

ordered and unordered

pair type

pair #include<utility>

C++ provides a pair type

- holds **exactly** two values
- is templated on the two values

```
pair<string, int> word_count;
```

single element with two parts, a string and an integer.



members and functions


```
make_pair("hi mom", 12);
```

- returns a pair<string, int> type
 - types inferred by the compiler

```
pair<string, int> wc={"hi mom", 12};
```

- make one and assign

```
wc.first or wc.second
```

- first element or second element
 - not a function, a data member!!!!
- 

can't print a pair

Much like a vector, or any compound pair, you cannot print a pair:

- you have to print the elements
- algorithms are your friends!



ordered associative containers

map, multimap, set, multiset

maps are not a sequence

It is important to remember that a map is not a sequence, there is no order to the elements of a map.

Well, that's kind of a lie, there is actually but the order changes as time goes on. More truthfully, one cannot count on the order of a map, as it constantly changes



bi-directional iterators

These containers yield bi-directional iterators (not a sequence, remember):

- no random access via []
- no pointer arithmetic
- no `itr < v.end()` because no less-than but does allow `itr != v.end()`
 - if there is no fixed order, then there cannot be a `<` (comparison) operator



ordered containers: map

map automatically inserts new elements such that they are ordered:

- each map element is a *pair*
 - (key, value) in that order
- order of map elements is based on the key
- if not specified, the order is based on a less-than compare on keys
- search for element is very fast

maintains order of keys

```

    { {"bill", "555-1212"}, {"jill", "555-2323"} }
      ^      ^
      |      |
    insert  insert
      |      |
    {"alex", "555-4545"} {"eric", "555-3434"}
  
```

looking at just the keys here!

albert, alex, barb, bill, chaz, chuck, eric, ... jill, john, julie, ..., mark, mary, ..., sally, samantha, ..., zach, zeke

find
 "alex", "555-4545"

initialization and keys

```
map<string, string> authors = {  
    {"Joyce", "James"}, {"Austen", "Jane"},  
    {"Dickens", "Charles"}  
};
```

Directly indicate the pairs

Only requirement on keys is that they must have a way to compare keys (these containers are all ordered). Either by default or you provide!3



by iteration

```
using Cnt = pair<char, long>;  
vector<Cnt> v = {{'a', 0}, {'b', 1}};  
map<char, long> m(v.begin(), v.end());
```

Push back pairs (of the correct type) onto the map.



much like python dict

- Every key has an associated value
- fast search is by key to find that value
 - cannot do the reverse, find value and look up key



map, 3 ways to insert


Not `push_back`, rather `insert`!

```
map<string,int> m;  
string word = "hello";  
m.insert({word, 1});  
m.insert(make_pair(word, 1));  
m.insert(pair<string,int>(word, 1));
```




map, return from insert

insert returns a pair<iterator,bool>

- if key is in map, then insert does nothing and the second element of the returned pair is false
 - if key is not in the map, the insert works and the second element of the returned pair is true
 - iterator points to element (whether added or already there)
- 

3 ways to erase(1)

```
map<string, int> m;  
size_t num;  
// removes every example of key  
// returns how many erased  
num = m.erase(key);
```



Can't do this, WHY????

```
auto itr = m.begin() + 2;  
// remove element pointed to by itr.  
// returns itr to next element  
m.erase(itr);
```



map as associative array

as an associative array. Beware, some weird stuff below!


```
map <string, int> word_dict;  
word_dict["bill"] = 10;  
++word_dict["fred"]; //?  
for(auto itr=word_dict.begin(),  
    itr != word_dict.end(), ++itr)  
    cout << itr->first<<endl; //?
```



[] operator

- Like python. The type in the [] is the key and the value is what is associated (what is returned and can be assigned).
- Unlike python. [] operator allows for **non-existing keys**. Any reference to a key that doesn't exist creates the key with the **default value type**.

```
map<int, double> m;
++m[15]; // default double is 0 , add 1
```



what does -> mean

What you iterate through in a map are **pairs**. A **map iterator points to a pair**. If you want to print the key of the pair via the iterator, you could type

```
(*itr).first;
```

