

## Lecture Topics

- templates (wrap-up)
  - miscellany
  - the Standard Template Library
  - specializing containers
- parallelism : why bother with it?

## Administrivia

- midterm
  - I looked over them (not thoroughly yet)
    - most people did fairly well
    - will try to grade this weekend
  - note that templates were not on midterm; they'll be on final
- Lab #2 handout & discussion
  - hopefully obvious, but referenced object should be deleted when count reaches 0
  - see p. 241 (Sec. 11.5.1) for the smart pointers discussion
  - due Thurs 19 March (handout has an error)

## Miscellany

- another code reduction method:  
use additional templates to eliminate pointless variation
  - example: Str. p. 347
  - use void\* variant of base container class
  - to implement all pointer variants
  - safe because casts occur within per-type template instance
  - inlining + optimization eliminates any overhead
- What if template isn't "right" for some types?
  - special case individual types, overriding the template
  - define before first instantiation of desired override
  - Str. Sec. 15.10.3 for details
- can also specialize based on groups of types
  - use overloading resolution mechanism to identify at compile time
  - compiler can optimize overhead away
  - Str. pp. 353-354 gives an example
- member functions can also be templates
  - but cannot be virtual (vtable must be defined before link!)
  - useful for allowing actions that compiler would allow anyway (compiler will give errors if template instantiated with bad type)
  - for example
    - if ALPHA<class T> is a template class
    - and BETA is derived from GAMMA
    - ALPHA<BETA> is NOT derived from ALPHA<GAMMA>
    - viewing the BETAs as GAMMAs might be useful, though
    - create a template constructor using implicit casts
      - works when compiler allows (e.g., derived to base pointer)
      - fails when it does not (e.g., unrelated class pointers)

## The Standard Template Library

- good starting reference: [www.cplusplus.com/reference/stl/](http://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

## **Parallelism: Why Bother with It?**

- few people ever do (fraction of software writers is small); why not?
  - gain is essentially constant, not scaled with problem
  - want a fixed gain?
    - wait a few years
    - or build your application in hardware instead
  - few people are motivated by “no gain, just pain”
- so why has anyone ever done it?
  - sometimes the pain is vanishingly small
    - little or no actual interaction between parts of the application
    - by design in database systems (my hypothesis from grad. school: for enough \$, any application can be made embarrassingly parallel)
  - more resources
    - memory
      - some commercial CFD codes in late 90s ran as separate processes primarily due to limits on virtual addr. space
      - large problems often don't fit in the memory available with a single-chip machine
    - memory bandwidth, I/O bandwidth
      - many applications limited by engineering decisions based on generic workloads
      - e.g., for database sorting benchmarks, the NOW cluster (100 machines) beat a commercial system designed for the server market because the NOW cluster had more disks
  - sometimes you need the fixed gain
    - supercomputing/computational science
    - grand challenge applications
    - h/w also works, but it's more expensive and restrictive (e.g., limits utility of algorithmic advances)

- recall first lecture...What has changed about exploiting parallelism?
- want a fixed gain?
  - wait a few years...sorry, that trend has ended
  - or build your application in hardware...only \$50-100M for a typical modern chip process!
- sometimes the pain is vanishingly small
  - maybe only use free/easy parallelism?
  - 2-processor system cost was the sweet spot briefly in mid-90s
  - before Intel/AMD entered server market
  - small SMP's are again popular as servers, but multi-core is now also ubiquitous on desks/laps
- more resources? no, not with one machine
  - in fact, fewer resources per processor
  - we'll need to be careful not to overtax resources
    - resource use/response time as a function of load
    - generally has a sharp rise as utilization nears 100%
  - methods for simplifying parallelism tend to exacerbate by temporally correlating resource use
- new viewpoint on parallelism
  - parallelism may be attractive to more people
    - zero- or tiny-pain variants
    - small fixed gains
  - particularly if “fixed” gain
    - can be made to scale with time
    - that is, process generations/density/number of cores
- How far can we push the envelope with good engineering?
  - research topic, but one that needed to be solved 10 years ago
  - potential for immediate impact