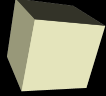
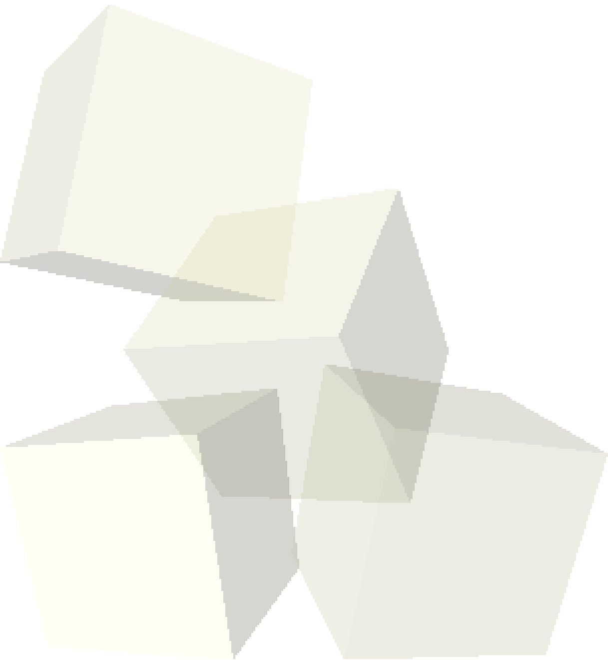
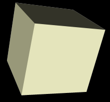


Unions and Recursion

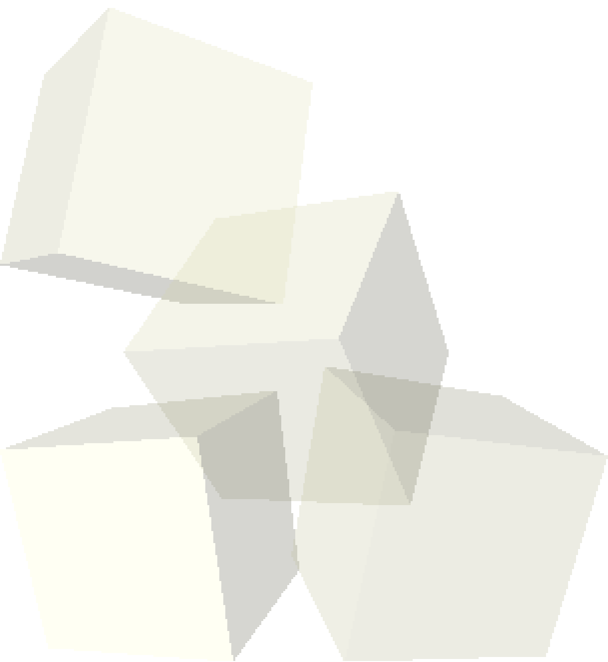
11/12/2007

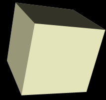




Opening Discussion

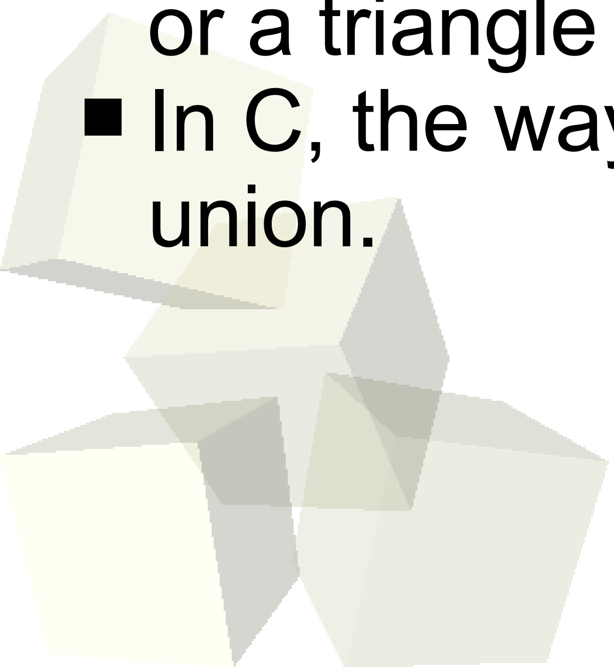
- Let's look at solutions to the interclass problem.

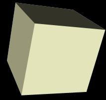




Motivating Unions

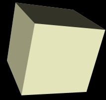
- There are many instances in programming when we would like to have a type that can represent one or more other types.
- Consider our geometry example from last class. We wanted a scene made of triangles and spheres. In many ways it would be superior to have a Geometry type that can be either a sphere or a triangle or something else we might add later.
- In C, the way we get this type of behavior is with a union.





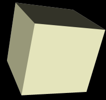
Unions

- You can think of a structure as a type that contains type1 and type2 and type3, etc. A union is a type that contains type1 or type2 or type3.
- So a struct can be used to hold a bunch of things while a union can be used to represent one of multiple things.
- Like structs, a union can either be tagged or used with a typedef.
  - union TagName {
    - ContentAlternatives
  - };
  - typedef union {
    - ContentAlternatives
  - };



