Sorting directory entries

- How many different ways to sort directory entries?
  - date last modified, size, type
  - name
    - handle numbers smartly?
    - handle capitalization smartly?
- C `scandir` function allows user to provide sorting function
- Function object: a class that can be used as a function
  - overload `operator()` so that object can be used syntactically as a function [like a function pointer]
  - STL supplies these as templated functions, e.g., `less<int>`

```cpp
template <class T>
struct less : public binary_function<T, T, bool> {
  bool operator() (const T& x, const T& y) {
    return x < y;
  }
};
```
templates and generic classes/functions

- **STL is the standard template library**
  - part of C++ as a library of generic functions, classes, and algorithms
  - uses no inheritance, but templates can require existence of operators/functions
    - in hash_map class, operator \( == \) is needed
    - in map class, operator \( < \) is needed

- **A templated class is not code, but a code generator**
  - separation of interface and implementation is tricky
  - implementation is a generator, not code
  - must instantiate generator at compile time
Using templated functions (classes)

- **Generic sort**

  ```
  template <class Type>
  void sort(vector<Type> & a, int numElts)
  // pre: numElts = number of elements in a
  // post: a[0] <= a[1] <= ... <= a[numElts-1]
  ```

- **What must be true of Type?**
  - Comparable
  - ???

- **Instantiate templated function/class**
  - bind template parameters, unify types

```
vector<string> vs; ... sort(vs,vs.size());
vector<Vector<int> > vvi; ... sort(vvi,vvi.size());
```
How to traverse directory entries

● Entries are a collection, how to let user traverse
  ➤ return entries for user to traverse
    • as a vector
    • as a c-style array
  ➤ let user provide collection to be filled in

● Issues?
  ➤ Should collection be copied?
  ➤ who allocates collection?
  ➤ how to change collection type?
  ➤ how to support filters?
Iterators: Patterns and STL

- Access a container without knowing how it’s implemented
  - libtapestry:
    - `first`, `isDone`, `next`, `current`
    - Iterators are part of an inheritance hierarchy: `Iterator`
  - STL
    - `begin`, `end`, `*`, `++` for pointer like syntax
    - No inheritance, all typedefs in each STL class
- What are iterator properties, who makes the iterator
  - `const`, `non-const`, `random-access`, ...
  - `makeIterator` uses new internally, who deletes?
    - Pointer Proxy: “smart pointers”, allocated on stack but act like heap-allocated pointers
Standard/STL classes vs libtapestry classes

- **libtapestry strings aren’t as efficient, e.g., no sharing on assignment**
  - reference counted, shared storage
  - copy on write
  - always copy (libtapestry)

- **shared vs static libraries**
  - static code stored in executable when compiled
  - shared/dynamic linked when program run
  - advantages?

```cpp
g++ string a = "hello"; string b = a; libtapestry.a
```

```
g++ string a = "hello"; string b = a; standard (libg++.so)
```

```cpp
b[0] = ‘j’; //what happens?
```
**STL classes (e.g., vector)**

- In general, the STL classes aren’t safe, e.g., bad index not checked
  - on our systems, a “safe” version of classes can be used by using `#define __STL_DEBUG` before any `#include`
  - specific differences in STL `vector` vs libtapestry `vector`

<table>
<thead>
<tr>
<th>STL</th>
<th>libtapestry</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size()</code></td>
<td><code>length()</code></td>
</tr>
<tr>
<td><code>push_back()</code></td>
<td><code>append()</code></td>
</tr>
</tbody>
</table>

- `append/push_back` works differently: `size()` in libtapestry is number of times append called
- libtapestry `vector, map, stack, queue` are as efficient as STL classes (sometimes more efficient) and are safe
Access to map classes

- Both STL and libtapestry use a templated *Pair* class
  - first, second in pair correspond to key, value in map
  - returned by *iterator* and *iterator->current()*
- libtapestry
  - includes, insert, getValue: see map.h
  - leads to redundant lookups in map implementation
- STL
  - operator[], insert
    - *map[key] = value*; inserts pair if not there
    - potential problem with *if(map[key] == …)*
- hash_map is NOT standard STL, map uses red-black tree