DATA STRUCTURES AND ALGORITHMS

Lecture 13

Learning outcome 5

SORTING WITH STL

Introduction

- STL comes with good algorithms that can satisfy all our sorting needs
- Basic function is `sort()` with complexity $O(n \log n)$ that comes in two versions:
  - `sort(begin, end)`
    - Sorts range [begin, end) in ascending order using `operator<`
  - `sort(begin, end, comparator)`
    - Sorts range [begin, end) using comparator function
    - Comparator function takes two objects and returns whether first object should come before the second
      - Similar, but not the same as with the priority queue (there the comparator was a structure)

An example of ascending sorting

```
int numbers[] = { 3, 1, 5, 2, 4};
sort(numbers, numbers + 5);
for (int i = 0; i < 5; i++) {
    cout << numbers[i] << ' ';
}
cout << endl;

vector<string> people({ "Zeljka", "Anica", "Mirko",
                                        "Ivana", "Branko" });
sort(people.begin(), people.end());
for (auto it = people.begin(); it != people.end(); ++it) {
    cout << *it << ' ';
}
cout << endl;
```

Example of descending and object sorting

```
struct Rectangle {
    int a;
    int b;
    Rectangle(int a, int b) {
        this->a = a;
        this->b = b;
    }
    int area() {
        return this->a * this->b;
    }
};

bool descending(int a, int b) {
    return a > b;
}

bool asc_rectangles(Rectangle a, Rectangle b) {
    return a.area() < b.area();
}
```


Example of descending and object sorting

```
int main() {
    int numbers [] = { 3, 1, 5, 2, 4};
    sort(numbers, numbers + 5, descending);
    for (int i = 0; i < 5; i++) {
        cout << numbers[i] << ' ';
    }
    cout << endl;

    vector<Rectangle> p({ Rectangle(10, 10), Rectangle(2, 2),
    Rectangle(4, 4) });
    sort(p.begin(), p.end(), asc_rectangles);
    for (auto it = p.begin(); it != p.end(); ++it) {
        cout << it->area() << ' ';
    }
    cout << endl;

    return 0;
}
```

Problem

- Prepare a vector with shuffled numbers from 1 to 2 million and sort it ascending. Let's display how long the sorting takes.
- Solution:

```
srand(unsigned(time(0)));
vector<int> v;
for (int i = 1; i <= 2000000; i++) { v.push_back(i); }
random_shuffle(v.begin(), v.end());

auto start = chrono::high_resolution_clock::now();
sort(v.begin(), v.end());
auto end = chrono::high_resolution_clock::now();

cout
   << "Duration: "
   << chrono::duration_cast<chrono::milliseconds>(end-start).count()
   << " ms" << endl;
```


Algorithms under the hood

- Visual Studio implements the sort() function using Intro sort and Insertion sort
- It works like this:
  - If < 32 items are sorted, use Insertion sort
  - If sorting >= 32 elements, use Intro sort
    - Start with Quick sort
      - As a pivot, it uses the median of the first, middle and last elements
    - If Quick sort breaks the defined recursion depth limit, it gets cancelled and Heap sort is used instead
      - Recursion depth limit is usually 2*log*n*

Stable sorting

- If we have the initial elements and sort them by key:

12	49	12	3
Mireau Miric	Yuro Yurich	Anna Anicci	Schtepf Stefi

- Unstable sorting can produce a result:

3	12	12	49
Schtepf Stefi	Anna Anicci	Mireau Miric	Yuro Yurich

- While stable sorting is guaranteed to yield this result:

3	12	12	49
Schtepf Stefi	Mireau Miric	Anna Anicci	Yuro Yurich

Example of stable sorting usage

- Example:
  - We have students sorted by last name and then first name
  - We want to also sort them by the number of points on the exam
  - In addition, we want students who have the same number of points to keep their alphabetical order

Student	Score
Ana Anić	51
Branka Brankić	93
Bruno Brunić	43
Iva Ivić	55
…	
Željko Željkić	93

Sorted by names

Student	Score
Željko Željkić	93
Branka Brankić	93
Iva Ivić	55
Ana Anić	51
…	
Bruno Brunić	43

Unstable

Student	Score
Branka Brankić	93
Željko Željkić	93
Iva Ivić	55
Ana Anić	51
…	
Bruno Brunić	43

Stable

Stable sorting

- In stable sorting, the elements retain their relative position after sorting
- Algorithm `sort()` is not stable
- Function `stable_sort()` is a stable version that can be used instead of `sort()`
  - Same usage `sort()`
  - Has complexity $O(n \log^2 n)$
    - If there is enough memory, it can complete in $O(n \log n)$
  - Uses Merge sort in combination with Insertion sort

Sorting a list

- Some containers do not make sense to sort (queue, stack, map, etc.)
- Function `sort()` can sort all other containers instead lists
  - List does not have a random access via `[]` nor `at()`
- `list<T>::sort()` is also a stable sort based on the Merge sort
  - Has the same complexity

Example of sorting a list

```
bool desc_rectangles(Rectangle a, Rectangle b) {
      return a.area() > b.area();
}

int main() {
      list<Rectangle> l({
          Rectangle(10, 10),
          Rectangle(2, 2),
          Rectangle(4, 4) });

      l.sort(desc_rectangles);

      for (auto it = l.begin(); it != l.end(); ++it) {
            cout << it->area() << ' ';
      }
      cout << endl;

      return 0;
}
```


