;login:
THE MAGAZINE OF USENIX & SAGE
August 2002 volume 27 • number 4

inside:

PROGRAMMING
Turoff: Practical Perl

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

practical perl

Programming with Iterators

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

Most programs deal with examining a sequence of values at some point. In this column, we investigate iterators, a way to simplify processing of a computed sequence of values. We conclude by revisiting a common problem: parsing a configuration file, this time by using iterators.

Introduction to Iterators

I recently worked on a project where I needed to use the iterator design pattern. Design patterns are common structures and behaviors that occur frequently in many programs and are used across programming languages and application domains. The iterator pattern describes a behavior where an object offers sequential access to a data structure composed of many individual elements.

Iterators are more common in strongly typed programming languages. Java, for example, contains an Iterator interface as part of the core language definition. Classes that use this interface provide next() and hasNext() methods to enable programs to examine a series of values sequentially, one at a time. One particularly interesting use of iterators involves traversing a binary tree. One way to examine each node in the tree would be to code up a breadth-first traversal algorithm every time you need it. An easier way to examine the tree is to make successive calls to a next() method to retrieve each item from the tree in the proper order.

You may not have heard about iterators before just because they're not particularly necessary in Perl. Because Perl has a generic list data type already, there isn't a pressing need to create generic interfaces or design patterns to access list-type objects in a sequential manner.

Here are a few common examples of how iterators are commonly used in Perl. Built-in operators like foreach, map, and grep can work with any kind of data in a list, because lists are generic containers:

```
foreach (@ARGV) {
    ## process items in a list
}
## transform a list of values
## into a list of squares
my @squares = map {$_ ** 2} 1..10;
## reduce a list of files
## to a list of writable files
my @writable = grep {-w $_} </htdocs/*>;
```

Iterating over the lines in a file using a while loop is another common technique:

```
while (<FILE>) {
    ## process each line of FILE
}
```

Recall that foreach will examine lists one element at a time, but while loops execute until the test condition evaluates to false. That's why the while(<FILE>) idiom is shorthand for this construct:

```
while (defined($_ = <FILE>)) {
    ## process each line of FILE
}
```

The point here is that a line is read from FILE each time the block is executed. Looking at the while loop this way shows that we're not examining a sequence of values in a list, but examining a sequence of values generated dynamically. It just so happens in this case that these computed values are lines from a file. We could just as easily iterate over a series of rows coming from a database query using the DBI module:

```
use DBI;
my $dbh = DBI->connect("dbi:SQLite:dbname=my_data", "", "");
    my $sth = $dbh->prepare("SELECT * FROM books");
    $sth->execute();
    while (my @row = $sth->fetchrow_array()) {
        ## ... process each row
    }
```

Iterators in Perl

In order to iterate over a sequence of computed values (like the ones returned by <FILE> or $sth->fetchrow_array()), it is necessary to return a series of values followed by some false value when the sequence is exhausted. One easy way to signal the end of a sequence is to return undef or an empty list. This is sufficient when examining a series of strings (lines from a file), a series of lists (rows from a database), or a series of numbers.

Generating a sequence of computed values involves maintaining some state variables so we can tell when the sequence is

exhausted. This is generally done with an object, but it can also be done with a closure. Closures are anonymous subroutines that maintain some private-state variables. They're like objects, except that they've been turned inside out. Where objects are pieces of data (like a hash) with some subroutines attached, closures are subroutines with some data attached.

Here is a function that creates closures, each of which will count from 1 to 10:

```
sub make_counter {
    my $i = 1;
    return sub {
    return if $i > 10;
    return $i++;
    }
}
```

In this example, a new variable $i and a new anonymous sub are created each time we call make_counter. Each closure we create maintains its own private value for $i. We can then call the closure 10 times to get the values 1 through 10. After that, we'll always return a false value. This satisfies the requirements for an iterator, so we read values from it one at a time, almost as if it were a file:

```
my $iterator = make_counter();

while(defined($_ = $iterator->())) {
    print; ## 12345678910
}
```

Combining Iterators

The iterators that are created by make_counter() are very simple and may not seem very worthwhile at first. But it is easy to combine iterators to produce more interesting results. Here is an iterator that filters values from our simple counter iterator and emits only the odd values:

```
sub odd_numbers {
    my $next = shift;
    return sub {
    my $i = $next->();
    while (defined($i) and ($i % 2) == 0) {
        $i = $next->();
    }
    return $i;
    }
}

my $iter = make_counter();
my $odd = odd_numbers($iter);

while (defined($_ = $odd->())) {
    print;  ## 13579
}
```

First, we ask for a value from the iterator $odd. Within $odd's closure, we ask for a value from its $next iterator until we find an odd value or the end of $next's sequence of values. The result, as expected, is a sequence of odd values from 1 to 10.

