`nofake` - simple `notangle` replacement for the `noweb` literate programming tool

December 14, 2021

Contents

1 `nofake`

`nofake` is a Perl script that acts as a simple substitute for the `notangle` command from the `noweb` literate programming system. When using `noweb`, program documentation and source code are kept together in a single file (by convention named with the `.nw` suffix). The `notangle` command takes such a *noweb* file and extracts the source code from it for compilation. However, not everybody who wants to compile the source code has a full `noweb` installation available. For these cases `nofake` can be shipped with the source code to provide a substitute for the `notangle` command. `nofake` has a much simpler and less general architecture than `notangle` but can handle most cases correctly.

`nofake` is derived from NegWeb 1.0.1 that was released by its author Darrell Johnson `<darrell@boswa.com>` into the public domain at `http://boswa.com/misc/negweb/`.

Note that this source is itself a literate program in `noweb` syntax. It has to use the delimiters `<<` and `>>` literally and hence has to escape them with a preceding `@`.

1.1 Manual Page

Perl manual pages are written in Perl's Plain Old Document (POD) format. They can be included into a script and extracted from there: `pod2man nofake`

`> nofake.1`; various other format can be generated as well, including HTML. This makes sure they do not get lost when the script is passed around. For the syntax, see the `perlpod` manual.

2 ⟨*manual page in POD format* 2⟩≡ (5)

```
=head1 NAME

nofake - simple notangle replacement for the Noweb literate programming tool

=head1 SYNOPSIS

B<nofake> [B<-R>I<chunk> ...] [B<-L>[I<format>]] [I<file>] ...

B<nofake> [B<--version>|B<-v>]

B<nofake> [B<--list-all>] [I<file>] ...

B<nofake> [B<--list-roots>] [I<file>] ...

=head1 DESCRIPTION

Noweb(1) is a literate-programming tool like Knuth's WEB. A noweb file
contains program source code interleaved with documentation. Extracting
the source code for compilation requires notangle(1). To allow source
code to be shipped to users not using noweb(1), B<nofake> offers the
most commonly used functionality of notangle(1) as a simple perl(1)
script. Alas, B<nofake> extracts source code from a file in noweb(1)
syntax: B<nofake> reads I<file> and extracts the code chunke named
I<chunk> to stdout. If no I<file> is provided, B<nofake> reads from
stdin, if no I<chunk> is named, B<nofake> extracts the chunk C<*>.

=head1 OPTIONS

=over 4

=item B<-R>I<chunk>

Extract chunk I<chunk> (recursively) from the noweb file and write it to
stdout.

=item B<-L>[I<format>]

B<nofake> emits cpp(1)-style C<#line> directives to allow a compiler
emit error messages that refer to I<file> rather than the extracted
source code directly.  The optional I<format> allows to provided the
format of the line directive: C<-L'#line %L "%F"%N'>. In I<format> C<%F>
```

```
indicates the name of the source file, C<%L> the line number, and C<%N>
a newline. The default C<#line %L "%F"%N> is suitable for C compilers.

=item B<--list-all>

List all I<chunks> in noweb files.

=item B<--list-roots>

List all I<chunks> in noweb files that are not referenced by other chunks.

=back

=head1 SYNTAX OF NOWEB FILES

The authoritive source for the syntax of noweb files is the noweb(1)
documentation. However, here is an example:

    <<hello.c>>=
    <<includes>>

    int main(int argc, char** argv)
    {
        <<say hello>>
        return 0;
    }

    <<say hello>>=
    printf("Hello World!\n");
    @

    <<includes>>=
    #include <stdio.h> /* for printf */
    @

A chunk is defined by C<E<lt>E<lt>chunkE<gt>E<gt>=> and reaches up
to the next definition or a line starting with C<@> followed by a
space or newline. A chunk can recursivley refer to other chunks:
chunk C<hello.c> refers to C<includes> and C<say hello>. A chunk
is referred to by C<E<lt>E<lt>chunkE<gt>E<gt>>. To use the
C<E<lt>E<lt>> and C<E<gt>E<gt>> character literally in a program,
precede them with a C<@>. Double C<@> on the first column to put
a literal C<@> there, applies only to the first column.

=head1 LIMITATIONS
```

```
The B<nofake> architecture is simpler than that of notangle(1) and
therefore one thing do not work. In particular:

=over 4

=item *

B<nofake> does not accept the B<-filter> I<command> option that
B<notangle> uses to filter chunks before they are emitted.

=back

=head1 COPYING

This software is in the public domain.

THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR AND COPYRIGHT HOLDER BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

=head1 AUTHOR

Christian Lindig <lindig@eecs.harard.edu>

Please note that this is a derived work and is not maintained
by Lindig, the canonical source for this version is
https://github.com/ctarbide/ctweb/blob/master/tools/nofake.nw.

=head1 SEE ALSO

noweb(1), notangle(1), perl(1), cpp(1)

Norman Ramsey, Literate programming simplified, IEEE Software
11(5):97-105, September 1994.
```

