Stanford C++ sets (5.5)

- `Set`: implemented using a linked structure called a *binary tree*.
  - pretty fast; elements are stored in **sorted order**
  - values must have a < operation

- `HashSet`: implemented using a special array called a *hash table*.
  - *very* fast; elements are stored in **unpredictable order**
  - values must have a `hashCode` function *(provided for most standard types)*
    - variant: `LinkedHashSet` *(slightly slower, but remembers insertion order)*

How to choose: Do you need the elements to be in sorted order?

- If so: Use `Set`.
- If not: Use `HashSet` for the performance boost.

Set members

```
#include "set.h"
#include "hashset.h"
```

Member	Set	HashSet	Description
`s.add(value);`	O(log *N*)	**O(1)**	adds given value to set
`s.clear();`	O(*N*)	O(*N*)	removes all elements of set
`s.contains(value)`	O(log *N*)	**O(1)**	`true` if given value is found
`s.isEmpty()`	O(1)	O(1)	`true` if set contains no elements
`s.isSubsetOf(set)`	O(*N* log *N*)	O(*N*)	`true` if *set* contains all of this one
`s.remove(value);`	O(log *N*)	**O(1)**	removes given value from set
`s.size()`	O(1)	O(1)	number of elements in set
`s.toString()`	O(*N*)	O(*N*)	e.g `"{3, 42, -7, 15}"`
`ostr << s`	O(*N*)	O(*N*)	print set to stream

Set operators

`s1 == s2`	`true` if the sets contain exactly the same elements
`s1 != s2`	`true` if the sets don't contain the same elements
`s1 + s2`	returns the **union** of s1 and s2 (elements from either)
`s1 += s2;`	sets s1 to the union of s1 and s2 (or adds a value to s1)
`s1 * s2`	returns **intersection** of s1 and s2 (elements in both)
`s1 *= s2;`	sets s1 to the intersection of s1 and s2
`s1 - s2`	returns **difference** of s1, s2 (elements in s1 but not s2)
`s1 -= s2;`	sets s1 to the difference of s1 and s2 (or removes a value from s1)

```
Set<string> set;
set += "Jess";
set += "Alex";
Set<string> set2 {"a", "b", "c"};   // initializer list
...
```

and differences using operators

Looping over a set

```
// forward iteration with for-each loop (read-only)
for (type name : collection) {
    statements;
}
```

- sets have no indexes; can't use normal `for` loop with index `[i]`
- `Set` iterates in sorted order; `HashSet` in unpredictable order

~~
```
for (int i = 0; i < set.size(); i++) {
    do something with set[i];          // does not compile
}
```
~~

it is
that doesn't work

Set exercise

isHappyNumber

- Write a function **isHappyNumber** that returns whether a given integer is "happy". An integer is "happy" if repeatedly summing the squares of its digits eventually leads to the number 1.
  - Examples:

139 is happy:	4 is unhappy:
$1^2+3^2+9^2 = 91$	$4^2 = 16$
$9^2+1^2 = 82$	$1^2+6^2 = 37$
$8^2+2^2 = 68$	$3^2+7^2 = 58$
$6^2+8^2 = 100$	$5^2+8^2 = 89$
$1^2+0^2+0^2 = 1$	$8^2+9^2 = 145$

$1^2+4^2+5^2 = 42$

$4^2+2^2 = 20$

$2^2+0^2 = 4$

that doesn't work there okay
all right um

  - How can this problem be solved using a set?

Structs

- C++ has an entity called a *structure* (**struct**).
  - Like a lightweight class; a collection of data and behavior.
  - By default, member variables and methods are public.

```
struct Date {           // declaring a struct type
    int month;
    int day;            // members of each Date structure
};
...

Date today;             // construct structure instances
today.month = 7;
today.day = 13;

Date xmas {12, 25};     // initialize on one line
```

