MORE CONTROL STRUCTURES AND SOME SORTING

Today

- This unit looks at a simple forms of sorting.
- We illustrate this by sorting some numbers that have been read into an array.
- First, though, we will use the reading of numbers as a prompt to look at two control structures we ignored before.

Reading information from a file (again)

- Consider the following problem statement:

  Read six numbers from a file into an array.

- With what we have covered so far, we would probably write something like the code in `files.cpp` (which you can download from the class web page).

- First we set up a file as the input stream:

```
ifstream myfile;
myfile.open("numbers-short.txt");
```

- Then we use a `for` loop to read in the six numbers:

```
for(counter = 0; counter < 6; counter++)
  {
    myfile >> numbers[counter];
  }
```

- This is fine so long as we know that there are at least six numbers in the file.
- Sometimes we are not so lucky.
- We want to read in up to six numbers, but there may not be as many as that.

- In such a case (which is not uncommon) we might want to do the following:

```
do {
    myfile >> numbers[counter];
    counter++;
  }
while(counter < 6 && myfile);
```

- The second `myfile` is `true` so long as the last read from the file returned something.
- Thus when we get to the end of the file, `myfile` is false, and the loop ends.

- This is an example of a `do/while` loop.
- This is subtly different from a `while` loop.
- The general form of the loop is:

```
do {

    <some instructions>

}
while(<a condition that is true or false>);
```

- Note the semicolon at the end of the line with the `while` on it.

- The big difference between `do/while` and `while` is how many times the *body* of the loop gets repeated.
  - The body of the loop is the bit between the { and the }
- In a `while` loop, if the condition is `false`, the body is *never* executed.
- In a `do/while` loop, even if if the condition is `false`, the body is executed *at least once*.
- This difference helps us to decide which control structure is best to use.

- Another way to achieve the same thing is to do:

```
while(counter < 6 && !myfile.eof()) {
  myfile >> numbers[counter];
  counter++;
}
```

- The function `myfile.eof()` returns true if we are at the end of the file.
- Thus, once again, when we get to the end of the file, the loop terminates.

- You can test the way that these examples work by running `files.cpp`, `files2.cpp` and `files3.cpp` from the course webpage.
- To test these, use the files of nubers `numbers.txt`, which holds more than 6 numbers, and `numbers-short.txt`, which holds less.
- Another useful function for handling files is `myfile.isopen()`, which will return false if a previous call to `myfile.open()` failed.
- Such a failure would occur, if you were opening a file for reading, if the file didn't exist (which is a problem that we have seen several times in the lab exercises).

Sorting

- Now we can read numbers into an array. Let's look at sorting them.
- We'll look at four different methods for sorting.
- These are:
  - blort sort
  - selection sort
  - insertion sort
  - bubble sort
- There are many other kinds of sorting...

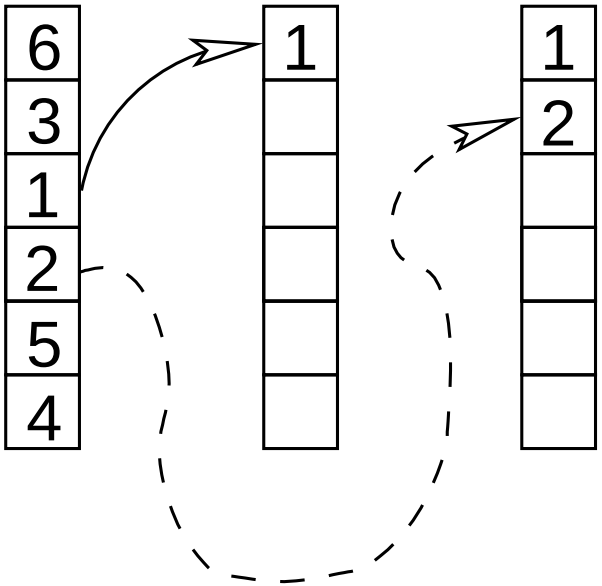
Blort sort

- "Fun but stupid"
- Works like this:
  1. Check to see if the array is ordered.
  2. If it is, then sort is over.
  3. Otherwise, shuffle the elements in the array and start over.
- This *will* work (eventually).
- However, it is not, generally, a good way to go.

