Perl Lists

- Note: we often refers to 'lists' as 'arrays', and vice-versa
- In Perl, all arrays are dynamic (they may grow and shrink as your program runs), so it’s useful to think of arrays as lists
- Here's a simple list of the first ten prime numbers:

```
1, 2, 3, 5, 7, 11, 13, 17, 19, 23
```

- Note: lists can also be enclosed in brackets, if needs be - here's another simple list:

```
(1, 2, 3, 4, 5)
```

- which can also be written as a list range:

```
(1..5)
```

List Ranges

- We can also generate a list using non-numeric values:

```
'Aa'..'Ak'
```

- is shorthand for:

```
'Aa', 'Ab', 'Ac', 'Ad', 'Ae', 'Af', 'Ag', 'Ah', 'Ai', 'Aj', 'Ak'
```

- If we have the following list:

```
'first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh'
```

- We can use the following shorthand to represent exactly the same thing:

```
qw(first second third fourth fifth sixth seventh)
```

- The `qw` stands for 'quoted words'

Naming Lists

- We use $ with scalar variables, and we use an @ with lists:

```
@small_primes = (1, 2, 3, 5, 7, 11, 13, 17, 19, 23);
```

- Note that the individual elements of the list are all scalars
- We can copy lists:

```
@copy_small_primes = @small_primes;
```

- And we can also add to lists as part of the copy:

```
@more_small_primes = (0, @small_primes, 29);
```

Printing Lists

- We can use *interpolation* to print out the values of lists:

```
print "Small primes is set to: @small_primes\n";
```

- will produce:

```
Small primes is set to: 1 2 3 5 7 11 13 17 19 23
```

- If we don't print from within an interpolated string, things can get strange:

```
print @small_primes;
```

- will produce:

```
123571113171923
```

- Note: by default, the interpolation will use a space character to separate list items on output - this space is stored inside the in-built `$"` variable, which is called `$LIST_SEPARATOR` if we ‘`use English;`‘ - changing `$"` to some other value will effect the output produced by interpolation

Individual Elements of Lists

- We can index into a list, and indexed elements always start at zero - remembering that list elements are all *scalars*, we can access the first element of a list like this:

```
$first_element = $small_primes[0];
```

- which will set the scalar `$first_element` to the value `1`, which is the first element in the `@small_primes` list (which is also a scalar) - note that we did not use `@small_primes[0]`

- Consider the following:

```
$small_primes[10] = 29;
print "Small primes is now set to: @small_primes\n";
```

- will produce as output:

```
Small primes is now set to: 1 2 3 5 7 11 13 17 19 23 29
```

How Many Elements?

- Assume we have a list called `@larry_wall` - we can access the highest value of its subscript as follows:

```
@larry_wall = qw(this is a list in honour of Larry Wall);
print "\@larry_wall is currently set to: @larry_wall.\n";
print "The highest subscript value of \@larry_wall is: $#larry_wall.\n";
$how_many = $#larry_wall+1;
print "Which means we have $how_many elements in the list.\n"
```

- produces as output:

```
@larry_wall is currently set to: this is a list in honour of Larry Wall.
The highest subscript value of @larry_wall is: 8.
Which means we have 9 elements in the list.
```

More Shrinking and Growing

- Let's assume a list will grow, and that it will have `1000` elements - we can tell Perl this (assuming the list is called `@fibonacci`) as follows:

```
$#fibonacci = 999;
```

- which will cause the creation of a list called `@fibonacci` with `1000` empty elements

- Later in our program, we might decide we don't want as big a list after all, so we can do the following to shrink the list:

```
$#fibonacci = 99;
```

- Note: if anything was stored beyond the `100`th element, it is now gone, and Perl's memory management *garbage collection* mechanism will reclaim the `RAM` it used

Using `for` with Lists

- Consider the following script:

```
$" =  "\t";
@squares = (1..10);
print "@squares\n";
for ($i = 0; $i < 10; ++$i)
{
  $squares[$i] *= $squares[$i];
}

print "@squares\n";
```

