# Project 1:
# The STL and You

A quick intro to the STL to give you tools to get started with stacks and queues, without writing your own!

Use a deque instead!

Speed up your output!

# The vector<> Template

- You must `#include <vector>`
- Basically a variable-sized array
- Implemented as a container template
- You must specify the type at compile time
- The size can be specified at run time
- For example:

```
vector<int> values;
```

# Adding to a Vector

- Starts empty with no room for values
- Use the `push_back()` member function to add a value to the end
- Parameter to `push_back()` must be same `<type>` as when vector was declared
- For example:

```
values.push_back(15);
```

# Accessing Vector Elements

- The `vector<>` template overloads `operator[]()`
- When the vector is not empty, you can access it with [0], [1], etc.
- Loop through all values:

```
for (size_t i = 0; i < values.size(); ++i)
  cout << values[i] << endl;
```

# Important Note

- These are not the only data structures you will need for Project 1!
- This is intended to help you with the “Routing Schemes” portion, where you have to remove/add when searching from the current location
- See Project 1 specification for more details; search for “Routing Schemes”

# STL Containers

- The STL containers are implemented as template classes
- There are many types available, but some of them are critical for Project 1
  - Stack
  - Queue
  - Deque (can take the place of both stack and queue)
- Common/similar member functions

# The STL Stack

- You must `#include <stack>`
- Create an object of template class, for example:

```
stack<int> values;
```

- You can push an element onto the top of the stack, look at the top element of the stack, and pop the top element from the stack

# The STL Queue

- You must `#include <queue>`
- Create an object of template class, for example:

```
queue<int> values;
```

- You can push an element onto the back of the queue, look at the front element of the queue, and pop the front element from the queue

# Common Member Functions

- The stack and queue containers use many of the same member functions
  - `void push(`*`elem`*`)` – add element to container
  - `void pop()` – remove the next element from the container
  - `bool empty()` – returns true/false
- The only difference is which end the `push()` operation affects

# Different Member Functions

- The stack uses:

  `<T> top()` – look at the “next” element (the top of the stack)

- The queue uses:

  `<T> front()` – look at the “next” element (the front of the queue)

# Using Stack/Queue in Project 1

- If you want to use stack and queue for the searching in Project 1, create one of each type
- Must use them inside a single function (which will probably be long)
  - Cannot make a template function, due to `.top()` versus `.front()`

# Don’t Make Two Functions!

- Don’t write a 100-line function for stack, and another 100-line function for queue
  - This is duplicated code
  - If you fix one you have to fix the other
- Instead, make one 100-line function, with a single `if` inside the loop
  - This is NOT significantly slower than two functions
  - Modern CPUs are good at predicting `if` result

# The Deque Container

- The deque is pronounced “deck”
  - Prevents confusion with dequeue (dee-cue)
- It is a double-ended queue
- Basically instead of being restricted to pushing or popping at a single end, you can perform either operation at either end

```
#include <deque>
```

# Deque Member Functions

- The deque provides the following:

```
void push_front(elem)
<T> front()
void pop_front()

void push_back(elem)
<T> back()
void pop_back()

bool empty()
```

# Using a Deque in Project 1

- If you want to use a single data structure for searching in Project 1, use a deque
- Always use `.push_back()`
- When you're supposed to use a stack, `.back()` and `.pop_back()` (everything happens at same end => stack)
- For a queue, use `.front()` and `.pop_front()`

# More Information

- More information on these STL data types can be found in the Josuttis textbook
  - Stacks and queues can be found in sections 12.1 and 12.2, respectively
  - Deques are in section 7.4
  - Vectors in section 7.3

# Dictionary Data Structure

- Create a `*` (pointer) and use dynamic memory
  - Can be problematic with a complex dictionary, because the number of words in the FILE might be smaller than the number of words in the dictionary
- Create a `vector<>`
  - Use the `.resize()` member function before reading the words from the file

# Creating/Initializing a Vector

- Here is an example of creating and initializing a 1D vector, with 10 entries, all initialized to -1:

```
int size = 10;
vector<int> values(size, -1);
```

