
HACKXCRACK

APRENDE VIM

“Enfocado a las
expresiones regulares”



BY: k133

hack  crack
WWW.HACKXCRACK.ES



Este obra está bajo una [licencia de Creative Commons Reconocimiento-NoComercial 3.0 España](#).

Índice

1. Dedicatoria	1
2. Introducción	2
3. ¿ Que es VIM ?	3
4. Instalación y características	4
5. Entorno del editor	5
6. Práctica con los modos principales	7
7. Tareas básicas para tratamiento de texto	10
8. Campos de actuación	12
9. Movimiento por el archivo	15
10. Movimiento por la línea	16
11. Búsquedas	17
12. Rangos	18
13. Marcas personalizadas	23
14. Comandos que pasan a modo insertar	24
15. Porta-papeles o buffer	25
16. Expresiones regulares	26
17. Editando el archivo de configuración	35
18. Expresiones regulares 2	37
19. Pestañas o Tabs	42
20. Sangrado de líneas	43
21. Otras características	45
21.1. Explorador de archivos	45
21.2. Autocompletar palabras y rutas	45
21.3. Abrir archivos desde el editor y manejos de buffers	47

21.4. Ejecución de comandos de sistema.	50
21.5. Corrección ortográfica	53
21.6. Guardar sesiones enteras	56
21.7. Creando nuestros propios Snippets (mapeos)	56
21.8. Gestión de pliegues	59
22. Conclusión y opinión personal	62

1. Dedicatoria

Quiero agradecer primero a toda la comunidad de www.hackxcrack.es por seguir manteniendo entre todos el foro, y particularmente a la pequeña familia del IRC ;). No me gusta dar nombres, pero quiero agradecer a “ksha”, gran persona y del que he aprendido muchísimo, y a “Neutron”, un buen amigo que seguro llegará lejos ;).

k133

2. Introducción

En este documento vamos a aprender a manejar VIM. Si, ese editor de texto tan “difícil” que sólo nos intenta complicar la existencia ;). La mayoría de la gente que no lo ha usado piensa que es complicado, como ya he dicho, pero ¿que entendemos por complicado ?. Por poner un ejemplo, si usted alguna vez ha jugado a algún juego de ordenador, piense en cuantas teclas debe memorizar, para controlar el juego. Teclas que ni siquiera tienen por qué cumplir un patrón establecido.

Pues bueno, VIM es similar, la mayoría de las teclas tienen su función (incluso puedes ir tirando el ratón a la basura por que no lo necesitarás ;)). No vamos a ver todas ni mucho menos, pero ayuda que su función derive de su nombre en inglés. En este documento vas a aprender lo necesario para hacer de VIM tu editor favorito (y dejar de usar nano ;)).

La estructura de este documento la dividiré en fases de aprendizaje, ya que al igual que todas las cosas, hay que ir aprendiéndolas poco a poco. Por lo tanto habrá una primera fase donde explicaré los comandos necesarios para poder editar cualquier archivo, y a partir de ahí ya iremos añadiendo cosas.

El documento estará enfocado a las expresiones regulares, esto quiere decir que le dedicó más espacio que a otras partes, al ser a mi parecer más importante. A pesar del dicho: "Si intentas resolver un problema con expresiones regulares, pasas a tener dos problemas", aplicarlas en VIM es eficiencia pura ;).

El entorno que usaré, donde instalaremos la herramienta y editaremos los archivos de configuración, será un Debian, ya que los sistemas más usados actualmente están basados en éste. Si usas otra distribución busca el comando apropiado para instalarlo o compilarlo, y localiza la ruta de los archivos de configuración. Una vez dentro del editor, el entorno es igual para todos.

¿Por que deberías aprender a usarlo?, VIM es un editor que consume un mínimo de recursos, y puedes ejecutarlo con tan solo la necesidad de una terminal, en sistemas con o sin interfaz gráfica, shell remota, VPS, etc. Es una de las herramientas que todo administrador debe saber usar, porque optimizaras tiempo y esfuerzo. De igual forma puede ser usado como IDE para programar, y lo mejor de todo, puedes personalizarlo al 100 %.

Si el editor de texto es una herramienta importante para ti, y eres de los que se preocupan por optimizar su trabajo (y si cumpliendo lo anterior aún no usas VIM), continua leyendo ;).

3. ¿ Que es VIM ?

Espero que con la introducción os hayáis animado. Vim (VI IMproved), en pocas palabras, es una mejora del ya antiguo editor de texto VI. Mejora en el sentido de que además de hacer todo lo que hace VI, es mas “user friendly”, y han añadido montones de cosas nuevas, por no hablar de la gran cantidad de plugins (que no veremos) que podemos añadir.

VIM esta escrito en C, el desarrollador es Bram Moolenaar, y tiene licencia Charityware (compatible con GPL). El editor es compatible con la mayoría de los sistemas (MacOS, OS/2, RiscOS, UNIX, DOS, Windows95..NT, *BSD, Linux, etc). También tiene su interfaz gráfica, que no vamos a usar.

VIM es de código abierto, y podréis encontrar instalado por defecto (en la mayoría de sistemas) al antiguo VI. Sabiendo usar VIM, también sabremos manejarnos en VI, así que no creáis que hay mucha diferencia entre ambos.



4. Instalación y características

Para instalar VIM basta con ejecutar:

```
# apt-get install vim
```

La versión que hay actualmente en Debian Squeeze es la 7.2.x, si lo instaláis en sistemas más actualizados como Ubuntu o la Wheezy de Debian, seguramente os venga ya con la versión 7.3.x .

VIM soporta la codificación de caracteres UTF-8, con lo cual podremos poner todo tipo de caracteres, además detecta el tipo que tiene el archivo (Dos, Unix, etc). Incluso podemos cambiar la codificación desde el propio editor.

Si el archivo que estamos editando esta comprimido (por ejemplo un .tar), también lo detectará y nos lo descomprime para poder editarlo.

Los comandos que vayamos ejecutando, nos los va guardando en un historial, al igual que en una terminal. Podremos movernos con las flechas entre los comandos y si escribimos los primeros caracteres del comando nos buscará en el historial las coincidencias.

Podemos realizar búsquedas de forma sencilla, incluso realizar una acción para cada aparición de la búsqueda.

Una cosa que hay que tener clara sobre VIM, para saber como trabaja, es que cuando nosotros nos disponemos a editar un archivo, hasta que no ejecutamos el comando de guardar, todos los cambios que estamos haciendo los esta guardando en un buffer (.swp). Por eso cuando la máquina se nos apaga inesperadamente el buffer queda ahí, y es posible recuperarlo si ejecutamos (estando en la misma ruta):

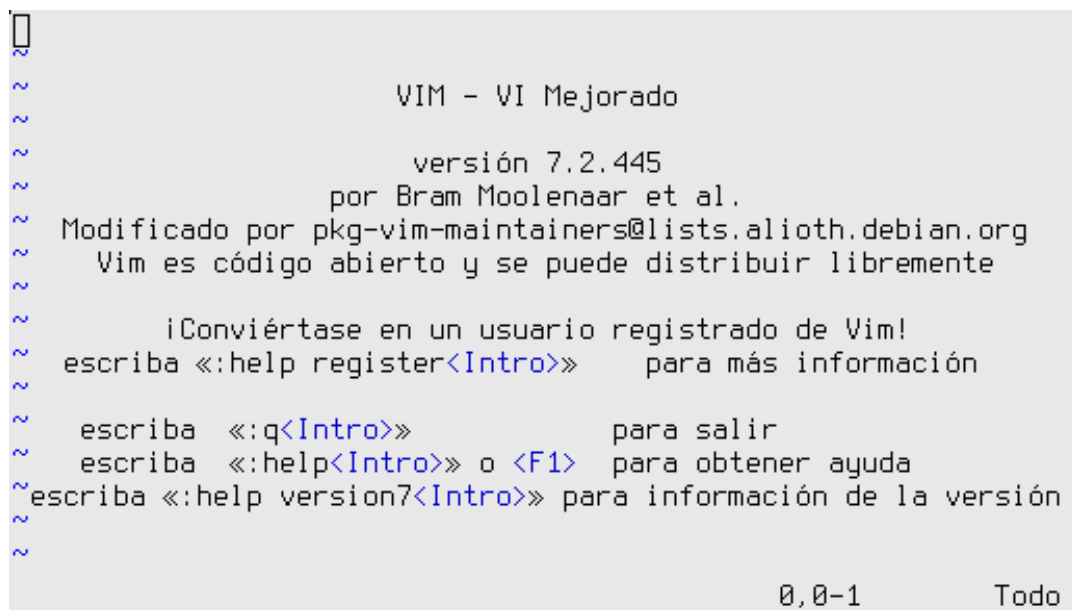
```
$ vim -r <FILE>
```

VIM viene con una opción para el resalto de sintaxis (después vemos como habilitarlo), y dispone de multitud de archivos de sintaxis para cada lenguaje (ver /usr/share/vim/vim72/syntax/).

También tiene soporte para el ratón, pero no lo habilitaremos, no pinta mucho usar el ratón en VIM :p.

Y nos podemos alargar durante todo el documento con las cosas que se pueden hacer en VIM pero ya lo iremos viendo.

5. Entorno del editor



```

VIM - VI Mejorado

    versión 7.2.445
    por Bram Moolenaar et al.
Modificado por pkg-vim-maintainers@lists.alioth.debian.org
Vim es código abierto y se puede distribuir libremente

    ¡Conviértase en un usuario registrado de Vim!
    escriba «:help register<Intro>»    para más información

    escriba «:q<Intro>»                para salir
    escriba «:help<Intro>» o <F1>      para obtener ayuda
    escriba «:help version7<Intro>»   para información de la versión

                                0,0-1      Todo
  
```

Figura 1: Entorno del editor.

Cuando ejecutamos VIM sin especificar un archivo, veremos la “intro” como muestra la imagen anterior. Ahí ya vemos los primeros comandos importantes, y con los que nos tenemos que ir quedando (luego los vemos).

Vamos a explicar el entorno, por ahora ¡no toquéis nada! :p. En cada línea no escrita vemos un ~ que marca donde empieza cada línea; al escribir nosotros, desaparecerá. Abajo tenemos una línea reservada para el programa (línea de estado), ahí es donde meteremos los comandos, nos mostrará también los errores en momentos puntuales, etc.

Lo que siempre va estar son los campos que vemos actualmente. Abajo en la esquina derecha nos muestra en que parte del archivo estamos. Puede poner:

- Todo** Si estamos visualizando todo el archivo en la pantalla.
- Comienzo** Si estamos viendo el principio del archivo.
- Final** Si estamos viendo la parte final el archivo.
- XX %** La posición respecto al total (100 %) del archivo en porcentaje.

Y al lado, un poco más hacia la izquierda esta el campo que muestra la posición del cursor. Primero muestra el número de línea en la que estamos, y separado por una coma, el número de la columna. Cuando empecemos a escribir, fijaros en como va avanzando la numeración. También os daréis cuenta de que a veces muestra un solo número de columna y otras veces

separa dos números con un guión. Eso se debe a que si introducimos un carácter fuera del ASCII (tildes por ejemplo), gracias a la codificación vemos el carácter correctamente, pero VIM lo interpreta como doble carácter, así que primero pone la columna según la interpreta, y luego, después del guión según la vemos nosotros.

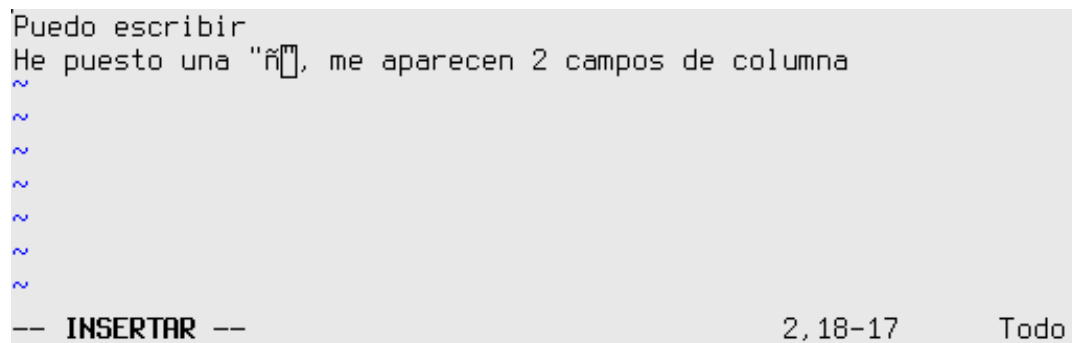
Nota

*Puedes obtener ayuda en cualquier momento con el comando **:help** acompañado de la palabra clave que desees buscar. Se abrirá una ventana partida que puedes cerrar con el comando **:close** .*

Con esto creo que ya sabemos las partes del entorno, así que “vamos a por él” ;).

6. Práctica con los modos principales

Ok, vamos a comenzar la práctica. Recomendando que os hagáis una pequeña chuleta con las teclas que vamos a comentar y su función. Por defecto el editor esta en **modo comando**, para comenzar a escribir se lo indicamos al editor pulsando la tecla **i**. En la línea de estado nos señala que estamos en el **modo insertar** texto.



```
Puedo escribir
He puesto una "ñ", me aparecen 2 campos de columna
~
~
~
~
~
~
~
-- INSERTAR --                2, 18-17      Todo
```

Figura 2: Modo insertar.

