COSC 220: Data Structures in C++

Data Structures: Queue Class

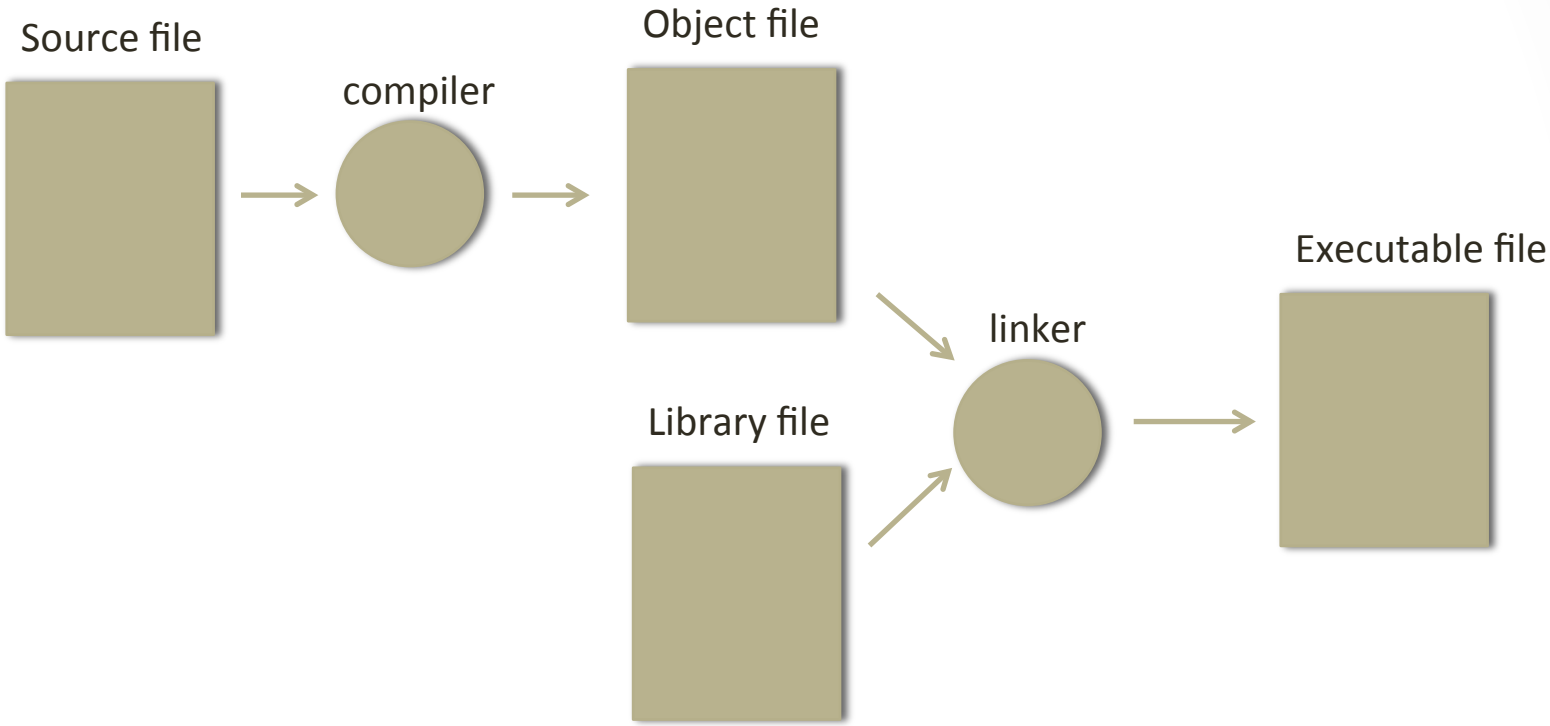
This week (week 8):
Queue, Map, and Set data structures will be discussed

Make sure the Static Stanford Libraries are working on your computer!! We got 4 working in 1 office hour last Friday!

