

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 “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

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](http://cppreference.com)
  - [cplusplus.com](http://cplusplus.com)
- Look up their syntax, constructors, member functions, etc.

# 3D Data Structure

- You could create a \*\*\* (triple pointer), don't
- Create a nested vector<>
  - Use the `.resize()` member function on each dimension after reading the sizes from the file
- For any choice, exploit locality of reference
  - Use subscripts in this order:  
`[room][row][col]`
- You could create a 1D structure and treat it as if it was 3D; don't do this either

# 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)

# Resizing, 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

# 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 resizing a 2D vector, with some number of rows and columns, all initialized to -1:

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

- Extend this upwards to 3 dimensions:  
rooms, size, size; make sure rooms is first!

# Creating/Initializing Structures

- If your 3D vector contains classes or structures, and you want to give them default values, the easiest way is inside the class or struct:
  - Use a constructor
  - Have default values specified for each member variable
- If you do this, you can leave out the default value entirely (the -1 in the int example)

# Converting char To/From int

- You will probably find that you need to perform conversions
- You can add and subtract integers and characters, and convert!
- For example, character to number:  
`level = static_cast<uint32_t>(square - '0');`
- Or number to character:  
`square = static_cast<char>('0' + level);`

# 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 (don't forget before you submit again)**
  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

# When to Use endl

- You want to use endl before calling `exit()`
- This causes the output stream to be *flushed*: you want to ensure that output completes before the program is terminated
- Typically used with `cerr` for the output of an error that is so bad the program can't continue running

# 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!

# 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!

# Backtracing the Path

- You can't start at the beginning and work your way to the end
  - Remember, the Start tile 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 tile is visited ONCE, there is exactly ONE tile “before” this one

# Backtracing Example

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

|  |   |  |   |
|--|---|--|---|
|  |   |  |   |
|  | S |  | C |
|  |   |  |   |
|  |   |  |   |