Manual / Tutorial de Vim

(Tutorial por Santiago Romero)

Introducción

Este pequeño tutorial pretende ser una introducción y a la vez una referencia rápida al editor Vim (Vi IMproved) tanto para aquellos que quieren empezar a utilizarlo como para aquellos que ya lo utilizan de una forma básica (abrir fichero, modificar, guardar y salir) y desean ampliar sus conocimientos sobre este fantástico editor.

El tutorial puede ser algo largo, pero está organizado de forma que puedas empezar a leerlo desde el principio y parar en el momento en que lo que se explica está por encima de tus necesidades. Es más, puedes dejarlo en un punto concreto que cubra tus necesidades y volver a leerlo pasado un tiempo, y volver a aprender cosas nuevas. La idea es que no sea necesario leerlo entero para que te sea útil.

Si eres novato y te abruma ver tantas combinaciones de teclado o explicaciones, tómatelo con calma. Lee un capítulo, y prueba todos los comandos u opciones editando un fichero de texto. Aplíca esos comandos cada vez que puedas, utilizando vim para hacer tus tareas de edición de textos, y verás como pasado un tiempo, lo que leíste la anterior vez te parece **básico** y puedes avanzar algo más en el tutorial.

La parte inicial del tutorial contiene bastante texto, porque se corresponde con el momento en que no estamos familiarizados con los comandos de Vim y las explicaciones necesitan ser más profundas, pero conforme avanzamos en el texto, las descripciones serán más someras, ya que si hemos llegado hasta allí, directamente estaremos asimilando los comandos y conceptos sin necesidad de explicaciones complejas.

Para seguir el tutorial simplemente os recomiendo que tengáis instalada alguna versión nueva de Vim (que podéis descargar en http://www.vim.org (http://www.vim.org) o bien utilizando el sistema de paquetes de vuestra distribución Linux) y sobre todo que tengáis ganas de aprender a utilizarlo. Además del típico VIM de línea de comandos, existen compilaciones de Vim con un Interfaz Gráfico (GUI (Graphical User Interface)), como gvim o la propia de Windows. Y es que recordad que existen versiones de Vim no sólo para UNIX / Linux, sino también para Windows, por ejemplo, de forma que las ventajas de utilizar Vim las podéis aprovechar también en ordenadores Windows que os obliguen a utilizar en Universidades, o en el trabajo, por ejemplo. Así que armados con Vim instalado para nuestro sistema operativo favorito y algo de tiempo para leer, entremos en materia.

Funcionamiento básico de Vim

Vim es un editor de textos, en contraposición a lo que se conoce como procesador de textos.

En un procesador de textos es muy importante **el formato del texto**: cursivas, negritas, títulos, centrado o justificado, color y tamaño de la fuente, etc.

Vim, en cambio, se utiliza para **editar texto**. Lo importante no es el formato del texto sino el texto en sí mismo. Así, Vim se utiliza para programar, para escribir emails, para editar textos, código HTML (HyperText Markup Language), ficheros de configuración del sistema, etc.

Los procesadores de texto están centrados en ofrecer muchas cosas para el formateado del documento, mientras que Vim está pensado para facilitar la labor de introducción y edición del texto. No es muy útil editar un fichero de configuración o programar con LibreOffice o Word al igual que no tiene mucho sentido utilizar Vim para editar un documento donde lo que prima es el formato (pese a que gracias al lenguaje de programación LATEX, esto se puede hacer en Vim).

Por eso, cuando quieras programar, editar ficheros de configuración, o simplemente, hacer tu trabajo con texto de una forma más rápida, lo mejor es utilizar un editor de texto. Y como veremos, Vim es especial para hacer esta labor, por encima de muchos otros editores.

El editor Vim es una evolución del clásico editor VI. VI es un editor que encontraremos presente en casi el 100% de los sistemas UNIX (y si no está presente por defecto se puede instalar), por lo que conocer su uso es prácticamente una obligación para los Administradores de Sistemas. Por suerte, Vim se diseñó heredando casi todas las teclas y opciones de VI, de modo que siguiendo este tutorial nos aseguramos los conocimientos necesarios para manejar VI a nivel básico y medio. Podéis pensar en VIM como un VI mejorado, al cual podréis aplicar la mayoría de conocimientos de movimiento y edición que veremos aquí.

