## Lecture Topics

- templates
  - the Standard Template Library
  - specializing containers
- profiling and tuning:
  - measuring operation timing
  - …

## Administrivia

- HW#2 due in one week
- midterm in two weeks

# The Standard Template Library

- good starting reference: `www.cplusplus.com/reference/stl/`

- vector — dynamic array
- list — doubly-linked list
- deque — double-ended queue
- bitset — array of bits/flags

- queue — deque adapted for FIFO-only access
- stack — deque adapted for LIFO-only access
- priority_queue — vector adapted for use as a heap

- (these are all implemented with red-black trees)
- set — set of unique items
- multiset — set of non-unique items
- map — set of key-value pairs (unique keys)
- multimap — set of key-value pairs (non-unique keys)

- look at the reference page above for other resources
  - I/O streams
  - strings
  - algorithms
  - iterators

# Specializing Containers

- example from 190
  - use a horse-racing example to introduce linked lists (of horses)
  - then we talk about sorted lists
  - What's the "natural" order for horses?
- sorting order
  - a container property, not an item property!
  - although some item types might have a natural order
- How do you express cleanly with C++ templates?
- answer: templates in C++ can have default values
  - create a "standard" comparison type (another template)
  - functions in comparison type use, for example, operator<
  - thus, by default, a type's operator< ends up being used
  - operator< defines the "natural" order for the type
  - however, comparison type can be provided explicitly
  - in which case it defines comparisons for that container type (usually an instance, but it's really a type)
- example…

```
class OddIsSmall {
public:
    boolean operator () (int a, int b) {
        if ((a % 2) == (b % 2)) {return (a < b);}
        return (1 == (a % 2));
    }
}
std::priority_queue<int,std::vector<int>,OddIsSmall> pq;
```

- push 0…9; get back 8, 6, 4, 2, 0, 9, 7, 5, 3, 1

- STL containers also allow allocation to be overridden

# Measuring Operation Timing

- scenario
  - you've written a library or module (e.g., NetLink)
  - how do you do "unit tuning"?
  - i.e., optimization without major driver apps

- be careful
  - avoid premature optimization
  - you need to know how library will be used before you implement it
  - with NetLink, networking and system call costs dominate library costs (as I mentioned, but we'll confirm)

- brainstorming opportunity [let them do it…]
  - given a short operation sequence (maybe even one)
  - what makes it hard to measure accurately?

- some issues
  - timer overhead
  - timer granularity
  - processor/resource virtualization
  - interrupts
  - hot/cold caches, TLBs (what's the right level for a module?)
  - pipelining, cache alignment, etc.

- a common strategy…
  - avoid timer issues by measuring many, not one
  - run on otherwise empty system to avoid conflicts
  - interrupts should be small relative to cost; use minimum timing result
  - assume (and force) hot caches
    - weak argument: important and frequent op won't have cold caches
    - reality: sometimes hard to know without actual application
  - ignore pipelining

- a timing and optimization study comparing C and C++
  - versions
    - netlink in C
    - NetLink in C++
  - operations
    - create & destroy server (completely local, but uses OS)
    - connect & close (TCP ping-pong)
    - connect & receive 1kB (real network use, akin to small web page)
  - gcc optimization levels
    - none: no optimizations, not even inlining of functions in class def'n
    - -O (means -O1in gcc): basic optimizations
    - -O9: optimize everything