This example shows another property of closures. Not only do closures turn objects inside out, but they turn logic inside out as well. Instead of skipping even values within the while loop, we weed them out beforehand, simplifying the while loop down to a single statement.

Note that we created the $odd iterator by modifying another iterator. This process can be extended, transforming a sequence of odd numbers into a sequence of odd numbers squared:

```
sub make_squares {
    my $next = shift;
    return sub {
    my $i = $next->();
    return unless $i;
    return $i ** 2;
    }
}

my $iter = make_counter();
$iter = odd_numbers($iter);
$iter = make_squares($iter);

while (defined($_ = $iter->())) {
    print;
}
```

We can go even further, adding another filter to transform this sequence of odd numbers squared into a running total of odd numbers squared, a running average of odd numbers squared, or something entirely different. No matter how we build the iterators up, the process of examining the final result remains the same: a simple while loop.

Parsing Configuration Files

Now that you understand the basic ideas behind iterators, it's time for a more practical example: parsing a configuration file. Let's start with a few simple requirements:

- Configuration files consist of a series of name-value pairs and are stored in a hash.
- Comments start with the # character and continue until end-of-line; all comments should be ignored.
- Lines consisting of nothing more than space characters should be ignored.

The first few requirements seem simple enough to implement with a standard while loop. It might look something like this:

```
while (<CONFIG>) {
    s/#.*$//;    ## delete comments until end-of-line
    ## skip blank lines
    while (m/^\s*$/) {
        $_ = <CONFIG>;
    }

    my ($name, $value) = m/^(.*?)=(.*?)$/;
    $config{$name} = $value;
}
```

If you look closely, there are some bugs caused by the inner loop. Only the first line's comments are deleted; after we've found a blank line (or a line with nothing but a comment), then the next non-blank line's comment will be kept. There are a lot of ways to fix this bug. If we had used iterators, these bugs would be easier to avoid.

First we need to read lines from a file using an iterator. Once that is done, we can then transform that stream of values by stacking one iterator on top of another until we're left with a stream of name-value pairs:

```
sub make_file_iterator {
    my $filename = shift;
    open(my $fh, $filename);
    return sub { return scalar <$fh>; }
}

sub strip_comments {
    my $next = shift;
    return sub {
        my $line = $next->();
        $line =~ s/#.*$//;
        return $line;
    }
}

sub skip_blanks {
    my $next = shift;
    return sub {
        my $line = $next->();
        while(defined ($line) && $line =~ m/^\s*$/) {
            $line = $next->();
        }
        return $line;
    }
}

my $config = make_file_interator("my.config");
$config = strip_comments($config);
$config = skip_blanks($config);

while (defined($_ = $config->())) {
    ## process name=value pairs
    my ($name, $value) = m/^(.*?)=(.*?)$/;
    $options{$name} = $value;
}
```

In this example, we start with three generic subroutines that create iterators. If another portion of our program needed to skip blank lines or strip comments, we could reuse these subroutines to generate iterators for that task. This allows us to maintain and debug code in one spot, rather than maintaining and debugging a few repeated lines in many places.

Another benefit is that our main program consists of three lines of initialization to set up the $config iterator, and a simple while loop that only sees valid values and operates on them. As requirements change over time, this main loop would need very little modification. Most of the changes could be handled by adding filters to the $config iterator to perform more pre-processing.

Conclusion

Iterators are a very powerful construct for processing a series of values. The kinds of iterators described here use closures for a simple and effective way to create and transform a series of values generated one at a time. Iterators simplify programming by separating out the pre-processing from the main processing for a series.