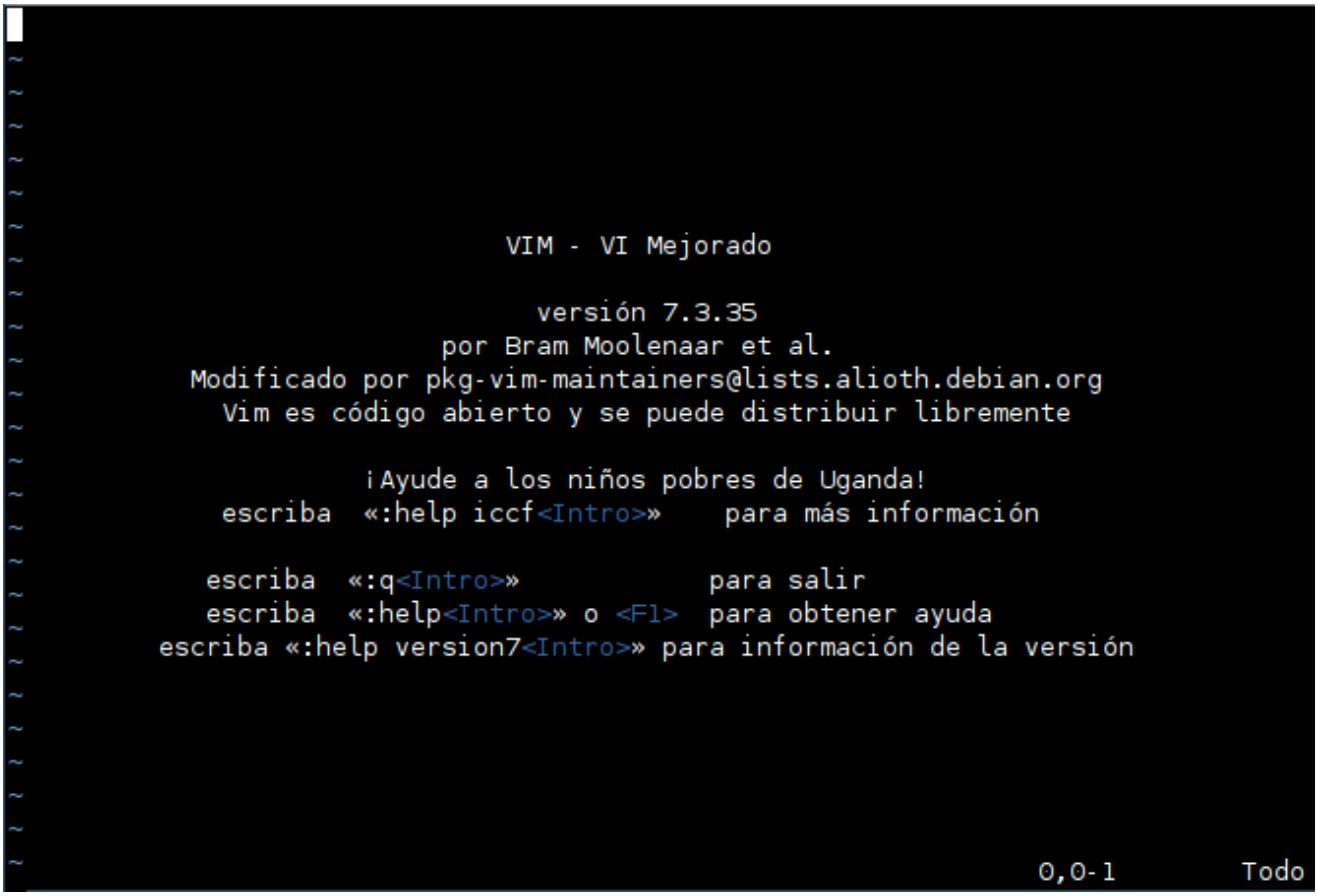
Instalación y ejecución de Vim

Si usas una plataforma Linux, lo más probable es que Vim ya esté instalado en tu sistema. En cualquier caso, puedes instalarlo con:

Variante	Sistema	Comando
Vim (modo texto)	CentOS / RedHat / Fedora	yum install vim-enhanced vim-minimal
gVim (modo gráfico)	CentOS / RedHat / Fedora	yum install vim-X11
Vim (modo texto)	Ubuntu / Debian / Mint	apt-get install vim-nox
gVim (modo gráfico)	Ubuntu / Debian / Mint	apt-get install vim-gtk

He separado la instalación de paquetes modo texto y modo gráfico por si alguien desea instalar ślo una de las 2 variantes, pero lo normal es tener ambas instaladas. Por ejemplo, podemos usar vim en terminales de texto para editar ficheros de configuración (cuando cambiamos a root en una consola) y gvim en proyectos de programación.

Para lanzar vim, basta con ejecutar **vim** en una terminal de texto (lo que creará un buffer vacío en vim), o bien **vim fichero** (que partirá con el contenido del fichero en el buffer).



También podemos irnos con el cursor directamente a una línea N contra del fichero editado si hacemos **vim +NUMERO fichero** (especialmente util si estamos editando un fichero tras un error de compilación de un programa, por ejemplo, y conocemos el número de línea del error).

Una vez dentro de Vim, podemos salir sin grabar el contenido ejecutando en modo comando **:q!** y grabando con **:x!**. En breve veremos qué quiere decir eso de "en modo comando".

Modo inserción y modo comando

Como muchos ya sabéis, a la hora de editar textos, Vim trabaja en varios modos: modo comando, modo inserción, modo visual… Se dice pues que **es un editor modal** (con varios modos de trabajo). En todo momento sabremos en cuál de los modos estamos gracias a la información que aparece en la barra de estado del editor (la última línea de la pantalla).

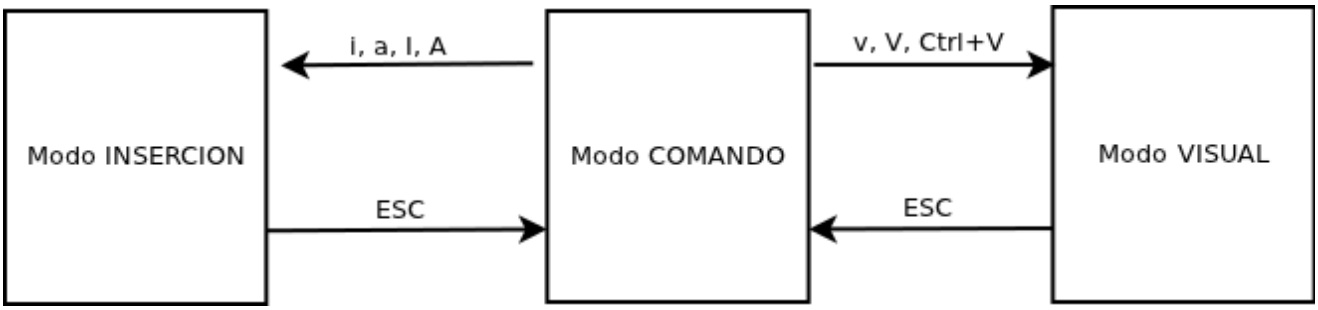
Los 2 principales modos de trabajo son **modo comando** y **modo inserción**. Para selección de texto con los cursores, también disponemos del **modo visual**.

En **modo comando** (el modo en que está Vim tras ejecutarlo) las teclas que pulsamos, en lugar de aparecer escritas en el documento, son interpretadas por Vim como comandos y nos permiten realizar acciones como grabar, salir, copiar, pegar, etc. Por ejemplo, pulsando **ZZ** en modo comando, no vamos a escribir dos zetas mayúsculas en el documento, sino que vamos a salir de vim grabando el fichero que estamos editando.

