Username: Pralay Patoria **Book:** The C++ Standard Library: A Tutorial and Reference, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

11.9. Sorting Algorithms

The STL provides several algorithms to sort elements of a range. In addition to full sorting, the STL provides variants of partial sorting. If their result is enough, you should prefer them because they usually have better performance.

Because (forward) lists and associative and unordered containers provide no random-access iterators, you can't use these containers (as a destination) for sorting algorithms. Instead, you might use associative containers to have elements sorted automatically. Note, however, that sorting all elements once is usually faster than keeping them always sorted (see Section 7.12, page 394, for details).

11.9.1. Sorting All Elements

Click here to view code image

```
void
sort (RandomAccessIterator beg, RandomAccessIterator end)

void
sort (RandomAccessIterator beg, RandomAccessIterator end, BinaryPredicate
op)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end)

void
stable_sort (RandomAccessIterator beg, RandomAccessIterator end,
             BinaryPredicate op)
```

- The first forms of `sort()` and `stable_sort()` sort all elements in the range `[` *beg* `,` *end* `)` with operator `<` .
- The second forms of `sort()` and `stable_sort()` sort all elements by using the binary predicate

  op`(`*elem1*`,`*elem2*`)`

  as the sorting criterion. It should return `true` if *elem1* is "less than" *elem2*.
- Note that *op* has to define a *strict weak ordering* for the values (see Section 7.7, page 314, for details).
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- The difference between `sort()` and `stable_sort()` is that `stable_sort()` guarantees that the order of equal elements remains stable.
- You can't call these algorithms for lists or forward lists, because both do not provide random-access iterators. However, they provide a special member function to sort elements: `sort()` (see Section 8.8.1, page 422).
- `sort()` guarantees a good performance (n-log-n). However, before C++11, this was only guaranteed "on average." So, if avoiding a worse complexity was important, you had to use `partial_sort()` or `stable_sort()` . See the discussion about sorting algorithms in Section 11.2.2, page 511.
- Complexity:
  - For `sort()` : n-log-n (approximately *numElems* `*log(` *numElems* `)` comparisons). Before C++11, the guarantee was: n-log-n *on average*.
  - For `stable_sort()` : n-log-n if there is enough extra memory (*numElems* `*log` (*numElems* `)` comparisons); otherwise, n-log-n `*` log-n (*numElems* `*(log` (*numElems* `))` ^{2} comparisons).

The following example demonstrates the use of `sort()` :

Click here to view code image

```
// algo/sort1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;
```

```
    INSERT_ELEMENTS(coll,1,9);
    INSERT_ELEMENTS(coll,1,9);

    PRINT_ELEMENTS(coll,"on entry: ");

    // sort elements
    sort (coll.begin(), coll.end());

    PRINT_ELEMENTS(coll,"sorted:   ");

    // sorted reverse
    sort (coll.begin(), coll.end(),       // range
          greater<int>());                // sorting criterion

    PRINT_ELEMENTS(coll,"sorted >: ");
}
```

The program has the following output:

Click here to view code image

```
on entry: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9
sorted:   1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
sorted >: 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1
```

See Section 6.8.2, page 228, for an example that demonstrates how to sort according to a member of a class.

The following program demonstrates how `sort()` and `stable_sort()` differ. The program uses both algorithms to sort strings only according to their number of characters by using the sorting criterion `lessLength()` :

Click here to view code image

```
// algo/sort2.cpp

#include "algostuff.hpp"
using namespace std;

bool lessLength (const string& s1, const string& s2)
{
    return s1.length() < s2.length();
}

int main()
{
    // fill two collections with the same elements
    vector<string> coll1 = { "1xxx", "2x", "3x", "4x", "5xx", "6xxxx",
                             "7xx", "8xxx", "9xx", "10xxx", "11", "12",
                             "13", "14xx", "15", "16", "17" };
    vector<string> coll2(coll1);

    PRINT_ELEMENTS(coll1,"on entry:\n ");

    // sort (according to the length of the strings)
    sort (coll1.begin(), coll1.end(),           // range
          lessLength);                          // criterion
    stable_sort (coll2.begin(), coll2.end(),    // range
                 lessLength);                   // criterion

    PRINT_ELEMENTS(coll1,"\nwith sort():\n ");
    PRINT_ELEMENTS(coll2,"\nwith stable_sort():\n ");
}
```