Para volver al modo comando basta con pulsar la tecla **<ESC>** (Escape). Es sencillo ¿no?. Una vez que estamos en modo insertar podemos movernos como en cualquier otro editor, con las flechas, y escribir. Para aclarar un poco más lo que para VIM es un “comando”, puede ser simplemente la pulsación de una tecla, por ejemplo, la propia **i** que pulsamos para entrar a modo insertar es un comando.

Nota

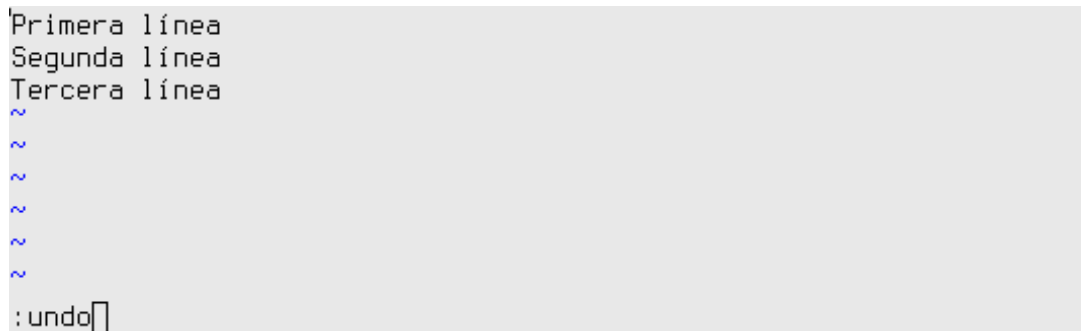
*También podemos movernos por el texto (estando en modo comando), con las teclas **h** (izquierda), **j** (abajo), **k** (arriba), **l** (derecha).*

Una cosa importante y que en la mayoría de documentos se lo saltan, es como deshacer/rehacer en VIM. A medida que vayáis usando VIM, alguna vez os pasará (y el que diga que no miente), que estando en modo comando nos ponemos a escribir tan alegremente, y claro, la mayoría de las teclas ejecutan una acción. Así que en un momento de descuido hemos podido borrar palabras, líneas o vete a saber el qué. Para deshacerlo, simplemente nos aseguramos de que estamos en modo comando y pulsamos la letra **u** tantas veces como sea necesario, he irá deshaciendo cada acción. En la línea de estado irá poniendo información de las acciones que va deshaciendo/rehaciendo.

En la línea de estado también podemos escribir comandos, pero par ello antes tenemos que poner el carácter **:** (estando en modo comando obviamente). Entonces veremos como el cursor

se desplaza hasta esta línea. Así el comando **u** visto anteriormente para deshacer, equivale a poner **:undo**. Para rehacer usamos el comando **:redo** (o la abreviatura **:red**).

Al poner estos comandos, podemos ayudarnos del tabulador que irá autocompletando con los comandos posibles, al igual que ocurre en la terminal del sistema.



```
Primera línea
Segunda línea
Tercera línea
~
~
~
~
~
~
~
:undo
```

Figura 3: Deshacer/rehacer

Nota

*Al rehacer, para no estar escribiendo continuamente **:redo**, tras ejecutar el comando la primera vez, podemos seguir rehaciendo pulsando **<CTRL+R>**.*

Y ha estas alturas, lo único que nos queda para poder decir que podemos “editar” cualquier archivo, es saber como guardarlo (o no) y como salir del editor. Guardar un archivo no es más que escribir el buffer en la unidad sólida, esto lo hacemos con **:write** (o la abreviatura **:w**). Para salir usaremos **:quit** (o la abreviatura **:q**). Si por el contrario queremos salir sin guardar ningún cambio usamos el comodín **!** (forzar) junto al comando. Quedaría así **:q!**, cierre forzoso.

Nota

*Podemos combinar los comandos guardar y salir, nos quedaría un solo comando **:wq**.*

Si hemos iniciado VIM sin especificar algún archivo, debemos poner el nombre al guardarlo por primera vez.

```
Primera línea
Segunda línea
Tercera línea
~
~
~
~
~
~
~
~
:w mi_archivo
```

Figura 4: Guardamos el archivo.

```
Primera línea
Segunda línea
Tercera línea
~
~
~
~
~
~
~
~
"mi_archivo" [Nuevo] 3L, 450 escritos      3,14-13      Todo
```

Figura 5: Tras ejecutar “:w mi_archivo”

En la línea de estado nos muestra el nombre el archivo, en este caso también especifica que lo ha creado (Nuevo), el número de líneas del archivo, y los caracteres escritos. Cuenta los finales de línea y como comentábamos los caracteres fuera del ascii los cuenta como dos.

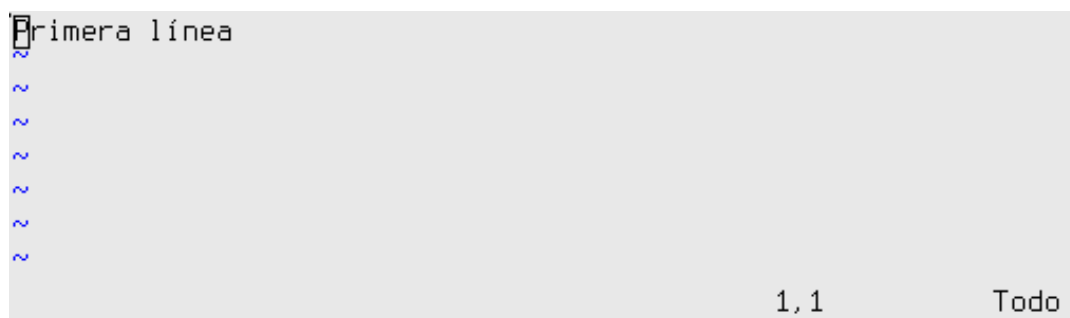


Figura 7: Tras ejecutar “2dd”

- Y ahora pegamos 2 veces el texto cortado, encima de la línea que nos queda:

2P

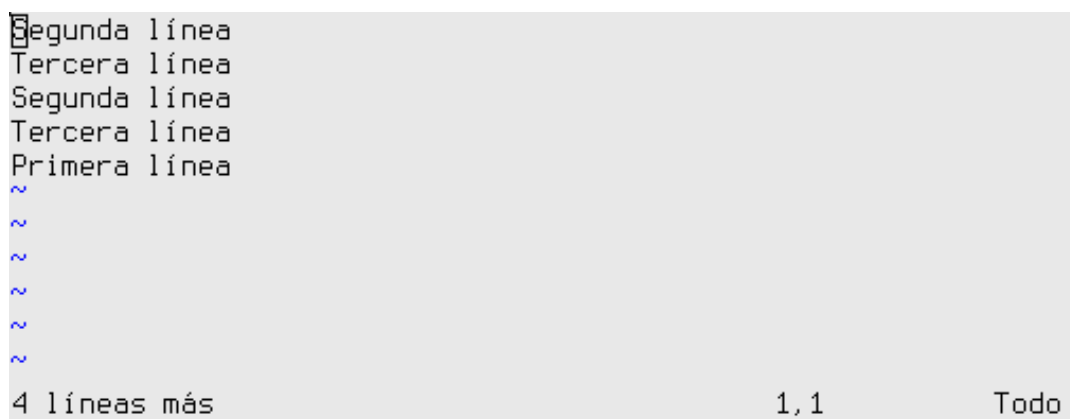


Figura 8: Tras ejecutar “2P”

8. Campos de actuación

Anteriormente hemos visto como copiar/cortar líneas, pero en VIM no tenemos por que limitarnos a eso, ¿y si queremos copiar solo unas palabra, o un párrafo, o todo el documento?. Tras ejecutar los comandos para copiar/cortar (**y**, **d**), VIM se queda esperando que le digamos que es lo que queremos copiar/cortar. Aquí es donde se empieza a complicar un poco, ya que necesitamos entender como trabaja VIM.

Hay varios tipos de “campos” que llevan asignados una letra, y esto es muy importante saberlo.

- Para referirnos a una palabra lo hacemos con **w** (word). VIM entiende por palabra a un conjunto de caracteres alfanuméricos separados por caracteres no alfanuméricos (’ ’, :, -, etc)

- Si nos referimos a una frase completa usamos **s** (sentence). VIM entiende por frase a un conjunto de caracteres acabados por punto . .

- Cuando nos referimos a un párrafo, con **p** (paragraph). El editor entiende por párrafo al texto delimitado por líneas en blanco.

- Para referirnos al texto encerrado entre paréntesis, con **b**.

- Para el texto encerrado entre llaves lo haremos con **B**.

Y aún queda una cosa más, hemos estado comentando los delimitadores que usa VIM para reconocer cada campo, pues bien también debemos indicar si queremos que incluya o no esos delimitadores. La **i** representa el campo sin incluirlos, y la **a** incluyéndolos.

Una vez que sabemos esto, solo tenemos que ir construyendo el comando, por ejemplo, queremos cortar cinco palabras a partir de la posición del cursor incluyendo delimitadores:

5 El número de palabras.

d Comando para cortar.

a Incluimos delimitador.

w Indicador de palabra.

5daw


```

Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas texto mas texto mas texto.
Mas texto mas texto.

Parrafo 3
Y mas texto 1 mas texto y mas texto.
Y mas texto y mas texto.

```

10, 13 Todo

Figura 9: Indicaciones del comando.

```

Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas texto mas texto mas texto.
Mas texto mas texto.

Parrafo 3
Y mas texto 2 texto.
Y mas texto y mas texto.

```

10, 13 Todo

Figura 10: Tras ejecutar “5daw”

Otro ejemplo, tenemos una función por ejemplo en C, que va entre llaves, queremos eliminar todo el contenido de la función dejando las llaves. Estando dentro de la función pondremos:

- d** Cortar
- i** Sin incluir llaves.
- B** Indicador de llaves.

diB

```
int contarCaracteres(char c[MAX],int i){
    char a;
    while((a=c[i]) != '\0'){
        i++;
    }
    return (i);
}
~
~
~
~
```

2,5 Todo

Figura 11: Indicaciones del comando.

```
int contarCaracteres(char c[MAX],int i){
}
~
~
~
~
~
~
~
~
~
~
```

5 líneas menos 1,40 Todo

Figura 12: Tras ejecutar “diB”

No hace falta que os aprendáis de golpe todos los comandos, tened la chuleta a mano para revisarlos, y a medida que lo uséis se os irá quedando.

w	Palabra		
s	Frase	i	Sin incluir.
p	Párrafo	a	Incluyendo.
b	Paréntesis		
B	Llaves		

9. Movimiento por el archivo

Como hemos quedado en no usar el ratón en VIM, estaréis pensando ¿tendré que estar flechita arriba, flechita abajo todo el rato !?. Obviamente no, en VIM podemos desplazarnos a cualquier sitio del archivo en un instante. Vamos a ver algunos de los comandos de desplazamiento y sería bueno que vayáis probándolos sobre la marcha.

Primero veremos como buscar la línea que queremos dentro del archivo. Cuando queremos ir a la primera línea del archivo usamos el comando **gg**, y para ir a la última **G**. Para ir a una línea concreta primero ponemos el número y luego **G**, quedando **12G** para ir a la línea doce.

Estos comando son de ejecución inmediata (sin el **:**). También podemos desplazarnos a una línea concreta con **:12**, en este ejemplo nos desplazamos a la línea doce, al igual que lo hacíamos con **12G**. Podemos ir al principio del archivo con **:0**, al igual que con **gg** y al final con **:\$** al igual que hacía **G**.

Nota

Para hacer scroll, se usan las teclas <AvPag> y <RePag>.

Principio	gg	:0
Intermedio	12G	:12
Final	G	:\$

10. Movimiento por la línea

Una vez que nos encontramos en la línea que queremos, tenemos mas opciones que la de movernos flechita derecha y flechita izquierda.

Si antes hemos visto que con **:0** y **:\$** nos movíamos entre el principio y el final del archivo, estando dentro de la línea usaremos lo mismo pero sin los **:**. Así si pulsamos **0** vamos al inicio de la línea, y si pulsamos **\$** iremos al final. Al no tener los dos puntos, quiere decir que son comandos de acción inmediata, nada más pulsarlos realizan la acción.

Y para movernos dentro de la línea podemos ir de palabra en palabra. Recordamos que el campo “palabra” esta asignado a la letra **w**, pues bien pulsando **w** avanzaremos hasta la siguiente palabra y pulsando **b** (before) nos movemos a la palabra anterior. Al igual que en otros comandos, podemos especificar el número de palabras que queremos avanzar o retroceder, poniendo un valor numérico antes del comando. Por ejemplo con **5w** avanzamos cinco palabras.

Principio	palabra	Final
0	b w	\$

11. Búsquedas

Otra forma de movernos por el documento es a través de las búsquedas. Como en la mayoría de editores de texto, nos busca coincidencias de la string que hemos especificado. En VIM, para iniciar una búsqueda pulsaremos **/**, a partir de ahí el cursor pasa a la línea de estado (abajo), y podremos escribir la string que queremos buscar. Tras pulsar **<ENTER>** el cursor se posicionará en la siguiente aparición de esa string en el documento. Si queremos hacer una búsqueda hacia atrás debemos iniciarla con **?** en vez de **/**.

Tras realizar la búsqueda podemos seguir moviéndonos a las siguientes coincidencias con **n**, o a las anteriores con **N**. Por ejemplo **/texto** inicia la búsqueda de la palabra “texto” hacia adelante, y **?texto** hacia atrás.

La búsqueda también admite algunos comodines, pero por ahora con quedarnos con esto es suficiente.

