Advanced Unix Tools	**Lecture 12**
CS214 Spring 2002	**Friday, March 15, 2002**

In this lecture, we introduce Perl arrays. These are useful because, for example, complete files can be read in as arrays of lines. An array, for our purposes, is just a sequence of values, stored in such a way that you can retrieve any element of the sequence efficiently.

An array variable is defined like a scalar variable. The main difference is that you have to indicate to Perl that the variable holds an array instead of a scalar value. You do that by prefixing the variable name by a @. The values of the array are specified by listing them in parentheses, separated by commas. For example:

```
@nums = ( 1,2,3,4,5,6,7,8 );
```

To access an element of the array, say the third element, the syntax is a bit different. Consider printing the third value in the array *@nums*:

```
print $nums[2];
```

Two things to notice: first, array elements start at position 0, so the third element is stored at array position 2; also since you are extracting a scalar value from an array, you indicate that the value is a scalar by using the *$* notation instead of the @ notation for the array name. As a rule, *@foo* refers to the array *foo* as a whole, while *$foo[..]* refers to the scalar content of the array.

Arrays have quite a few uses. First, recall the *for* loop:

```
for $i (1,2,3,4,5) {
  print $i;
}
```

The list of elements over which to iterate can contain an array, at which point the iteration will occur over all the elements of the array. Since we are refering to the array as a whole, we use the @ notation for the array. For example,

```
@committee =  ( "Joe", "Dexter", "Delia" );
for $name (@committee) {
  print "member : $name\n";
}
```

Another place where arrays are handy is to split up strings. The function *split* takes a regular expression describing where to split the string, and a string to be split, and returns the array of substrings after the split. For example,

```
@result = split (/ /, "this is a sentence");
for $w (@result) {
  print "$w\n";
}
```

splits up the string at every space into the words *this*, *is*, *a*, and *sentence*, and outputs them one by line. The converse function *join* takes an array of strings and a joining string, and concatenates all the substrings together, inserting the joining string between them. Hence:

```
@strings = ("Hello", "world", "and", "welcome!");
$result = join (' ', @strings);
print "$result\n";
```

outputs the string *Hello world and welcome!*.

A final use of arrays that I'll mention is as a way to read in a file as a whole. Recall that if you do a *<HANDLE>* where *HANDLE* is some input handle, you read in a single line of the corresponding handle (typically, a file). Actually, this depends on the context of use. If you are assigning the value of *<HANDLE>* to a scalar variable, you are indeed reading a single line from the file. If you assign the result to an array variable, you end up reading the whole file, putting each line of the file into a different position in the array. This leads to an easy way to iterate over all the lines in a file:

```
@file = <FILEHANDLE>;
for $line (@file) {
  play with $line
}
```

Perl also has a notion of *associative arrays* builtin, that the rest of the world knows as *dictionaries*. They are indicated by the special prefix *%*, hence *%foo* represents a dictionary variable. I will refer you to the documentation for more information on those.