- With lists, we can use the `foreach` looping construct to make this more convenient:

```
@squares(1..10);
foreach $s (@squares)
{
  $s *= $s;
}
```

- Note: `$s` is a synonym for the current element of `@squares`, so updates to `$s` effect the current element of `@squares`

Lists as Structures

- You can *simulate* a simple `struct` (or `record`) using lists, as well as extract individual values (fields) from the struct, change them, then put them back into the list, as follows:

```
@coord_3D = (100, 200, -200);

print "@coord_3D\n";

($x, $y, $z) = @coord_3D;

print '$x = ' . $x . ', $y = ' . $y . ', $z = ' . $z, "\n";

$x /= 2;
$y /= 2;
$z /= 2;

print '$x = ' . $x . ', $y = ' . $y . ', $z = ' . $z, "\n";

@coord_3D = ($x, $y, $z);
print "@coord_3D\n";
```

Lists and Built-in Functions

- A large number of Perl functions return lists as their result

```
($sec, $min, $hour, $month_day, $month, $year, $week_day,
$year_day, $summer_time) = gmtime;
++$month;
$year += 1900; # Look: no Y2K bugs here!
print "The date is: $month_day/$month/$year\n";
```

- will produce as output:

```
The date is: 7/11/1999
```

- The `gmtime` function is interesting, as it can provide a *list* as its result (above), or a *scalar* - for example, the following code:

```
print $dt = gmtime, "\n";
```

- will produce as output:

```
Sun Nov  7 07:24:30 1999
```

More on List Functions

- Again, taking `gmtime` as our example, let's say we are only interested in the *current time*, so we can do this:

```
($sec, $min, $hour) = gmtime;
```

- We can also retrieve the rest of the `gmtime` list as one thing (i.e., a list) as follows:

```
($sec, $min, $hour, @the_rest) = gmtime;
```

- When it comes to assignments, the RHS of the assignment is completely evaluated before the assignment takes place, and, as an example, here's the Perl version of the classic "let's swap the contents of two variables" problem:

```
($a, $b) = ($b, $a);
```

- Note: Perl does not require the use of a temporary variable

Calling Context

- Consider the following:

```
@my_list = qw(this is my list);

print "@my_list\n";
print $size = @my_list, "\n";

@my_list = (@my_list, qw(and a little bit more));

print "@my_list\n";
print scalar @my_list, "\n";
```

- will produce as output:

```
this is my list
4
this is my list and a little bit more
9
```

Calling Context and Files

- We already know that the next line reads from the filehandle `MYFILE` and assigns the line read to the variable `$line`:

```
$line = <MYFILE>;
```

- If we assign what's read to a list, we get the *entire file*:

```
@file = <MYFILE>;
```

- This can be useful, but dangerous ...

- With `@file` set as above, we can use it like any other list and pass it to functions that know about lists - `chomp` is one such function:

```
chomp(@file);
```

- which will remove the new-line character from every line read into the list `@file`

Calling Context Example

- Here is yet another version of our first Perl script:

```
#!/usr/bin/perl

@file = (<>);
print @file;
```

- Warning: although this works, it is *harder to extend than the original* - how would you go about only printing lines from the `/etc/fstab` file that contained the pattern `/cdrom/` anywhere in the line?

- Note that the following would definitely **NOT** work:

```
@file = (<>);
print @file if /cdrom/;
```

Calling Context and Subroutines

- Perl allows us to write subroutines that return either a *list* or a *scalar* based on the context within which they are called:

```
sub the_first_primes {
  @primes = (1, 2, 3, 5, 7, 11, 13, 17, 19, 23);

  return wantarray ? @primes : ($#primes+1);
}

@list = the_first_primes;

$scalar = the_first_primes;

print "The subroutine returned the list value:   @list\n";
print "The subroutine returned the scalar value: $scalar\n";
```

- will produce as output:

```
The subroutine returned the list value:   1 2 3 5 7 11 13 17 19 23
The subroutine returned the scalar value: 10
```