The program has the following output:

```
on entry:
 1xxx 2x 3x 4x 5xx 6xxxx 7xx 8xxx 9xx 10xxx 11 12 13 14xx 15 16 17

with sort():
 2x 3x 4x 17 16 15 13 12 11 9xx 7xx 5xx 1xxx 8xxx 14xx 10xxx 6xxxx

with stable_sort():
 2x 3x 4x 11 12 13 15 16 17 5xx 7xx 9xx 1xxx 8xxx 14xx 6xxxx 10xxx
```

Only `stable_sort()` preserves the relative order of the elements (the leading numbers tag the order of the elements on entry).

11.9.2. Partial Sorting

Click here to view code image

```
void
partial_sort (RandomAccessIterator beg, RandomAccessIterator sortEnd,
              RandomAccessIterator end)

void
partial_sort (RandomAccessIterator beg, RandomAccessIterator sortEnd,
              RandomAccessIterator end, BinaryPredicate op)
```

- The first form sorts the elements in the range `[` *beg* `,` *end* `)` with operator `<`, so range `[` *beg* `,` *sortEnd* `)` contains the elements in sorted order.
- The second form sorts the elements by using the binary predicate

  op(*elem1*,*elem2*)

  as the sorting criterion, so range `[` *beg* `,` *sortEnd* `)` contains the elements in sorted order.
- Note that *op* has to define a *strict weak ordering* for the values (see Section 7.7, page 314, for details).
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- Unlike `sort()`, `partial_sort()` does not sort all elements but stops the sorting once the first elements up to *sortEnd* are sorted correctly. Thus, if, after sorting a sequence, you need only the first three elements, this algorithm saves time because it does not sort the remaining elements unnecessarily.
- If *sortEnd* is equal to *end*, `partial_sort()` sorts the full sequence. It has worse performance than `sort()` on average but better performance in the worst case. See the discussion about sorting algorithms in Section 11.2.2, page 511.
- Complexity: between linear and n-log-n (approximately *numElems* `*log(` *numSortedElems* `)` comparisons).

The following program demonstrates how to use `partial_sort()`:

Click here to view code image

```
// algo/partialsort1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,3,7);
    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,1,5);
    PRINT_ELEMENTS(coll);

    // sort until the first five elements are sorted
    partial_sort (coll.begin(),      // beginning of the range
                  coll.begin()+5,    // end of sorted range
                  coll.end());       // end of full range
    PRINT_ELEMENTS(coll);

    // sort inversely until the first five elements are sorted
    partial_sort (coll.begin(),      // beginning of the range
                  coll.begin()+5,    // end of sorted range
                  coll.end(),        // end of full range
                  greater<int>());   // sorting criterion
    PRINT_ELEMENTS(coll);

    // sort all elements
    partial_sort (coll.begin(),      // beginning of the range
                  coll.end(),        // end of sorted range
                  coll.end());       // end of full range
    PRINT_ELEMENTS(coll);
}
```

The program has the following output:

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 7 6 5 5 6 4 4 3 4 5
7 6 6 5 5 1 2 2 3 3 4 4 3 4 5
1 2 2 3 3 3 4 4 4 5 5 5 6 6 7
```

Click here to view code image

```
RandomAccessIterator
partial_sort_copy (InputIterator sourceBeg, InputIterator sourceEnd,
```

`RandomAccessIterator` *destBeg*, `RandomAccessIterator` *destEnd*)

`RandomAccessIterator`
partial_sort_copy (`InputIterator` *sourceBeg*, `InputIterator` *sourceEnd*,
`RandomAccessIterator` *destBeg*, `RandomAccessIterator` *destEnd*,
`BinaryPredicate` *op*)

- Both forms are a combination of `copy()` and `partial_sort()` .
- They copy elements from the source range [*sourceBeg,sourceEnd*) sorted into the destination range [*destBeg,destEnd*) .
- The number of elements that are sorted and copied is the minimum number of elements in the source range and in the destination range.
- Both forms return the position after the last copied element in the destination range (the first element that is not overwritten).
- If the size of the source range [*sourceBeg,sourceEnd*) is not smaller than the size of the destination range [*destBeg,destEnd*) , all elements are copied and sorted. Thus, the behavior is a combination of `copy()` and `sort()` .
- Note that *op* has to define a *strict weak ordering* for the values (see Section 7.7, page 314, for details).
- Complexity: between linear and n-log-n (approximately *numElems* `*log(` *numSortedElems* `)` comparisons).

The following program demonstrates some examples of `partial_sort_copy()` :

Click here to view code image