El **modo inserción** sí que nos permite introducir caracteres en el fichero, en la posición actual del cursor, al estilo de los editores básicos a los que estamos acostumbrados. Estando en modo inserción, si pulsamos **ZZ**, se insertarán dos zetas mayúsculas en la posición actual del cursor, tal y como cabría esperar en un editor normal. Cuando estamos en modo inserción aparece la cadena **–INSERTAR–** en la barra de estado del editor (la última línea de pantalla).

Para pasar al modo inserción desde el modo comando se utiliza la tecla/comando **i**, y para volver al modo comando se utiliza la tecla ESC.

La siguiente imagen ilustra los diferentes modos y las teclas que cambian entre ellos:



El hecho de disponer de 3 modos y tener que pasar de uno a otro puede parecer algo confuso o incluso un engorro, pero es justo la mejor baza de vim; es lo que le proporciona su potencia, lo que los demás editores no pueden hacer: aplicar comandos al texto. Es mucho más cómodo borrar una línea completa con el comando **dd** en vim que seleccionar la línea con el teclado o ratón y borrarla con la tecla DEL o SUPR en otro editor.

Alguien podría decir: *"""bueno, seguro que cualquier otro editor también tiene un atajo de teclado para borrar la línea actual completa."""*

Bien, imaginemos que la tiene (suele ser CTRL+Y), pero … ¿y si quieres borrar las 30 líneas siguientes a la del cursor (incluída esta)? ¿Vas a pulsar 30 veces el atajo de teclado? ¿Vas a seleccionar con el ratón o el teclado las 30 líneas? ¿Y si te digo que en modo comando de Vim, lo puedes hacer simplemente tecleando **30dd**?

Y es que **30dd** para vim significa **30 x dd**, o sea, **ejecuta 30 veces el comando dd**, es decir, **borra 30 líneas**.

Esto sólo se puede hacer gracias a la potencia del modo comando.

Cuando se es un novato en vim, y uno está acostumbrado a otros editores que cree más potentes, puede pensar que vim es un editor arcaico y obsoleto sólo por el hecho de que se utiliza integramente con el teclado y es modal.

La realidad es toda la contraria: es precisamente eso lo que permite que Vim sea mucho más potente que el resto de editores.

Vim es muy muy muy potente, no es un simple editor. La clave de Vim es estar el mayor tiempo que se pueda en modo comando, pasando a modo inserción sólo cuando se requiera introducir texto en el documento. Cuando estemos escribiendo emails o documentos de texto es muy probable que estemos casi todo el tiempo en modo inserción (a menos que queramos corregir algo que hayamos escrito), pero programando o editando ficheros de configuración ocurrirá justo lo contrario. Los atajos de teclado del modo comando se hacen algo complicados de entender al principio, pero tras el uso continuado de Vim se desarrolla en nuestra mente la forma de utilizar esos comandos de forma totalmente intuitiva, sin pararnos a pensar en ellos.

Cuando empieza a utilizarse vim sólo se conoce el funcionamiento básico, pero con el tiempo uno comienza a descubrir toda la potencia de este genial editor y empieza a cambiar la concepción de tiene de él: Vim no es sólo un editor, es una forma de vida en UNIX.

El fichero .vimrc

En vim podemos modificar muchos parámetros del editor mientras editamos los ficheros. Por ejemplo, tecleando en modo comando **:set number** (dos puntos, set number, intro), Vim activará la numeración de líneas (no dentro del fichero en sí, sino visualmente), algo que puede ser útil para programar.

Otro ejemplo, tecleando **:syntax on**, activaremos para el fichero actual el coloreado de sintaxis, es decir, que las palabras especiales que el editor entienda como que tienen un significado concreto aparecerán en diferentes colores. Si estamos programando en C, por ejemplo, las palabras claves aparecerán de un color, las cadenas de otro, etc (algo realmente útil a la hora de programar).

Pues bien, cualquier tipo de opción, macro, comando o función que vim entienda puede ser incluída en el fichero .vimrc en el directorio $HOME de nuestro usuario (o en un fichero _vimrc en el directorio de instalación de Vim o en el padre del Escritorio del usuario en Windows) de forma que se aplique como opción por defecto cuando

lancemos Vim. Así, podemos crear un fichero .vimrc (por defecto normalmente no existirá), que contenga algo como lo siguiente:

```
set nocompatible
set number
set ruler
syntax on
```

Esto hará que siempre que editemos un fichero, aparezca numeración de líneas (set number), un indicador de fila y columna en la barra de estado (set ruler) y resaltado de sintaxis (si está definida para el tipo de fichero que estamos editando) activado. Es algo así como **el fichero de opciones de vim para nuestro usuario** (y sólo para nuestro usuario). Existe un fichero de opciones general /etc/vimrc (normalmente) cuyos cambios afectan a todos los usuarios cuando arrancan vim, pero lo que incluyamos en nuestro .vimrc sólo afectará a vim cuando lo ejecutemos con nuestro usuario del sistema.

Así, podemos utilizar dicho fichero para indicar aquellas configuraciones con las que estemos más cómodos, de forma que podamos adaptar vim a nuestras necesidades. Es normal que en estos momentos iniciales no conozcamos vim lo suficiente como para hacernos un .vimrc decente, pero para empezar os recomiendo algo como lo que sigue:

```
" Fichero .vimrc de mi usuario
" Los comentarios se ponen con comillas dobles
set nobackup
set ruler

" nocompatible permite funciones que VI no soporta
set nocompatible

set tabstop=4
set sw=4
set expandtab

set vb
set noerrorbells
syntax on
```

Las posibilidades del fichero .vimrc son muy grandes, ya que no sólo soporta comandos simples de configuración sino que tiene un lenguaje propio que nos permite hacer casi cualquier tipo de cosa.

Poco a poco podréis ampliar este fichero con más opciones, macros, etc, que iremos viendo a lo largo del tutorial. De momento os recomiendo que creéis uno con, como mínimo, las opciones que acabamos de mostrar en el ejemplo anterior.