As a short cut to the above, you can use

```
itr -> first;
```

"call the member of what the iterator points to"




cannot change a key

Again, iteration is through pairs and the key is a `const` value

- you can view but cannot change a key value via iteration!

```
map<int, int>pt={{2,2}, {4,4}};
for(auto itr=pt.begin(),itr!=pt.end(),
    itr++){
itr->second=itr->second + 2;
// itr->first = itr->first + 4 // error
}
```



count words example

Pretty straight forward to print in word order

Printing in occurrence order is a little work.



find

Can't use [] to check for a value,
because it adds it if it is not there

- `m.find(key)` //itr to key (or end)
- `m.count(key)` // occurrences (1)



can provide a compare function

ordered map (all the ordered types)
maintain an order of pair elements
based on keys

- default is `less_than`

You can provide your own function and
change the order

- easier when we get to classes!



summary

```
pair<K,V> → pair<string, string>
```

```
map<K,V> → map<string, string>
```

- *dual templated*
- *has .first and .second*
- *collection of pairs*
 - *dual templated, the pair template*
 - *.first is the "key",*
 - *.second is the "value"*
- *ordered based on each pair's key*
- *default order: less than (<)*
- *adding/erasing a pair changes map order*
- *keys are const*
 - *they cannot be changed in place*
- *key lookup is fast*
 - *because of the ordering*
- *only one example of a key in the map*
 - *there is however a multimap!*



Sets

Fairly simple

Sets are like you expect

- are templated for one type
- hold only one example of any element
 - if you add another example of an element it is ignored



insert/erase are the same

insert on a set returns a pair just like before

- now, the iterator points to the base type, not a pair
- erase erases all examples of the key
 - only one, so...



iterators on sets are const

You can iterate through a set, but the iterator is const

- cannot change a key in place.



summary

$\text{map}\langle K, V \rangle \rightarrow \text{map}\langle \text{string}, \text{string} \rangle$

- collection of pairs
 - dual templated, pair template
- ordered based on each pair's key
- default order: less than (<)
- adding/erasing a pair changes order
- keys are const,
 - they cannot be changed in place
- key lookup is fast
 - because of the ordering
- only one example of a key in the map
 - there is however a multimap!

$\text{set}\langle T \rangle \rightarrow \text{set}\langle \text{string} \rangle$

- collection of elements
 - single template, element type
- ordered based on the elements
- default order: less than (<)
- adding/erasing an element changes order
- elements are const
 - they cannot be changed in place
- element lookup is fast
 - because of the ordering
- only one example of an element in the set
 - there is however a multiset!



set like algorithms

Interestingly there are no methods for sets like union, intersection etc.

Instead, there are generic algorithms which can be used on any container to get that kind of behavior



set algorithms (2)

for the algorithms to work, they must be working with a ***sorted container***

- weird/undefined behavior if not already sorted




set algorithms(3)


General form:

```
algorithm(src1-iter, src1-iter  
          src2-iter, src2-iter,  
          dest-iter);
```

assumption is src1 and src2 are sorted.
dest-iter is either another container or
an output iterator



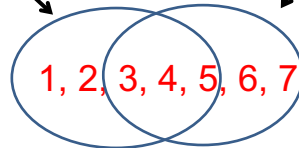
set algorithms(4)

- `set_union`
 - as you would guess
 - `set_intersection`
 - as you would guess
 - `set_difference`
 - those things in src1 not found in src2.
order dependent!
 - `set_symmetric_difference`
 - those things found in src1 and src2 that
are not common between them
- 

set_union

$s1 = \{1, 2, 3, 4, 5\}$

$s2 = \{3, 4, 5, 6, 7\}$

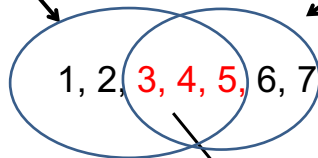


returns $\{1, 2, 3, 4, 5, 6, 7\}$

set_intersection

$s1 = \{1, 2, 3, 4, 5\}$

$s2 = \{3, 4, 5, 6, 7\}$

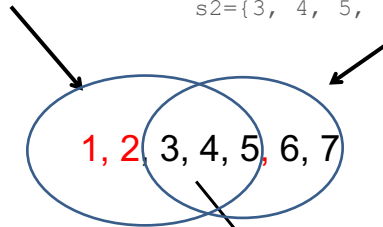


returns $\{3, 4, 5\}$

set_difference (s1 - s2, order matters)