```
// algo/partialsort2.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll1;
    vector<int> coll6(6);        // initialize with 6 elements
    vector<int> coll30(30);      // initialize with 30 elements

    INSERT_ELEMENTS(coll1,3,7);
    INSERT_ELEMENTS(coll1,2,6);
    INSERT_ELEMENTS(coll1,1,5);
    PRINT_ELEMENTS(coll1);

    // copy elements of coll1 sorted into coll6
    vector<int>::const_iterator pos6;
    pos6 = partial_sort_copy (coll1.cbegin(), coll1.cend(),
                              coll6.begin(), coll6.end());

    // print all copied elements
    copy (coll6.cbegin(), pos6,
          ostream_iterator<int>(cout," "));
    cout << endl;

    // copy elements of coll1 sorted into coll30
    vector<int>::const_iterator pos30;
    pos30 = partial_sort_copy (coll1.cbegin(), coll1.cend(),
                               coll30.begin(), coll30.end(),
                               greater<int>());

    // print all copied elements
    copy (coll30.cbegin(), pos30,
          ostream_iterator<int>(cout," "));

    cout << endl;
}
```

The program has the following output:

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 3
7 6 6 5 5 5 4 4 4 3 3 3 2 2 1
```

The destination of the first call of `partial_sort_copy()` has only six elements, so the algorithm copies only six elements and returns the end of `coll6` . The second call of `partial_sort_copy()` copies all elements of `coll1` into `coll30`, which has enough room for them, and thus all elements are copied and sorted.

11.9.3. Sorting According to the *n*th Element

Click here to view code image

```
void
nth_element (RandomAccessIterator beg, RandomAccessIterator nth,
             RandomAccessIterator end)

void
nth_element (RandomAccessIterator beg, RandomAccessIterator nth,
             RandomAccessIterator end, BinaryPredicate op)
```

- Both forms sort the elements in the range [*beg* , *end*), so the correct element is at the *n*th position, and all elements in front are less than or equal to this element, and all elements that follow are greater than or equal to it. Thus, you get two subsequences separated by the element at position *n*, whereby each element of the first subsequence is less than or equal to each element of the second subsequence. This is helpful if you need only the set of the *n* highest or lowest elements without having all the elements sorted.
- The first form uses operator `<` as the sorting criterion.
- The second form uses the binary predicate

  `op(elem1,elem2)`

  as the sorting criterion.
- Note that *op* has to define a *strict weak ordering* for the values (see Section 7.7, page 314, for details).
- Note that *op* should not change its state during a function call. See Section 10.1.4, page 483, for details.
- The `partition()` algorithm (see Section 11.8.5, page 592) is also provided to split elements of a sequence into two parts according to a sorting criterion. See Section 11.2.2, page 514, for a discussion of how `nth_element()` and `partition()` differ.
- Complexity: linear on average.

The following program demonstrates how to use `nth_element()`:

Click here to view code image

```
// algo/nthelement1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    deque<int> coll;

    INSERT_ELEMENTS(coll,3,7);
    INSERT_ELEMENTS(coll,2,6);
    INSERT_ELEMENTS(coll,1,5);
    PRINT_ELEMENTS(coll);

    // extract the four lowest elements
    nth_element (coll.begin(),      // beginning of range
                 coll.begin()+3,    // element that should be sorted correctly
                 coll.end());       // end of range

    // print them
    cout << "the four lowest elements are:  ";
    copy (coll.cbegin(), coll.cbegin()+4,
          ostream_iterator<int>(cout," "));
    cout << endl;

    // extract the four highest elements
    nth_element (coll.begin(),      // beginning of range
                 coll.end()-4,      // element that should be sorted correctly
                 coll.end());       // end of range

    // print them
    cout << "the four highest elements are: ";
    copy (coll.cend()-4, coll.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // extract the four highest elements (second version)
```

```
    nth_element (coll.begin(),      // beginning of range
                 coll.begin()+3,    // element that should be sorted correctly
                 coll.end(),        // end of range
                 greater<int>());   // sorting criterion

    // print them
    cout << "the four highest elements are: ";
    copy (coll.cbegin(), coll.cbegin()+4,
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

The program has the following output:

```
3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
the four lowest elements are:  2 1 2 3
the four highest elements are: 5 6 7 6
the four highest elements are: 6 7 6 5
```

11.9.4. Heap Algorithms

In the context of sorting, a *heap* is used as a particular way to sort elements. It is used by heapsort. A heap can be considered a binary tree that is implemented as a sequential collection. Heaps have two properties:

1. The first element is always (one of) the largest.
2. You can add or remove an element in logarithmic time.

A heap is the ideal way to implement a priority queue: a queue that sorts its elements automatically so that the “next” element always is (one of) the largest. Therefore, the heap algorithms are used by the `priority_queue` container (see Section 12.3, page 641). The STL provides four algorithms to handle a heap:

1. `make_heap()` converts a range of elements into a heap.
2. `push_heap()` adds one element to the heap.
3. `pop_heap()` removes the next element from the heap.
4. `sort_heap()` converts the heap into a sorted collection, after which it is no longer a heap.

In addition, since C++11, `is_heap()` and `is_heap_until()` are provided to check whether a collection is a heap or to return the first element that breaks the property of a collection to be a heap (see Section 11.5.5, page 554).

As usual, you can pass a binary predicate as the sorting criterion. The default sorting criterion is operator `<` .

Heap Algorithms in Detail

void
make_heap (RandomAccessIterator *beg*, RandomAccessIterator *end*)

void
make_heap (RandomAccessIterator *beg*, RandomAccessIterator *end*,
BinaryPredicate *op*)

- Both forms convert the elements in the range `[` *beg* `,` *end* `)` into a heap.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op(*elem1*,*elem2*)

- You need these functions only to start processing a heap for more than one element (one element automatically is a heap).
- Complexity: linear (at most, `3*` *numElems* comparisons).

void
push_heap (RandomAccessIterator *beg*, RandomAccessIterator *end*)

void
push_heap (RandomAccessIterator *beg*, RandomAccessIterator *end*,
BinaryPredicate *op*)

- Both forms add the last element that is in front of *end* to the existing heap in the range `[` *beg* `,` *end* `-1` `)` so that the whole range `[` *beg* `,` *end* `)` becomes a heap.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op(*elem1*,*elem2*)

- The caller has to ensure that, on entry, the elements in the range `[` *beg* `,` *end* `-1 )` are a heap (according to the same sorting criterion) and that the new element immediately follows these elements.
- Complexity: logarithmic (at most, `log(` *numElems* `)` comparisons).

```
void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end)

void
pop_heap (RandomAccessIterator beg, RandomAccessIterator end,
          BinaryPredicate op)
```

- Both forms move the highest element of the heap `[` *beg* `,` *end* `)` , which is the first element, to the last position and create a new heap from the remaining elements in the range `[` *beg* `,` *end* `-1 )` .
- *op* is an optional binary predicate to be used as the sorting criterion:

  op (*elem1*, *elem2*)

- The caller has to ensure that, on entry, the elements in the range `[` *beg* `,` *end* `)` are a heap (according to the same sorting criterion).
- Complexity: logarithmic (at most, `2*log(` *numElems* `)` comparisons).

```
void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end)

void
sort_heap (RandomAccessIterator beg, RandomAccessIterator end,
           BinaryPredicate op)
```

- Both forms convert the heap `[` *beg* `,` *end* `)` into a sorted sequence.
- *op* is an optional binary predicate to be used as the sorting criterion:

  op (*elem1*, *elem2*)

- Note that after this call, the range is no longer a heap.
- The caller has to ensure that, on entry, the elements in the range `[` *beg* `,` *end* `)` are a heap (according to the same sorting criterion).
- Complexity: n-log-n (at most, *numElems* `*log(` *numElems* `)` comparisons).

Example Using Heaps

The following program demonstrates how to use the different heap algorithms:

[Click here to view code image]

```
// algo/heap1.cpp

#include "algostuff.hpp"
using namespace std;

int main()
{
    vector<int> coll;

