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!

DO NOT!!!

- Copy and paste code from a PDF and expect it to compile
  - PDF files sometimes use Unicode characters to make things like - look nice, but it isn’t a “minus” sign
  - PDF files have hidden characters (called elisions) to make spacing look good
- Anything in here is short enough to retype

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 Algorithm” portion, where you have to remove/add when searching from the current location
- See Project 1 specification for more details; search for “The Algorithm”

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

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, you *could* create one of each type
- Must use them inside a single function (which will probably be long), or write two almost identical functions
  - Cannot make a template function, due to `.top()` versus `.front()`
- This is not the best way to proceed

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
- This is the *search container* in the spec
- **Always** use `.push_back()`
- When you're supposed to use a stack, use `.back()` and `.pop_back()`
- For a queue, use `.front()` and `.pop_front()`

More Information

- More information on these STL data types can be found online
  - cppreference.com
  - cplusplus.com
- Look up their syntax, constructors, member functions, etc.

2D or 3D Data Structures

- Create a ** or *** (double or triple pointer)
  - Bad choice, too much work to do on your own
- Create a nested vector<>
  - Create the vector with the right size initially
  - Use the `.resize()` member function on each dimension before reading the file
- 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:

```
uint32_t 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 then Resizing

- If instead you want to declare the vector then read the size, then change the size of the vector:

```
vector<int> oneDimArray;
uint32_t size;
cin >> size;
oneDimArray.resize(size, -1);
```

Creating/Initializing a 2D Vector

- Here is an example of creating and initializing a 2D vector, all initialized to -1:

```
uint32_t rows;
uint32_t cols;
cin >> rows >> cols;
vector<vector<int>> twoDimArray(rows, vector<int>(cols, -1));
```

- Each “row” is itself a `vector<int>`
- You can extend this upward to 3 dimensions!

Resizing Multiple Dimensions

- If you have a 2D vector:

```
uint32_t rows, cols;
vector<vector<int>> twoDimArray;
cin >> rows >> cols;
twoDimArray.resize(rows, vector<int>(cols, -1));
```

- When you have a 3D vector, extend this syntax upward to three dimensions
- There's two items inside every set of ( )
  - The number of elements
  - What each element is

About Data Structures

- Be willing to make different types of data for different purposes
- Don't try to make one type of data that can be used for every purpose (the map, backtracing, and deque)
  - If you do this you'll have memory trouble
- Make different data types for different purposes as needed

Speeding up Input/Output

- C++ `cin` and `cout` can be slow, but there are several ways to speed it up:
  - DO turn off synchronization of C/C++ I/O
  - DO use `'\n'`
  - DON'T use string streams
    - This has **NO** real time benefit when using the latest version of g++, and it wastes memory
  - DON'T produce a `string` object containing all your output (no speed gain, wastes memory)

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 as part of the map, or in list mode
- The map, stack/queue/deque do not have this information
- You have to save it separately!

Backtracing the Path

- You can’t start at the beginning and work your way to the end
  – Remember, the Start might have had 4 possible places to go
- Think about it this way: when you’re at the goal, how did you get here?
  – Since each location is visited at most ONCE, there is exactly ONE location “before” this one

Backtracing Example

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

	@		$