A Simple Tangler Program

Andrew L. Johnson

March 22, 2000

Abstract

The following is a slightly modified excerpt from chapter 9 of my book *Elements of Programming with Perl*, published by Manning Publications. It is reprinted here with permission of the publisher. This program is a simplified `tangler` program for use with single file literate programs written in the `noweb` syntax.

A Simple Tangler

Now that we know what the chunk definition and reference syntax is, we can build a limited tangler program to allow us to write our perl programs using `noweb`'s syntax, intermixing code chunks and documentation chunks (we are in a documentation chunk right now) throughout the source file.

We want our tangler to operate similarly to `notangle`, allowing a `-R` option to specify the root chunk and a `-L` option to include line directives. We will add two differences. The `notangle` program simply prints the tangled code to STDOUT so you have to redirect it to a file yourself. We will assume that the root chunk-name is also the filename you want to use for the tangled code. So, running our tangler program with a root option of `-Rblah` will create a file named ‘blah’ and write the tangled code to it. The second difference is that if no `-R` option is given, our program will automatically find all root chunks and print them to their respective files based on their chunk-names (a root chunk will be any chunk that is not used inside another chunk definition).

We shall call our tangler `pqtangle` for Perl Quick Tangler, and write it in a file named *pqtangle.nw*. Our initial program outline (our root chunk) looks like:

1 ⟨*pqtangle* 1⟩≡
```
#!/usr/bin/perl -w
use strict;
⟨declare variables 2a⟩
⟨get options 2b⟩
⟨open and parse file for chunks 3b⟩
⟨make array of root chunks 4b⟩
⟨print root chunks 5a⟩
⟨subroutine definitions 5b⟩
```

We will need variables to reflect the two options our program can accept (the -R and -L options for a root chunk and to turn on line directives respectively. We will also use variables to hold patterns containing double left angle brackets and double right angle brackets:

2a ⟨*declare variables* 2a⟩≡ (1) 3a▷

```
  my $Root;     # root option
  my $Line_dir; # format option
  my $la = '<' . '<';
  my $ra = '>' . '>';
```

Defines:
`$la`, *used in chunks 3–5, 7, and 8.*
`$Line_dir`, *used in chunks 2b and 7–9.*
`$ra`, *used in chunks 3–5 and 7–9.*
`$Root`, *used in chunks 2b and 4b.*

Notice that we've used the `\%def` syntax on the chunk-ending line to mark these two variables as defined in this chunk. These def lines are ignored by the tangler, but we have to recognize them if we want to recognize general `noweb` syntax—and we may as well use them so we can weave our documentation (and you can see what the cross-reference information available looks like if you visit the above mentioned site). We will consider other variables shortly.

There is a simple technique for extracting options (and their potential arguments) from the command line arguments. Recall that all the command line arguments are in the `@ARGV` array when the program runs. We can simply use a `while` loop to pull the first item out of this array if it starts with a dash, then we test that the option is valid (in this case, begins with either -R or -L) and set our option variables appropriately: For the root option, we simply capture everything following the -R into the `\$1` variable and assign it to the `\$Root` variable. For the line directive we assign a generic string to the variable that contains placeholders for the line number, filename, and newline that we will substitute in as needed during the program:

2b ⟨*get options* 2b⟩≡ (1)

```
  while ($ARGV[0] =~ m/^-/) {
     $_ = shift @ARGV;
     last                          if m/^--$/;
     $Root   = $1                  if m/^-R(.*)/;
     $Line_dir = '#line %L "%F"%N' if m/^-L/;
  }
```

Uses `$Line_dir` *2a and* `$Root` *2a.*

The convention for command line options is that they must all come before any other arguments and that anything after a `--` argument is not to be treated as an option (to allow you to pass other arguments that start with a dash but are not options).