List Functions

- The `reverse` function will take a list and return it in reversed order

- Here's a quick one-liner to *reverse* the line contents of any file:

```
print reverse(<>);
```

- The `sort` function is a bit more useful:

```
@sorted = sort qw(one two three four five);
print "@sorted\n";
```

- To sort in *descending* order, pass the results from `sort` to `reverse`:

```
@sorted = reverse sort qw(one two three four five);
print "@sorted\n";
```

Processing Every List Element

- The `grep` and `map` functions allow us to apply a computation to *every* element in a list
- The `grep` function returns a list of the elements for which the evaluation produces a true value
- So, applying the following script to `/etc/passwd`:

```
@bash_entries = grep { m[/bin/bash$] } <>;
print "Matching Entries are:\n@bash_entries\n";
```

- will produce as output a list consisting of the `/etc/passwd` entries for users of the Bash Shell, which gave this output on my home system:

```
Matching entries are:
root:x:0:0:root:/root:/bin/bash
postgres:x:17:17:Postgres User:/home/postgres:/bin/bash
barryp:x:500:100:Paul Barry:/home/barryp:/bin/bash
```

Getting Just The Results

- The `map` function allows us to return a list consisting of the *results* of the computation:

```
@bash_users = map { m[(^\w+).*/bin/bash$] ; $1 } <>;

print "Bashers on this system are:\n";

foreach $b (@bash_users)
{
  print "  User = ", $b, "\n" if defined( $b );
}
```

- will produce (again on my home Linux system):

```
Bashers on this system are:
  User = root
  User = postgres
  User = barryp
```

Splitting Lists

- We can use the `split` function to divide a list into parts based on a delimiter
- Note: `split` returns the parts as a list
- Again, we will use the `/etc/passwd` file as an example - here's an *extract* from the file on my home computer:

```
majordom:x:16:16:Majordomo:/:/bin/false
postgres:x:17:17:Postgres User:/home/postgres:/bin/bash
mysql:x:18:18:MySQL User:/var/lib/mysql:/bin/false
nobody:x:65534:65534:Nobody:/:/bin/false
barryp:x:500:100:Paul Barry:/home/barryp:/bin/bash
```

- Note that the file has a record on each line with 7 fields, each separated by a single `:` character

Splitting Passwords

- Here's a script to print out the users that use some sort of shell:

```
#!/usr/bin/perl

# We don't want to '/usr/bin/perl -w' here, as Perl will complain
# about the throw-away variables $a, $b, $c, $d, and $e.

while (<>)
{

  $passwd_line = $_;   # Copy the current line contents.
  chomp( $passwd_line );
  ($user, $a, $b, $c, $d, $e, $shell) = split /:/, $passwd_line;
  print "User: '", $user, "' uses: ", $shell, "\n" unless $shell eq "";
}
```

- Results of Splitting Passwords

```
User: 'root' uses: /bin/bash
User: 'sync' uses: /bin/sync
User: 'shutdown' uses: /sbin/shutdown
User: 'halt' uses: /sbin/halt
User: 'majordom' uses: /bin/false
User: 'postgres' uses: /bin/bash
User: 'mysql' uses: /bin/false
User: 'nobody' uses: /bin/false

User: 'barryp' uses: /bin/bash
```

More Splits

- The *pattern* and *string* arguments to `split` are optional - if missing, `split` will split `$_` on whitespace, so:

```
$_ = "this is a test of split";
@line_list = split;
foreach $w (@line_list)
{
  print $w, "\n";
}
```

- will produce as output each word contained within `$_` on its own line
- Check `man perlfunc` for more details on `split`
- The `join` function allows us to create a delimited record:

```
print join ':', $user, $a, $b, $c, $d, $e, $shell;
```

Shifting

- The `shift` function *returns* and *removes* the first element of the list:

```
@nums = qw(one two three four five);
$first_num = shift @nums;  # $first_num is set to 'one'.
print "@nums\n";
```

- We can add to the start of a list using the `unshift` function:

```
unshift @nums, 'one';
print "@nums\n";
```

