Extracted from:

Practical Vim

Edit Text at the Speed of Thought

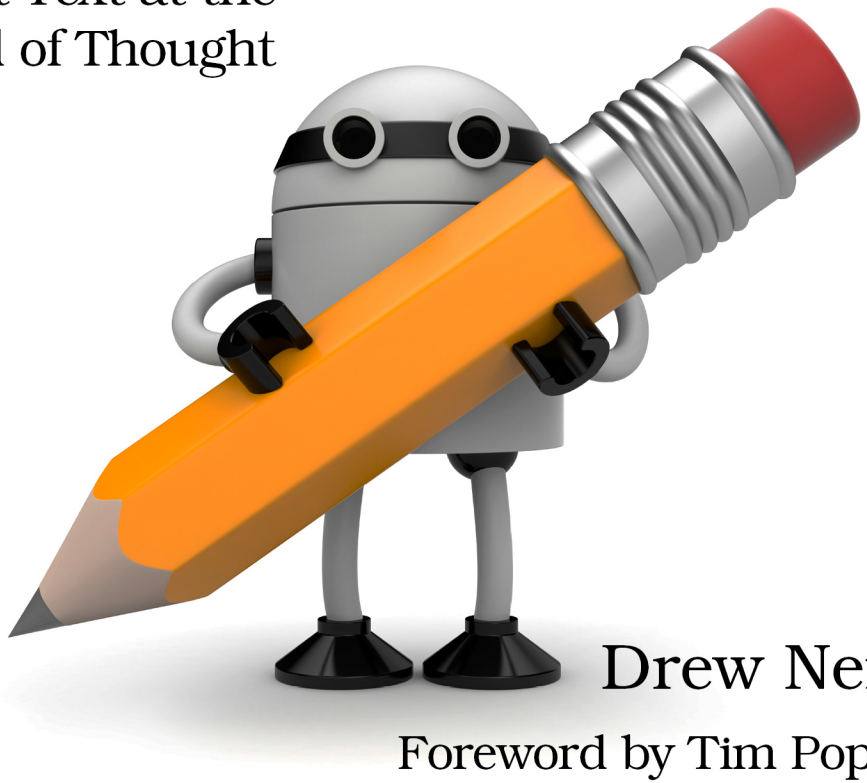
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Practical Vim

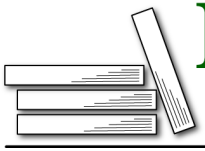
Edit Text at the Speed of Thought



Drew Neil

Foreword by Tim Pope

Edited by Kay Keppler



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Kay Keppler (editor)
Potomac Indexing, LLC (indexer)
Molly McBeath (copyeditor)
David J. Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-98-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2012

Tip 64

Record and Execute a Macro

Macros allow us to record a sequence of changes and then play them back. This tip shows how.

Many repetitive tasks involve making multiple changes. If we want to automate these, we can record a macro and then execute it.

Capture a Sequence of Commands by Recording a Macro

The `q` key functions both as the "record" button and the "stop" button. To begin recording our keystrokes, we type `q{register}`, giving the address of the register where we want to save the macro. We can tell that we've done it right if the word "recording" appears in the status line. Every command that we execute will be captured, right up until we press `q` again to stop recording.

Let's see this in action:

Keystrokes	Buffer Contents
`qa`	foo = 1 bar = 'a' foobar = foo + bar
`A;<Esc>`	foo = 1; bar = 'a' foobar = foo + bar
`Ivar␣<Esc>`	var foo = 1; bar = 'a' foobar = foo + bar
`q`	var foo = 1; bar = 'a' foobar = foo + bar

Pressing `qa` begins recording and saves our macro into register `a`. We then perform two changes on the first line: appending a semicolon and prepending the word `var`. Having completed both of those changes, we press `q` to stop recording our macro (`:h q`).

We can inspect the contents of register `a` by typing the following:

```
⇛ :reg a
❮ --- Registers ---
  "a   A;^[Ivar ^[
```

It doesn't make for easy reading, but the same sequence of commands that we recorded moments ago should be recognizable. The only surprise might be that the symbol ^[is used to stand for the Escape key. See *Keyboard Codes in Macros, on page ?*, for an explanation.

Play Back a Sequence of Commands by Executing a Macro

The `@{register}` command executes the contents of the specified register (see :h @ ⓘ). We can also use `@@`, which repeats the macro that was invoked most recently.

Here's an example:

Keystrokes	Buffer Contents
{start}	var foo = 1; bar = 'a' foobar = foo + bar
`j`	var foo = 1; bar = 'a' foobar = foo + bar
`@a`	var foo = 1; var bar = 'a'; foobar = foo + bar

`j@@`	var foo = 1;
	var bar = 'a';
	var foobar = foo + bar;

We've executed the macro that we just recorded, repeating the same two changes for each of the subsequent lines. Note that we use `@a` on the first line and then `@@` to replay the same macro on the next line.

