Vim registers: The basics and beyond

brianstorti.com/vim-registers

February 9, 2015

Vim’s register are that kind of thing you don’t think you need until you learn about it. Then, it’s hard to imagine life without it and it becomes essential in your workflow. It’s still common for people to use `vim` for years without knowing how to properly work with registers, which I think is a shame, and just a basic understanding of what they are and how they work can make you a lot more productive (and avoid a couple of annoyances).

If you have no idea what I’m talking about

You can think of registers as a bunch of spaces in memory that `vim` uses to store some text. Each of these spaces have a identifier, so it can be accessed later. It’s no different than when you copy some text to your clipboard, except you usually have just one clipboard to copy to, while `vim` allows you to have multiple places to store different texts.

The basic usage

Every register is accessed using a double quote before its name. For example, we can access the content that is in the register `r` with `"r`.

You could add the selected text to the register `r` by doing `"ry`. You are copying (`y`anking) the selected text, and then adding it to the register `"r`. To paste the content of this register, the logic is the same: `"rp`. You are `p`asting the data that is in this register.

You can also access the registers in insert/command mode with `Ctrl-r` + register name, like in `Ctrl-r r`. It will just paste the text in your current buffer. You can use the `:reg` command to see all the registers and their content, or filter just the ones that you are interested with `:reg a b c`.

```
:reg a b c
--- Registers ---
"a   register a content
"b   register b content
"c   register c content
```

The unnamed register

`vim` has a unnamed (or default) register that can be accessed with `""`. Any text that you delete (with `d`, `c`, `s` or `x`) or yank (with `y`) will be placed there, and that’s what `vim` uses to `p`aste, when no explicit register is given. A simple `p` is the same thing as doing `""p`.

Never lose a yanked text again

It has happened to all of us. We yank some text, than delete some other, and when we try to paste the yanked text, it's not there anymore, `vim` replaced it with the text that you deleted, then you need to go there and yanked that text again.
Well, as I said, `vim` will always replace the unnamed register, but of course we didn't lose the yanked text, `vim` would not have survived that long if it was that dumb, right?

`vim` automatically populates what is called the **numbered registers** for us. As expected, these are registers from `"0` to `"9`.
`"0` will always have the content of the latest yank, and the others will have last 9 deleted text, being `"1` the newest, and `"9` the oldest. So if you yanked some text, you can always refer to it using `"0p`.

The read only registers

There are 4 read only registers: `".`, `"%`, `":` and `"#`
The last inserted text is stored on `".`, and it's quite handy if you need to write the same text twice, in different places, not needing to yank and paste.

`"%` has the current file path, starting from the directory where `vim` was first opened. What I usually use it for is to copy the current file to the clipboard, so I can use it externally (running a script in another terminal, for instance). You could execute `:let @+=@%` to do that. `let` is used to write to a register, and `"+` is the clipboard register, so we are copying the current file path to the clipboard.

`":` is the most recently executed command. If you save the current buffer with `:w`, "w" will be in this register. A good way to use it is with `@:`, to execute this command again. For example, if you execute a substitute command in one line, like in `:s/foo/bar`, you can just to go another line and execute `@:` to run this substitution again.

`"#` is the name of the alternate file, that you can think of it as the last edited file (it's a bit more complex than that, go to `:h alternate-file` if you want to understand it better). It's what `vim` uses to switch between files when you use `Ctrl-^`, and you could do the same thing with `:e Ctrl-r #`. I rarely use this, but hopefully you are more creative than I am.

The expression and the search registers

The expression register (`"=`) is used to deal with results of expressions. This is easier to understand with an example. If, in insert mode, you type `Ctrl-r =`, you will see a "=" sign in the command line. Then if you type `2+2 <enter>`, `4` will be printed. This can be used to execute all sort of expressions, even calling external commands. To give another example, if you type `Ctrl-r =` and then, in the command line, `system('ls') <enter>`, the output of the `ls` command will be pasted in your buffer.

The search register, as you may have imagined, is where the latest text that you searched with `/`, `?`, `*` or `#` is. If, for example, you just searched for `/Nietzsche`, and now you want to replace it with something else, there is no way you are going to type "Nietzsche" again, just do `:%s/<Ctrl-r />/mustache/g` and you are good to go.

Macros

You may already be familiar with `vim`'s macros. It's a way to record a set of actions that can be executed multiple times (`:h recording` if you need more information). What you probably didn't know is that `vim` uses a register to store these actions, so if you use `qw` to record a macro, the register `"w` will have all the things that you did, it's all just plain text.

The cool thing about this is that, as it is just a normal register, you can manipulate it as you want. How many times have you forgotten that step in the middle of a macro recording and had to do it all over again? Well, fixing that is as simple as editing a register.

For example, if you forgot to add a semicolon in the end of that `w` macro, just do something like `:let @W='i;'`. Noticed the upcased `W`? That's just how we append a value to a register, using its upcased name, so here we are just appending the command `i;` to the register, to enter insert mode (`i`) and add a semicolon. If you need to edit something in the middle of the register, just do `:let @w='<Ctrl-r w>`, change what you want, and close the quotes in the end. Done, no more recording a macro 10 times before you get it right.

Another cool thing about this is that, as it's just plain text in a register, you can easily move macros around, applying it in other `vim` instance, or sharing it with someone else. Think about it, if you have that register in your clipboard, you can just execute it with `@+` (`"+` is the clipboard register). Try it, just write "ivim is awesome" anywhere, then copy it to your clipboard, and execute `@+` in a `vim` buffer. How cool is that?

Wrapping up

Understanding how registers work is quite simple, and although you are not going to use them every 5 minutes, it certainly will avoid some annoyances, like losing a yanked text, of having to record a macro again. I covered the things that I use the most, but there is more. If you are curious about what a small delete or a black hole register is, you should definitely read the short and easy to follow documentation in `:h registers`. And if you want to learn more about `vim` in general, the book Practical Vim is a great resource.

Interested in learning Kubernetes?

I just published a new book called Kubernetes in Practice, you can use the discount code **blog** to get 10% off.

Get fresh articles in your inbox

If you liked this article, you might want to subscribe. If you don't like what you get, unsubscribe with one click.