We will need a few more variables declared at the file scope to allow us to store various bits of information relating to the chunks we find. We will need an array to hold the list of all the root chunks we find, a hash to mark which chunks are are used in other chunks, and a hash of arrays to store file and line number information where each chunk starts in the source file:

3a ⟨*declare variables* 2a⟩+≡ (1) ◁2a
```
my (@roots, %used, %chunks, $file);
```
Defines:
`%chunks`, *used in chunks 4–6.*
`$file`, *used in chunks 3c, 4a, and 7c.*
`@roots`, *used in chunks 4b and 5a.*
`%used`, *used in chunk 4.*

The next thing we need to do is open the source file and parse it for chunks, recording information about each chunk we find:

3b ⟨*open and parse file for chunks* 3b⟩≡ (1)
```
⟨open the source file 3c⟩
while ( <FILE> ) {
    ⟨find and parse chunks 3d⟩
}
```

This is a simple tangling program, and we only allow one source file to be specified on the command line. Now that we have already pulled out any options from `@ARGV`, we can simply test this array to make sure there is still at least one more item in it (the filename) store that value in our `\$file` variable, and open the file:

3c ⟨*open the source file* 3c⟩≡ (3b)
```
unless (@ARGV) {die "no file given for processing\n"}
$file = $ARGV[0];
open(FILE,$file)||die "can't open $file $!";
```
Uses `$file` *3a.*

We will declare two additional lexical variables to be used during our parsing loop on the file, one to hold each line in turn and one to hold the current line number (actually the current line number plus 1 so that if we find a chunk definition the line number points to the first line of code immediately following the chunk tag).

3d ⟨*find and parse chunks* 3d⟩≡ (3b)
```
my $line = $_;
my $line_no = $. + 1;
if ($line =~ m/^$la([^>]+)$ra=\s*$/) {
    ⟨parse chunk 4a⟩
}
```
Defines:
`$line`, *used in chunks 4a and 6–9.*
`$line_no`, *used in chunks 4a and 7c.*
Uses `$la` *2a and* `$ra` *2a.*

The regular expression above simply finds lines that have match the chunk definition tag and captures the chunk name into the `\$1` variable. This expression assumes that a `>` will not be part of the chunk name and that no chunk name will be entirely empty.

The first thing we when parsing a chunk itself is to record where in the file the chunk begins. We do this using the `tell()` function. This function returns the current byte offset into the file represented by the filehandle. Later we can use this value to `seek()` directly to this position in the file when we are actually tangling out the code. Then we can continue looping through the chunk (while the current line does not match either a single `@` or one followed by a `\%def` list of identifiers. Inside this loop we grab the next line of the file, and see if it contains a reference to another chunk—if it does, we use that chunk name as a key in our `\%used` hash and add one to its value (this hash will thus be a record of every chunk that was used inside of another chunk).

4a ⟨*parse chunk* 4a⟩≡ (3d)
```
my $begin_offset = tell FILE;
while (($line !~ m/^(\@\s*$|\@\s*\%def)/)) {
   $line = <FILE>;
   $used{$1} += 1 if $line =~ m/^\s*$la([^>]+)$ra\s*$/;
}
push @{$chunks{$1}}, "$begin_offset:$file:$line_no";
```
Uses `%chunks` *3a,* `$file` *3a,* `$la` *2a,* `$line` *3d,* `$line_no` *3d,* `$ra` *2a, and* `%used` *3a.*

When we finished reading through the chunk, we used the 'auto-vivification' syntax to push a string containing the offset, filename, and line number information of the chunk we just parsed. In other words, the `\%chunks` hash contains keys which are chunk names and whose values hold an array of offset/information strings for every location where that chunk's definition is continued throughout the file. The `\$1` variable here is the one matched in the outer `if` statement, the one matched inside the inner `while` loop was localized to that block which is now out of scope.