Existe otro comando que bien puede ir en esta sección de búsqueda, el comando **f** (find) nos busca la siguiente aparición del carácter que indiquemos a continuación. Por ejemplo si pulsamos **fx** el cursor se posicionará encima del siguiente carácter **x**. O si queremos ir al final de la frase bastaría un **f.** . Para buscar el carácter hacia atrás usaremos la **F**.

Nota

Por defecto la búsqueda en VIM diferencia mayúsculas de minúsculas.

Atrás	Adelante	Anterior	Siguiente
?	/	N	n

12. Rangos

Ahora que ya conocemos un poco más como movernos por el documento, vamos a ver como realizar tareas orientadas a esos movimientos. Pondremos de ejemplo el comando cortar **d**. Recordamos que al pulsarlo se queda esperando a que digamos el que queremos cortar. Anteriormente probamos como decirle que borrarse distinto campos de actuación predefinidos en VIM, tales como una palabra **w** o el contenido entre unas llaves **B**. Pues bien también podemos indicarle un movimiento, de los que hemos visto.

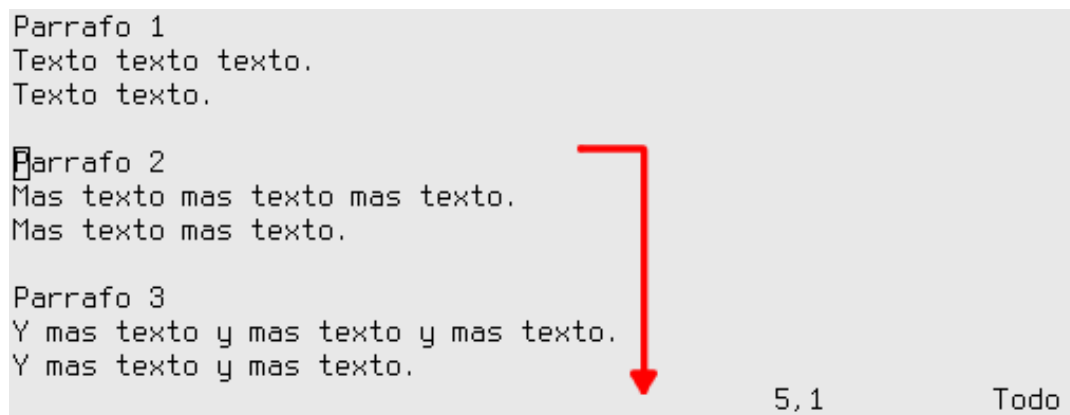
Así, si pulsamos **dG** le estamos diciendo que corte desde la línea en la que estamos hasta el final del documento. Y así podemos indicarle cualquier movimiento de los que vimos, **dgg** cortará desde la línea actual al principio del documento. Imaginaos lo que hace **d12G** ;). Igualmente, podemos usar los movimientos dentro de la línea. El comando **d\$** corta desde la posición actual del cursor, hasta el final de la línea. ¿ Que hará **d0** ? .

Incluso podemos especificar el movimiento con una búsqueda, por ejemplo el comando **d/texto** cortará desde la posición actual del cursor, hasta la próxima aparición de la palabra “texto” (sin incluirla).

Lo mismo podemos hacer con otros comandos como el de copiar **y**. Hay más tipos de movimientos (más específicos), pero no los voy a nombrar porque, a mi parecer, no son eficientes para la mayoría de usuarios. Vamos a ver unos ejemplo:

- Queremos cortar desde la posición actual hasta el final del archivo.

dG



```
Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas texto mas texto mas texto.
Mas texto mas texto.

Parrafo 3
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
```

5,1 Todo

Figura 13: Indicaciones del comando.

```
Parrafo 1
Texto texto texto.
Texto texto.
█
~
~
~
~
~
~
~
7 líneas menos 4,0-1 Todo
```

Figura 14: Tras ejecutar “dG”

- Queremos cortar desde la posición actual hasta el final de la línea:

d\$

```
Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas texto █ mas texto mas texto.
Mas texto mas texto.

Parrafo 3
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
6,11 Comienzo
```

Figura 15: Indicaciones del comando.

```
Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas texto
Mas texto mas texto.

Parrafo 3
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
```

6,10 Comienzo

Figura 16: Tras ejecutar “d\$”

- Aquí queremos cortar desde la posición actual hasta la siguiente aparición de la palabra “Parrafo 3”:

d/Parrafo 3

```
Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas texto mas texto mas texto.
Mas texto mas texto.

Parrafo 3 → No se incluye
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
```

5,1 Todo

Figura 17: Indicaciones del comando.


```
'Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 3
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
~
~
~
~
4 líneas menos                    5,1          Todo
```

Figura 18: Tras ejecutar “d/Parrafo 3”

Estos comandos son de actuación inmediata, menos el que usa la búsqueda que tenemos que pulsar <ENTER>, pero también podemos especificar rangos de actuación algo más complejos si usamos los comandos precedidos por `:`. Como ya hemos visto, al pulsar `:` el cursor pasa a la línea de estado, y recordamos que también vimos algunos movimientos como `:$` para ir al final del archivo.

Debemos añadir otros dos caracteres especiales que nos serán de utilidad, `%` que representa todas las líneas del documento, y `.` que indica la línea actual. Pues bien vamos a ver algunos de los rangos más relevantes en estos ejemplos:

- Para cortar una sola línea:

```
:12 d
```

- Para cortar la línea actual:

```
:. d
```

- Para cortar de principio a fin del documento (varias formas):

```
:0,$ d
```

```
:1,$ d
```

```
: % d
```

- Para cortar desde la línea actual al principio del documento.

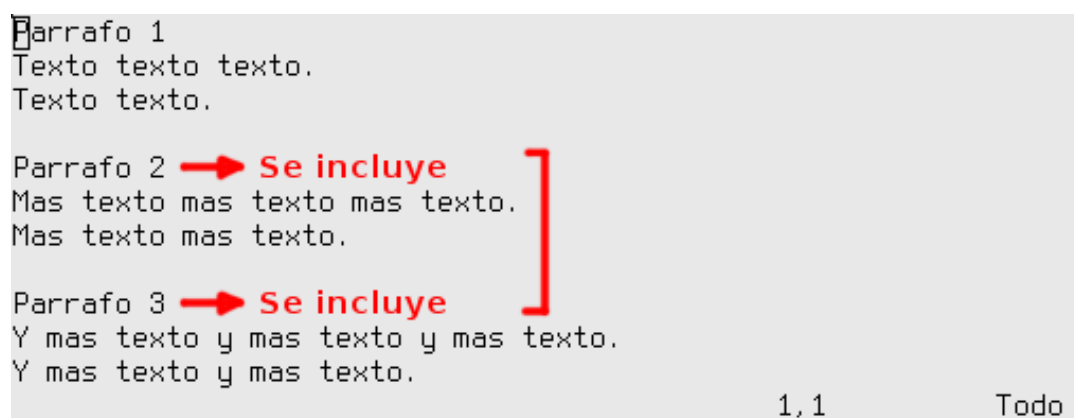
:0, d

- Para cortar desde la línea actual hasta 3 líneas más:

:.+3 d

También podemos especificar búsquedas en este tipo de comandos, por ejemplo vamos a borrar las líneas entre dos búsquedas, en concreto desde la cadena **Parrafo 2**, hasta la cadena **Parrafo 3**:

:/Parrafo 2/,Parrafo 3/ d



```

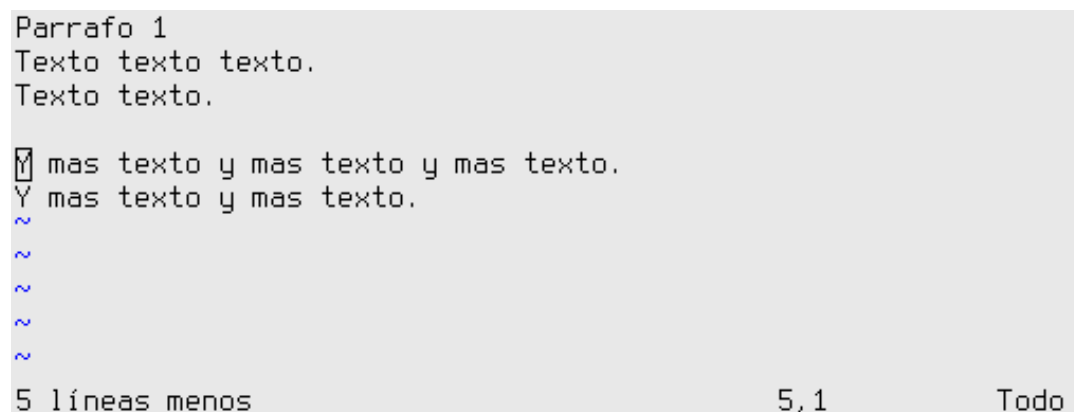
Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2 → Se incluye
Mas texto mas texto mas texto.
Mas texto mas texto.

Parrafo 3 → Se incluye
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
  
```

1, 1 Todo

Figura 19: Indicaciones del comando.



```

Parrafo 1
Texto texto texto.
Texto texto.

[ ] mas texto y mas texto y mas texto.
[ ] mas texto y mas texto.
[ ]
[ ]
[ ]
[ ]
[ ]
  
```

5 líneas menos 5, 1 Todo

Figura 20: Tras ejecutar “:/Parrafo 2/,Parrafo 3/ d”

13. Marcas personalizadas

VIM nos permite la libertad de definir nuestras propias marcas, y situarlas en la posición que queramos. Luego, podremos usar estas marcas para definir rangos como lo hemos hecho anteriormente. Para definir una marca lo haremos con el comando **m**, seguido de una letra del alfabeto (inglés). Entonces... podemos definir tantas marcas como letras tiene el alfabeto... unas 26... NO !! no nos olvidemos de que VIM diferencia mayúsculas de minúsculas ;).

Es más, las marcas con mayúsculas tienen un significado especial, y es que se consideran globales. En el caso en que estemos editando varios archivos, si en el primer archivo hemos definido la marca **mA**, cualquier movimiento que hagamos a esa marca desde los otros archivos, nos moverá al archivo donde la definimos. Por lo tanto solo podrá haber una marca por letra en mayúsculas, y en minúsculas tendremos marcas individuales por cada archivo.

Para referirnos a una marca lo podemos hacer con un apóstrofe ' o con un acento grave ` . Con el apóstrofe nos estaremos refiriendo al inicio de la línea donde esta definida la marca, y con el acento grave iremos justo a la posición dentro de la línea. Para estos movimientos podemos usar el comando **g** que vimos anteriormente, así con **g'a** nos moveremos a la línea de la marca. Con el acento grave no se suele usar ya que es más engorroso (hay que pulsar dos veces para ponerlo).

Podemos incluir estas marcas en los rangos, por ejemplo:

- Para cortar desde la línea con marca **a**, hasta la actual:

```
: 'a,. d
```

- Para cortar desde la marca **a** a la **b**.

```
: 'a,'b d
```

Marcas individuales	Marcas globales
'a, ..., 'z	'A, ..., 'Z

14. Comandos que pasan a modo insertar

Para hacer un pequeño receso, vamos a ver algún comando interesante, que nos permite realizar una acción o movimiento y pasar a modo inserción. Estos comando son muy útiles y seguro les vais a dar bastante uso (o eso espero ;)).

El comando **o** nos abre una línea en blanco debajo de la actual, y **O** por encima. Podemos ejecutar el comando estando en cualquier parte de la línea, además nos pasa a modo insertar. Podemos usar esto por ejemplo para empezar un nuevo párrafo de forma rápida, sin importar donde esta situado el cursor dentro de la línea.

Otro comando bastante útil es **A**, que nos lleva el cursor al final de la línea actual y pasa a insertar. Este siempre lo suelo utilizar después de que he tenido que retroceder en la línea para corregir alguna palabra. Con una sola pulsación, tenemos el cursor al final de la línea, preparado para seguir escribiendo. De igual manera, el comando **I** nos posiciona el cursor al principio de la línea.

El comando **s** corta el carácter de debajo del cursor y pasa a insertar. Por ejemplo, si nos hemos dejado de poner una tilde, en un solo paso cortamos la vocal y pasamos a modo insertar para escribir la corrección.

La mayúscula **S** corta toda la línea actual y nos deja el cursor en modo insertar. Por ejemplo, si ha mitad de línea nos damos cuenta de que hemos metido la pata en algo, cortamos todo y ha empezar de nuevo. Ahora me viene a la cabeza algo que seguro que os ha pasado alguna vez, que de tanto apretar la tecla “backspace” nos hemos pasado poco/bastante de lo que teníamos pensado borrar. Con VIM esto ya no será un problema ;).

Y el comando **C** que nos corta desde la posición actual del cursor hasta el final de la línea, y nos pasa a insertar.

Estos son los comandos básicos, los que creo son más útiles, hay muchos más (más específicos), incluso una misma función la pueden realizar 2 comandos distintos. Por ejemplo el comando **cc** equivale al comando **S**, pero es mejor aprender bien unos pocos, y saberlos aplicar a la hora de la verdad ;).

15. Porta-papeles o buffer

Otro arma potente que tiene VIM es el porta-papeles, o mejor dicho, los porta-papeles. Al igual que sucede con las marcas, disponemos de tantos porta-papeles (que en realidad son buffers), como letras del abecedario inglés, y también habrá diferencia entre mayúsculas y minúsculas.

Primero vamos a ver como definirlo. Nos referiremos al buffer como **“x**, y irá situado antes de colocar el comando. Por ejemplo el comando **“a5dd**, guarda en el buffer **a** las cinco líneas que cortamos. Si después queremos recuperar el contenido, por ejemplo pegarlo, bastaría hacer un **“ap**.