- The following does nothing (except *waste* valuable processor cycles):

```
unshift @nums, (shift @nums);
```

Pop and Push

- The pop function operates on the end of the list, returning the last element as its result:

```
@nums = qw(one two three four five);
$last_num = pop @nums;  # $last_num is set to 'five'.
print "@nums\n";
```

- The push function (surprise, surprise) appends an element to the end of a list:

```
push @nums, 'five';
print "@nums\n";
```

Simple List Data Structures

- Using `push` and `pop` on any list makes it behave like a `LIFO` stack
- The same is also true of using `shift` and `unshift` together
- Using `push` and `shift` together, or using `unshift` and `pop` together, lets you treat the list as a `FIFO` queue
- If you use all four together, then lists can behave like double-ended queues (or `deques`)

Splice

- Splicing and dicing of lists is possible with the `splice` function, which has the general form:

```
splice @some_list, $offset, $length, @some_other_list;
```

- where `$offset` is a location within `@some_list` to start at (remembering that we start counting at *zero*), `$length` is the number of elements to remove, and `@some_other_list` is a list to put into `@some_list` in place of the elements removed

Splice Example

- For example, given:

```
@nums = (1, 2, 3, 4, 5, 6);
@num_words = qw(three four four_and_a_half five);
splice @nums, 2, 3, @num_words;
print "@nums\n";
```

- will produce as output:

```
1 2 three four four_and_a_half five 6
```

More on Splice

- If we omit a replacement list, `splice` simply deletes the entries from the original list:

```
@nums = (1, 2, 3, 4, 5, 6);
splice @nums, 2, 3;
print "@nums\n";
```

- will produce as output:

```
1 2 6
```

- If we omit a length value, `splice` deletes from the offset to the end of the list

- As you may have guessed, `splice` returns the items removed in the form of a list

Lists and Subroutines

- Now that we know about lists, we can revisit subroutines to learn how Perl *passes arguments* into subroutines
- Yes, you guessed it, they get *passed as lists*
- Fans of 'traditional' programming languages prepare to be *shocked*: strange though it may seem, you do not have to provide named arguments to your subroutines in Perl (Gasp! Horror! *How can this be?!*)
- Within your subroutine, the list of arguments passed in (if any) are available to you in the Perl list variable `@_` (which is called `@ARG` if we '`use English;`')
- This is really important: the values passed in are *passed by reference* - changes to the values within the subroutine change the corresponding values in the calling code

Call By Value

- To allow you to *preserve* the contents of the variables within the calling code, Perl requires you to create variables that will be local to your subroutine, and that will contain *copies* of the passed by reference variables

- We do this with the `my` function (in combination with `shift`):

```
sub my_copy {
  my $copy_one = shift @_;   # Note: the @_ is optional;
  $copy_one = "one has been changed";
  $_[0] = "two has been changed";  # Why zero?
}

$one = "this is one";
$two = "this is two";

my_copy ($one, $two);

print '$one = ' . $one, "\n";
print '$two = ' . $two, "\n";
```

Command Line Arguments

- Our script can receive arguments from the command line (i.e., the bash shell)
- If provided, these are available within our Perl scripts as elements of a list called @ARGV
- This next script takes two arguments: a string to search for, and a file to search in:

```
#!/usr/bin/perl -w

$string = $ARGV[0];
$file = $ARGV[1];

open FILE, $file or die "Cannot open $file\n";

while (<FILE>)
{
  print if /$string/;
}
```

More on Command Line Arguments

- The previous script was useful, but it gets into trouble if we feed it *more than one file* to process
- We already know (from our very first Perl script) that the `while (<>)` code will work on *multiple files*, and rather than rewrite code to simulate this, we'd like our script to *take advantage* of this default behaviour
- But, what do we do with the first argument, which is the string to search for?
- The solution is surprisingly simple:

```
#!/usr/bin/perl -w

$string = shift;    # We shift the first command-line argument.
while (<>)          # We use the default behaviour as is.
{
  print if /$string/;
}
```