Sorting a part of the range

- `partial_sort(begin, middle, end)`
  - The elements in front of the `middle` are the smallest elements in the entire range and are sorted
  - There is no guarantee how all other elements will be arranged
  - Uses Heap sort, complexity is $O(n \log m)$, where m is the distance from `begin` to `middle`

Problem

- Display three lowest elements from the vector.
- Solution:

```
vector<int> v({ 8, 3, 1, 5, 2, 4, 6, 7, 9, 10, 13, 11, 15,
12, 14, 16, 17, 19, 20, 18 });

partial_sort(v.begin(), v.begin() + 3, v.end());

for (auto it = v.begin(); it != v.begin() + 3; ++it) {
    cout << *it << ' ';
}
cout << endl;
```


Sorting check

- `is_sorted(begin, end)`
  - Return whether the range [begin, end) is sorted
- `is_sorted_until(begin, end)`
  - Returns the iterator to the first element in the range [begin, end) that is not sorted
  - If it returns the end(), the entire range is sorted

Example

```
srand(unsigned(time(0)));
vector<int> v;
for (int i = 1; i <= 35; i++) { v.push_back(i); }

random_shuffle(v.begin(), v.end());
for (auto it = v.begin(); it != v.end(); ++it) {
        cout << *it << ' ';
}
cout << endl;

cout << "Is sorted: " << is_sorted(v.begin(), v.end()) << endl;

cout << "Sorted elements: ";
auto first_unsorted = is_sorted_until(v.begin(), v.end());
for (auto it = v.begin(); it != first_unsorted; ++it) {
        cout << *it << ' ';
}
cout << endl;
```


Sorting *nth* element

- `nth_element(begin, nth, end)` in linear complexity modifies the range in a way:
  - At position *nth* is the value that would be there if the array was sorted (that value is now in place)
  - The elements in front of *nth* are certainly not larger than the element on *nth*
  - The elements behind *nth* are certainly not smaller than the element on *nth*
- Task: calculate which number is the third smallest number in the range of shuffled numbers from 1 to 2 million. Let’s compare the execution speed compared to sorting the entire range.

Solution

```
srand(unsigned(time(0)));
vector<int> v;
for (int i = 1; i <= 2000000; i++) { v.push_back(i);   }
random_shuffle(v.begin(), v.end());

auto start = chrono::high_resolution_clock::now();
nth_element(v.begin(), v.begin() + 2, v.end());
auto end = chrono::high_resolution_clock::now();

cout
  << "Duration: "
  << chrono::duration_cast<chrono::milliseconds>(end-start).count(
  << " ms" << endl;

cout << *(v.begin() + 2) << endl;
```

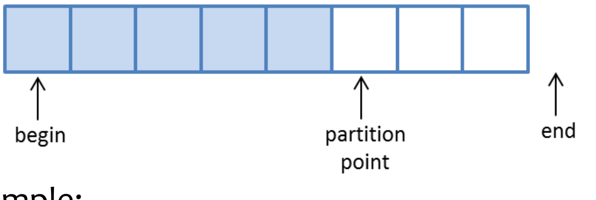

Conclusion

- The main tool
  - Function `sort()` sorts the range according to the criteria
- We want to sort while maintaining the previous relationship of equal elements
  - Function `stable_sort()`, but performance might be lower
- We want to find the value that would be in the *nth* place of the sorted elements
  - Function `nth_element()`
- We want to find all the values from the beginning to the *nth* position of the sorted elements
  - Function `partial_sort()`

PARTITIONING

Partitioning

- Sometimes we need to effectively divide the elements of a container into those that meet the condition and those that do not.



- In this example:
  - Blue-marked elements satisfy the condition
  - White-marked are those who do not satisfy the condition
  - The partitioning point is the first element that does not satisfy the condition

Image taken from fluentcpp.com

Partitioning functions

- The basic functions are:
  - `partition(begin, end, predicate)`
    - Performs partitioning according to the predicate and returns the iterator to the partitioning point
    - A predicate is a function that receives a value and returns a bool
    - Complexity is $O(n)$
  - `is_partitioned(begin, end, predicate)`
    - Checks if the range is partitioned
  - `partition_point(begin, end, predicate)`
    - Returns the iterator to the partitioning point

Problem

- Let's make a vector with shuffled numbers from 1 to 2 million and partition it so that first come even and then odd. Display the first odd number.
- Solution:

```
bool is_even(int n) {
    return n % 2 == 0;
}
int main() {
    srand(unsigned(time(0)));
    vector<int> v;
    for (int i = 1; i <= 2000000; i++) { v.push_back(i); }
    random_shuffle(v.begin(), v.end());
    auto start = chrono::high_resolution_clock::now();
    auto pp = partition(v.begin(), v.end(), is_even);
    auto end = chrono::high_resolution_clock::now();
    cout << "Partition point: " << *pp << endl;
    return 0;
}
```

BINARY SEARCH

Binary search

- Binary search is a procedure in which we look for a value in a sorted container in complexity $O(\log n)$
- Two versions:
  - `binary_search(begin, end, value)`
    - Returns true if the requested value exists within the specified sorted range
  - `binary_search(begin, end, value, comparator)`
    - Returns true if the requested value exists in the specified sorted range, but using the same comparator function as when sorting

Problem

- Prepare a vector with shuffled values of 1 to 2 million. Look for the value of 1,234,456 in the following ways and discuss durations:
  - Linear search in unsorted vector
  - Linear search on a sorted vector
  - Sort + binary search
  - Binary search without sorting