Comandos básicos: movimiento, inserción y borrado

Un primer contacto con Vim puede ser tan simple como editar un fichero con **vim fichero**, pasar a modo inserción pulsando **i** (i minúscula), moverse por el documento, cambiar e introducir texto, volver a modo comando (pulsando **ESC**), y salir del editor grabando los cambios en el fichero pulsando **ZZ** (2 zetas mayúsculas) o con **:x!**. Como toma de contacto inicial es suficiente y puede servir para perder el miedo al hecho de que Vim tenga 2 modos principales de funcionamiento (comando e inserción).

Lo siguiente que debemos hacer con nuestro editor es aprender a movernos por el texto. Supongamos que hemos creado/editado un documento con Vim, y tenemos que movernos por él (y añadir/cambiar cosas). Como siempre, en modo inserción (si tenemos bien configurada la variable $TERM del sistema) podremos movernos

con las teclas clásicas de los demás editores: cursores, Inicio, Fin, RePág, AvPág, etc.

No obstante, la potencia real de Vim la encontramos con las posibilidades de movimiento definidas en el modo comando. Aparte de que en modo inserción estamos muy limitados (movimiento en las 4 direcciones, principio y fin de línea, y anterior y siguiente página), algunas combinaciones de teclado no tienen por qué funcionar en ciertas máquinas, Sistemas Operativos o configuraciones de teclado (en Solaris, AIX, HPUX, o utilizando telnet/ssh contra otra máquina). El movimiento en modo comando es mucho más estándar (al utilizar teclas básicas del teclado y no teclas extendidas) y nos permite mucho más juego.

La regla general de Vim es moverse y trabajar siempre en modo comando y sólo pasar a modo inserción para introducir, borrar o modificar texto de nuestro documento (volviendo a modo comando al acabar el cambio), ya que el modo comando es el lugar donde podremos usar todas las opciones que en otros editores no se pueden realizar.

Veamos los diferentes comandos básicos de movimiento, inserción y borrado (siempre en modo comando):

Comando	**Significado**
h	Mover el cursor a la izquierda.
j	Mover el cursor hacia abajo.
k	Mover el cursor hacia arriba.
l	Mover el cursor hacia la derecha.
i	Insertar texto en la posición actual del cursor (Insert), pasando a Modo Inserción. Se permanece en modo inserción hasta que se sale explícitamente de él.
ESC	Salir del modo inserción y volver a modo comando. En modo comando, permite cancelar muchos de los comandos que se están ejecutando.
x	Borrar el caracter bajo el cursor (equivale a la tecla Del/Supr).
X	Borrar el caracter a la izquierda del cursor (equivale a la tecla Borrar/Backspace).
J	Juntar la línea actual con la siguiente (Join), eliminando el retorno de carro entre ellas.
u	Deshacer la última acción (Undo). Si lo pulsamos más veces desharemos acciones anteriores.
CTRL+R	Rehacer la última acción (Redo). Si lo pulsamos más veces reharemos acciones posteriores deshechas.
a	Insertar texto en la siguiente posición tras el cursor (Append). Es similar a **i**, salvo que el texto no se inserta en la posición actual del cursor sino a su derecha.
A	Poner el cursor al final de la línea y pasar a modo inserción (añadir texto al final).
o	Crear una línea vacía, en blanco, bajo la línea actual, y pasar a modo inserción con el cursor posicionado en dicha línea. Es mucho más cómodo que (como en otros editores) tener que pulsar FIN y ENTER para crear una línea en blanco.
O	Crear una línea vacía, en blanco, sobre la línea actual. Sería el equivalente en otros editores a ARRIBA, ARRIBA, FIN, ENTER.
dd	Borrar la línea actual (sobre la que está el cursor).
D	Borrar desde la posición actual del cursor hasta el final de la línea.

Como podéis ver, la existencia de ciertos comandos (como **o**, **O**, o **dd**) está pensada para evitar la mayor cantidad de pulsaciones de teclas/ratón posible. Borrar líneas con dd es mucho más rápido y sencillo que llevar la mano al ratón o a SHIFT+cursores en otros editores, e induce a muchos menos errores.

También, pulsar **J** (jota mayúscula) para juntar (Join) una línea con la línea siguiente es mucho más rápido que bajar a la siguiente línea, irse al principio de la misma, y pulsar borrar para subirla a la línea anterior.

Counts o repetidores

En la mayoría de comandos de Vim podemos añadir **counts**, que es como se conoce a los **repetidores** del comando. El count es un número que se teclea antes del comando para que se repita varias veces. Unido a la potencia del modo comando nos da mucho juego para la edición. Veamos unos cuantos ejemplos:

Comando	Significado
10dd	Repetir 10 veces el comando **dd**, es decir borrar 10 líneas empezando desde la línea actual. Es el equivalente a teclear manualmente 10 veces **dd**, y mucho más rápido que seleccionar 10 líneas a mano con ratón o cursores.
5x	Repetir 5 veces el comando **x**, es decir, borrar 5 caracteres empezando desde el carácter actual. Equivale a pulsar manualmente 5 veces el comando **x**.
60i-<ESC> <ENTER>	Insertar 60 guiones consecutivos. Este comando se teclea en modo comando pulsando 6, 0, i, guión, y pulsando la tecla ESCAPE y luego pulsar ENTER para validar el comando. Al hacerlo, estamos diciendo que se repita 60 veces la secuencia **i guión ESCAPE**, es decir, pasar a modo inserción, escribir un guión, y volver al modo comando pulsando ESCAPE. El 60 que hay delante lo repite 60 veces, con lo cual tenemos 60 guiones en pantalla. ¿No es mucho más cómodo al programar, para introducir separadores de comentarios, que pulsar el guión 60 veces o durante varios segundos mientras miramos la columna en la que estamos?
10iHola<ENTER> <ESC><ENTER>	(pulsar ESC y ENTER como las teclas ESC y ENTER, no tecleando la cadena) Aparece la palabra Hola 10 veces en pantalla, cada vez en una línea propia. Su significado, al igual que en el ejemplo anterior, sería **repite 10 veces la secuencia i, Hola, ENTER, ESC**, que pasa a modo inserción, escribe Hola, pasa a la siguiente línea con ENTER, y vuelve a modo comando.