HW#3 online Weds

COSC 220: Data Structures in C++

Review of the Compilation Process:



1. **Source file**: text of your code
2. **Compiler** → translate source to **object file**
3. **Object file**: machine-language instructs
4. Object file combined with other object files via Linker → **Executable file**
5. Other object files: predefined obj files called **libraries**
6. **Libraries**: contain machine-lang instructs for various operations req'd by program
7. **Linking**: combining object files into an executable

COSC 220: Data Structures in C++

Data Structures: Queue Class

The Queue Class

Counterpart to Stack Class.
FIFO: first in first out
Stack → LIFO

Stack: push → add to top of stack
pop: → remove from top of stack

e.g. nested function call: last function called is first to return

Queue counterpart to Stack
enqueue → add element to end of queue
dequeue → remove element from front of queue

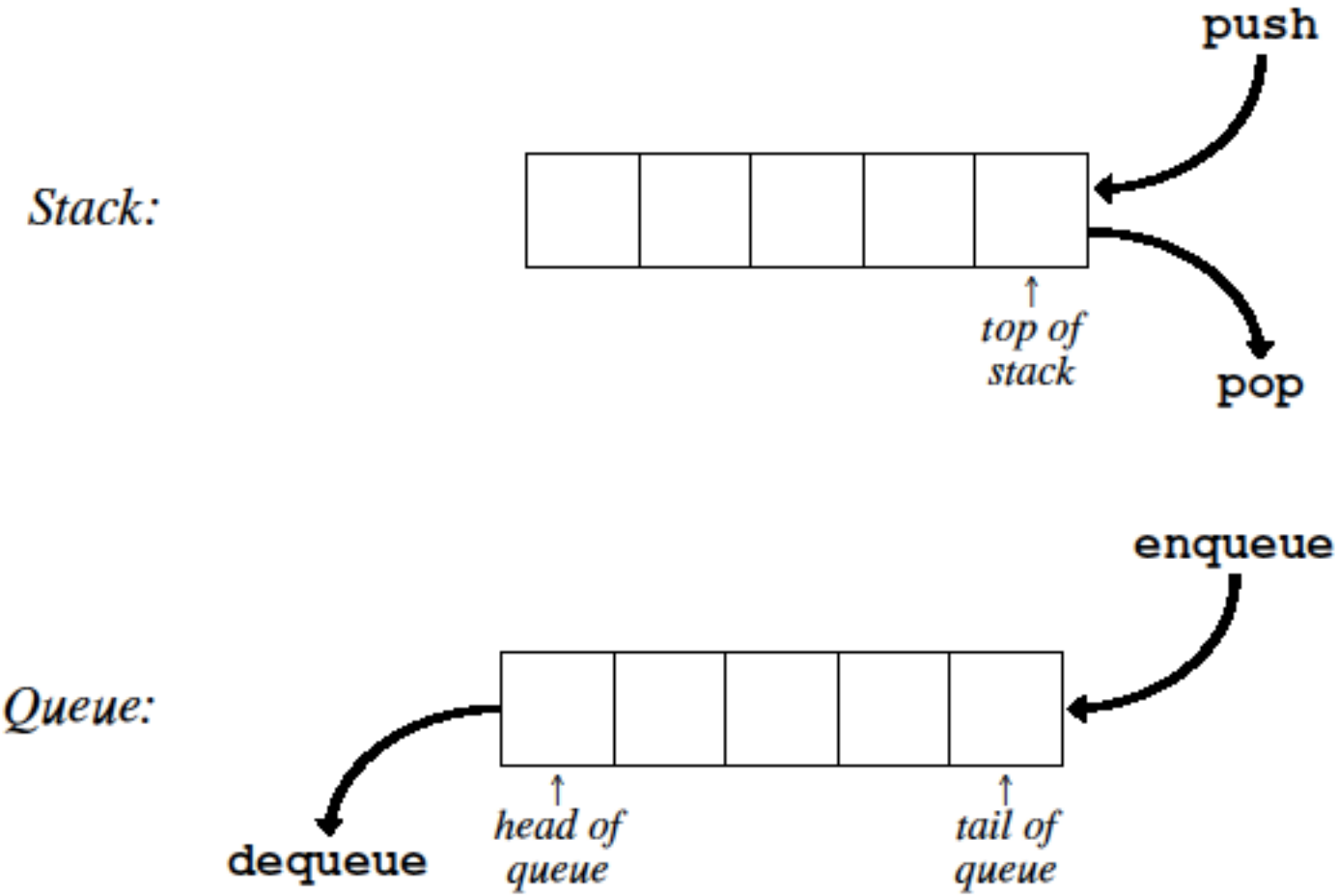
COSC 220: Data Structures in C++

Data Structures: Queue Class

Queue Class:

enqueue → add element to end of queue

dequeue → remove element from front of queue



COSC 220: Data Structures in C++

Data Structures: Queue Class

The Queue Class

TABLE 5-4 Entries exported by the `queue.h` interface

Constructor

`Queue<`*type*`>()`	Creates an empty queue capable of holding values of the specified type.