1.2 Implementation

5 ⟨*nofake* 5⟩≡

```
#!/usr/bin/env perl
#
# Generated from nofake.nw. Do not edit! Edit nofake.nw instead and
# run nofake on it:
#
#   ./nofake -Rnofake nofake.nw > nofake.pl
#   mv nofake.pl nofake
#   chmod a+x nofake
#
# The manual page is at the end of this file in Perl's
# POD format. You can format it using pod2man(1):
#
#   pod2man nofake > nofake.1
#   nroff -man nofake.1 | more
#
# The noweb source for this version is available at:
#
#   https://github.com/ctarbide/ctweb/blob/master/tools/nofake.nw
#
# This software is in the public domain; for details, see the manual
# page.

eval 'exec perl -wS $0 ${1+"$@"}'
    if 0;

use 5.008; # perl v5.8.0 was released on July 18, 2002
use strict;
use warnings FATAL => qw{uninitialized void inplace};
use Carp ();
$SIG{__DIE__} = \&Carp::confess;

local $\ = "\n";

sub version {
    print <<'EOF';
    (c) 2021 C. Tarbide <ctarbide@tuta.io>
    (c) 2002 Christian Lindig <lindig@eecs.harvard.edu>

    NoFake is derived from the public domain NegWeb 1.0.1
    http://boswa.com/misc/negweb by Darrell Johnson <darrell@boswa.com>.
EOF
}
```

⟨*defaults* 6⟩

⟨*many utils* 15⟩
⟨*line directive* 7a⟩
⟨*read file* 8b⟩
⟨*sub extract* 12a⟩
⟨*sub usage* 12b⟩

⟨*process command line and extract chunk* 13⟩

```
__END__
```

⟨*manual page in POD format* 2⟩

1.3 Variables

All (code) chunks are stored in the global hash table `chunks`. A chunk may contain references to other chunks. These are ignored when a chunk is read and expanded when then chunk itself is expanded.

6 ⟨*defaults* 6⟩≡ (5)

```
my $lineformat = '#line %L "%F"%N';   # default format for #line directive
my $sync = 0;                         # default: do not emit #line directives

my %chunks = ();                      # hash table for chunks
my %chunks_options = ();              # hash table for chunks options

my $list_roots = 0;                   # list only root chunks
```

1.4 Functions

If we compile source code that was extracted from a noweb file, we want the error messages point to the noweb file and not the file extracted from it. Therefore we emit `#line` directives that connect the extracted file back with the noweb file. They are understood by many compilers such that they can emit error messages pointing to the noweb file.

The `line_directive` returns the string to be used as a line directive. The formatting is taken from `lineformat` that is controlled by the `-L` command line option.

7a ⟨*line directive* 7a⟩≡ (5)

```
sub line_directive {
    my ($fname, $line) = @_;
    my $ret = $lineformat;
    $ret =~ s/%F/$fname/g;
    $ret =~ s/%L/$line/g;
    $ret =~ s/%N/\n/g;
    return $ret;
}
```

The `read_line` function reads a file line-by-line and recognizes chunk definitions. Each chunk is put under its name into the global `chunks` hash table. The `sync` flag controls whether line directives are emitted.

Relevant `man noweb` section: Documentation chunks begin with a line that starts with an at sign (`@`) followed by a space or newline.

A `@@` at the first column must yield a `@`.