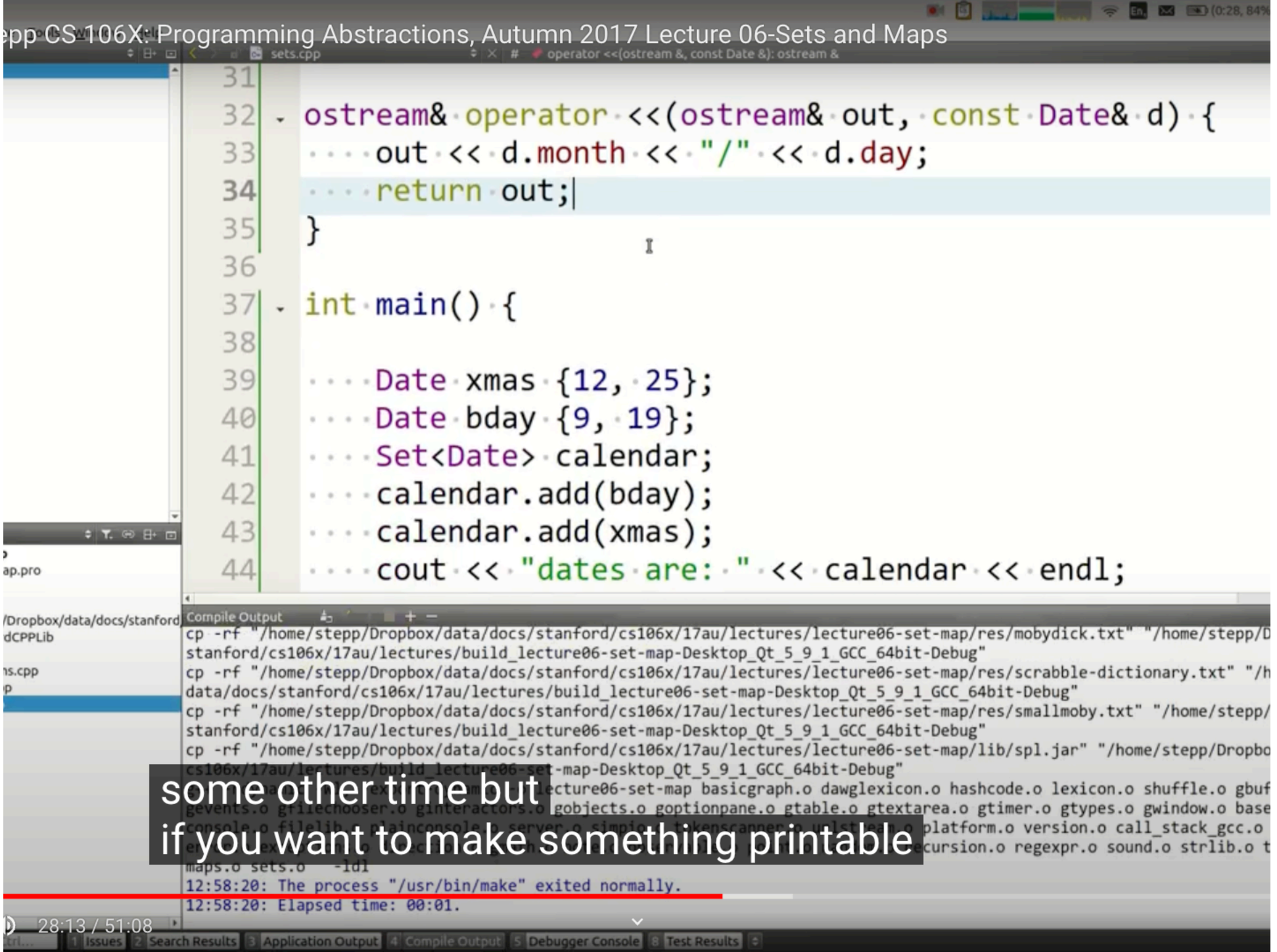
no methods but just two instance variables

Set/HashSet of structs

- By default, our libs can't store your struct in a `Set` or `HashSet`.
  - To be in a `Set`, your struct needs to be sortable using a **< operator**.
  - To be in a `HashSet`, you need **==**, **!=**, and a **hashCode** function.
    - We will talk about these ideas in much more detail later.

```
// operator overloading; define < operation on dates
// (required in order to put Date in a Set)
bool operator <(const Date& d1, const Date& d2) {
    return d1.month < d2.month
        || (d1.month == d2.month && d1.day < d2.day);
}

// converts a Date into an integer key
// (required in order to put Date in a HashSet)
int hashCode(const Date& date) {
    return date.month * 100 + date.day;
}
```



some other time but
if you want to make something printable

Stanford Lexicon (5.6)

```
#include "lexicon.h"
```

- A set of words optimized for dictionary and prefix lookups

Member	Big-Oh	Description
`Lexicon name;` `Lexicon name("file");`	O(*N*len*)	create empty lexicon or read from file
`L.add(word);`	O(*len*)	adds the given word to lexicon
`L.addWordsFromFile("f");`	O(*N*len*)	adds all words from input file (one per line)
`L.clear();`	O(*N*len*)	removes all elements of lexicon
`L.contains("word")`	O(*len*)	`true` if word is found in lexicon
`L.containsPrefix("str")`	O(*len*)	`true` if `s` is the start of any word in lexicon
`L.isEmpty()`	O(1)	`true` if lexicon contains no words
`L.remove("word");`	O(*len*)	removes word from lexicon, if present
`L.removePrefix("str");`	O(*len*)	removes all words that start with prefix
`L.size()`	O(1)	number of elements in lexicon
`L.toString()`	O(*N*)	e.g. `{"arm", "cot", "zebra"}`

