义内存分配器

- 用std::vector搭配我么自定义的内存分配器：

```
int elements[] = { 1, 2, 3, 4, 5 };
const int n = sizeof(elements) / sizeof(int);

vector<int, wj::MyAllocator<int> > myVector(elements, elements + n);
for_each(myVector.begin(), myVector.end(), PrintElements<int>());
```

```
template <typename T>
struct PrintElements : std::unary_function<T, void>
{
    void operator()(const T& t) const {
        std::wcout << t << L" ";
    }
};
```

```
1 2 3 4 5
Press any key to continue . . .
```


Thanks☺

Zhang wenjie
wjzhangb@163.com