Los 2 últimos ejemplos son bastante ilustrativos de la potencia de Vim en modo comando. Lo que en otros editores requiere varios segundos de presión de la tecla **-** para poner una raya horizontal (por ejemplo, en comentarios en C o C++ para separar funciones o clarificar los comentarios), en Vim se puede hacer con un simple comando, y sin miedo a poner guiones de más ni de menos o estar fijándose en la columna mientras los añadimos. Le pedimos a Vim que añada 60 guiones y lo hará directamente y sin posibilidad de error.

El último ejemplo nos muestra cómo repetir N veces una determinada frase en nuestro documento. No es necesario escribir, seleccionar, y pegar, pegar, pegar y pegar mientras contamos las frases que llevamos hasta tener nuestras 10 frases escritas como en otros editores. A Vim le decimos que repita la inserción 10 veces y lo hace sin necesidad de intervención extra por nuestra parte.

Los multiplicadores de comandos son muy útiles y pueden aplicarse a muchos de los comandos que veremos en este tutorial, aunque no lo digamos explícitamente aquí.

La manipulación del fichero (abrir, guardar, salir)

Los comandos básicos a la hora de editar ficheros son:

Comando	Significado
:w	Grabar los cambios del fichero actual.
:w nombre	Grabar el contenido actual del buffer en un fichero de nombre **nombre**.

Comando	Significado
:q!	Salir del editor sin grabar ningún cambio en el fichero actual (descartando cualquier cosa que hayamos hecho desde su apertura o última vez que grabamos).
ZZ	Salir del editor grabando los cambios en el fichero actual. También sirve **:x** o **:wq!**.
CTRL+G	Obtener información en la barra de estado del nombre del fichero que estamos editando, línea actual, número de líneas, en qué porcentaje del fichero estamos, y número de columna.
:e fichero	Abrir un fichero en el buffer actual (si está vacío) o en uno nuevo (si tenemos el buffer actual en uso)

En el caso de que hayamos abierto con **:e** más de un fichero, podemos movernos entre ellos con:

Comando	Significado
:bn	Siguiente buffer (fichero).
:bp	Anterior buffer (fichero).
:bd	Cerrar buffer (fichero) actual.

Hablaremos más sobre los buffers en un capítulo posterior, pero resulta interesante saber moverse entre múltiples ficheros abiertos.

Más sobre movimiento y cambios

Una vez hemos digerido lo básico sobre Vim (básico pero que ya nos permite hacer gran cantidad de cosas), vamos a ver más opciones de edición con respecto a la modificación del texto. Soy consciente de que hemos visto muchos comandos y atajos, pero tenéis que tener en cuenta que se aprenden con el uso (no por memorización) y que probablemente usando el 33% de lo que veremos ya seremos mucho más productivos que con un editor normal.

Modo reemplazar

Si estando en modo inserción pulsamos la tecla INSERT, pasaremos a modo REEMPLAZAR, donde el texto que introduzcamos modificará el texto bajo el cursor en lugar de añadirlo o insertarlo. Pulsando INSERT de nuevo volveremos a modo inserción (en realidad, INSERT sirve para conmutar entre ambos modos), y pulsando ESC volveremos a modo comando.

Si en modo comando queremos reemplazar un sólo carácter, podemos hacerlo mediante el comando **r**. Nos posicionamos sobre el carácter que queremos modificar, pulsamos **r** seguido del carácter correcto, y cambiaremos el carácter bajo el cursor por aquel que hemos tecleado tras la r. Por ejemplo, **ra** reemplazará el cáracter bajo el cursor por una **a**, sin salir del modo comando. Es ligeramente más rápido que pasar a modo inserción, borrar el carácter, introducir el nuevo y pulsar ESC para volver a modo comando. Obviamente, podemos aplicar modificadores para repetir el comando más veces. De esta forma, **10ra** cambiará los 10 carácteres a partir de la posición actual del cursor por carácteres **a**. Finalmente, **R** (mayúscula) permite reemplazar todos los caracteres hasta el final de la línea actual.

Deshacer cambios

En vim, podemos deshacer múltiples pasos con la tecla **u**, y rehacer pasos de nuevo con **Ctrl-R**.

Esto es diferente del ví clásico, donde sólo podías deshacer un paso, y volviendo a dar a "u" deshacías lo deshecho, es decir, hacías un undo del undo.

Para rizar más el rizo, vim dispone incluso de funcionalidad de undo especificando el tiempo. Podemos volver el documento atrás en el tiempo con el comando **:earlier**, y volver de nuevo adelante el tiempo necesario con **:later**:

```
:earlier 1h   <-- Volvemos el documento a como estaba hace 1 hora
:later 10m    <-- Ahora avanzamos 30 minutos (a como estaba hace 60-10=50m)
```

Especificando rangos de texto

En Vim podemos aplicar las búsquedas, borrados y modificaciones no sólo a la línea actual o a la totalidad del documento, sino que también podemos hacer referencia a porciones del texto:

Rango definido con…	**Significado**
%	Todo el documento.
n,M	Bloque de líneas desde la n a la M.
'a,'b	Bloque de líneas entre las marcas a y b.
'<,'>	Bloque de texto seleccionado en modo visual.

Por ejemplo:

```
:17,20d                -> Borrar líneas de la 17 a la 20
:'a,'b s/hola/adios/g  -> Cambiar cadena en el bloque entre las marcas 'a y 'b
```

El rango "Bloque de texto seleccionado" ('<,'>) aparece automáticamente cuando tenemos seleccionado texto en modo visual y pulsamos ":" para teclear un comando.

Movimiento más avanzado

Hemos dicho que con el modo comando de Vim tenemos muchas más opciones que con el modo inserción (o que en otros editores), pero hasta ahora sólo hemos visto una ínfima parte de las posibilidades de Vim. Como veremos ahora, no tenemos porqué movernos carácter a carácter, línea a línea o página a página. Vamos a poder movernos a la palabra anterior y la siguiente, a cualquier parte del fichero, etc.