s1={1, 2, 3, 4, 5}

s2={3, 4, 5, 6, 7}

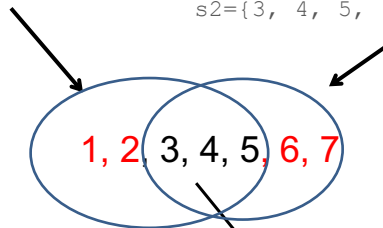


returns {1, 2}

set_symmetric_difference

s1={1, 2, 3, 4, 5}

s2={3, 4, 5, 6, 7}



returns {1, 2, 6, 7}

What about repeats?

These algorithms work on any STL container. What happens with repeats?

Remember, if you want to hold on to repeats you need to insert them into a container that allows repeats.



set_union

v1={1, 2, 2, 2, 3, 4, 5}

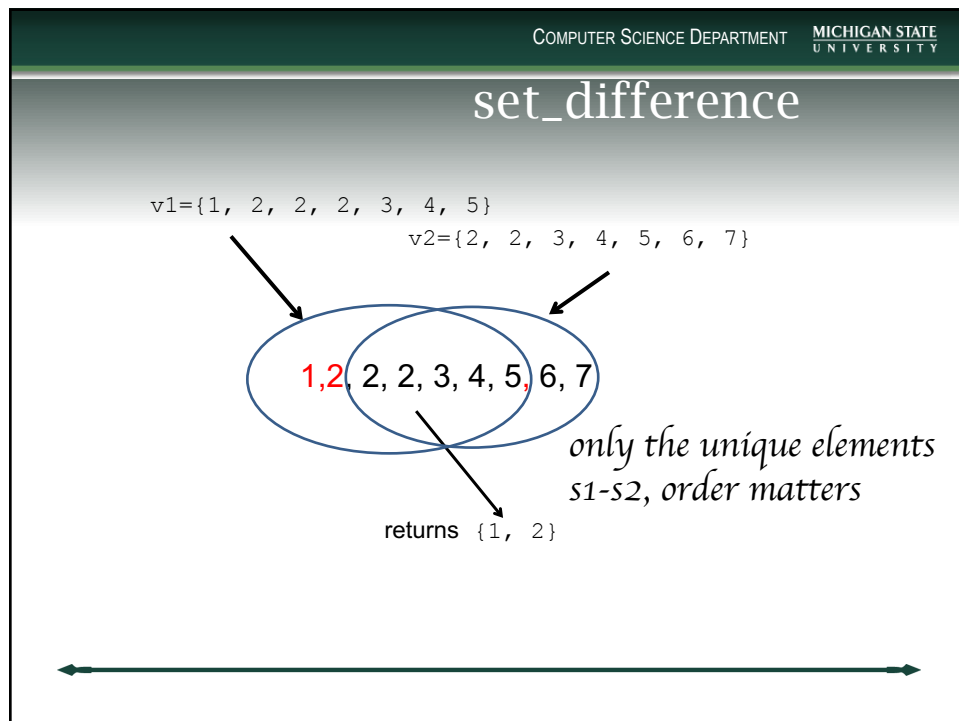
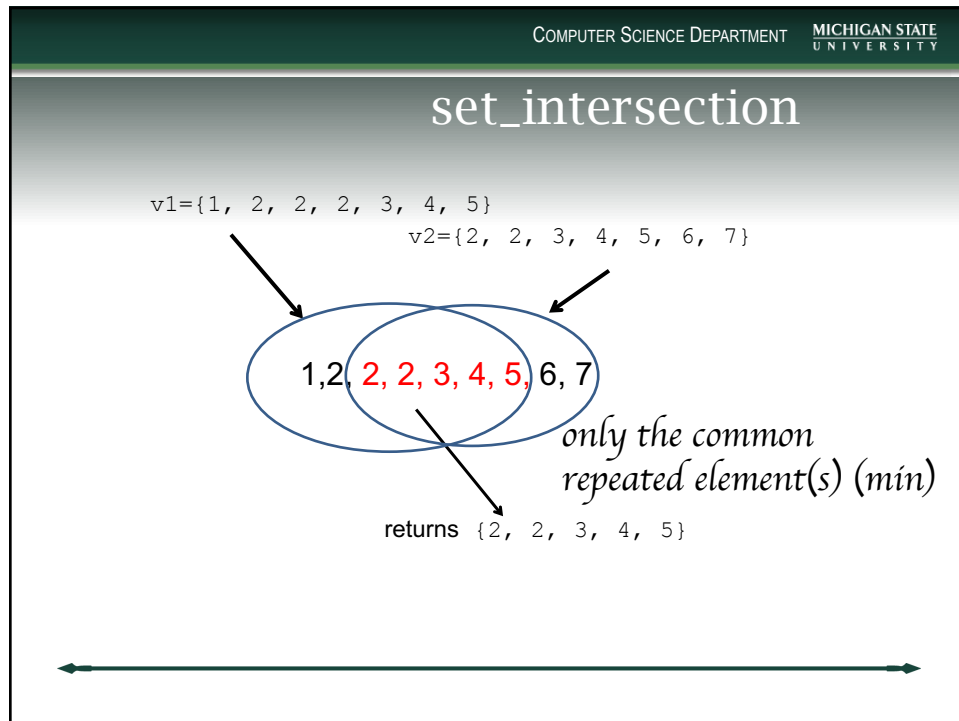
v2={2, 2, 3, 4, 5, 6, 7}