At this point we know the byte offset location of where every chunk definition starts in the file, and the line number of the first line of code in that chunk definition. We also have a record of every chunk that was used in another chunk (and hence can not be a root chunk itself). We will now populate our `@roots` array with the root chunks we need to extract. If we were given the `-R` option, we will only store that one root chunk name in this array. If that option was not given we want to extract all the root chunks—we can do this by simply looping through the keys of the `\%chunks` hash (the keys are all the chunk definitions encountered) and pushing them onto the `@roots` array if they were not used (are not seen in the `\%used` hash).

4b ⟨*make array of root chunks* 4b⟩≡ (1)
```
if ($Root) {
    @roots = ($Root);
} else {
    foreach my $key (keys %chunks) {
        push @roots, $key if not $used{$key};
    }
}
```
Uses `%chunks` *3a,* `$Root` *2a,* `@roots` *3a, and* `%used` *3a.*

Printing out the root chunks is simply a matter of opening a file for writing for each root chunk name and printing out the tangled version of that chunk (which we do using a function called `print\_chunk()`):

5a ⟨*print root chunks* 5a⟩≡ (1)
```
  foreach my $root (@roots) {
      open(PROGRAM, ">$root") || die "can't open $root: $!";
      print_chunk($root,'');
  }
  close PROGRAM;
```
Uses `@roots` *3a.*

The `print\_chunk()` subroutine takes two arguments, the name of the chunk to print and a string representing the current indentation level (which is empty for the beginning of root chunks)—this indentation argument will be discussed shortly.

Now we can start defining our chunk of subroutine definitions, which at the moment has only one:

5b ⟨*subroutine definitions* 5b⟩≡ (1) 7b▷
  ⟨*sub print_chunk* 5c⟩

The subroutine to print the chunk begins by assigning the two arguments to lexical variables. This function will be called recursively to print out any chunks contained within the current chunk being printed. For this reason, we need to ensure that the current chunk name is actually defined in the `\%chunks` hash (there may have been a typo in the name of an embedded chunk reference). Then we may print the chunk.

5c ⟨*sub print_chunk* 5c⟩≡ (5b)
```
  sub print_chunk {
      my ($chunk,$whitespace) = @_;
      ⟨make sure chunk is defined 5d⟩
      ⟨print the chunk 6a⟩
  }
```
Defines:
  `$chunk`*, used in chunks 5d and 6a.*
  `$whitespace`*, used in chunks 8b and 9a.*

Testing whether the current chunk is defined is a simple matter of checking its name in the `\%chunks` hash—if we do not find it we issue a warning and return from the function (we should probably use `die()` here, but by using a warning we can continue and perhaps find other such errors which the user can then fix all at once).

5d ⟨*make sure chunk is defined* 5d⟩≡ (5c)
```
  unless (exists $chunks{$chunk}) {
      warn "undefined chunk name: $la$chunk$ra\n";
      return;
  }
```
Uses `$chunk` *5c,* `%chunks` *3a,* `$la` *2a, and* `$ra` *2a.*

Printing out the chunk is a fairly complex task and we need to break this down into more manageable units before proceeding. First we will assign the list of chunk information strings (offsets and line numbers for chunk definitions) into a lexical array variable. Then we will seek to position to begin processing a chunk, set a shebang line flag (we will discuss this shortly), set up and print the line directive if necessary, and tangle out (print) the current chunk.

6a ⟨*print the chunk* 6a⟩≡ (5c)
```
my @locations = @{$chunks{$chunk}};
foreach my $item (@locations) {
      ⟨get location info and seek to offset 6b⟩
      ⟨set flag for shebang line 6c⟩
      ⟨set and print line directive 7a⟩
      ⟨tangle out current chunk 8a⟩
}
```
Defines:
`$item`, *used in chunk 6b.*
`@locations`, *never used.*
Uses `$chunk` *5c and* `%chunks` *3a.*

Remember that each chunk was assigned a list of strings, each of which was a colon separated list containing the offset into the file, the filename itself, and the line number. We simply split such a string at the colons into three lexical variables and then `seek()` to the offset in the file where the chunk definition begins (or is continued). The `seek()` function takes three arguments: A file-handle to seek on, a byte offset position, and a third argument which tells it to move to the position relative to a certain point (0 means move directly to that position, 1 means move that many bytes forward from the current position, and 2 means move to byte position given relative to the end of the file—in which case it usually only makes sense to use a negative byte offset). We use only the 0 flag for the third argument. The `seek()` function returns 1 if successful and 0 if it failed, so we can test it and `die()` just as we do for a call to the `open()` function.