Comando	**Significado**
w	Mueve el cursor al principio de la siguiente palabra de la línea actual, o de la siguiente línea si estamos en la última palabra de la línea.
b	Mueve el cursor al principio de la anterior palabra de la línea actual, o de la anterior línea si estamos en la primera palabra de la línea.
e	Igual que **w**, pero coloca el cursor en el último carácter de la siguiente palabra (al final de la palabra en lugar de al principio).

Comando	Significado
ge	Igual que **b**, pero coloca el cursor en el último carácter de la anterior palabra.
W, B, E y gE	Iguales que **w**, **b**, **e** y **ge**, pero con una peculiaridad. En mayúsculas, nos movemos de palabra en palabra considerando como separador de palabra sólo los espacios en blanco y retornos de carro, mientras que en minúsculas, Vim utiliza un modo **inteligente** con más separadores de palabras, como el guión o la barra. Por ejemplo, en el caso de tener la frase **cadena1-cadena2 cadena3** o **cadena1/cadena2 cadena3** con el cursor sobre el primer carácter, **w** avanzaría el cursor hasta primera letra de **cadena2**, mientras que **W** lo avanzaría hasta la primera letra de **cadena3**.
$	Mueve el cursor al final de la línea (equivalente a la tecla Fin).
0	Mueve el cursor al principio de la línea (equivalente a la tecla Inicio).
^	Mueve el cursor al primer carácter no blanco de la línea. Perfecto a la hora de programar, cuando queremos corregir cosas en el código, normalmente indentado con espacios o tabuladores al principio de las líneas.
{	Mueve el cursor al anterior párrafo (o bloque de código).
}	Mueve el cursor al siguiente párrafo (o bloque de código).
f<carácter>	Realiza una búsqueda en la línea actual del carácter indicado. Por ejemplo, **fx** mueve el cursor a la primera aparición del carácter **x** desde la posición actual. Muy útil para ir rápidamente a partes concretas de una línea sin llevar la mano al ratón (por ejemplo, para corregir una **h** que sea un error ortográfico, pulsando **fh**.
F<carácter>	Igual que el comando anterior, pero realizando la búsqueda hacia atrás en la línea actual (empezando desde la posición actual del cursor).
t<carácter> y T<carácter>	Similares a **f<carácter>** y **F<carácter>** salvo que posicionan el cursor en el carácter anterior a la letra buscada.
; y ,	Repiten la ejecución del último comando **f, F, t o T** hacia adelante (;) o hacia atrás (,).
ESC	En el caso de búsquedas **f, F, t o T**, permite cancelar la búsqueda.
%	Al pulsarlo sobre un paréntesis abierto o cerrado **(,)**, corchete abierto o cerrado **[,]** , o llave abierta o cerrada **{, }**, mueve el cursor a la pareja de dicho elemento. Por ejemplo, si estamos programando y queremos saber cuál es el paréntesis que cierra el paréntesis sobre el cual está el cursor, pulsamos % y vim nos lleva directamente a él. Como también funciona con corchetes y llaves, podemos encontrar fácilmente qué llave cierra un bloque de código, o qué if/for/while/loquesea es el que ha abierto una determinada llave de cierre en un programa en C que estemos depurando.
<NUMERO>G	Ir a la línea número NUMERO del documento. Por ejemplo, **100G** nos llevaría a la línea número 100. Es especialmente útil a la hora de programar, cuando tenemos que ir a líneas concretas del programa donde el compilador nos ha reportado errores.

Si no estamos programando pero queremos utilizar números de líneas (porque nos parece más cómodo), podemos hacer uso de las siguientes opciones de modo comando:

Comando	Significado
:set number	Activa la numeración de líneas.
:set nonumber	Desactiva la numeración de líneas.
:set ruler	Activa en la barra de estado una indicación de la columna y fila actual.

Cualquiera de estas opciones las podemos poner en nuestro fichero .vimrc para que se apliquen a todos los documentos que editemos, o cambiarlas en cualquier momento en modo comando.

Comando	Significado
gg	Ir a la primera línea del documento (equivale a **1G**)
G	Sin número delante, G nos lleva a la última línea del documento.
<NUMERO>%	Nos lleva a un porcentaje concreto del fichero. Por ejemplo **50%** nos lleva a la mitad del fichero, y **95%**, casi al final del mismo.
CTRL+F	Scrollea una pantalla completa hacia adelanta (F de Forward).
CTRL+B	Scrollea una pantalla completa hacia atrás (B de Backward).
CTRL+E	Scrollea la pantalla en una sóla línea hacia arriba.
CTRL+Y	Scrollea la pantalla en una sóla línea hacia abajo.
CTRL+U	Scrollea media pantalla de texto hacia abajo (el equivalente a hacer medio RePág). Puede sonar raro el hecho de scrollear **medias pantallas**, pero en determinadas situaciones puede ser útil (si no queremos perder de vista texto ya leído cuando avanzamos, por ejemplo).
CTRL+D	Scrollea media pantalla de texto hacia arriba (como hacer medio AvPág). Permite así avanzar el documento media página sin perder de vista el texto donde está el cursor.
zz	Sin modificar la posición actual del cursor, modifica la **ventana de visualización del fichero** de forma que la línea actual acabe centrada en pantalla y podamos ver el contexto. Por ejemplo, supongamos que estamos en la parte de abajo de la pantalla con el cursor en la última línea y necesitamos ver con facilidad y claridad qué líneas hay sobre y bajo ella. En otros editores usaríamos la tecla de Abajo hasta centrar un poco la línea en pantalla y luego subiríamos hacia arriba para volver a la línea en que estábamos. En Vim basta con pulsar zz para centrar la línea actual en pantalla sin mover la posición del cursor para nada.
zt	Igual que zz pero posicionando la línea en la parte superior de la pantalla (t viene de top) lo que nos permite ver con claridad la línea actual y muchas líneas posteriores.
zb	Igual que zt, pero posicionando la línea en la última posición de la ventana de pantalla, lo que nos permite ver la línea actual y muchas líneas anteriores. En ambos 3 comandos no se modifica la posición del cursor en el documento, sólo la manera de verlo en pantalla.