Las mayúsculas también tiene un significado especial, no representan buffers propios. Cuando especificamos un buffer como puede ser **“A**, lo que hará, es añadir el contenido que asignemos, a la letra en minúsculas. Así siguiendo el ejemplo anterior, donde metimos cinco líneas en el buffer **a**, tras ejecutar el comando **“A3dd**, no se sobrescribirá el buffer, sino que se añaden esas líneas. El buffer **a** pasa a contener ocho líneas.

Para entenderlo mejor, si nos manejamos algo en la terminal, equivaldría a hacer una redirección con **> (a)** o con **>> (A)** . También hay una serie de buffers especiales, que no veremos aquí, con saber manejar estos nos sobra.

Buffers (principales)	Buffers (agregan)
“a, ..., “z	“A, ..., “Z

16. Expresiones regulares

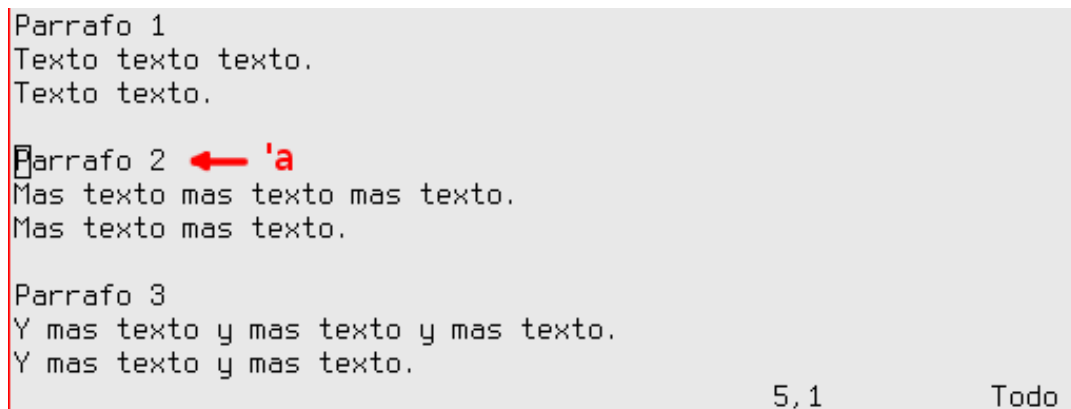
Y ahora volviendo al tema de operaciones con rangos, hay un comando que se lleva el “premio gordo”, el comando sustituir, de forma abreviada **:s**. Lo que tiene de especial es que admite el uso de expresiones regulares, muy similar al comando “sed” en la terminal.

Como hemos visto anteriormente, primero debemos especificar el rango donde actuará el comando, luego ira la estructura del comando, vemos un ejemplo:

- Queremos sustituir la palabra “texto”, por su cadena espejo “otxet”, dentro del rango comprendido entre la marca **a** y el final del documento.

Colocaremos la marca en la línea actual del cursor que vemos en la imagen:

ma



```
Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2 ← 'a
Mas texto mas texto mas texto.
Mas texto mas texto.

Parrafo 3
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
```

5, 1 Todo

Figura 21: Indicaciones del comando.

Y ahora ejecutamos el comando:

:’a,\$ s/texto/otxet/

```

Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas otxet mas texto mas texto.
Mas otxet mas texto.

Parrafo 3
Y mas otxet y mas texto y mas texto.
[ mas otxet y mas texto.
4 sustituciones en 4 líneas

```

Rango
'a,\$

11,1 Todo

Figura 22: Tras ejecutar el comando.

Explicaremos la estructura del comando y lo que ha pasado. El comando `sustituir` consta de dos campos principales; el primero, es el campo que contendrá la cadena origen, que admitirá **regex**. Es la cadena que se buscará, y que se sustituirá por la cadena contenida en el segundo campo. Nótese que los campos usan un separador, en este caso el carácter `/`. Pero no tiene que ser siempre ese carácter, puede ser cualquier signo de puntuación o carácter especial (luego vemos un ejemplo).

Si os habéis fijado solo se ha sustituido la primera coincidencia de cada línea, esto ocurre así por defecto. Para que se sustituyan todas las apariciones de la cadena origen, debemos añadir el modificador **g** al final (luego vemos otros tipos de modificadores), veamos como quedaría.

```
:a,$ s/texto/otxet/g
```

```

Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
Mas otxet mas otxet mas otxet.
Mas otxet mas otxet.

Parrafo 3
Y mas otxet y mas otxet y mas otxet.
[ mas otxet y mas otxet.
10 sustituciones en 4 líneas

```

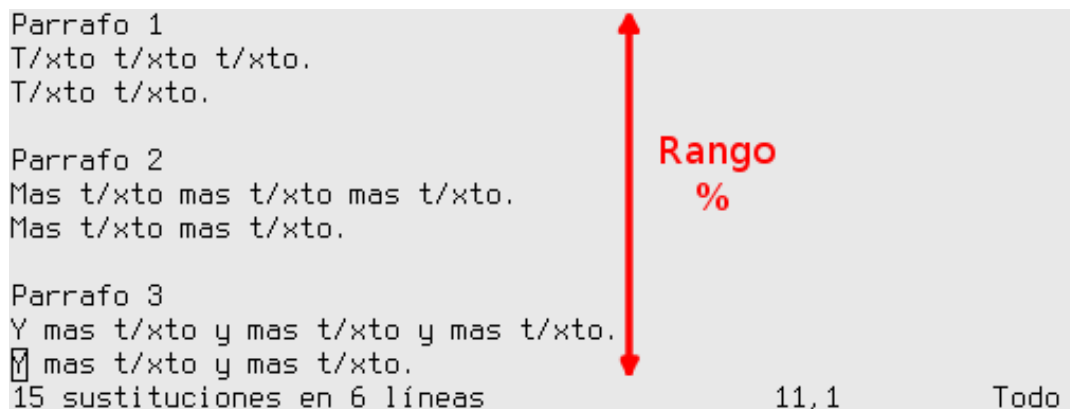
11,1 Todo

Figura 23: Tras ejecutar el comando.

- Queremos sustituir todas las vocales **e** del documento por el carácter `/`.

En este caso el carácter de sustitución será /, con lo cual si usamos como separador el mismo carácter, nos dará error. Recurriremos entonces a otro carácter, como decíamos anteriormente:

```
: %s+e+/+g
```



```

Parrafo 1
T/xto t/xto t/xto.
T/xto t/xto.

Parrafo 2
Mas t/xto mas t/xto mas t/xto.
Mas t/xto mas t/xto.

Parrafo 3
Y mas t/xto y mas t/xto y mas t/xto.
Y mas t/xto y mas t/xto.
15 sustituciones en 6 líneas
11,1
Todo

```

Figura 24: Tras ejecutar el comando.

Como rango hemos, puesto % que recordamos se refiere a todo el documento, y al igual que antes, añadimos la **g** al final para sustituir todas las apariciones en cada línea. Pero no estamos obligados a cambiar los caracteres de separación de campos, podemos anular el significado especial del carácter, precediéndolo de una contra barra \. El comando anterior lo podremos poner también de esta manera:

```
: %s/e\//g
```

Antes de continuar vamos a ver lo tipos de comandos o modificadores más usados, que podemos añadir al final del comando “sustituir”.

- c** Nos pregunta antes de cada sustitución, y nos dará las siguientes opciones:
 - y Reemplazarlo.
 - n Pasar a la siguiente coincidencia.
 - a Reemplazar todo (no nos vuelve a preguntar en las coincidencias restantes).
 - CTRL+E Ver mas texto, hacia adelante.
 - CTRL+Y Ver mas texto, hacia atrás.
- g** Que sustituya todas las ocurrencias de cada línea.
- i** Que no sea sensible a mayúsculas.

Vamos a poner un ejemplo del uso de estos modificadores.

- Primero vamos a sustituir la primera aparición de la vocal **e** por la **E** mayúscula. Vamos a poner también que nos pida confirmación.

: %s/e/E/c

```
Parrafo 1
Texte texto texto.
Texte texto.

Parrafo 2
Mas texto mas texto mas texto.
Mas texto mas texto.

Parrafo 3
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
¿Reemplazar con E (y/n/a/q/l/^E/^Y)?
```

Nos pregunta
que hacer

2,2 Todo

Figura 25: Tras ejecutar el comando.

Nos pide confirmación, podemos ir dando a la **y** para ir una por una, aunque ya que no tenemos inconveniente en sustituir todas, podemos pulsar la **a**.

```
Parrafo 1
TExto texto texto.
TExto texto.

Parrafo 2
Mas tExto mas texto mas texto.
Mas tExto mas texto.

Parrafo 3
Y mas tExto y mas texto y mas texto.
y mas tExto y mas texto.
6 sustituciones en 6 líneas
```

11,1 Todo

Figura 26: Tras acabar de ejecutarlo.

- Ahora sustituiremos tanto la **e** como la **E**, por el número **5**, usando para ello el modificador para ignorar diferencias entre mayúsculas/minúsculas.

: %s/e/5/gi

```

Parrafo 1
T5xto t5xto t5xto.
T5xto t5xto.

Parrafo 2
Mas t5xto mas t5xto mas t5xto.
Mas t5xto mas t5xto.

Parrafo 3
Y mas t5xto y mas t5xto y mas t5xto.
[ ] mas t5xto y mas t5xto.
15 sustituciones en 6 líneas                11,1                Todo

```

Figura 27: Tras ejecutar el comando.

Que no se nos olvide añadir el **g** !!!

Pero esta funcionalidad sería un poco “pobre” si no hiciéramos uso de las expresiones regulares. Estas expresiones nos permitirán seleccionar lo que queremos buscar de forma bastante precisa. Y es muy recomendable saber usarlo, ya que nos va a ahorrar mucho tiempo, cuando necesitemos modificar un archivo, o extraer ciertas partes del mismo.

Primero vamos a definir como llamaremos a los campos del comando `sustituir`. Aunque tienen ya un nombre establecido, para hacerlo simple lo llamaremos:

s/campo1/campo2/

A continuación vamos a ver una serie de caracteres y expresiones especiales con significado. La mayoría de ellos son comunes para los diferentes tipos de estándares sobre expresiones regulares, aunque en algún lenguaje pueden variar.

\n	Salto de línea (en campo1).
\t	Tabulador.
\s	Espacio.
_s	Salto de línea o espacio.
\S	No espacio.
\d	Dígito.
\D	No dígito.
\u	Mayúsculas (en campo1). Si esta en campo2 pasa a mayúscula el carácter que le sigue.
\U	No mayúsculas (en campo1). Si esta en campo2 pasa a mayúscula los caracteres que le siguen.
\l	Minúsculas (en campo1). Si esta en campo2 pasa a minúscula el carácter que le sigue.
\L	No minúsculas (en campo1). Si esta en campo2 pasa a minúscula los caracteres que le siguen.
\E	En el campo2 marca cuando acaban el \U y el \L.
\r	Introduce un salto de línea (en campo2).
^	Principio de línea.
\$	Final de línea.
.	Cualquier carácter, menos el salto de línea.
*	Que lo anterior aparezca 0 o más veces.
\+	Que lo anterior aparezca 1 o más veces.
\=	Que lo anterior aparezca 0 o 1 vez.
\{n,m\}	Que lo anterior aparezca de n a m veces.
\{n\}	Que lo anterior aparezca n veces.
\{,m\}	Que lo anterior aparezca como máximo m veces.
\{n,\}	Que lo anterior aparezca como mínimo n veces.
\<	Hace referencia al principio de cualquier palabra.
\>	Hace referencia al final de cualquier palabra.
\ 	Sirve como “or”
\(\)	Creación de grupos
\1	Contenido del grupo referenciado por su posición, en este caso será el primer grupo.
&	Contenido del patrón especificado en el campo1, deberemos ponerlo en el campo2.
[ab]	Posibles caracteres que puede adquirir un único carácter. Los caracteres especiales pierden su significado.
[^ab]	Posibles caracteres que NO puede adquirir un único carácter.

Esto como se aprende es a base de practicar, así que vamos a ver unos ejemplos.

- Vamos a coger todas las palabras de 3 caracteres y las vamos a poner en mayúscula.

```
: %s\<...\>\U&/g
```

```
Parrafo 1
Texto texto texto.
Texto texto.

Parrafo 2
MAS texto MAS texto MAS texto.
MAS texto MAS texto.

Parrafo 3
Y MAS texto y MAS texto y MAS texto.
☐ MAS texto y MAS texto.
10 sustituciones en 4 líneas          11,1          Todo
```

Figura 28: Tras ejecutar el comando.

En el campo1 hemos puesto que busque los inicios de palabra (\<), que le sigan tres caracteres cualquiera (...), y que seguidamente este el final de la palabra (\>). Y en el campo2 irá la sustitución, en este caso pasará a mayúsculas (\U), el propio contenido marcado en el campo1, que serán las palabras con tres caracteres que encuentre.

- Vamos a sustituir las líneas totalmente vacías por unos cuantos guiones a modo de separador.

```
: %s/^$/---/
```

```
Parrafo 1
Texto texto texto.
Texto texto.
---
Parrafo 2
Mas texto mas texto mas texto.
Mas texto mas texto.
☐--
Parrafo 3
Y mas texto y mas texto y mas texto.
Y mas texto y mas texto.
: %s/^$/---/          8,1          Todo
```

Figura 29: Tras ejecutar el comando.

En el campo1 ponemos que la línea empiece (^) y seguidamente que este ya el final (\$). Así cogerás las líneas que no tengan nada. El campo2 no tiene ningún misterio. Podríamos haber dejado la opción de que la línea pueda contener 0 o más espacios en blanco (\s*).