7b ⟨*read line vars* 7b⟩≡ (8b)

```
my $chunk = '';
```

8a ⟨*read line* 8a⟩≡ (8b)

```
if ($line =~ m{^<<(.+?)>>=\s*$}) {
    $chunk = $1;
    if (!$chunks_options{$chunk}) {
        $chunks_options{$chunk} = {nextractions => 0};
    }
    $chunks_options{$chunk}->{nchunks}++;
} elsif ($chunk) {
    if ($line =~ m{^\@(?:$|\s)}) {
        $chunk = '';
    } else {
        # regular line inside chunk
        $line =~ s,^\@\@,\@,;
        set_many(%chunks, $chunk, $ARGV, $., $line . "\n");
    }
} else {
    if ($line =~ m{^(|.*?[^\@])<<} and $1 !~ m{\[\[}) {
        Carp::carp("WARNING: Unescaped << in documentation chunk at line ${ARGV}:${.}.");
    }
}
```

8b ⟨*read file* 8b⟩≡ (5)

```
sub read_file {
    local @ARGV = @_;
    ⟨read line vars 7b⟩
    while (my $line = <>) {
        chomp($line);
        ⟨read line 8a⟩
    }
}
```

The `extract` function takes a chunk name and extracts this chunk recursively from the `chunks` hash table. The output is returned as a string.

While we look for chunk names in lines to extract we have to be careful: if a chunk name delimiter is preceded by a an `@` it does not denote a chunk, but the delimiter literally.

When dealing with line numbers, `nofake` prioritize document structure, `notangle` is more liberal in breaking lines to put line directives.

8c ⟨*setup $before_has_content* 8c⟩≡ (9a 10)

```
my $before_has_content = $before !~ m{^\s*$};
```

9a ⟨*setup $indent and $before_has_content* 9a⟩≡ (10)

```
my $indent;
⟨setup $before_has_content 8c⟩
if ($before_has_content) {
    $indent = ' ' x length($before);
} else {
    $indent = $before;
}
```

These special items will be processed at the top level, after all files have been read.

9b ⟨*propagate header special items* 9b⟩≡ (10)

```
while (@chunkreflines and ref($chunkreflines[0])) {
    # propagate special items early, before processing lines
    push(@res, shift(@chunkreflines));
}
```

Defining $line_new will effectively replace the current line and search for more references. This allows more than one chunk reference per line.

10 ⟨*extract and process chunk reference* 10⟩≡ (11a)

```
my @chunkreflines = extract($prefix . $before, $fname, $lnum, $chunkref);
⟨propagate header special items 9b⟩
if (@chunkreflines > 1) {
    # many lines
    ⟨setup $indent and $before_has_content 9a⟩
    my $first = shift(@chunkreflines);
    my $last = pop(@chunkreflines);
    if ($before_has_content or $first ne "\n") {
        push(@res, $before . $first);
    } else {
        push(@res, "\n");
    }
    for (@chunkreflines) {
        if (ref($_)) {
            # deal with special items later
            push(@res, $_);
        } elsif ($_ ne "\n") {
            push(@res, $indent . $_);
        } else {
            # no need to indent an empty line
            push(@res, "\n");
        }
    }
    if ($after) {
        chomp($last);
        $line_new = $indent . $last . $after . "\n";
    } elsif ($last ne "\n") {
        push(@res, $indent . $last);
    } else {
        # no need to indent an empty line
        push(@res, "\n");
    }
} elsif (@chunkreflines) {
    # just 1 line
    if ($after) {
        chomp(my $tmp = $chunkreflines[0]);
        $line_new = $before . $tmp . $after . "\n";
    } else {
        ⟨setup $before_has_content 8c⟩
        if ($before_has_content or $chunkreflines[0] ne "\n") {
            push(@res, $before . $chunkreflines[0]);
        } else {
```

```
            push(@res, "\n");
        }
    }
} else {
    Carp::carp("WARNING: Code chunk <<${chunkref}>> is empty (at ${fname}:${lnum}).");
}
```

11a ⟨*process line* 11a⟩≡ (12a)

```
my $found_chunk_ref;
for (;;) {
    $found_chunk_ref = ($line =~ /^(|.*?[^\@])<<(.*?[^\@])>>(.*)/);
    if ($found_chunk_ref) {
        my $before = $1;
        my $chunkref = $2;
        my $after = $3;
        if ($chunks_options{$chunkref}) {
            my $line_new = undef;
            ⟨extract and process chunk reference 10⟩
            if (defined($line_new)) {
                $line = $line_new;
                next;
            }
        } else {
            Carp::carp("WARNING: Code chunk <<${chunkref}>> does not exist (at ${fname}:${lnum}).");
        }
        $lnum = -1; # force line number directive
    } else {
        $line =~ s/\@(<<|>>)/$1/g;
        push(@res, $line);
    }
    last;
}
```

A chunk referenced at least once is definitely a non-root chunk, so we need to render all chunks at most once

11b ⟨*remove chunk data if listing roots* 11b⟩≡ (12a)

```
if ($list_roots and @input) {
    $chunks{$chunk} = [$input[0], $input[1], ""];
}
```

Special item `0` is a line number directive.