In this example, we played the macro back by running `j@a` (and subsequently `j@@`). Superficially, this has some resemblance to the Dot Formula. It involves one keystroke to move (`j`) and two to act (`@a`). Not bad, but there's room for improvement.

We have a couple of techniques at our disposal for executing a macro multiple times. The setup differs slightly for each technique, but more importantly, they react differently on encountering an error. I'll explain the differences by way of a comparison with Christmas tree lights.

If you buy a cheap set of party lights, the chances are that they will be wired in series. If one bulb blows, they all go out. If you buy a premium set, they're more likely to be wired in parallel. That means any bulb can go out, and the rest will be unaffected.

I've borrowed the expressions *in series* and *in parallel* from the field of electronics to differentiate between two techniques for executing a macro multiple times. The technique for executing a macro in series is brittle. Like cheap Christmas tree lights, it breaks easily. The technique for executing a macro in parallel is more fault tolerant.

Execute the Macro in Series

Picture a robotic arm and a conveyor belt containing a series of items for the robot to manipulate (Figure 4, *Vim's macros make quick work of repetitive tasks*, on page 7). Recording a macro is like programming the robot to do a single unit of work. As a final step, we instruct the robot to move the conveyor belt and bring the next item within reach. In this manner, we can have a single robot carry out a series of repetitive tasks on similar items.

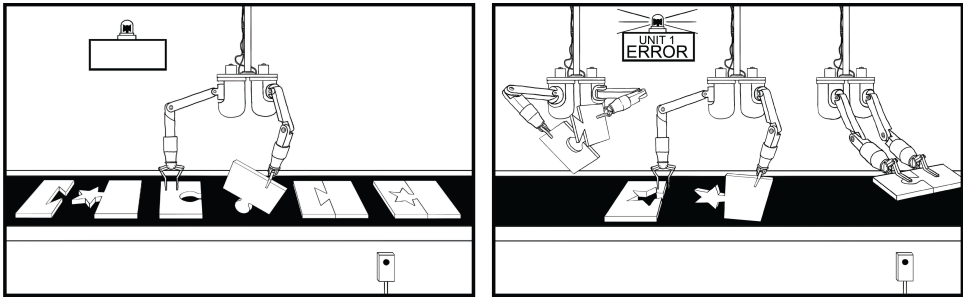


Figure 4—Vim's macros make quick work of repetitive tasks

One consequence of this approach is that if the robot encounters any surprises, it sounds an alarm and aborts the operation. Even if items on the conveyor belt still need to be manipulated, the work stops.

Execute the Macro in Parallel

When we execute the macro in parallel, it's as though we've dispensed with the conveyor belt entirely. Instead, we deploy an assemblage of robots,[1] all programmed to do the same simple task. Each is given a single job to do. If it succeeds, very well. If it fails, no matter.

Under the hood, Vim always executes macros sequentially, no matter which of these two techniques we use. The term *in parallel* is intended to draw an analogy with the robustness of parallel circuits. It is not meant to suggest that Vim executes multiple changes concurrently.

1. http://all-sorts.org/of/robots

In Tip 67, on page ?, as well as Tip 69, on page ?, we'll see examples of a macro being executed both in series and in parallel.

Tip 65

Normalize, Strike, Abort

Executing a macro can sometimes produce unexpected results, but we can achieve better consistency if we follow a handful of best practices.

When we execute a macro, Vim blindly repeats the sequence of canned keystrokes. If we aren't careful, the outcome when we replay a macro might diverge from our expectations. But it's possible to compose macros that are more flexible, adapting to do the right thing in each context.

The golden rule is this: when recording a macro, ensure that every command is repeatable.

Normalize the Cursor Position

As soon as you start recording a macro, ask yourself these questions: where am I, where have I come from, and where am I going? Before you do anything, make sure your cursor is positioned so that the next command does what you expect, where you expect it.

That might mean moving the cursor to the next search match (`n`) or the start of the current line (`0`) or perhaps the first line of the current file (`gg`). Always starting on square one makes it easier to strike the right target every time.

Strike Your Target with a Repeatable Motion

Vim has a rich vocabulary of motions for getting around a text file. Use them well.

Don't just hammer the `l` key until your cursor reaches its target. Remember, Vim executes your keystrokes blindly. Moving your cursor ten characters to the right might get you where you need to go right now as you record the macro, but what about when you play it back later? In another context, moving the cursor ten places to the right might overshoot the mark or stop short of it.