Methods

`size()`	Returns the number of elements currently in the queue.
`isEmpty()`	Returns `true` if the queue is empty.
`enqueue(`*value*`)`	Adds *value* to the tail of the queue.
`dequeue()`	Removes the element at the head of the queue and returns it to the caller. Calling `dequeue` on an empty queue generates an error.
`peek()`	Returns the value at the head of the queue without removing it. Calling `peek` on an empty queue generates an error.
`clear()`	Removes all the elements from a queue.

COSC 220: Data Structures in C++

Data Structures: Queue Class

The Queue Class useful in simulations:

→waiting in line modeling
→traffic modeling: car following and cars acceleration from light, see math 354

Class Exercise 1: write a function that reverses a queue. Keep in mind you cannot see any internal rep'n of the queue

```
void reverseQueue(Queue<int> & queue);
```

COSC 220: Data Structures in C++

Data Structures: Queue Class

The Queue Class useful in simulations:

-waiting in line modeling
-traffic modeling: car following and cars acceleration from light, see math 354

Class Exercise 1: write a function that reverses a queue. Keep in mind you cannot see any internal rep'n of the queue

```
void reverseQueue(Queue<int> & queue){
    int qSize = queue.size();
    Stack<int> stack;
    for (int i = 0; i < qSize; i++){
        stack.push(queue.dequeue());
    }
    for (int i = 0; i < qSize; i++){
        queue.enqueue(stack.pop());
    }
}
```

COSC 220: Data Structures in C++

Data Structures: Queue Class

Write a function isPalindrome that returns boolean based on if a phrase is a palindrome or not using both the stack and queue classes.

```
bool isPalindrome(string str);
```

COSC 220: Data Structures in C++

Data Structures: Queue Class

Write a function isPalindrome that returns boolean based on if a phrase is a palindrome or not using both the stack and queue classes.

```
bool isPalindrome(string str) {
    Queue<char> queue;
    Stack<char> stack;
    long int n = str.length(); // method length for class
string
    for (int i = 0; i < n; i++ ){
        queue.enqueue(str[i]);
        stack.push(str[i]);
    }
    for (int i = 0; i < n / 2; i++ ){
        if (queue.dequeue() != stack.pop()) return false;
       // note compare FIFO to LIFO
    }
    return true;
}
```

COSC 220: Data Structures in C++

Data Structures: Map Class

The Map Class
Associative arrays
Closest thing to a dictionary seen in Python
Association between a key and value

maps: sometimes called *symbol tables*
constructor: **Map**<*key type*, *value type*> map

Associative arrays:
map[key] = value;
same as:
map.put(key, value);

COSC 220: Data Structures in C++

Data Structures: Map Class

TABLE 5-5 Entries exported by the `map.h` interface

Constructors

`Map<`*key type*`,` *value type*`>()`	Creates an empty map associating keys and values.

Methods

`size()`	Returns the number of key/value pairs contained in the map.
`isEmpty()`	Returns `true` if the map is empty.
`put(`*key*`,` *value*`)`	Associates the specified key and value in the map. If *key* has no previous definition, a new entry is added; if a previous association exists, the old value is discarded and replaced by the new one.
`get(`*key*`)`	Returns the value currently associated with *key* in the map. If there is no such value, `get` generates an error.
`remove(`*key*`)`	Removes *key* from the map along with any associated value. If *key* does not exist, this call leaves the map unchanged.
`containsKey(`*key*`)`	Checks to see whether *key* is associated with a value. If so, this method returns `true`; if not, it returns `false`.
`clear()`	Removes all the key/value pairs from the map.

Operators

map`[`*key*`]`	The `Map` class overloads the square bracket operator so that a map acts as an *associative array* indexed by the key value. If the key does not exist in the map, the square bracket operator creates a new entry and sets its value to the default for that type.

COSC 220: Data Structures in C++

Data Structures: Map Class

See AirportCodes.cpp (www.fredpark.com/teaching)

1. airport code and airport info will be key value pair

2. Class exercise, join a partner and think about an interesting application using the map class. Write pseudo code and then implement it.