11c ⟨*push line number directive* 11c⟩≡ (12a)

```
push(@res, [0, $fname, $lnum]);
```

12a ⟨*sub extract* 12a⟩≡ (5)

```
my %being_extracted = ();
sub extract {
    my ($prefix, $parent_fname, $parent_lnum, $chunk) = @_;
    if ($being_extracted{$chunk}) {
        Carp::croak("ERROR: Code chunk <<$chunk>> is used in its own definition (at ${parent_fname}:$
    }
    $being_extracted{$chunk}++;
    $chunks_options{$chunk}->{nextractions}++;
    my @res = ();
    my @input = get_many($chunks{$chunk});
    ⟨remove chunk data if listing roots 11b⟩
    my $lnum_previous = -1;
    while (my ($fname, $lnum, $line) = splice(@input, 0, 3)) {
        if ($sync and $lnum != $lnum_previous + 1) {
            ⟨push line number directive 11c⟩
        }
        ⟨process line 11a⟩
        $lnum_previous = $lnum;
    }
    $being_extracted{$chunk}--;
    return @res;
}
```

12b ⟨*sub usage* 12b⟩≡ (5)

```
sub usage {
    my $arg = shift @_;

    print STDERR <<EOF;
    Unknown command line argument "$arg". See the manual page for help
    which is also included at the end of this Perl script.

    nofake [-Rchunk ...] [-L[format]] [file] ...
    nofake [--version|-v]
    nofake [--list-all] [file] ...
    nofake [--list-roots] [file] ...
EOF
}
```

We parse the command line and start working: reading the noweb files and then extracting each chunk in the `chunks` variable.

13 ⟨*process command line and extract chunk* 13⟩≡ (5)

```
my @chunks = ();
my @files = ();
my $list_all = 0;
my $no_op = 0;

while ($_ = shift(@ARGV)) {
    if (/^\-L(.*)/)             { $sync = 1; if ($1 ne '') { $lineformat = $1 }}
    elsif (/^\-R(.+)/)          { push(@chunks, $1) }
    elsif (/^\-(v|-version)$/)  { version(); exit(0) }
    elsif (/^\--list-all$/)     { $list_all = 1 }
    elsif (/^\--list-roots$/)   { $list_roots = 1 }
    elsif (/^\-filter$/)        { shift(@ARGV) }  # ignore
    elsif (/^\--no-op$/)        { $no_op = 1 }
    elsif (/^(\-.*)$/)          { usage($1); exit(1) }
    else                        { push(@files, $_) }
}

if (!@chunks) {
    push(@chunks, '*');
}

if (!@files) {
    push(@files, '-');
}

read_file($_) for @files;

if ($no_op) {
    # just read lines, useful for validating documentation chunks
} elsif ($list_all) {
    print("<<${_}>>") for sort(keys(%chunks_options));
} elsif ($list_roots) {
    ⟨output roots 14d⟩
} else {
    ⟨output lines 14c⟩
}
```

14a ⟨*output lines of $chunk* 14a⟩≡ (14c)

```
local $\ = undef;
my ($first_fname, $first_lnum) = get_many($chunks{$chunk});
for my $item (extract('', $first_fname, $first_lnum, $chunk)) {
    if (ref($item) eq 'ARRAY') {
        print line_directive($item->[1], $item->[2]);
    } else {
        print $item;
    }
}
```

14b ⟨*verify $chunk is defined* 14b⟩≡ (14c)

```
if (!$chunks_options{$chunk}) {
    Carp::croak("ERROR: The root chunk <<${chunk}>> was not defined.");
}
```

14c ⟨*output lines* 14c⟩≡ (13)

```
for my $chunk (@chunks) {
    ⟨verify $chunk is defined 14b⟩
}
for my $chunk (@chunks) {
    ⟨output lines of $chunk 14a⟩
}
```