Lexicon example

```
#include "lexicon.h"
...

// read file into a dictionary lexicon
Lexicon dictionary;
dictionary.addWordsFromFile("dict.txt");

// look up words/prefixes in the dictionary
string word = getLine("type a word: ");
if (dictionary.contains(word)) {
    cout << "That word is in the dictionary!" << endl;
} else if (dictionary.containsPrefix(word)) {
    cout << "Some words start with " << word << endl;
}
```

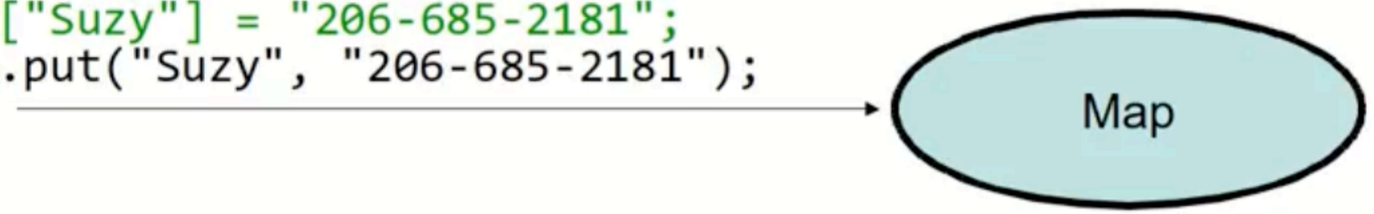
print the words in sorted order
yep uh okay

Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

    Store pair ("Suzy", "206-685-2181").

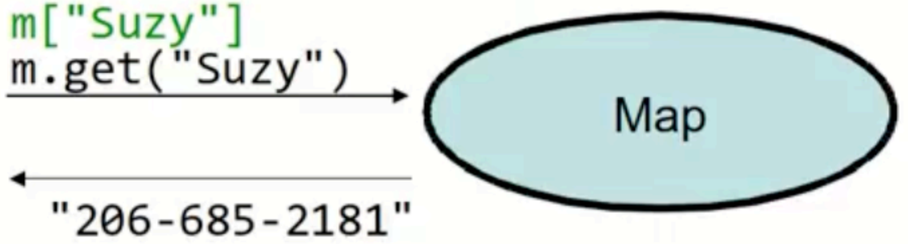
```
//      key           value
// m["Suzy"] = "206-685-2181";
   m.put("Suzy", "206-685-2181");
```



  - Later, we can supply only the key and get back the related value:

    What is Suzy's phone number?

```
// m["Suzy"]
   m.get("Suzy")
```



Map implementation

- in the Stanford C++ library, there are two map classes:
  - **`Map`**: implemented using a linked structure called a *binary search tree*.
    - pretty fast for all operations; keys are stored in **sorted order**
    - both kinds of maps implement exactly the same operations
    - the keys' type must be a comparable type with a < operation
  - **`HashMap`**: implemented using a special array called a *hash table*.
    - *very* fast, but keys are stored in unpredictable order
    - the keys' type must have a hashCode function (but most types have one)
- Requires 2 type parameters: one for keys, one for values.

```
// maps from string keys to integer values
Map<string, int> votes;
```

Map members

Member	Map	HashMap	Description
`m.clear();`	O(*N*)	O(*N*)	removes all key/value pairs
`m.containsKey(key)`	O(log *N*)	**O(1)**	`true` if map has a pair with given key
`m[key]` or `m.get(key)`	O(log *N*)	**O(1)**	returns value mapped to given key; if not found, **adds** it with a default value
`m.isEmpty()`	O(1)	O(1)	`true` if the map contains no pairs
`m.keys()`	O(*N*)	O(*N*)	a `Vector` copy of all keys in map
`m[key] = value;` or `m.put(key, value);`	O(log *N*)	**O(1)**	adds a key/value pair; if key already exists, **replaces** its value
`m.remove(key);`	O(log *N*)	**O(1)**	removes any pair for given key
`m.size()`	O(1)	O(1)	returns number of pairs in map
`m.toString()`	*O(N)*	*O(N)*	e.g. `"{a:90, d:60, c:70}"`
`m.values()`	O(*N*)	O(*N*)	a `Vector` copy of all values in map
`ostr << m`	O(*N*)	O(*N*)	prints map to stream