- Vamos a poner en mayúsculas la primera y la última palabra de cada línea:

```
: %s/^\([^ ]+\)(.*)\s\([^ ]+\.)$ \U\1\E\2\U\3/
```

```
Parrafo 1
TEXT0 texto TEXT0.
TEXT0 TEXT0.

Parrafo 2
MAS texto mas texto mas TEXT0.
MAS texto mas TEXT0.

Parrafo 3
Y mas texto y mas texto y mas TEXT0.
Y mas texto y mas TEXT0.
6 sustituciones en 6 líneas          11,1          Todo
```

Figura 30: Tras ejecutar el comando.

Vamos por partes, en el campo1 buscamos el inicio de la línea (^), y creamos tres grupos, dos de ellos contendrán las palabras que queremos poner en mayúscula y el otro contendrá todo el texto restante. El primer grupo, delimitado por los paréntesis \(\) debe estar en el inicio de la línea, y cogerá cualquier carácter que no sea un espacio ([^]) y indicamos que este debe aparecer 1 o más veces (\+). Dentro del corchete hemos puesto un espacio real, no podemos usar el carácter especial \s por que dentro no lo interpreta como tal. Cuando llegue a un espacio (terminación de una palabra), cierra el primer grupo y sigue ejecutando la regex.

En el segundo grupo cogeremos cualquier carácter (.) y decimos que pueda aparecer 0 o más veces (*). Por lo tanto cogerá el espacio con el que se cerró el primer grupo e irá cogiendo caracteres hasta que encuentre la "señal de stop". Entonces analizará el resto de la regex.

A continuación le indicamos que deberá parar en un espacio (\s) pero para que sepa en cual de todos, necesita mirar el resto de la regex. Y en el tercer grupo al igual que en el primero, le indicamos que coja caracteres que no sean espacio ([^]), que sean 1 o más (\+), y que finalice en punto (\.). Con el carácter de final de línea (\$) a continuación, el tercer grupo seleccionará justo la última palabra de la línea, y el espacio donde parará el segundo grupo será el anterior a la última palabra. El punto final lo podemos sacar del grupo, pero bueno no habrá problema al pasar a mayúsculas.

En el campo2 ponemos que queremos pasar a mayúsculas (\U) todos los caracteres del primer grupo (\1), luego parará de pasar a mayúsculas (\E). Ponemos el segundo grupo con el texto central según esta (\2), añadiendo el espacio que usamos para marcar la regex en el campo1. Y pasamos a mayúsculas (\U) el tercer grupo (\3).

Como veis, no es más que ir mirando qué es lo que queremos ir marcando, y aplicarle la regex adecuada. La mayoría de errores se producen por fallos en la sintaxis, ya que como hemos visto muchos caracteres necesitan ser escapados con la contra-barra. Por eso ante un error lo primero mirar que la sintaxis este bien y luego ya ir mirando la regex parte por parte.

17. Editando el archivo de configuración

Hasta ahora hemos estado editando archivos con la configuración estándar del editor, pero podemos hacernos un VIM a medida ;). Vamos a ver una serie de configuraciones que pueden resultarnos muy útiles. El archivo de configuración se encuentra en `/etc/vim/vimrc`. El carácter usado como comentario es la comilla “.

Nota

¡IMPORTANTE! Al editar el archivo “/etc/vim/vimrc” estamos aplicando la configuración para todos los usuarios del sistema, si usted comparte el pc con otras personas o no tiene los permisos suficientes, cree un archivo `~/.vimrc` y añada toda la nueva configuración en él.

Para habilitar el resalto de sintaxis, podemos descomentar la línea que pone **`syntax on`**. Así según el tipo de archivo que abramos usará un resalto de sintaxis de los que almacena en `/usr/share/vim/vim72/syntax/`. Para quitar el resalto desde el editor podemos lanzar el comando **`:syntax off`**.

También podemos habilitar la numeración de las líneas, añadimos al final del archivo **`set number`**. Para quitar la numeración podemos ejecutar el comando **`:set nonumber`**.

Nota

*La mayoría de “seteos” se niegan añadiendo **`no`** delante.*

Seguramente os acordaréis que os dije que no podíamos ver los comandos de actuación inmediata que íbamos tecleando, pues bien era mentira :p. Podemos descomentar la línea que pone **`set showcmd`**, así veremos lo que vamos pulsando. Se muestran en la línea de estado, a la izquierda de la numeración de “línea,columna”.

También podemos descomentar la línea **`set incsearch`**, así cuando estemos realizando una búsqueda, nos irá buscando en tiempo real mientras escribamos.

Si añadimos también la opción **`set hlsearch`**, que resalta las búsquedas, tenemos una gran ayuda a la hora de crear la expresión regular. A medida que vayamos escribiéndola, se irá resaltando el texto que marca. Muy muy útil ;). Si queremos quitarlo **`:set nohlsearch`**.

Si nos dedicamos a programar nos será muy útil la opción **set autoindent**, que nos mantendrá la indentación de una línea a otra.

Y para los que estéis hartos de los tabuladores (como yo), la opción **set expandtab**. Al introducir un tabulador, en realidad nos lo convierte en espacios, y podemos definir el número de espacios que pondrá por cada tabulación con **set tabstop=2** (por ejemplo).

Para habilitar o deshabilitar el ajuste de línea, tenemos las opciones **:set wrap** (habilitar) y **:set nowrap** (deshabilitar).

Hay muchas más opciones, mas específicas, que podéis investigar si os quedáis con ganas. Una vez que hemos añadido las opciones, el editor quedaría tal que así:

```
1 Parrafo 1
2 Texto texto texto.
3 Texto texto.
4
5 Parrafo 2
6 Mas texto mas texto mas texto.
7 Mas texto mas texto.
8
9 Parrafo 3
10 Y mas texto y mas texto y mas texto.
11 Y mas texto y mas texto.
/^([ ])*\s[]
```

Figura 31: Con nuevas opciones.

Vemos como está la numeración de líneas y el resalto en las búsquedas.

18. Expresiones regulares 2

Hay otro comando que es incluso más potente que el de sustituir **:s**, es el comando global **:g**. Este comando nos permite buscar líneas que contengan una cadena de texto o una expresión regular sobre un rango especificado, y una vez que encuentra ese texto podremos ejecutar un comando, incluso podemos ejecutar el propio comando de sustituir dentro de el global.

El comando global por defecto toma como rango todo el archivo (**%**) a diferencia del sustituir que tomaba la línea actual (**.**). La sintaxis es esta:

:g/texto/comando

Vamos a poner unos ejemplos, con su correspondiente explicación:

- Queremos eliminar las líneas en blanco de un archivo:

:g/^\s*\$ /d

```

1 Parrafo 1
2 Texto texto texto.
3 Texto texto.
4 Parrafo 2
5 Mas texto mas texto mas texto.
6 Mas texto mas texto.
7 Parrafo 3
8 Y mas texto y mas texto y mas texto.
9 Y mas texto y mas texto.
~
~
:g/^\s*$ /d                                7,1      Todo

```

Figura 32: Tras ejecutar el comando.

Hemos hecho una búsqueda de las líneas que empiecen (**^**), pudiendo contener 0 o más espacios en blanco (**\s***) y terminando seguidamente la línea (**\$**). Tras lo cual vamos a ejecutar el comando borrar (**d**) sobre las líneas coincidentes.

- Vamos a añadir unos caracteres en las líneas con el título del párrafo, al principio y al final:

:g/Parrafo\s[0-9]\+/s/## & #####/

```

1 ## Parrafo 1 #####
2 Texto texto texto.
3 Texto texto.
4
5 ## Parrafo 2 #####
6 Mas texto mas texto mas texto.
7 Mas texto mas texto.
8
9 ## Parrafo 3 #####
10 Y mas texto y mas texto y mas texto.
11 Y mas texto y mas texto.
3 sustituciones en 3 líneas      8,0-1      Todo

```

Figura 33: Tras ejecutar el comando.

En este caso seleccionamos el texto “Parrafo”, seguido de un espacio (\s) y de uno o más dígitos ([0-9]\+), ya que no sabemos si el número de párrafos llega a 9, 20 ó 500. Para seleccionar los dígitos podíamos haber usado el carácter especial \d. En esta ocasión usamos el comando sustituir, pero en el campo1 no ponemos nada por que al buscar la cadena con :g, hemos seleccionado ya lo que queríamos. Así que solo queda poner en el campo2 unas almohadillas y entre medio el contenido de la búsqueda (&), en este caso no del propio comando sustituir, sino del global.

- Vamos a corregir el fallo ortográfico (hecho a posta :p) en la palabra “Parrafo”:

:g/Parrafo/s/a/á/

```

1 Párrafo 1
2 Texto texto texto.
3 Texto texto.
4
5 Párrafo 2
6 Mas texto mas texto mas texto.
7 Mas texto mas texto.
8
9 Párrafo 3
10 Y mas texto y mas texto y mas texto.
11 Y mas texto y mas texto.
3 sustituciones en 3 líneas      9,1      Todo

```

Figura 34: Tras ejecutar el comando.

En este caso si que hemos especificado en el comando `sustituir` que queríamos cambiar solo la **a** por la **á**, nótese que al no poner el modificador **g**, solo sustituirá la primera aparición de esa letra (como así queríamos).

Aún queda otra utilidad por ver sobre el comando global. Nos permite negar una búsqueda, lo que quiere decir que podemos seleccionar lo que NO coincida con la búsqueda. Para ello basta añadir un **!**:

:g!/texto/comando

- Vamos a guardar las líneas que no estén vacías o las de los títulos de párrafo, en otro archivo llamado “prueba2”:

:g!/^\s*\$ \|Parrafo/. w >> prueba2

```

1 Parrafo 1
2 Texto texto texto.
3 Texto texto.
4
5 Parrafo 2
6 Mas texto mas texto mas texto.
7 Mas texto mas texto.
8
9 Parrafo 3
10 Y mas texto y mas texto y mas texto.
11 mas texto y mas texto.
"prueba2" 1L, 25C añadido
11,1 Todo

```

Figura 35: Tras ejecutar el comando.

En la imagen vemos como se guarda el texto correctamente. Una cosa importante, el archivo lo tenemos que haber creado antes, ya que si no VIM dará un error al no poder abrir un archivo inexistente. Lo que hacemos aquí es seleccionar las líneas vacías (`^\s*$`) y usamos el indicador `\|` (or), para indicar otra opción, en este caso las líneas que contengan la cadena “Parrafo”. Posteriormente le decimos que la línea actual (.) la guarde (w), agregándola (>>) al archivo “prueba2”. Esto último aunque parezca lioso es el comando guardar, donde hemos puesto un rango (la línea actual) y un archivo donde guardarlo, podéis ejecutarlo por separado **:. w >> prueba2** para comprobarlo.

Lo único que hay que saber es como trabaja el comando global. Este va buscando línea a línea y ejecutando el comando en cada línea que encuentra coincidencia. Por eso el punto indica la línea actual donde se encuentre el comando, y esa línea la llevamos al archivo. De

ahí la importancia de poner el >>, ya que se realizaran múltiples inserciones al archivo. De lo contrario nos salta el error al intentar sobrescribir un archivo. Así es como queda el archivo:

```
1 Texto texto texto.  
2 Texto texto.  
3 Mas texto mas texto mas texto.  
4 Mas texto mas texto.  
5 Y mas texto y mas texto y mas texto.  
6 Y mas texto y mas texto.  
~  
~  
~  
~  
~  
~  
"prueba2" 6L, 146C 1,1 Todo
```

Figura 36: Nuevo archivo creado.

- Y un último ejemplo, vamos cambiar las ip's de un hipotético archivo de logs:

```
1 login desde 192.168.1.30  
2 logs, logs  
3 71.240.123.3 me ataca  
4 logs, logs, logs  
5 sshd detecta petición desde 32.23.12.178  
6 intento de conexion desde 32.23.12.178 rechazado  
7 logs, logs, logs, logs  
8 acceso desde 230.76.181.8 a la zona hipermegasegura  
9 login como root desde 154.214.197.251  
10 logs, logs  
1,1 Todo
```

Figura 37: Falso archivo de logs.

Tenemos este archivo con ip's variadas, y queremos, por ejemplo, cambiar todas a "127.0.0.1".

```
: %s/(\d{1,3}\.){3}\d{1,3}/127.0.0.1/g
```

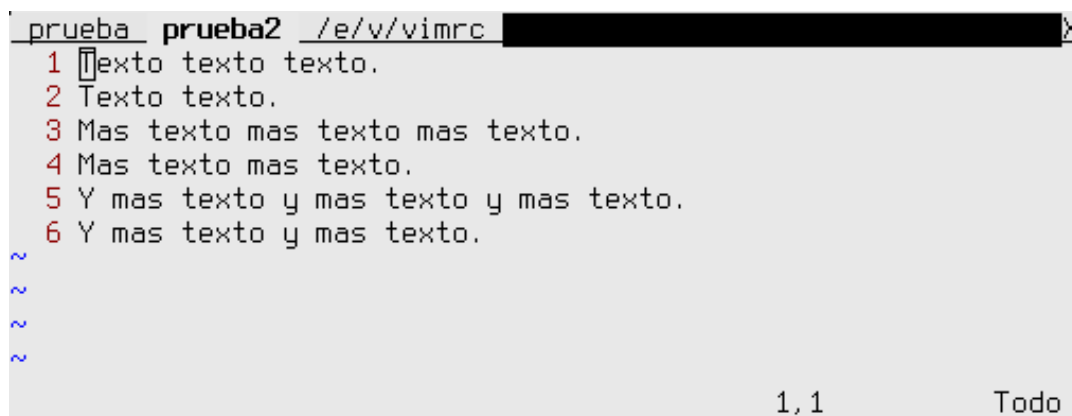
```
1 login desde 127.0.0.1
2 logs, logs
3 127.0.0.1 me ataca
4 logs, logs, logs
5 sshd detecta petición desde 127.0.0.1
6 intento de conexión desde 127.0.0.1 rechazado
7 logs, logs, logs, logs
8 acceso desde 127.0.0.1 a la zona hipermegasegura
9 login como root desde 127.0.0.1
10 logs, logs
6 sustituciones en 6 líneas          9,1      Todo
```

Figura 38: Tras ejecutar el comando.