    INSERT_ELEMENTS(coll,3,7);
    INSERT_ELEMENTS(coll,5,9);
    INSERT_ELEMENTS(coll,1,4);

    PRINT_ELEMENTS (coll, "on entry:           ");

    // convert collection into a heap
    make_heap (coll.begin(), coll.end());

    PRINT_ELEMENTS (coll, "after make_heap():  ");

    // pop next element out of the heap
    pop_heap (coll.begin(), coll.end());
    coll.pop_back();

    PRINT_ELEMENTS (coll, "after pop_heap():   ");

    // push new element into the heap
    coll.push_back (17);
```

```
        push_heap (coll.begin(), coll.end());

        PRINT_ELEMENTS (coll, "after push_heap():  ");

        // convert heap into a sorted collection
        // - NOTE: after the call it is no longer a heap
        sort_heap (coll.begin(), coll.end());

        PRINT_ELEMENTS (coll, "after sort_heap(): ");
    }
```

The program has the following output:

Click here to view code image

```
on entry:             3 4 5 6 7 5 6 7 8 9 1 2 3 4
after make_heap():    9 8 6 7 7 5 5 3 6 4 1 2 3 4
after pop_heap():     8 7 6 7 4 5 5 3 6 4 1 2 3
after push_heap():    17 7 8 7 4 5 6 3 6 4 1 2 3 5
after sort_heap():    1 2 3 3 4 4 5 5 6 6 7 7 8 17
```

After `make_heap()` , the elements are sorted as a heap:

```
9 8 6 7 7 5 5 3 6 4 1 2 3 4
```

Transform the elements into a binary tree, and you'll see that the value of each node is less than or equal to its parent node (Figure 11.1). Both `push_heap()` and `pop_heap()` change the elements so that the invariant of this binary tree structure — each node not greater than its parent node — remains stable.

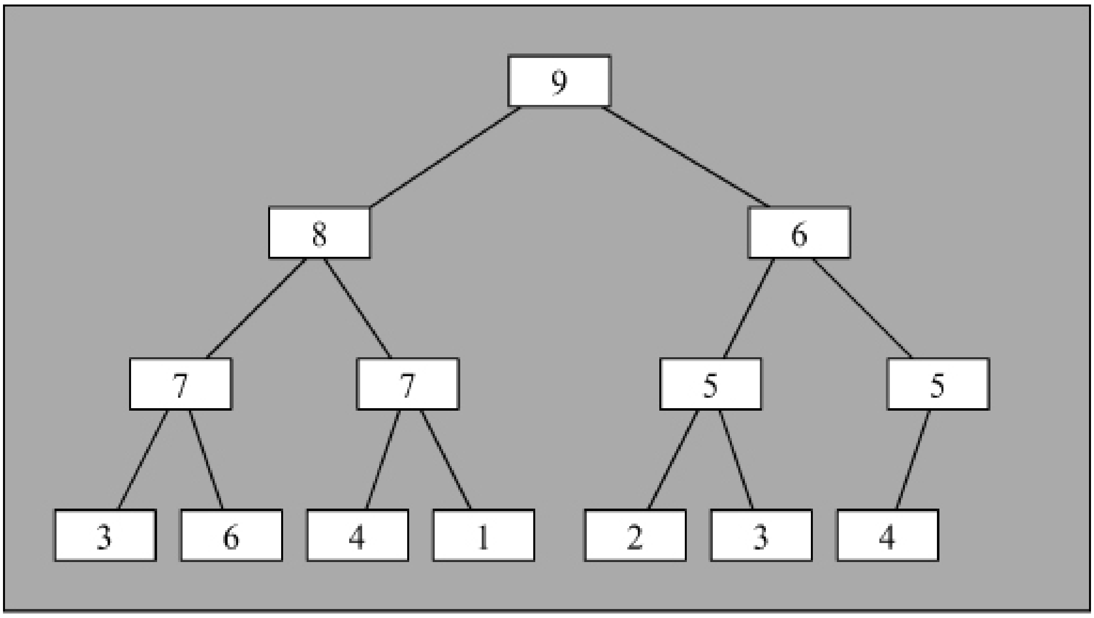

Figure 11.1. Elements of a Heap as a Binary Tree