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()

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
- When you're supposed to use a stack, `push_front()`
- For a queue, `push_back()`
- Always 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

3D Data Structure

- Create a *** (triple pointer)
- Create a nested vector<>
  - Use the `.resize()` member function on each dimension before reading the file
- Create a nested array<>
- For any choice, exploit locality of reference
  - Use subscripts in this order:
    - [level][row][col]

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> oneDimArray(size, -1);
```

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

Creating/Initializing a Vector

- Here is an example of creating and initializing a 2D vector, with 10 rows and 20 columns, all initialized to -1:

```
int rows = 10;
int cols = 20;
vector<vector<int>> twoDimArray(rows,vector<int>(cols,-1));
```

Speeding up Output

- C++ `cout` can be slow, but there are several ways to speed it up:
  - Use `'\n'`
  - Use string streams
  - Turn off synchronization of C/C++ I/O

`'\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

String Streams

- Basically a string object that you can use << or >> with (most of this example is output only)

```
#include <sstream>
```

- Create an object:

```
    ostringstream os;
```

- Send things to it:

```
    os << "Text!" << '\n';
```

- Just before program exits:

```
    cout << os.str();
```

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 must 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 memory leak
- The only 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.

Finding the Path

- Once you reach the goal, you have to display the path that found it
  - Either on top of the map, or in list mode
- The map, 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 square might have had 4 or more possible places to go
- Think about it this way: when you’re at the goal, how did you get here?
  - Since each squareis visited ONCE, there is exactly ONE square “before” this one

Backtracking Example

- When you're at the goal, how did you get here? What square were you on when the goal square was added to the stack/queue/deque?
  - Every square must remember the "previous" square
- If you're using queue-based routing, it was the square to the west

	S		R