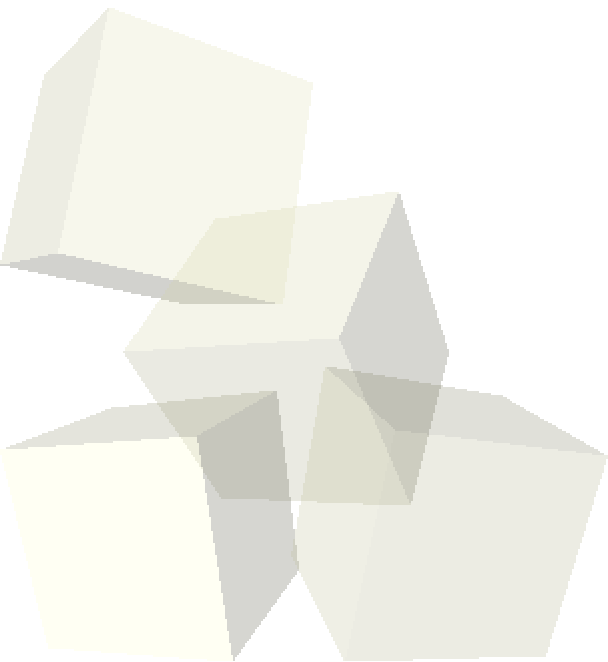
Using Unions

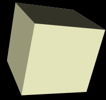
- Unions are almost never used alone. You nearly always put a union in a struct and often use an enum as well.
- The reason is that the union has no internal way of telling you which of the alternatives it is actually being used for. So you typically put a union inside of a struct where the struct includes a field that tells you which option of the union is being used. The other field can in just an int or you can use an enum.
- Because of this, code with unions almost always includes switch statements.



Problems with Unions

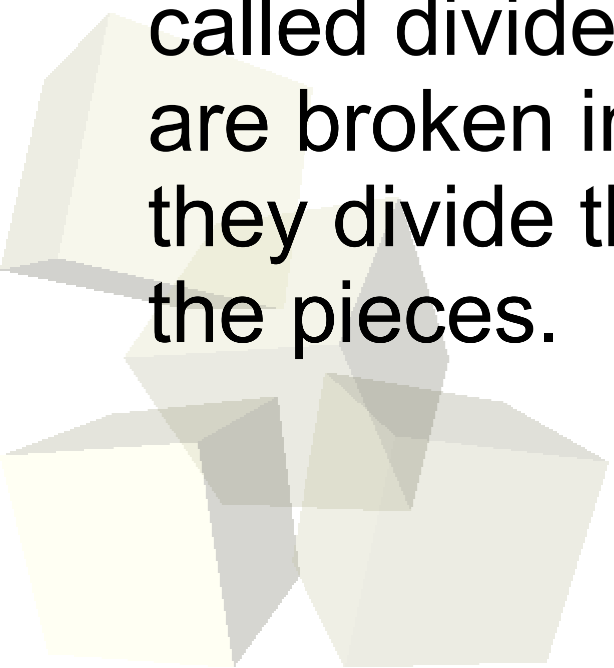
- Using unions requires care because they don't tell you which option you are using.
- This capability of a single type to represent multiple types is generally called polymorphism. The form we have in C with unions is quite limited. We'll see a lot more of this and what we can do with it in Java.

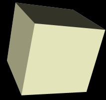




Recursive Sorts

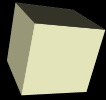
- Earlier in the semester we saw three different sorts that are simple to write, but are rather slow if we have large arrays.
- We can get better performance in our sorts using more complex algorithms.
- Merge sort and quicksort are more efficient sorts that use recursion. They both use an approach called divide and conquer where large problems are broken into smaller pieces. They differ in how they divide the problem and how the recombine the pieces.





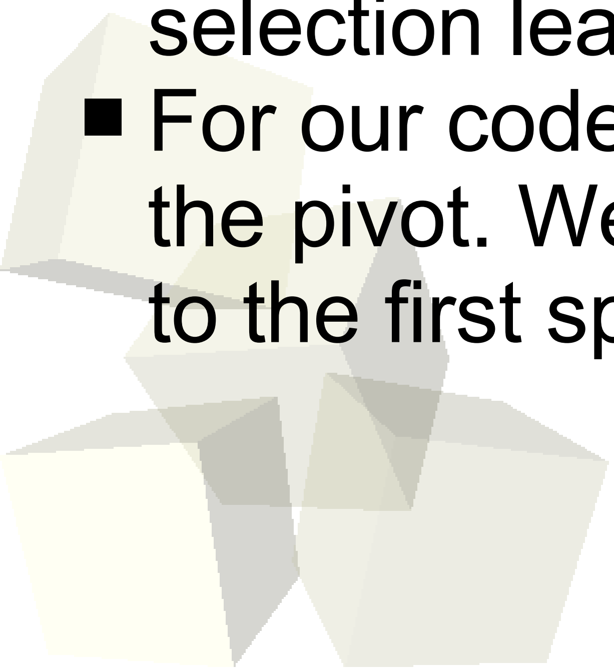
Merge sort

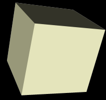
- Merge sort is simple to describe, but difficult to implement.
- Going down the recursion we divide the array in half continually until we get to single elements, which are like tiny sorted arrays.
- As the recursion returns back up the stack we merge the sorted pieces. We can merge two sorted arrays in linear time. The real work is on the way back up.
- This always gives O(n log n) performance.
- Unfortunately, merge sort can't be done in place. A second array is needed and that hurts efficiency and makes it a lot harder to code well.



Quicksort

- Quicksort can sort in place. At each step we pick a special element called the pivot. We move elements so that the pivot is in its proper place. Then we recursively call quicksort on the parts below and above the pivot.
- All the work happens going down the recursion.
- Expected behavior is $O(n \log n)$, but poor pivot selection leads to worst case of $O(n^2)$.
- For our code we will just take the first element as the pivot. We could easily pick another and swap it to the first spot.





Minute Essay

- What is one of the challenges associated with using unions?
- Interclass Problem – Write code that uses a union. Try to be creative in what you are using the union for.