- Since 10 values already exist, read data directly into them using `values[i]`, do NOT `.push_back()` more values

# About Vector Size

- We’ll be covering this in more detail later
- A vector has two different “sizes”
  - The number of values currently contained
  - The maximum size before the vector must be resized
- For example, you could have 5 elements in use out of a maximum size of 8
  - The `.size()` member function returns 5

# About Vector Growth

- If you declare a vector as shown in a previous slide, it is “presized” to have the number of elements that you requested
  – For example, current size 10, max size 10
- If you `.push_back()` another element, it must increase in size (generally doubling)
  – Becomes current size 11, max size 20
- This size increase takes time (copying, etc.) and wastes memory (45% unused)

# Presizing, Resizing, and Reserving

- You can also resize a vector after it is created, or reserve a certain number of elements
- The `.resize()` member function changes both the current and maximum sizes
- The `.reserve()` member function only changes the maximum size

# About the Complex Dictionary

- If you presize or resize the dictionary, you might have to change halfway through, from storing elements directly into `dictionary[i]`, to adding them with `dictionary.push_back(stuff);`
- Instead, use `.reserve(size)`
  - Use the size declared on line 2
- Probably only have to grow once

# Speeding up Input/Output

- C++ `cin` and `cout` can be slow, but there are several ways to speed it up:
  - Turn off synchronization of C/C++ I/O
  - Use `'\n'`
  - Use string streams
    - This has no real benefit when using the latest version of g++

# Synchronized I/O

- What if you used both `printf()` (from C) and `cout` (C++) in the same program?
  - Would the output order always be the same?
  - What if you were reading input?
- To insure consistency, C++ I/O is synchronized with C-style I/O
- If you're only using one method, turning off synchronization saves time

# Turning off Synchronized I/O

- Add the following line of code in your program, as the first line of `main()`
- It should appear before ANY I/O is done!

```
ios_base::sync_with_stdio(false);
```

# Warning!

- If you turn off synchronized I/O, and then use `valgrind`, it will report potential memory leaks
  – Appears as 122KB that is “still reachable”
- The simplest way to get accurate feedback from `valgrind` is to:
  1. Comment out the call to `sync_with_stdio()`
  2. Recompile
  3. Run `valgrind`
  4. Un-comment the sync/false line
  5. Proceed to edit/compile/submit/etc.

# `'\n'` versus `endl`

- Whenever the `endl` object is sent to a stream, after displaying a newline it also causes that stream to “flush”
  - Same as calling `stream.flush()`
- Causes output to be written to the hard drive RIGHT NOW
  - Doing this after every line takes up time
- Using `'\n'` does not flush

# Finding the Path

- Once you reach the goal, you have to display the path that found it
  - Either words or modification mode
- The stack/queue/deque do not have this information
- You have to save it separately!

# Backtracking the Path

- You can’t start at the beginning and work your way to the end
  - Remember, the Start word might have had 4 other words that were 1 letter different
- Think about it this way: when you’re at the goal, how did you get here?
  - Since each word is visited ONCE, there is exactly ONE word “before” this one

# Backtracking Information

- When you’re at the goal word, how did you get here?  What word were you on when the goal word was added to the stack/queue/deque?
  - Every dictionary word must remember the “previous” word

# Simple/Complex Dictionary

- The dictionary in memory should always be simple (just words)
- For example, the complex dictionary FILE on page 3 of the project spec should result in the following 12 words in MEMORY:
- chip, chop, junk, star, tsar, ship, shop, shot, stop, pots, let, leet

# Strings / Arrays

- You can treat a string as an array of characters, by using the [] operator:

```
string word;
getline(cin, word);

for (size_t i = 0; i < word.size(); ++i)
  cout << word[i] << endl;
```

# Processing Strings

- For a complex dictionary, you may need some string functions to help you determine where a special character is located, break a string into pieces, etc.
- Look at the following:

  http://www.cplusplus.com/reference/string/string/

  http://www.cplusplus.com/reference/string/string/find/

  http://www.cplusplus.com/reference/string/string/find_first_of/

  http://www.cplusplus.com/reference/string/string/substr/