14d ⟨*output roots* 14d⟩≡ (13)

```
my @all_chunks = sort(keys(%chunks_options));
# first pass, extract all chunks
for my $chunk (@all_chunks) {
    my $opts = $chunks_options{$chunk};
    my ($first_fname, $first_lnum) = get_many($chunks{$chunk});
    extract('', $first_fname, $first_lnum, $chunk);
}
# second pass, report
for my $chunk (@all_chunks) {
    my $opts = $chunks_options{$chunk};
    if ($opts->{nextractions} == 1) {
        print("<<${chunk}>>");
    }
}
```

15 ⟨*many utils* 15⟩≡ (5)

```
sub set_many (\%@) {
    my ($h, $k, @rest) = @_;
    if (defined($h->{$k})) {
        if (ref($h->{$k}) eq 'ARRAY') {
            push(@{ $h->{$k} }, @rest);
        } else {
            $h->{$k} = [$h->{$k}, @rest];
        }
    } elsif (@rest > 1) {
        $h->{$k} = [ @rest ];
    } elsif (@rest) {
        $h->{$k} = $rest[0];
    } else {
        Carp::carp("no value specified");
    }
}
sub get_many {
    # () yield undef in scalar context
    return () unless defined($_[0]);              # return () instead of (undef)
    return @{ $_[0] } if ref($_[0]) eq 'ARRAY';   # return @{arg}
    @_;                                           # return (arg)
}
```

This was taken from `perl\5.8.3\bin\runperl.bat` and tested under Windows XP SP3 (x86). Simply extract it to somewhere in the PATH, see documentation below in DESCRIPTION section.

16 ⟨*nofake.bat* 16⟩≡

```
@rem = '--*-Perl-*--
@echo off
if "%OS%" == "Windows_NT" goto WinNT
perl.exe -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
:WinNT
perl.exe -x -S %0 %*
if NOT "%COMSPEC%" == "%SystemRoot%\system32\cmd.exe" goto endofperl
if %errorlevel% == 9009 echo You do not have Perl in your PATH.
if errorlevel 1 goto script_failed_so_exit_with_non_zero_val 2>nul
goto endofperl
@rem ';
#!perl -w
#line 15
$0 =~ s|\.bat||i;
unless (-f $0) {
    $0 =~ s|.*[/\\]||;
    for (".", split ';', $ENV{PATH}) {
        $_ = "." if $_ eq "";
        $0 = "$_/$0" , goto doit if -f "$_/$0";
    }
    die "`$0' not found.\n";
}
doit: exec "perl.exe", "-x", $0, @ARGV;
die "Failed to exec `$0': $!";
__END__

=head1 NAME

runperl.bat - "universal" batch file to run perl scripts

=head1 SYNOPSIS

        C:\> copy runperl.bat foo.bat
        C:\> foo
        [..runs the perl script `foo'..]

        C:\> foo.bat
        [..runs the perl script `foo'..]
```

```
=head1 DESCRIPTION

This file can be copied to any file name ending in the ".bat" suffix.
When executed on a DOS-like operating system, it will invoke the perl
script of the same name, but without the ".bat" suffix.  It will
look for the script in the same directory as itself, and then in
the current directory, and then search the directories in your PATH.

It relies on the C<exec()> operator, so you will need to make sure
that works in your perl.

This method of invoking perl scripts has some advantages over
batch-file wrappers like C<pl2bat.bat>:  it avoids duplication
of all the code; it ensures C<$0> contains the same name as the
executing file, without any egregious ".bat" suffix; it allows
you to separate your perl scripts from the wrapper used to
run them; since the wrapper is generic, you can use symbolic
links to simply link to C<runperl.bat>, if you are serving your
files on a filesystem that supports that.

On the other hand, if the batch file is invoked with the ".bat"
suffix, it does an extra C<exec()>.  This may be a performance
issue.  You can avoid this by running it without specifying
the ".bat" suffix.

Perl is invoked with the -x flag, so the script must contain
a C<#!perl> line.  Any flags found on that line will be honored.

=head1 BUGS

Perl is invoked with the -S flag, so it will search the PATH to find
the script.  This may have undesirable effects.

=head1 SEE ALSO

perl, perlwin32, pl2bat.bat

=cut


__END__
:endofperl
```

2 List of all chunks