Word-wise motions, such as `w`, `b`, `e`, and `ge` tend to be more flexible than character-wise `h` and `l` motions. If we recorded the motion `0` followed by `e`, we could expect consistent results each time we executed the macro. The

cursor would end up on the last character of the first word of the current line. It wouldn't matter how many characters that word contained, so long as the line contained at least one word.

Navigate by search. Use text objects. Exploit the full arsenal of Vim's motions to make your macros as flexible and repeatable as you can. Don't forget: when recording a macro, using the mouse is *verboten*!

Abort When a Motion Fails

Vim's motions can fail. For example, if our cursor is positioned on the first line of a file, the `k` command does nothing. The same goes for `j` when our cursor is on the last line of a file. By default, Vim beeps at us when a motion fails, although we can mute it with the 'visualbell' setting (see :h 'visualbell').

If a motion fails while a macro is executing, then Vim aborts the rest of the macro. Consider this a feature, not a bug. We can use motions as a simple test of whether or not the macro should be executed in the current context.

Consider this example: We start by searching for a pattern. Let's say that the document has ten matches. We start recording a macro using the `n` command to repeat the last search. With our cursor positioned on a match, we make some small change to the text and stop recording the macro. The result of our edit is that this particular region of text no longer matches our search pattern. Now the document has only nine matches.

When we execute this macro, it jumps to the next match and makes the same change. Now the document has only eight matches. We execute the macro again and again, until eventually no matches remain. If we attempt to execute the macro now, the `n` command will fail because there are no more matches. The macro aborts.

Suppose that the macro was stored in the `a` register. Rather than executing `@a` ten times, we could prefix it with a count: `10@a`. The beauty of this technique is that we can be unscrupulous about how many times we execute this macro. Don't care for counting? It doesn't matter! We could execute `100@a` or even `1000@a`, and it would produce the same result.

Tip 66

Play Back with a Count

The Dot Formula can be an efficient editing strategy for a small number of repeats, but it can't be executed with a count. Overcome this limitation by recording a cheap one-off macro and playing it back with a count.

In Tip 3, on page ?, we used the Dot Formula to transform this:

the_vim_way/3_concat.js
```
var foo = "method("+argument1+","+argument2+")";
```

What we want is for it to look like this:

```
var foo = "method(" + argument1 + "," + argument2 + ")";
```

The Dot Formula meant that we could complete the task simply by repeating `;.` a few times. What if we faced the same problem but on a larger scale?

```
x = "("+a+","+b+","+c+","+d+","+e+")";
```

We can approach this in exactly the same way. But when we have to invoke the two commands `;.` so many times to complete the job, it starts to feel like a lot of work. Isn't there some way that we could apply a count?

It's tempting to think that running `11;.` would do the trick, but it's no use. This instructs Vim to run the `;` command eleven times, and then the `.` command once. The equivalent mistake is more obvious if we run `;11.`, which tells Vim to invoke `;` once and then `.` eleven times. We really want to run `;.` eleven times.

We can simulate this by recording one of the simplest possible macros: `qq;.q`. Here, `qq` tells Vim to record the following keystrokes and save them to the `q` register. Then we type our commands `;.` and finish recording the macro by pressing `q` one final time. Now we can execute the macro with a count: `11@q`. This executes `;.` eleven times.

Let's put all of that together. (See Table 16, *Recording and Playing a Macro with a Count*, on page 11.)

The `;` command repeats the `f+` search. When our cursor is positioned *after* the last + character on the line, the `;` motion fails and the macro aborts.

Keystrokes	Buffer Contents
{start}	x = "("+a+","+b+","+c+","+d+","+e+")";
f+	x = "("+a+","+b+","+c+","+d+","+e+")";
s + <Esc>	x = "(" +a+","+b+","+c+","+d+","+e+")";
qq;.q	x = "(" + a +","+b+","+c+","+d+","+e+")";
22@q	x = "(" + a + "," + b + "," + c + "," + d + "," + e +")";

Table 16—Recording and Playing a Macro with a Count

In our case, we want to execute the macro ten times. But if we were to play it back eleven times, the final execution would abort. In other words, we can complete the task so long as we invoke the macro with a count of ten or more.

Who wants to sit there and count the exact number of times that a macro should be executed? Not me. I'd rather provide a count that I reckon to be high enough to get the job done. I often use 22, because I'm lazy and it's easy to type. On my keyboard, the @ and 2 characters are entered with the same button.

Note that it won't always be possible to make approximations when providing a count to a macro. It works in this case because the macro has a built-in safety catch: the ; motion will fail if no more + symbols are left on the current line. See *Abort When a Motion Fails*, on page 9, for more details.