#### What is abstraction?

- Workable answer a *blurring* of details
- Idea: agree to ignore certain details (for now)
  - e.g., with procedural abstraction idea is to convert original problem to a series of simpler problems
- Works for data types too

   Think (and write code) in terms of abstract data types like Lists, Stacks, Trees, ...
  - What should matter what you can do with a List
  - What should not matter what goes on inside the List
  - Assume the ADT works just use it!

### A Priority Queue ADT

- ADT is defined by its interface what it does
- Never mind how it works think about that later

### Interface is enough to use ADT

- Easy way to sort let a priority queue do it void easySort(PQItem a[], int n) { int i; PriorityQueue pq;
  - initialize(&pq);
    for (i=0; i<n; i++) /\* put all items in priority queue\*/
     insert(a[i], &pq);
    for (i=n-1; i>=0; i--) /\* items come out sorted \*/
  - a[i] = remove(&pq);
- *)* /\* There are more efficient ways to sort, but that's not the point. \*/
  The point is that we can use it without knowing how it works.
- Abstraction is good!

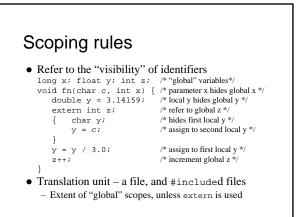
### Of course, it does have to work

- Many ways to implement text covers 2: – Maintain a *sorted list* of items:
  - insert some work: insure item is inserted in order
  - remove easy: remove the first item
  - Keep items in an *unsorted array*:
    insert easy: append item as last array element
    remove harder: search for highest priority item,
  - and move last array element to emptied slot
- Binary tree method works best later topic

### Decomposition and C modules

- So user just needs the interface:
  - e.g., #include "PriorityQueue.h"
  - Which may vary between implementations but better not to
- The implementation is in a separate file:

   Usually PriorityQueue.c, and separately compiled
   This file also has #include PriorityQueue.h in it
- This organization has at least two major benefits:
   Implementation details hidden from user
  - User less likely to mess it up, & doesn't have to think about it - Critical interface declarations stored in a single place



# Compiling, linking, & make files Compiling only – e.g., gcc -c pgm.c – Creates object file called pgm.o (or pgm.obj in DOS) Linking only – e.g., gcc pgm.o -o pgm – Makes executable file called pgm (or pgm.exe in DOS) Can automate process with a Makefile:

pgm: pgm.o # dependency

gcc pgm.o -o pgm # action (*tab is required*) pgm.o: pgm.c

gcc -c pgm.c

 Then just type "make" – Unix tool executes the actions as necessary to satisfy the dependencies

## Dealing with multiple modules

- Imagine a program for factorial, consisting (for illustrative purposes only) of 3 modules: <u>factorial.h</u> - contains the function prototype <u>factorial.c</u> - implements the function <u>testfac.c</u> - uses the function - Both .c files #include "factorial.h"
- <u>Makefile</u> separately compiles testfac and factorial, then links them
  - If just change factorial.c make recompiles that file only and relinks to existing testfac.o

### Abstract lists

- Text's ch. 4 lists more abstract than ch. 2 - Info stored as ItemType
  - Then typedef int ItemType, or any other type

     #include ItemInterface.h redefined as necessary
  - List node operations are very general: void setLink(NodePointer, NodePointer) NodePointer getLink(NodePointer) void setItem(NodePointer, ListItem) /\* where typedef ItemType ListItem \*/ ListItem getItem(NodePointer)
- Idea is to hide the implementation details

## Even more abstract lists

- One way: store info as void \*
  - Then can point to anything
  - Only way to apply polymorphic abstraction in C
- Another way: hide internal data structures completely give no access to nodes
  - Not just function implementations can be hidden
  - Necessary to provide an iterator mechanism, because user has no direct access to links
    - Simplifies list usage, and prevents tampering

### **Basic List ADT**

- <u>basiclist.h</u> (very) abstract data type for lists – Allows handling of any type of data:
  - typedef void \*InfoPointer;
  - Completely hides implementation details:
    - typedef struct ListTag \*ListPointer;
    - Structure declared here; defined in basiclist.c
    - Might be implemented as array or other way user doesn't
  - have to know; user can't mess it up – Requires initialization to set things up:
    - ListPointer createList(void);
    - In this case, have to allocate space for list structure, and initialize all pointers to NULL

- Basic list ADT (cont.)Accessor functions access info, not nodes
  - InfoPointer firstInfo(ListPointer); InfoPointer lastInfo(ListPointer); InfoPointer currentInfo(ListPointer);
  - User cannot incorrectly handle nodes
     e.g., can never set node->link = node;
- Insert functions do not copy info, just pointers void insertFirst(InfoPointer, ListPointer);
   Can also insert last, or before or after current
- Delete functions return copies of deleted pointers InfoPointer deleteFirst(ListPointer);
  - Can also delete last or current