De nuevo podemos utilizar multiplicadores en todos los comandos anteriores para evitarnos pulsaciones innecesarias de teclas:

Comando	Significado
20w	Avanzar 20 palabras.
3fx	Avanzar el cursor a la tercera aparición de la letra **x** en la línea actual, desde la posición del cursor.

Por último, respecto a comandos de movimiento, existen 3 comandos muy especiales que nos permiten posicionar el cursor al principio, medio y final de la pantalla. Ojo, no principio, medio y final del fichero, sino de la pantalla, de lo que vemos en nuestro monitor:

Comando	Significado
H	Posiciona el cursor al principio de la pantalla (sin hacer scroll de ella).
M	Posiciona el cursor en el centro de la pantalla.

Comando	Significado
L	Posiciona el cursor en la parte baja de la pantalla.

Nótese lo útil que puede ser los comandos **w**, **W**, **b** y **B** para moverse a derecha e izquierda en un párrafo palabra a palabra, a una velocidad mucho más rápida que utilizando los cursores. Y además podemos agregar multiplicadores, de modo que 6w nos moverá el cursor 6 palabras a la derecha, que puede equivaler a ahorrarse 40-50 pulsaciones de cursor o levantar la mano del teclado para llevarla al ratón.

El concepto de "palabra" de Vim es `[a-zA-Z0-9_]`, es decir, que forma parte de una misma palabra todos los carácteres alfanuméricos (a-z y 0-9) además del carácter de subrayado (underscore) "_". Si queremos excluir el subrayado de la consideración de palabra, podemos hacerlo con esta opción del vimrc:

```
set iskeyword-=_
```

Operadores

Existen una serie de comandos en Vim que se comportan como operadores, actuando sobre los comandos de movimiento. Por ejemplo, el operador de borrado **d** (delete), se puede anteponer a comandos de Vim para modificar su comportamiento.

Así, si el comando **w** se mueve hasta la siguiente palabra, el comando **dw**, borra desde la posición del cursor hasta el final la palabra actual y se mueve hasta la siguiente palabra (recordemos que podríamos utilizar **dW** para borrar la palabra completa hasta el siguiente espacio sin contar separadores especiales). De igual forma, **4dw** realiza 4 veces dw, es decir, borra 4 palabras. Nótese que además de **4dw** también podríamos haber escrito **d4w**, que hubiera tenido el mismo efecto. Resumiendo, los 2 comandos siguientes son 2 formas diferentes de hacer lo mismo:

Comando	Significado
4dw	Repetir 4 veces **dw**, es decir, borrar 4 palabras.
d4w	Borrar el resultado de **4w**, es decir, borrar el resultado de moverse 4 palabras.

Así, si el comando **$** nos mueve hasta el final de la línea actual, el comando **d$** borra desde la posición actual del cursor hasta el final de la línea. O, por ejemplo, si el comando **100G** nos lleva a la línea 100, el comando **d100G** borra desde la línea actual hasta la línea 100. Del mismo modo, podemos utilizar **db** o **DB** para borrar la palabra a la izquierda del cursor, o **d^** para borrar desde la posición actual hasta el principio de la línea.

Otro operador interesante es **c**. El operador **c** significa cambio. Se comporta exáctamente igual que **d**, pero al acabar pone el cursor en modo inserción. El sentido es el siguiente: si con **dw** (o **dW**) borramos la palabra actual, con **cw** hacemos lo mismo pero además se pone el editor en modo inserción para que introduzcamos texto, lo que efectivamente resulta en que hemos cambiado la palabra actual. Se le pueden aplicar los mismos modificadores y opciones (como **4cw**, **c100G**, etc). El equivalente de cambio de **dd** (borrar línea completa) es **cc** (cambiar línea completa).

Como algunos modificaciones y operadores se utilizan tanto, Vim nos proporciona unos **atajos** de una sola letra para ejecutarlos:

Atajo	Equivalente	Significado
x	dl	Borrar el carácter bajo el cursor.
X	dh	Borrar el carácter a la izquierda del cursor.
D	d$	Borrar hasta el final de la línea.

Atajo	Equivalente	Significado
C	c$	Cambiar el texto hasta el final de la línea.
s	cl	Cambiar un carácter.
S	cc	Cambiar la línea completa.

Gracias a la potencia de Vim, entre operadores y multiplicadores podemos hacer la edición muchísimo más rápida. Veamos algunos comandos más avanzados:

Comando	Significado
dw	Borrar desde el cursor hasta el final de la palabra actual. Por ejemplo, si estamos encima de la letra **m** de la palabra **automóvil**, ejecutando **dw** quedaría tan sólo la palabra **auto**. Recuerda que **w** avanza hasta el siguiente separador de palabra y **W** hasta el siguiente espacio entre palabras, de modo que también podemos usar **dW** si es lo que nos interesa.
db	Borrar desde el cursor hasta el principio de la palabra actual. Por ejemplo, si estamos encima de la letra **m** de la palabra **automóvil**, ejecutando **db** quedaría tan sólo la palabra **móvil**.
diw	Borrar la palabra bajo el cursor (completa), desde su principio hasta su final, estemos donde estemos dentro de la palabra.
daw	Borrar la palabra bajo el cursor, igual que **diw**, pero en este caso si existe un espacio tras la palabra también lo borra.
dis	Borrar la frase (no línea, sino frase hasta el próximo punto) sobre la que está el cursor.
das	Igual que dis, pero si existe un espacio tras la frase también lo elimina.
dG	Borrar desde la posición actual del cursor hasta el final del fichero.
dgg	Borrar desde la posición actual del cursor hasta el principio del fichero.

Cambiando la letra **d** por una **c**, los comandos anteriores se transforman en comandos de cambio, pasando a modo inserción tras ser ejecutados.

Soy consciente de que habrá gente que en este punto dirá """*bueno, yo para borrar una palabra no me voy a 'aprender' un comando, para eso la borro a mano*""". Así pensé yo también al leer por primera vez el manual de Vim.