1, 2, 2, 2, 3, 4, 5, 6, 7

max of the repeated element(s)

returns {1, 2, 2, 2, 3, 4, 5, 6, 7}





COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

set_symmetric_difference

$v1 = \{1, 2, 2, 2, 3, 4, 5\}$
 $v2 = \{2, 2, 3, 4, 5, 6, 7\}$

opposite of intersection

returns $\{1, 2, 6, 7\}$

COMPUTER SCIENCE DEPARTMENT MICHIGAN STATE UNIVERSITY

multiset, multimap

This slide displays a large block of text, likely representing a data structure or a list of elements, used in the context of multiset and multimap operations. The text is organized into columns and rows, with some elements highlighted in red.

multiple examples, same key, allowed

For both multiset and multimap,
multiple examples of a key are allowed

- multimap is nice for "overloaded" keys (one word, multiple definitions).
- cannot use [] for either
- find is useful here




more multi

- insert here returns the iterator, not a pair
 - insert always works, since multiple keys
- count can now return more than 1,0
- find is the first element with the key
 - or end if not there




find all examples of key

```
...  
mm = {{1,2}, {1,3}, {2,3}, {1,4}};  
auto cnt = m.count(1);  
auto itr = m.find(1)  
while (cnt){  
    cout << itr->second;  
    ++itr;  
    --cnt;  
}
```



more multimap methods

- lower_bound : first occurrence
 - upper_bound : last occurrence
 - equal_range : a pair of iterators. first is first occurrence and last is last occurrence.
- 

assumes equal keys right next to
each other

clearly this all assumes that equal keys
are right next to each other, and that is
true for multimap

multimap is "arranged" based on keys.



unordered containers

unordered_map, unordered_set,

unordered_multimap, unordered_multiset

a difference of implementation

The unordered types do not necessarily introduce any new capabilities from the point of view of the user.

Rather than provide a new interface, they provide a new underlying implementation



order vs hashing

If the elements of a container are ordered, search for an element is very fast:

- binary search

Another approach is called hashing

- make a key out of some processing of the value being stored



Bottom line, order is easy

The bottom line is that the ordered containers are pretty straight forward to work with and, until we get to hashing, easier to understand.