En esta expresión regular le decimos, primeramente que debe buscar el contenido de un grupo `(\)` un número exacto de veces, en este caso debe aparecer tres veces (`{3}`). El grupo estará formado por un dígito cualquiera (`\d`), que puede aparecer de una a tres veces (`{1,3}`), y por un punto (`\.`), que será el punto de separación de los octetos. Una vez que cogemos los tres primeros campos de la IP, solo nos queda un número (`\d`) que podrá aparecer de una a tres veces (`{1,3}`), al igual que antes. En el campo2 ponemos la ip a sustituir y añadimos el modificador `g` por si hay más de una ip por línea.

19. Pestañas o Tabs

Una de las utilidades que más me gusta es las posibilidad de abrir pestañas. Nos ahorra mucho espacio, al no tener que abrir una terminal nueva. Tendremos a golpe de vista los archivos que tenemos abiertos, cosa que no ocurre si editamos varios archivos de una vez, con el método que conocíamos hasta ahora. El comando es **:tabnew archivo**, pudiendo especificar el archivo mediante su ruta absoluta o relativa. De igual modo que en la terminal, podemos usar la tecla <TAB> para ir autocomplementando.



```
prueba prueba2 /e/v/vimrc
1 texto texto texto.
2 Texto texto.
3 Mas texto mas texto mas texto.
4 Mas texto mas texto.
5 Y mas texto y mas texto y mas texto.
6 Y mas texto y mas texto.
~
~
~
~
1, 1 Todo
```

Figura 39: Archivos en pestañas.

Como vemos, en la parte de arriba se reserva una línea para las pestañas. Y solo se ve la primera letra de cada directorio de la ruta al archivo “vimrc”. Para ir de una pestaña a otra usamos los comandos **gt** (siguiente pestaña) y **gT** (anterior pestaña). Otra ventaja es que podemos compartir el porta-papeles, así podemos copiar unas líneas en el archivo “a” y pegarlas en “b”.

20. Sangrado de líneas

Para aplicar un sangrado tenemos el comando `>` o `<`, al que podemos aplicar un rango de actuación o un movimiento. Para la línea actual podemos usar `>>` o `<<`, esto insertará un nivel de sangrado. Podemos especificar un número de líneas `5>>` (las cinco líneas siguiente), o un movimiento por el archivo. Por ejemplo `>G` aplica un nivel de sangrado desde la línea actual hasta el final de archivo, si recordamos con `G` nos movíamos la final del archivo. También podemos especificar una búsqueda, por ejemplo situados en la primera línea, si queremos subir un nivel al primer párrafo `>/^\s*$`.

```

1  Parrafo 1
2  Texto texto texto.
3  Texto texto.
4
5  Parrafo 2
6  Mas texto mas texto mas texto.
7  Mas texto mas texto.
8
9  Parrafo 3
10 Y mas texto y mas texto y mas texto.
11 Y mas texto y mas texto.
3 líneas >ed 1 vez                                1,3          Todo

```

Figura 40: Tras ejecutar el comando.

Simplemente le decimos que aplique un nivel desde la posición actual hasta que encuentre esa búsqueda. Si recordamos, VIM identificaba al párrafo con la letra **p**, así que podemos ahorrarnos trabajo con este comando `>ip`. Con **i** le decimos que no coja la línea en blanco (lo que el editor entiende por separación entre párrafo), y con **p** le decimos el rango de actuación (el párrafo actual).

El resultado será el mismo, la ventaja es que podemos situarnos en cualquier línea dentro del párrafo, al ejecutar el comando. Ya os dije que era importante saberse los campos de actuación predefinidos por VIM ;).

Nota

Para establecer el número de espacios que insertará este comando, añadir al archivo de configuración (`/etc/vim/vimrc`) la opción `set shiftwidth=2` (por ejemplo). Para que los cambios tengan efecto podemos cerrar y abrir el archivo de nuevo, o lo guardamos y ejecutamos `:e` sin parámetros que volverá a abrir el mismo archivo.

Hay una opción especial para la programación en C, no se si lo he dicho antes pero VIM esta pensado para la programación, en concreto en C. Y tiene una opción para el sangrado o autoindentación de este tipo de archivos. Debemos añadir al archivo de configuración: **autocmd FileType c setlocal cindent** . Definimos así una opción que se habilitará solo para archivos de C. A mi me gusta, pero ya cada uno que decida.

21. Otras características

VIM tiene muchas más posibilidades y utilidades. A continuación voy a comentar algunas que me parecen interesantes, y que no se han tratado anteriores secciones por que no son algo de “primera necesidad”. Este apartado servirá como complemento al resto del documento, para aquellos curiosos, que se hayan quedado con ganas de más ;).

21.1. Explorador de archivos

VIM también nos sirve como explorador de archivos, quiere decir esto que puede editar directorios. Podemos lanzar por ejemplo un **vim /etc/apt**, y nos abrirá ese directorio en una pantalla como la de un archivo normal, pero con unas funcionalidades distintas. Primero tendremos una cabecera con la ruta donde nos encontramos, como esta ordenando el contenido, una pequeña ayuda, y la lista de directorios y archivos que contiene ese directorio. Podemos navegar por el árbol de directorios, y para editar cualquier archivo nos posicionamos encima y damos <ENTER>, fácil ¿no? ;).

```

" =====
" Netrw Directory Listing
" /etc/apt
" Sorted by      name
" Sort sequence: [\\/]$,\\<core\\%(\\.\\.d\\+\\)\\=\\>,\\.h$,\\.c$,\\.cpp$
" Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:s
" =====
../
apt.conf.d/
[.]references.d/
sources.list.d/

10,1      Comienzo

```

Figura 41: Modo explorador de archivos.

21.2. Autocompletar palabras y rutas

Y otra de las muchas utilidades que nos ayudará, es el autocompletado de las palabras. Pulsando la combinación **CTRL+N** mientras escribimos (en modo insertar), se despliega un menú con las posibles coincidencias. La lista de coincidencias las saca a partir de las palabras que ya hemos escrito anteriormente. Por ejemplo si hemos llamado a una función “leerArchivoDatos”, para no tener que escribirlo de nuevo (con el peligro de equivocarnos), pulsamos la combinación y nos aparece el menú donde lo seleccionaremos.

```

1 #include<stdio.h>
2 //... código aquí ...
3 //Función que leerá un archivo y bla bla
4 struct contador leerArchivoDatos(FILE *f, struct contador co
  ntar);
5
6 int main(){
7     leerá
8     leerArchivoDatos
9     leerá[]
10
-- Completar palabra clave (^N^P) coincidencia 1 de 2

```

Figura 42: Autocompletar palabras.

Vemos el menú que se abrió encima de la palabra a autocompletar. Por defecto lo completa con la primera aparición de la lista, pero basta con movernos por el menú y seleccionar la que queramos.

Hay bastantes más combinaciones para el autocompletado, pero una que puede sernos útil es **CTRL+X CTRL+F**. Nos autocompleta el path o ruta que estemos escribiendo, por ejemplo:

```

1 Los repositorios estan en /etc/apache2/[]
~
~
~
~
~
~
~
~
~
~
~
~
-- Completar nombre de archivo (^F^N^P) coincidencia 1 de 3

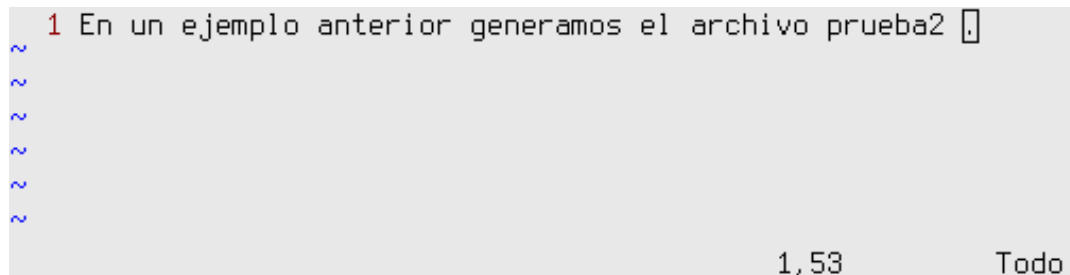
```

Figura 43: Autocompletar ruta.

Nos puso por defecto la primera opción, basta seleccionar la correcta. Al igual que la anterior, solo necesita que le pongamos los primeros caracteres y en base a eso nos da la lista con las posibilidades.

21.3. Abrir archivos desde el editor y manejos de buffers

Otra curiosidad sobre este editor, es que “podemos abrir un archivo desde otro”, con tan solo tener escrito el path o ruta del archivo en cuestión. El comando que usaremos es **gf**. Para entenderlo mejor vemos un ejemplo:



```
1 En un ejemplo anterior generamos el archivo prueba2
~
~
~
~
~
~
1,53 Todo
```

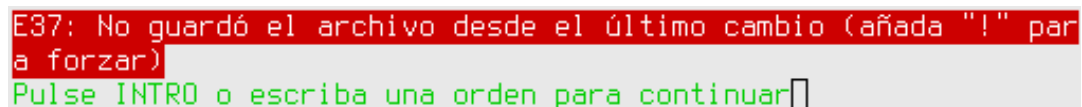
Figura 44: Archivo inicial.

Aquí tenemos escrito el archivo **prueba2**. Este es el nombre de un archivo válido, si recordáis este archivo le generamos en un ejemplo de expresiones regulares, anteriormente. En este caso al estar en la misma ruta en donde hemos abierto el archivo actual, con ponerlo tal cual sobra, pero si no debemos especificar la ruta absoluta o la relativa a partir de nuestro directorio activo. Por ejemplo: **/etc/vim/vimrc**.

Nota

*Para ver cual es el directorio activo, donde estamos, podemos usar el comando **:pwd**.*

Antes de ejecutar el comando, debemos asegurarnos de que el cursor esta situado sobre el archivo que queremos abrir. Y también debemos guardar el archivo, ya que al abrir el otro archivo, cerraremos el actual. De todas formas, nos avisará de ello.



```
E37: No guardó el archivo desde el último cambio (añada "!" para forzar)
Pulse INTRO o escriba una orden para continuar
```

Figura 45: Error al no guardar.

Una vez guardado, nos abre el nuevo archivo sin problemas, pero ¿y si queremos volver al inicial ?.

```

1 Texto texto texto.
2 Texto texto.
3 Mas texto mas texto mas texto.
4 Mas texto mas texto.
5 Y mas texto y mas texto y mas texto.
6 Y mas texto y mas texto.
~
~
~
~
~
"prueba2" 6L, 146C                                1,1      Todo

```

Figura 46: Archivo “prueba2” abierto.

Si recordamos al principio del documento dije que los archivos realmente son buffers, así que cuando hemos cerrado el archivo, hemos cerrado su buffer, pero este no desaparece de la memoria. Podemos ver los buffers con el comando **:buffers** .

```

4 Mas texto mas texto.
5 Y mas texto y mas texto y mas texto.
6 Y mas texto y mas texto.
~
~
~
~
~
:buffers
1 #      "prueba"                línea 1
2 %a     "prueba2"               línea 1
Pulse INTRO o escriba una orden para continuar

```

Figura 47: Buffers.

Como vemos, cada archivo tiene un buffer numerado, seguido de unos caracteres o flags (ahora vemos su significado), seguido del propio nombre y la línea en la que teníamos el cursor. Los caracteres significan:

%	Buffer actual.
#	Buffer cerrado.
a	Buffer activo (cargado y visible).
h	Buffer oculto (cargado pero oculto).
-	Buffer no es modificable.
=	Buffer con permiso de solo lectura.
+	Buffer es modificable.
x	Buffer con errores de lectura.

Analizando esos caracteres, vemos que el buffer número uno, correspondiente al archivo inicial, tiene el flag de buffer cerrado (como estaba previsto). Los flags del buffer número dos, indican que es el actual y es visible. Podemos saltar de un buffer a otro especificando su número, por ejemplo con **:buffer 1** volvemos a cargar o abrir el primer archivo. Tras ejecutar el comando podemos ver con **:buffers**, como ahora el primero es el actual, y el segundo esta cerrado.

Retomando el hilo de esta sección, que por si ha alguno se le olvidó, estábamos viendo como abrir archivos a partir del nombre o ruta directamente escrita en el archivo. El método anterior, con el comando **gf**, es un poco engorroso ya que nos cierra el archivo para abrir el otro. Hay otra alternativa que posiblemente nos contente más, y es abrirlo en una ventana partida.