6b ⟨*get location info and seek to offset* 6b⟩≡ (6a)
```
my ($offset, $filename, $line_number) = split /:/,$item;
seek(FILE, $offset,0) || die "seek failed on $filename: $!";
my $line=<FILE>;
```
Defines:
`$filename`, *used in chunks 7–9.*
`$line_number`, *used in chunks 7–9.*
`$offset`, *used in chunk 8b.*
Uses `$item` *6a and* `$line` *3d.*

In a perl program, you often use a 'shebang' line as the first line in the program. This line must be the first line in the file so we do not want a line directive emitted before this line. Instead, we would like to emit a line directive immediately following the shebang line and pointing to the next line in the code chunk. Here we merely test if the current code line is a shebang line and set a flag if it is.

6c ⟨*set flag for shebang line* 6c⟩≡ (6a)

```
my $shebang_special = 0;
$shebang_special = 1 if $line =~ m/^#!/;
```

Defines:
`$shebang_special`, *used in chunks 7a and 9b.*
Uses `$line` *3d.*

At this point we need to create a formatted line directive, substituting in the correct information for the placeholders we used in the `\$Line\_dir` variable. We will format this line directive in a separate function. Then we need to print out this line directive, but only if the current line is not a shebang line, or an embedded chunk reference (in which case, we will want to print a line directive when processing that chunk). We use a simple set of logical ORs which will terminate at the first expression that is true, and thus only print out the line directive when it is needed.

7a ⟨*set and print line directive* 7a⟩≡ (6a)

```
my $line_dir;
if ($Line_dir) {
   $line_dir = make_line_dir($line_number,$filename);
   $line =~ m/^\s*$la.*?$ra\s*$/ ||
            $shebang_special   ||
            print PROGRAM  "$line_dir";
}
```

Defines:
`$line_dir`, *used in chunks 7–9.*
Uses `$filename` *6b,* `$la` *2a,* `$line` *3d,* `$Line_dir` *2a,* `$line_number` *6b,* `$ra` *2a, and* `$shebang_special` *6c.*

Since we have just used the `make\_line\_dir()` function, we should go ahead and define it here—this also illustrates the point about being able to write our literate source in the order that makes sense for discussion. First let's add to the subroutine definitions chunk:

7b ⟨*subroutine definitions* 5b⟩+≡ (1) ◁5b
  ⟨*sub make_line_dir* 7c⟩

Now, the function to format our line directive is a simple set of substitutions operations. The function is passed parameters for the current line number and the filename, which we immediately assign to lexical variables. We then declare a new lexical `\$line\_dir` variable and assign it the value of our file scoped `\$Line\_dir` variable which holds the line directive string with placeholders. Finally, we simply replace the placeholders with their proper values and return the value of the `\$line\_dir` variable.

7c ⟨*sub make_line_dir* 7c⟩≡ (7b)

```
sub make_line_dir {
    my ($line_no,$file) = @_;
    my $line_dir = $Line_dir;
    $line_dir =~ s/\%L/$line_no/;
    $line_dir =~ s/\%F/$file/;
    $line_dir =~ s/\%\%/%/;
```

```
        $line_dir =~ s/\%N/\n/;
        return $line_dir;
    }
```
Uses `$file` *3a,* `$Line_dir` *2a,* `$line_dir` *7a, and* `$line_no` *3d.*

In order to tangle out our chunk, we use a similar loop to that used when parsing the chunks in the first place—that is, we continually loop while the current line does not match a chunk terminating line (we must make sure we read in another line in both of blocks of the `if/else` statement or we would be looping forever on the same line). Inside the loop the current line might be an embedded chunk reference, in which case we will need to tangle out that embedded chunk. Note, we are capturing the leading whitespace if there is an embedded chunk reference, as well as the chunk name—this way we can call the `print\_chunk()` routine and pass it a string representing the current indentation level so our tangled code has the appropriate indentation. If the line does not contain a chunk reference, we will simply print the code line (and print out a line directive following it if it was a shebang line).

