Iterators and Pointers

- Iterators extend convenience and efficiency of pointer operations to non-array collections
- Can perform standard pointer operations such as dereferencing, pointer arithmetic, and incrementing/decrementing
- For iteration, random access is less efficient than incrementing a pointer!
Sets

- `set<T>` contains elements of type `T`
  - must be unique; duplicates ignored
  - `s.insert(const T& value)` member function adds value to set `s`
  - can use `=` to assign sets
  - `s.clear()` member function empties `s`
- `multiset<T>` allows duplicates
  - `insert()`, `clear()` available
  - `count(const T& value)` member function returns number of occurrences of `value`

Maps

- `map<T1, T2>` contains key-value pairs
  - unique key of type `T1`, value of type `T2`
  - overloads `[]` for retrieving value with key
- `multimap<T1, T2>`: duplicate keys
  - `lower_bound(T1)`, `upper_bound(T1)` member functions return iterators defining sequence of values matching key
  - iterators are pointers to `pair` objects
  - `insert(make_pair(key, value))` adds key-value pair to multimap
  - `erase(key)` removes key
Pairs

- A pair represents an ordered pair
- To declare: `std::pair<T_1, T_2> p(x_1, x_2)` where `x_i` is of type `T_i`
- A pair has members first and second, so `p.first` is `x_1` and `p.second` is `x_2`
- An iterator `i` from a map points to a pair, where `i->first` is the key and `i->second` is the value

Working with Collections

- `s.erase(i)` erases element pointed to by the iterator `i` from the collection `s`
- `s.erase(x)` erases all elements equal to `x` from the collection `s`, where `x` is a key for a map or multimap, not a value
- `i = s.find(x)` looks up an element `x` (or key, for map or multimap) in a collection and returns iterator `i` pointing to matching elements, or pairs for maps (equal to `end()` if `x` is not found)
Sparse Matrices

• In many applications that require solution of systems of linear equations of the form $Ax = b$, the matrix $A$ is large but sparse (most entries are zero)
• Inefficient to represent such matrices using a two-dimensional array
• Can only use vectors in certain cases
• An alternative data structure is required
• Must be able to quickly access elements, iterate over rows, columns

Sparse Matrices with Maps

• Can represent a sparse matrix using a `map<pair<int, int>, double>`
• Internally, map uses a hash table for efficient element access
• Can use a `map<int, set<int>>` to keep track of locations of nonzero entries in row/column
• Matrix operations should only require work proportional to number of nonzero entries
Next Time

• Catch-up day!
• Project 6 preview