COSC 220: Data Structures in C++

Data Structures: Map Class

The Map Class
Associative arrays
Closest thing to a dictionary seen in Python
Association between a key and value

maps: sometimes called *symbol tables*
constructor: **Map**<*key type*, *value type*> map

Associative arrays:
map[key] = value;
same as:
map.put(key, value);

COSC 220: Data Structures in C++

Data Structures: Set Class

The Set Class

Models the mathematical abstraction of a set

This is an unordered collection of elements occurring only once

Lots of algorithmic applications.

COSC 220: Data Structures in C++

Data Structures: Set Class

The Set Class

TABLE 5-6 Entries exported by the `set.h` interface

Constructor

`Set<type>()`	Creates an empty set containing values of the specified type.

Methods

`size()`	Returns the number of elements in the set.
`isEmpty()`	Returns `true` if the set is empty.
`add(`*value*`)`	Adds the value to the set. If the value is already in the set, no error is generated, and the set remains unchanged.
`remove(`*value*`)`	Removes the value to the set. If the value is not present, no error is generated, and the set remains unchanged.
`contains(`*value*`)`	Returns `true` if the value is in the set.
`clear()`	Removes all elements from the set.
`isSubsetOf(`*set*`)`	Returns `true` if this set is a subset of the set passed as an argument.
`first()`	Returns the first element of the set in the ordering specified for its value type.

Operators

s_1 `+` s_2	Returns the ***union*** of s_1 and s_2, which consists of the elements in either or both of the original sets.
s_1 `*` s_2	Returns the ***intersection*** of s_1 and s_2, which consists of the elements common to both of the original sets.
s_1 `-` s_2	Returns the ***set difference*** of s_1 and s_2, which consists of the all elements in s_1 that are not present in s_2.
s_1 `+=` s_2 s_1 `-=` s_2 s_1 `*=` s_2	The `+`, `-`, and `*` operators can be combined with assignment just as they can with numeric values. For `+=` and `-=`, the right hand value can be a set, a single value, or a list of values separated by commas.

COSC 220: Data Structures in C++

Data Structures: Set Class

Scrabble or a spell checker: can use a set to store the words since definitions are not needed

Set of words with no associated definitions is called a *lexicon*
A set based implementation from a file looks like:

```
Set<string> lexicon;
ifstream infile;
infile.open("EnglishWords.txt");
if (infile.fail()) error("Can't open
EnglishWords.txt");
while (true) {
   string word;
   getline(infile, word);
   if (infile.fail()) break;
   lexicon.add(word);
}
infile.close();
```

COSC 220: Data Structures in C++

Data Structures: lexicon class

Set of words with no associated definitions is called a *lexicon*
Set based implementation is useful but not efficient.

Use lexicon class.
Load “EnglishWords.dat” contains reasonably complete list of English words.

```
Lexicon english("EnglishWords.dat"); // lexicon class
```

COSC 220: Data Structures in C++

Data Structures: lexicon class

TABLE 5-7 Entries exported by the `lexicon.h` interface

Constructors

`Lexicon()`	Creates an empty lexicon.
`Lexicon``(`*file*`)`	Initializes a lexicon by reading data from a file.

Methods

`size()`	Returns the number of words in the lexicon.
`isEmpty()`	Returns **`true`** if the lexicon is empty.
`add``(`*word*`)`	Adds a new word to the lexicon, if it is not already present. All words in a lexicon are stored in lower case.
`addWordsFromFile``(`*file*`)`	Adds all the words in the named file to the lexicon. The file must either be a text file, in which case the words are listed on separate lines, or a compiled data file specifically formatted for the lexicon.
`contains``(`*word*`)`	Returns **`true`** if *word* is in the lexicon.
`containsPrefix``(`*prefix*`)`	Returns **`true`** if any of the words in the lexicon start with the specified prefix.
`clear()`	Removes all the words from a lexicon.

COSC 220: Data Structures in C++

Iterating over a collection

C++11 → can do a range based for loop

```
for (type variable : collection){
        body of loop
}
```

example: iterate through all words in English lexicon and select only those with two letters:

```
for (string word : english){
        if (word.length() == 2) {
                cout << word << endl;
        }
}
```

colon operator ":" is a mechanism called an *iterator*