Vimos casi al principio del documento como eran las ventanas, si recordáis al pedir ayuda con el comando **:help**, nos abre una ventana. Para abrir un archivo en una ventana, ejecutaremos el comando **CTRL+W CTRL+F** (posicionados sobre el nombre del archivo), con la ventaja de poder ver los dos archivos, y no cerrar el inicial.

```

1 Texto texto texto.
2 Texto texto.
3 Mas texto mas texto mas texto.
4 Mas texto mas texto.
5 Y mas texto y mas texto y mas texto.
prueba2                               1,1      Comienzo
1 En un ejemplo anterior generamos el archivo prueba2 .
~
~
~
prueba                                1,47      Todo
"prueba2" 6L, 146C
```

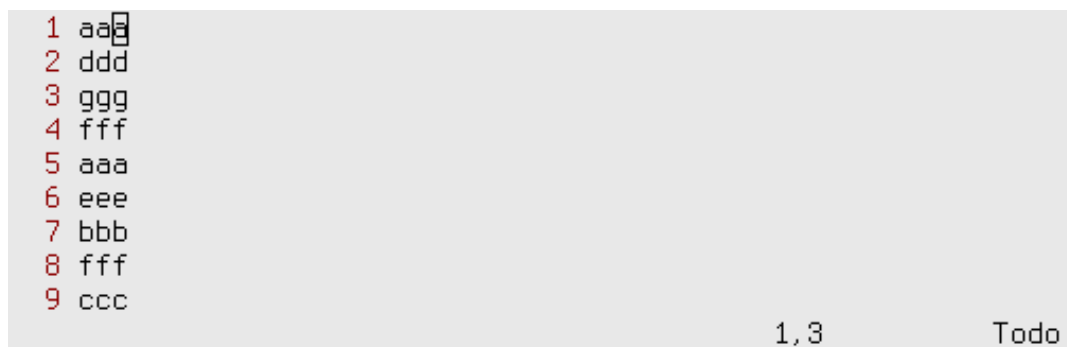
Figura 48: Archivos en ventana.

Aquí vemos los dos archivos, arriba el que hemos abierto. La nueva ventana ocupará la mitad de líneas que la inicial. Ahora el archivo donde nos movemos es el que acabamos de abrir. Para cambiar el control a la ventana de abajo, con el archivo inicial, usamos la combinación **CTRL+W** seguido de la flecha hacia abajo, o hacia arriba si queremos subir a la ventana de arriba desde la de abajo. Para cerrar cualquiera de las ventanas usamos el comando **:close**.

21.4. Ejecución de comandos de sistema.

VIM también puede ejecutar otro tipo de comandos, diferentes a los de su propio entorno. Podemos ejecutar comandos del sistema, tales como “sort”, “uniq”, etc, y todo sin salir del editor claro ;). Para ello usamos el comando **:!** , que al igual que los demás le tendremos que pasar un rango de actuación y el propio comando a ejecutar.

Para esto hay que tener un poco de idea de los comandos básico del sistema y de su utilización. Vamos a ver unos ejemplos, partiendo del archivo que vemos a continuación:



```
1 aaa
2 ddd
3 ggg
4 fff
5 aaa
6 eee
7 bbb
8 fff
9 ccc
```

1,3 Todo

Figura 49: Archivo original.

- Queremos ordenar todo nuestro archivo alfabéticamente:

```
:%!sort
```

```
1 aaa
2 aaa
3 bbb
4 ccc
5 ddd
6 eee
7 fff
8 fff
9 ggg
9 líneas filtradas 1,1 Todo
```

Figura 50: Tras ejecutar el comando.

Aquí hay poco que explicar, el comando **sort** realiza esa función, y aplicamos el rango a todo el archivo con **%**.

- Ahora además vamos a quitar las líneas duplicadas:

```
: %!sort | uniq
```

```
1 aaa
2 bbb
3 ccc
4 ddd
5 eee
6 fff
7 ggg
~
~
9 líneas filtradas 1,1 Todo
```

Figura 51: Tras ejecutar el comando.

En este caso hemos usado una tubería para enlazar los dos comandos necesarios.

Podemos usar comandos que generen una salida. Pero en este caso sobrescribirá el rango seleccionado, me explico, si seleccionamos un rango de todo el archivo (**%**), y ejecutamos el comando **echo hola**, nos dejará una sola línea con esa palabra. Para solucionarlo podemos dejar una línea vacía, y especificar como rango esa única línea, que como estará vacía no importa que lo sobrescriba.

Pero esa es la forma cutre ;), para comandos con salida, podemos enlazarlo con el comando **:read** . Este comando lo que hace es leer de la fuente que nosotros le proporcionemos y escribirlo donde le indiquemos. En nuestro caso queremos que lea desde un comando, vemos un ejemplo:

- Queremos hacer un listado de los host que están en nuestra red, partimos de un archivo tal que así:

```
1 [A] continuación un reporte con nmap:
```

```
~  
~  
~  
~  
~  
~  
~  
~
```

```
1,1 Todo
```

Figura 52: Archivo original.

Ahora vamos a ejecutar nmap, junto con read:

```
:1read ! nmap -sP 192.168.1.1/24
```

```

1 A continuación un reporte con nmap:
2
3 Starting Nmap 6.00 ( http://nmap.org ) at 2013-07-05 04:09
  CEST
4 Nmap scan report for 192.168.1.1
5 Host is up (0.0050s latency).
6 Nmap scan report for 192.168.1.75
7 Host is up (0.00011s latency).
8 Nmap done: 256 IP addresses (2 hosts up) scanned in 2.80 se
  conds
7 líneas más

```

Figura 53: Tras ejecutar el comando.

Y “voilà” tenemos el reporte hecho ;). Como rango hemos puesto la primera línea, así pondrá la salida a continuación. Ya según el nivel de manejo de los comandos, y de la propia imaginación de cada uno, le iréis encontrando utilidades.

Nota

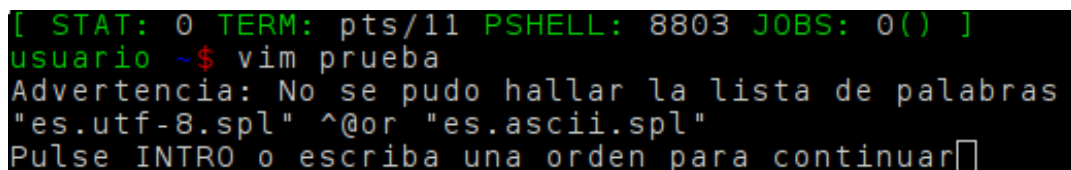
*Se puede abrir temporalmente una shell con el comando **:shell** . En esta nueva shell, podemos trabajar igual que en cualquier otra, pero al cerrarla (con **exit**) volveremos al editor.*

21.5. Corrección ortográfica

Para ser una buena herramienta de edición de texto, no podía faltar el corrector ortográfico. Siguiendo unos sencillos pasos lo tendremos totalmente operativo.

Primero vamos a añadir lo necesario a nuestro archivo de configuración **/etc/vim/vimrc** . Añadiremos el seteo **set spell spelllang=es,en** al final del archivo (por ejemplo). Como veis, como lenguaje principal ponemos el español (es), y de secundario el inglés (en), ya que actualmente las palabras en inglés, están a la orden del día.

Pero aún nos quedan cosas que hacer, si intentamos editar algún archivo con esta configuración, VIM nos saltará con una advertencia, que nos da una buena pista de lo que nos falta.



```
[ STAT: 0 TERM: pts/11 PSHELL: 8803 JOBS: 0() ]
usuario ~$ vim prueba
Advertencia: No se pudo hallar la lista de palabras
"es.utf-8.spl" ^@or "es.ascii.spl"
Pulse INTRO o escriba una orden para continuar
```

Figura 54: Error por falta de archivos.

Efectivamente, nos falta por descargar los diccionarios. Primero vamos a crear el directorio donde se alojarán, y nos situaremos allí.

```
[ STAT: 0 TERM: pts/11 PSHELL: 8803 JOBS: 0() ]
usuario ~$ mkdir -p ~/.vim/spell
48 file    23 dir    43 oculto
[ STAT: 0 TERM: pts/11 PSHELL: 8803 JOBS: 0() ]
usuario ~$ cd ~/.vim/spell
0 file    0 dir    0 oculto
[ STAT: 0 TERM: pts/11 PSHELL: 8803 JOBS: 0() ]
usuario ~/.vim/spell$
```

Figura 55: Creando el directorio.

Y ahora descargamos los archivos necesarios, en nuestro caso los dos diccionarios tanto de español como de inglés. Para la codificación escogemos los utf-8, para un mejor soporte. Lo podemos obtener en la pagina oficial de VIM, en concreto son: <http://ftp.vim.org/vim/runtime/spell/es.utf-8.spl> y <http://ftp.vim.org/vim/runtime/spell/en.utf-8.spl>.

```
[ STAT: 0 TERM: pts/11 PSHELL: 8803 JOBS: 0() ]
usuario ~/.vim/spell$ wget -q http://ftp.vim.org/vim/
/runtime/spell/{es,en}.utf-8.spl
2 file    0 dir    0 oculto
[ STAT: 0 TERM: pts/11 PSHELL: 8803 JOBS: 0() ]
usuario ~/.vim/spell$ ls
en.utf-8.spl es.utf-8.spl
```

Figura 56: Descargando los diccionarios.

Y ya podemos editar cualquier archivo con soporte de corrección ortográfica. Lo que hará es resaltar aquellas palabras que detecte están mal escritas. Aquí vemos como quedaría:

```
1 vim permite la correccion ortográfica.
2 Mi sistema es Debian
~
~
~
~
~
2,21 Todo
```

Figura 57: Resalto de palabras.

En este caso vemos como las palabras al inicio de línea o tras un punto, en minúsculas, son resaltadas con azul. Y los demás errores ortográficos con un tono rosa. A la hora de ver las posibles sugerencias, tenemos un amplio abanico de comandos, vamos a ver los más útiles. Todos los debemos ejecutar posicionando el cursor encima de la palabra.

Con el comando **z=** nos da una lista de sugerencias numeradas, para elegir cualquiera de estas, ponemos el número y pulsamos <ENTER>.

```
1 "Vim"
2 "Him"
3 "Dim"
4 "Him"
5 "Jim"
6 "Kim"
Escriba un número e <Intro> o pulse con el ratón (la omisión ca
-- Más --
```

Figura 58: Lista de sugerencias.

Si nos gusta más los menús, como los de autocompletar, y no queremos estar saliendo a modo comando, podemos usar el comando **CTRL+X s**. Así estando en modo insertar podemos abrir un menú con las sugerencias.

```
1 Vim permite la corrección ortográfica.
2 Mi sistema es corrección
~ correction
~ colección
~ corrections
~ corrupción
~ correccional
-- Sugerencia de ortografía (s^N^P) coincidencia 1 de 100
```

Figura 59: Menú de sugerencias.

Nota

*Cuando realicemos una corrección, podemos ejecutar el comando **:spellrepall**, que corregirá automáticamente todas las palabras (mal escritas), que coincidan con la que acabamos de corregir.*

Y para aquellas palabras que no vienen contempladas en la lista, podemos agregarlas con el comando **zg**, o si cambiamos de opinión, quitarlas con **zw**. Por ejemplo la palabra “Debian” viene como mal escrita, al ser un termino fuera del diccionario, pero la podemos agregar, así en futuros casos, no nos volverá a aparecer con el resalto.

```

1 Vim permite la corrección ortográfica.
2 Mi sistema es Debian.
~
~
~
~
~
Añadiendo pala...ell/es.utf-8.add"          2,18      Todo

```

Figura 60: Palabra añadida a la lista.

Nota

Para movernos entre las palabras mal escritas, podemos usar los comandos `Js` (siguiente), y `Is` (anterior).

21.6. Guardar sesiones enteras

Cuando editamos archivos grandes, en los que tenemos unas cuantas marcas, hemos copiado varias líneas en distintos buffers, y no queremos perderlas al cerrar el archivo, podemos guardar toda esa información. VIM dispone de un comando para guardar una sesión completa con todos los registros, marcas, seteos, etc, cuya sintaxis es **`:mksession <FILE>`**. En concreto, el comando **`:set sessionoptions`** contiene todo lo que se guardará. Debemos especificar un nombre para el archivo que se creará con la información.

Para cargar una sesión, una vez abierto el archivo de forma normal, ponemos el comando **`:source <FILE>`**. Tanto al crear el archivo de sesión como al cargarlo, VIM no notifica nada en la barra de estado, sólo si se llegase a producir algún error.

Nota

*VIM permite la ejecución de comandos al lanzar el programa con el parámetro `-c`. Podemos incorporar el comando anterior, si abrimos el archivo con ese parámetro: **`vim -c “:source <FILE>” <ARCHIVO>`**.*

21.7. Creando nuestros propios Snippets (mapeos)

Hay muchos comandos dedicados a “mapear”, dependiendo del modo donde queramos que se haga, o de su recursividad. Vamos a ver un comando que puede ahorrarnos algo de tiempo,

a los programadores sobre todo. Puesto que nos interesa que el snippet salte a medida que vamos escribiendo usaremos la opción **inoremap**. La “i” indica el modo insertar, el “nore” indica que no será recursivo (si tenemos definidos otros mapeos), y “map” pues el comando en sí.

El mapeo lo incluiremos en el archivo de configuración (vimrc). La sintaxis del comando es:

inoremap campo1 campo2

Donde “campo1” es una combinación de caracteres, que una vez escritos se convertirán en el “campo2”. Por ejemplo uno sencillito sería este: **inoremap { }<left>**. Cuando abramos un corchete, inmediatamente nos lo convierte a la estructura del “campo2”, que básicamente lo cierra y nos mueve el cursor un espacio a la izquierda, para que quede entre medio.

