Why C++?
- a better C
  - Type safe, e.g., I/O streams
  - Better support for ADTs, encapsulation
  - Better libraries
- object-oriented programming
  - Add inheritance to encapsulation
  - OO isn’t a silver bullet, but it helps in dealing with the complexity of software development
- non OO programming
  - sometimes non-class approach has merits
- generic programming
  - STL, the standard (template) library

Classes, an Overview
- Classes encapsulate both state and behavior
  - Think about behavior when designing classes
  - Don’t think about state, implementation at design time
- State is private, behavior is public, though it’s often useful to have private helper functions
  - A class is often called an object factory, it’s used to create potentially lots of instances of the class
- Classes collaborate and communicate
  - Parameters --- passes a, gets a
  - Containment --- has a, has a pointer to
  - Inheritance --- is a

Classes, C++ specifics
- Create and destroy classes via constructor and destructor
  - Default constructor needed for vector (map, etc.)
  - Destructor frees resources “if you create it, clean it up”
- Clients expect classes to be const correct, use const properly
  - Accessor methods are const (where does the const go?)
  - Mutator methods are non-const
  - Sometimes logical constness and physical constness are at odds, the keyword mutable can help
- Classes can declare friends, but there are often alternatives
- Classes can overload operators, nice syntactically for clients
  - Sometimes ugly for implementors

Overloading operators
(see getlines.cpp and filesorter2.cpp)
- See online Howto for “standard” overloading: +, -, *, <<,
- operator ()
  - Instances can be used syntactically as functions
  - Varying number of parameters/arguments
- operator []
  - Often need const and non-const versions
  - Cumbersome in map, must insert an element (no lookup)
- operator ++() and operator++(int)
  - Difference between pre and post increment?
STL, Generic Programming

- **Common container classes/data structures**
  - vector: growable array
  - map: dictionary, (search tree, hashtable)
  - set: union, intersection, membership

- **Common algorithms**
  - copy, sort, count, find, transform, ...

- **STL, Standard (Template) Library**
  - not just implementations, but way of thinking
  - little/no inheritance, lots of templates
  - algorithms and functions generalized too

- **In general not safe, e.g., a[x] can cause segmentation fault**
  - Uses “esoteric” C++, now supported in varying degrees

Making an STL-like Adapter

- **The Adapter pattern**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces

- **What makes a class act like an STL iterator, for example?**
  - STL classes model concepts, not enforced by inheritance, but by convention and template instantiation
  - See *Generic Programming and the STL*, Matthew Austern

- **See dir_iterator in diradapter.h**
  - What’s needed for use in algorithm copy?
  - What conventions are used in creating adapter class?