Selection sort

- Selection sort uses an *auxilliary* array
  - Another array that collects the sorted numbers.
  - After sorting these values are copied back into the original array.
- The basic algorithm to order the array from *lowest* to *highest* is
  1. Select the smallest element still left in the array.
  2. Add it to the end of the auxilliary array.
- The next slide shows this in operation.

- Here are the first couple of steps in sorting.

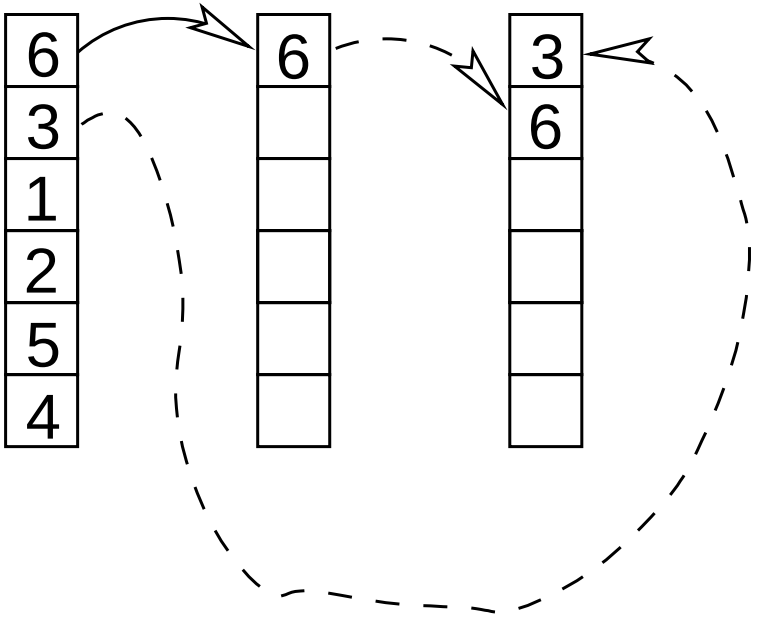


- In the first, 1 is found to be the smallest element and copied to the end (in this case its also the first spot in the array since the auxilliary array starts out empty) of the auxilliary array.
- In the second step, 2 is the smallest remaining element in the array, and is copied to the current end position in the auxilliary.

Insertion sort

- In insertion sort we do the following:
  1. We take elements from the array that is neing sorted and we *insert* them into the correct place in the auxilliary array.
     (This typically requires moving other elements to make room).
  2. One all the elements have been inserted into the auxilliary array they are copied back into the original array.
- The next slide shows this in operation.

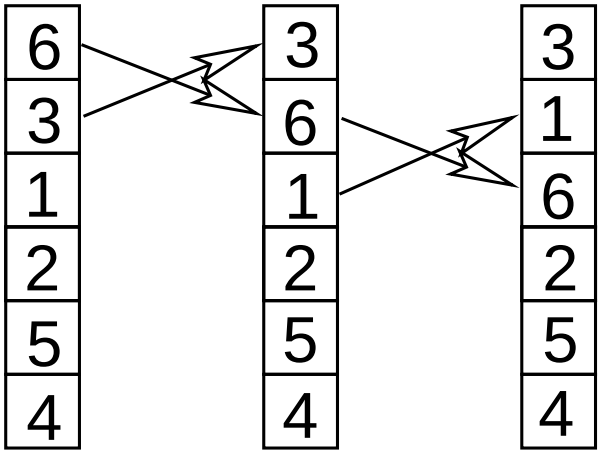
- First 6 is put into the auxilliary array in order — this is easy since it is the first element.



- The we put 3 into the array. To do this we need to push 6 down.
- Next we will insert 1 into the array — to do that we'll need to move both 3 and 6 down.

Bubble sort

- Bubble sort repeatedly compares adjacent members of the array, swapping elements if they aren't in order.



- In the first step, 3 and 6 are swapped.
- Next 1 and 6 are swapped.
- Lower values "bubble up" through the array.

Summary

- This lecture discussed two things.
- First it considered different ways of reading information in from a file.
  - We looked at a couple of ways of detecting the end of a file.
- Then we considered how to sort things.
  - In particular, we looked at blort sort, selection sort, insertion sort and bubble sort.
- Next time we'll look in detail at how to program these sorts.