Nota

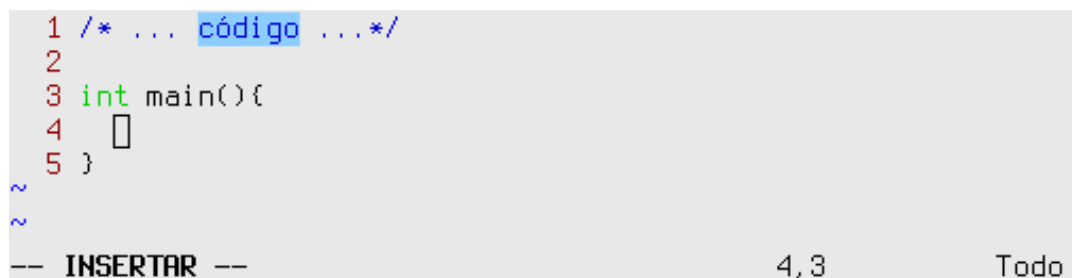
Las teclas se pueden representar, como se ha visto en el ejemplo, para la flecha a la izquierda use <left>.

Aquí una lista de la notación que usan algunas teclas especiales:

<up>	Flecha arriba	<f1> .. <f12>	Teclas de función
<down>	Flecha abajo	<insert>	Insert
<left>	Flecha izquierda	<end>	End
<right>	Flecha derecha	<pageup>	Re Pág
<esc>	Escape	<pagedown>	Av Pág
<cr>	Enter	<c-tecla>	CTRL+<TECLA>
<space>	Espacio	<s-tecla>	SHIFT+<TECLA>
<tab>	Tabulador	<c-s-tecla>	CTRL+SHIFT+<TECLA>
<bs>	Backspace		
	Supr		

Esto da muchas posibilidades de cara a hacer nuestros propios snippets, y cada uno seguro que ya esta pensando en como darle uso ;). El ejemplo anterior le podemos refinar, quedando así:

```
inoremap { {<cr><cr><up><tab>
```



```

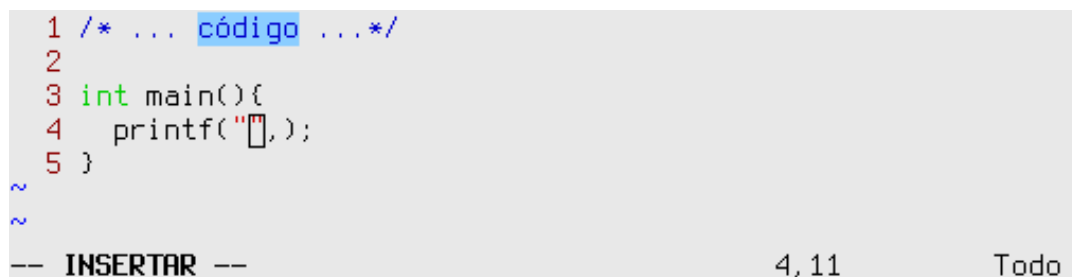
1 /* ... código ... */
2
3 int main(){
4     □
5 }
~
~
-- INSERTAR --                                4,3          Todo

```

Figura 61: Snippet.

Como veis no tiene ninguna complicación, es ir poniendo lo que nosotros haríamos, sustituyendo las teclas especiales por su notación. Un último ejemplo:

```
inoremap printf printf("");<left><left><left><left> .
```



```

1 /* ... código ... */
2
3 int main(){
4     printf("□,");
5 }
~
~
-- INSERTAR --                                4,11         Todo

```

Figura 62: Otro snippet.

Ojo, por que si lo definimos tal cual, cada vez que estemos editando cualquier archivo, nos seguirá haciendo los mapeos. Por eso lo mejor es especificar cuando queremos que se usen, o mejor dicho con que tipos de archivos. Si recordáis, cuando vimos la indentación, había una especial para c, que la metíamos en el comando **autocmd FileType c**. Pues haremos lo mismo con los comandos para el mapeo, nos quedaría tal que así:

```
autocmd FileType c inoremap { {<cr><cr><up><tab>
```

Esto lo añadimos al archivo de configuración, por supuesto. Así podéis ir definiendo mapeos para cada lenguaje de programación, o bueno para lo que queráis, si le echáis imaginación ;).

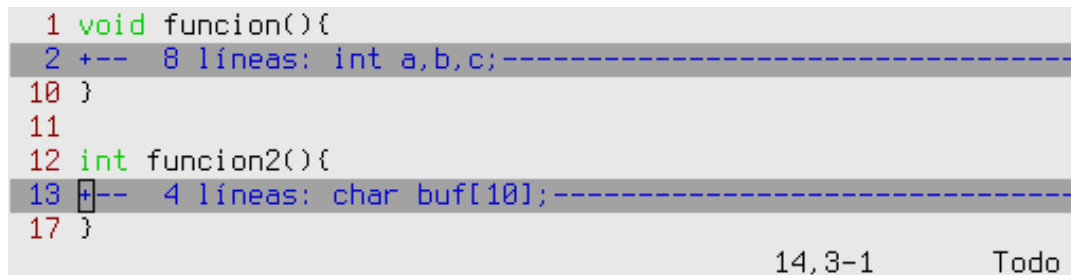
Nota

Cuando estemos escribiendo el snippet, VIM va evaluando las tecla pulsadas, y puede que nos parezca que el texto no se esta escribiendo. Pero ojo, por que si dejamos de escribir o no escribimos el snippet medianamente rápido, dejará de analizar las teclas y el mapeo no tendrá efecto.

21.8. Gestión de pliegues

Cuando estamos programando, puede que queramos ocultar el contenido de las funciones, o de las variables, con el fin de ver el código más organizado. VIM tiene muchas opciones y tipos de pliegues, incluso los podemos personalizar. Pero para seguir un método general vamos a ver algunos de los tipos de pliegues predefinidos.

Vamos a empezar por el método quizás mas orientado a la programación, y es el que se rige en base a nivel de sangrado. Para ello añadimos en el archivo de configuración el comando **set foldmethod=indent**. Una buena practica en la programación es aplicar un nivel de sangrado acorde con las funciones y subfunciones que vamos poniendo. Y este método se aprovecha de esta regla no escrita, pudiendo llegar a ser muy útil.



```
1 void funcion(){
2 +-- 8 líneas: int a,b,c;-----
10 }
11
12 int funcion2(){
13 +-- 4 líneas: char buf[10];-----
17 }
```

14, 3-1 Todo

Figura 63: Pliegues según sangrado.

Podemos ver en esa imagen dos pliegues, uno en cada función, vamos a analizar su estructura. Según el nivel de pliegue, va a tener más o menos guiones después de signo de suma. En este caso los dos son unos pliegues de primer nivel y tienen dos guiones (+--). Si dentro tuviésemos una función con distinto sangrado (un for por ejemplo), tendría un pliegue de segundo nivel, al estar contenido en el principal y se representaría con tres guiones al inicio (+---). A continuación nos indica el número de líneas que contiene el pliegue, y nos incluye también el contenido de la primera línea del pliegue.

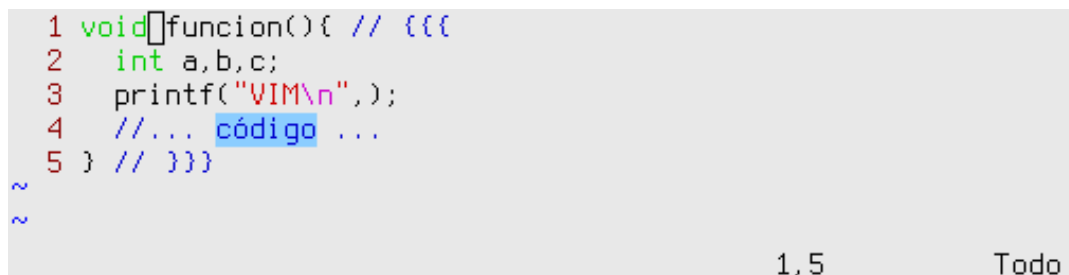
Para abrir el pliegue nos posicionamos encima y movemos el cursor en sentido horizontal (izquierda o derecha). Podemos copiar el pliegue, o cortarlo como si se tratase de una línea cualquiera. Hay una serie de comandos específicos para gestionar los pliegues:

zo	Abre el pliegue actual (open).
zO	Abre el pliegue actual, incluso los anidados a este.
zc	Cierra el pliegue actual (close).
zC	Cierra el pliegue actual, incluso los anidados a este.
zr	Abre TODOS los pliegues.
zR	Abre TODOS los pliegues, hasta los que están anidados.
zm	Cierra TODOS los pliegues.
zM	Cierra TODOS los pliegues, hasta los que están anidados.

El otro método que vamos a ver es el que se basa en un marcador. Quiere decir esto, que nosotros pondremos la cadena que inicia el pliegue y la que lo finaliza. Vamos a cambiar el tipo de pliegue en el archivo de configuración, por este otro **set foldmethod=marker**.

Por defecto las marcas que usa este método son {{{ para el inicio y }}} para el cierre final. Lo bueno de este método es que al poner nosotros las marcas, podemos poner pliegues en cualquier documento, no solo en archivos de programación. Por ejemplo, para tener organizado un documento por capítulos, y que cada uno tenga su pliegue.

En este caso vamos a poner el pliegue en una función, quedaría tal que así:



```

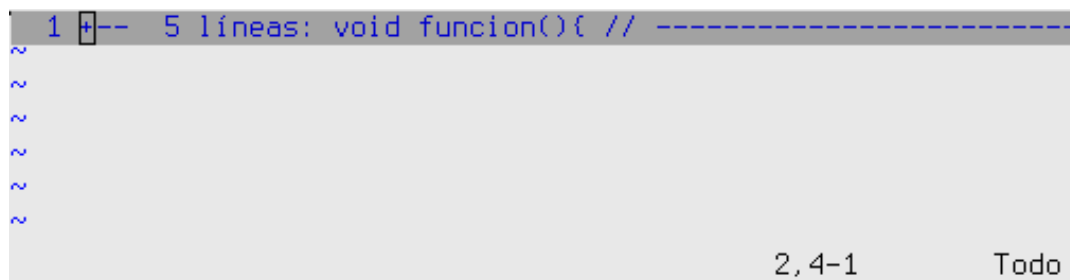
1 void funcion(){ // {{{
2   int a,b,c;
3   printf("VIM\\n",);
4   /*... código ...
5 } // }}}

```

Figura 64: Pliegues según marcas.

Aquí vemos como la cadena de comienzo del pliegue, la hemos puesto en el inicio de la función, luego veremos para qué. Y la cadena de finalización en el final, con la intención de abarcar toda la función, obviamente. En este caso no lo hemos hecho, pero podríamos haber numerado el nivel del pliegue. Este nivel se añade a la marca de comienzo, quedando {{{1}, para el primer nivel. Si tenemos algún otro pliegue dentro de este le podemos poner {{{2}, etc. Esta numeración no es obligatoria, ni mucho menos, esta pensada para que nos sea útil

a nosotros identificar el nivel donde nos encontramos. Una vez cerrado el pliegue (con **zc**) lo veremos tal que así:



```
1 +-- 5 líneas: void funcion(){ // -----
~
~
~
~
~
```

2, 4-1 Todo

Figura 65: Pliegue cerrado.

La razón por la que le pusimos el comienzo del pliegue, al inicio de la función, es para que al cerrar el pliegue nos aparezca esa línea con la declaración de la función. Bueno que para gustos los colores, ya sabréis vosotros como os viene mejor ponerlos ;).

Nota

*Hay mas métodos, pero también más “engorrosos”, si queréis investigarlo más **:help foldmethod** y **:help foldmarker** .*

22. Conclusión y opinión personal

Y después de esto, creo que lo único que le falta a VIM, es una combinación de teclas para que venga alguien a abanicarnos mientras programamos ;). Fuera bromas, espero que os haya gustado este pequeño documento. Quiero recalcar que lo que se ha visto aquí no es mas que la punta del iceberg. Este editor ofrece innumerables opciones, incluso tiene su propio lenguaje de scripting (archivos .vim), para realizar nuestras propias macros (también es compatible con otros lenguajes de scripting). Aquí tenéis el repositorio oficial por si queréis echar un vistazo <http://www.vim.org/scripts/index.php>.

Este es un editor que requiere un pequeño sacrificio al principio, pero que es capaz de recompensarnoslo a la larga. Hay mucha documentación en internet, por ejemplo podéis ver la documentación en <http://vimdoc.sourceforge.net> , y encontrar muchos consejos y trucos en http://vim.wikia.com/wiki/Vim_Tips_Wiki.

Y una última reflexión, y es que a mitad del recorrido del aprendizaje, se me planteó una duda y es la siguiente... ¿ Merece la pena invertir tantas horas en el aprendizaje de un editor de texto, ya sea VIM o cualquier otro ?. Al final he llegado a la conclusión de que si lo merece, ¡pero ojo! cuando lo que vas a aprender no te va a ayudar en el día a día, no merece la pena seguir adelante. Después de haber leído varios libros, recorrerme montones de web's, y lo más importante haber practicado mucho, me tope con el lenguaje de scripting de VIM y me pregunte ¿ me es necesario ?. Y la verdad que por mucha ilusión que tenía hice un stop. Quizá dentro de unos años lo retome como hobby, pero lo que quiero decir, no hay que perder de vista lo que realmente necesitas.

Y es por eso, que en este documento me he intentado centrar en lo que a mi parecer es imprescindible saber sobre VIM, a partir de aquí, que cada uno decida su camino. Saludos y gracias por leerme ;).

