21. Algorithms & Maps

containers ⟷ iterators ⟷ algorithms

21.1 some STL algorithms

½ open interval

```
r = find(b,e,v)          first ==v in [b:e)
r = find_if(b,e,p)       first p(x) in [b:e)
x = count(b,e,v)         count # of ==v in [b:e)
x = count_if(b,e,p)      count # of p(-) true in [b:e)
sort(b,e)                sort [b:e) using op<
sort(b,e,p)              sort using p(-,-) as <
copy(b,e,b2)             copy [b:e) to [b2:b2+(e-b))
                         assume large enough
merge(b,e,b2,e2,r)       merge sorted seq [b:e) and
                         [b2:e2) into sorted [r:r+(e-b)+(e2-b2))
equal(b,e,b2)            all elts == [b,e) and [b2+(e-b))
x = accumulate(b,e,i)    x = i + Σ[b:e)
x = accumulate(b,e,i,op) same, but use op instead of +
```

default comparison ==
ordering <

```
#incl <algorithm>
```

21.2 find

```
template <typename iterator, typename T>
iterator find(const iterator &begin, const iterator &end,
              const T &value) {
    iterator i = begin;
    while (i != end && *i != value) ++i;
    return i
}
```

To call it:

```
vector<int>::iterator p = find(V.begin(), V.end(), x)
if(p == V.end()) not found
        else found @ *p
```

Find is generic

all it requires: ~~iterator has copy ctor, ++, !=~~

iterator i has: copy ctor, i != i, *i, ++i

T x has: x != x

works on list<string>

deque<complex<double>>

vector<foo>, etc.

21.3 find_if

```
template <typename itor, typename pred>
itor find_if(const itor &begin, const itor &end,
             pred p) {
    itor i = begin;
    while (i != end && !p(*i)) ++i;
    return i;
}
```

defn: predicate: fn that returns true or false

ex: `bool odd(int x) { return x % 2; }`

```
auto p = find_if(l.begin(), l.end(), odd);
```

in C++11 also:

```
auto p = find_if(l.begin(), l.end,
                 [](int x){return x % 2;});
```

↑ lambda

~~21.4 Function objects~~

21.4 Function objects

C++21
4

- eg. want to find ~~int~~ x > 41

```
class larger_than {
  const int val;
public:
    larger_than(int v):val(v){ }
    bool operator()(int x)const { return x > val }
}
  /// val const so has to be field int. can't op=
  /// default ctor suppressed
x = find_if(v.begin(), v.end(), larger_than(41));
                                  ↑ ctor
```

~ find_if expr p(*i) means ~
p.operator()(operator*(i))

Abstraction

fn obj is a function that carries state

ex:
```
class foo {
    bar x;
public:
    foo(const bar &xx):x(xx){ }
    const bar& state()const { return x; }
    void reset(const bar &xx) { x = xx; }
    Qux operator()(const baz &bz) {
        ~ whatever ~
    }
}
```

STL mostly uses fn objs for parameterization of effects
esp. searching, sorting, copy

ex:
```
sort (v.begin(), v.end(), compare_by_name())
sort (w.begin(), w.end(), cmp_alphabetically())
```

21.5 Numerical algorithms

```
template<typename itor, typename num>
num accumulate (~~itor~~ const itor &begin,
            const itor &end, num init) {
   for (itor i = begin; i != end; ++i) init += *i
   return init
}
```

— work on containers of doubles, ints,
or anything with operator +=

ex:
```
string allargs = accumulate (argv + 1,
         argv + argc, "");
```

~~[illegible]~~
~~[illegible]~~

C++21
6

```
template <class itor, class T, class binop>
T accumulate (itor i, itor end, T init,
              binop op) {
    while (i != last) {
        init = op (init, *i); ++i
    }
    return init
```

// functional programming calls this
// fold_left
// ocaml: fold_left (+) 0 list

21.6 Associative Containers

map, set — red black trees

unordered_map, unordered_set — hash table, collision resolution by chaining

21.6.1 MAPS

ex: read words, print freq count in lexi. order

```
map<string, int> words;
string s;
while (cin >> s) ++words[s];
for (auto i = words.begin(); i != words.end(); ++i)
    cout << i->first << ":" << i->second << endl;
```

note: the iterator is a pair
map<~>::value_type is pair<string, int>

in c++11:

```
for (auto &i : words) cout << ~~~ .
```

Maps are red/black trees
(balanced binary search trees)
insert/delete/find: O(lg n) time

simplified header:

```
template <typename key, typename value,
          class cmp = less <key>>
class map {
    typedef pair<key,value> value_type
    iterator begin()
    iterator end()
    value &operator[](const key &)
    iterator find(const key &)
    void erase(iterator p)
    pair<iterator, bool>
        insert(const value_type &)
```

success?

21.6.4 Unordered_map

hash table lookup O(1) time
if: - load factor low enough
- good hash func

21.6.5 sets, unordered_sets

keys only - no values

21.7 Copying

copy (b, e, b2) copy [b:e) to [b2:b2+(e-b))
assume space

copy_if(b, e, b2, p) only copy x if p(x) true
- returns end of copied seq.

```
template <class intor, class outtor>
outtor copy (intor i, intor end, outtor to) {
    while (i != end) *to++ = *i++;
    return to;
}
```

example: list<foo> lf; vector<foo> vf;
copy(vf.begin(), vf.end, lf.begin())

ex2: vector<string> argvec(argc - 1);
copy(argv+1, argv+argc, argvec);

ex: read words using >>, print sorted one per line

```
istream_iterator cini(cin);
istream_iterator eof; // default init
ostream_iterator couti (cout, "\n");
```

```
vector<string> vs (cini, eof);
sort (vs.begin(), vs.end());
copy (vs.begin(), vs.end, couti);
```

use unique_copy to avoid dup words

Unix tr '[\t]' '\n' < infile | sort | uniq > outfile

21.8 Sorting & Searching

template <class random_iterator>
void sort(random_itor begin, random_itor end)

3rd param ~~<less>~~
bool less(a,b)
default operator<
will use O(n lg n) sort

sort - uses quicksort, O(n lg n)
but worst case $O(n^2)$

partial_sort ~ uses heapsort
- exactly O(n lg n)
but until 2-5 times slower than qsort
- but can stop after n elts
- ex: find k smallest elts of n

stable_sort = mergesort
- O(n lg n) guaranteed
- order of equal keys preserved
- uses extra memory

each sort:
sort(begin, end)
sort(begin, end, cmpfn)

Binary search

requires random access iterators

bool binary_search(Ran b, Ran e, const Val&)
4th arg cmpless
- assumes sorted
- if not ⇒ undefined (∞ loop?)

p = lower_bound(b,e,v) — first occur of v in [b,e)

p = upper_bound — p → first value > v

pair(p1,p2) = equal_range
- binary search for v, bounded by [p1,p2)