8a ⟨*tangle out current chunk* 8a⟩≡ (6a)
```
  while ($line !~ m/^(\@\s*$|\@\s\%def)/) {
      if ($line =~ m/^(\s*?)$la([^>]+)$ra\s*$/) {
          ⟨tangle out embedded chunk 8b⟩
      } else {
          ⟨print out line 9a⟩
          ⟨take care of shebang line 9b⟩
      }
  }
```
Uses `$la` *2a,* `$line` *3d, and* `$ra` *2a.*

To tangle out an embedded chunk, we first get the current offset into the file and calculate our new indentation level. Then we make a recursive call to the `print\_chunk()` function to deal with the embedded chunk. Following that we simply reset our indentation to its previous value, `seek()` back to where we left off in the file and read in a new line from the file. We will also have to print a new line directive indicating our position in the file again.

8b ⟨*tangle out embedded chunk* 8b⟩≡ (8a)
```
  my $offset = tell FILE;
  my $addedspace = $1;
  $whitespace = $addedspace.$whitespace;
  &print_chunk($2,$whitespace);
  $whitespace = substr($whitespace,length($addedspace));
  seek(FILE,$offset,0) || die "can't seek on $filename: $!";
  $line_number += 1;
  $line = <FILE>;
  ⟨add returning line directive 8c⟩
```
Uses `$filename` *6b,* `$line` *3d,* `$line_number` *6b,* `$offset` *6b, and* `$whitespace` *5c.*

We need to add a new line directive to indicate that we have returned back the position in the current chunk. We need only do this if the current line is not another chunk reference.

8c ⟨*add returning line directive* 8c⟩≡ (8b)
```
if ($Line_dir) {
    $line_dir = make_line_dir($line_number,$filename);
    print PROGRAM $line_dir if $line !~ /^\s*$la.*?$ra\s*$/;
}
```
Uses `$filename` *6b,* `$la` *2a,* `$line` *3d,* `$Line_dir` *2a,* `$line_dir` *7a,* `$line_number` *6b, and* `$ra` *2a.*

If the current line is not a chunk reference we simply need to print it out with the correct amount of leading whitespace, read in another line line from the file and increment our line number counter.

9a ⟨*print out line* 9a⟩≡ (8a)
```
print PROGRAM $whitespace,$line;
$line = <FILE>;
$line_number += 1;
```
Uses `$line` *3d,* `$line_number` *6b, and* `$whitespace` *5c.*

Finally, if the current line is a shebang line then we want to add a line directive directly after it, and reset the shebang flag to 0.

9b ⟨*take care of shebang line* 9b⟩≡ (8a)
```
if ($Line_dir && $shebang_special) {
    $line_dir = make_line_dir($line_number,$filename);
    print PROGRAM "$line_dir" if $line !~ m/^\s*$la[^>]+$ra\s*$/;
    $shebang_special = 0;
}
```
Uses `$filename` *6b,* `$line` *3d,* `$Line_dir` *2a,* `$line_dir` *7a,* `$line_number` *6b,* `$ra` *2a, and* `$shebang_special` *6c.*

That concludes the `pqtangle` program. The whole program is about 95 lines of code and won't be listed here. Of course, you already have enough experience reading the chunk syntax that you can assemble it from the literate listing above. If you do not wish to enter the code manually, the tangled script is available at the site mentioned in the previous section.

Identifier Index

Chunk Index