Lo que ocurre después es que las primeras semanas que usas Vim, para borrar una palabra entras en modo edición y usas Supr o la tecla de borrar. Pasado un tiempo, te das cuenta de lo cómodo que es usar **x** directamente en modo comando (pudiendo deshacer así cualquier borrado parcial con **u**). En algún momento tras algo más de tiempo, usarás **dw** y borrarás la palabra completa, y finalmente cuando te sientes ante cualquier otro editor te sentiras totalmente limitado de no poder hacer un **4dw** para borrar cuatro palabras de golpe.

¿Qué tal borrar desde la posición actual del cursor hasta el final del párrafo o bloque de código con **d}**? ¿O borrar desde la posición del cursor hasta la aparición de la cadena XYZ con **d/XYZ**? Las posibilidades son infinitas.

El comando punto "."

Este comando sirve para **repetir el último comando que haya producido un cambio en el documento**. Es decir, si lo último que ejecutamos fue un comando **dd** (borrar línea), con el comando **.** lo repetimos. Si fue un comando **dw** (borrar palabra), con el punto repetimos la ejecución (pero esta vez sobre la palabra actual).

El operador punto es extremadamente útil, y podemos verlo con un ejemplo (presente en el manual de Vim). Supongamos que estamos editando HTML (HyperText Markup Language) y queremos borrar algunas imágenes de la página (no todas, sólo algunas). Habría que buscar algunos **<img src="imagen" alt="texto">** y borrarlos. Supongamos que buscamos con / la cadena **/<img** y con ello ponemos el cursor sobre el símbolo **<** de **<img**. En un editor normal tendríamos que movernos con los cursores (o el ratón) para seleccionar el tag html completo o borrarlo con la tecla de borrar o de suprimir. En Vim la cosa se resume en:

```
/<img      <--- Buscar <img y poner el cursor sobre el "<"
df>        <--- Borrar desde el cursor hasta el primer ">", incluídos ambos
```

Recordemos que el comando **fCARACTER** busca la primera aparición del carácter indicado a partir de la posición actual del cursor, de modo que en si estamos posicionados en el **<** de <img src="imagen" alt="texto">, la búsqueda **f>** buscará la primera aparición de **>**, es decir, el símbolo del final del tag HTML (HyperText Markup Language) de img. Con el modificador **d** que le hemos puesto delante, le estamos diciendo que borre ese texto desde la posición actual del cursor hasta el final del tag. Así, **df>** elimina el tag <img> completo, tenga la cantidad de letras y texto que tenga. Lo hace vim sólo, de forma automática, porque nosotros le hemos dicho **borra desde la posición actual hasta el primer > que encuentres**.

Pues bien, una vez borrado este primer tag IMG con 2 simples comandos (buscar y borrar), si pulsamos **n** nos iremos al siguiente tag img . Si nos interesa borrar ese tag que hemos encontrado, le damos al punto **.**, y repetimos la última operación anterior que modifica el documento, es decir, el **df<**. Si no nos interesa, le damos a **n** de nuevo y pasamos al siguiente. ¿Es o no es útil?

Así, podemos saltar de un tag img a otro con "n" y, o bien borrarlo con "." si nos interesa borrarlo, o saltarlo pulsando de nuevo "n" para ir al siguiente match de la búsqueda. Esto es infinitamente más rápido y seguro (a prueba de errores) que realizar la operación manualmente.

Ahora como segundo ejemplo, supongamos que no queremos borrar un tag <img> sino un enlace <a>. El ejemplo anterior no nos vale exáctamente para <a> porque el tag completo no tiene un símbolo **>** sino dos, y **df>** sólo borraría hasta el primero:

```
<a href="http://enlace"> texto del enlace </a>
```

En este caso sólo tenemos que aprovecharnos de los multiplicadores: Si no estamos buscando el primer símbolo **>**, sino en segundo, todo se reduce a:

```
/<a href     <--- Buscar la cadena "<a href" y posicionar el cursor en "<"
d2f>         <--- Borrar desde la posición actual hasta el segundo ">" encontrado
```

Es decir, en lugar de borrar el resultado de **f>** ejecutamos el borrado de **2f>**, que es repetir 2 veces el **f>** con lo que encontramos el segundo cierre de tag.

También, en lugar de 'f', podemos utilizar 't' para realizar la misma búsqueda pero hasta la posición anterior al carácter buscado. De esta forma, supongamos que queremos borrar en el siguiente ejemplo todo desde la posición actual del cursor (en la "M" en nuestro ejemplo) hasta el paréntesis:

```
void MiFuncion_larga( int x );
```

En ese caso, bastará con pulsar **dt(** para borrar hasta la "a" anterior al paréntesis, dejando el nombre de la función listo para ser tecleado de nuevo.

Pensad por un momento la diferencia entre un editor convencional y vim con lo que hemos visto hasta ahora. Alguien podría pensar **es que es complicado tanto comando**. Piensa que no los aprendes de memoria, sino con el uso (y casi por lógica, por ejemplo, dw = delete word).

Copiar y pegar

Cuando borramos texto (con **dd**, **dw**, **x** o similares), dicho texto (líneas, palabras o incluso un simple carácter) se almacena en un buffer interno. Digamos que no se borra sino que se **corta**. Podemos pegar el último texto borrado utilizando el comando **p**. Esta es la primera lección de este capítulo: p = paste = pegar. Cabe destacar que una línea cortada con **dd**, al ser pegada con **p** será insertada debajo de la línea actual del cursor. Si lo que pegamos no es una línea completa sino una porción de texto, entonces será insertado a la derecha de la posición actual del cursor.

Existe una variante de **p** que es **P**, cuya diferencia es que pega el texto a la izquierda de la posición actual del cursor para porciones de texto, o en la línea sobre el cursor para líneas completas.

Como siempre, podemos aprovechar los multiplicadores para ahorrarnos trabajo: Con **dd** podemos cortar una línea y con, por ejemplo, **10p** podemos pegar 10 copias de la linea cortada.

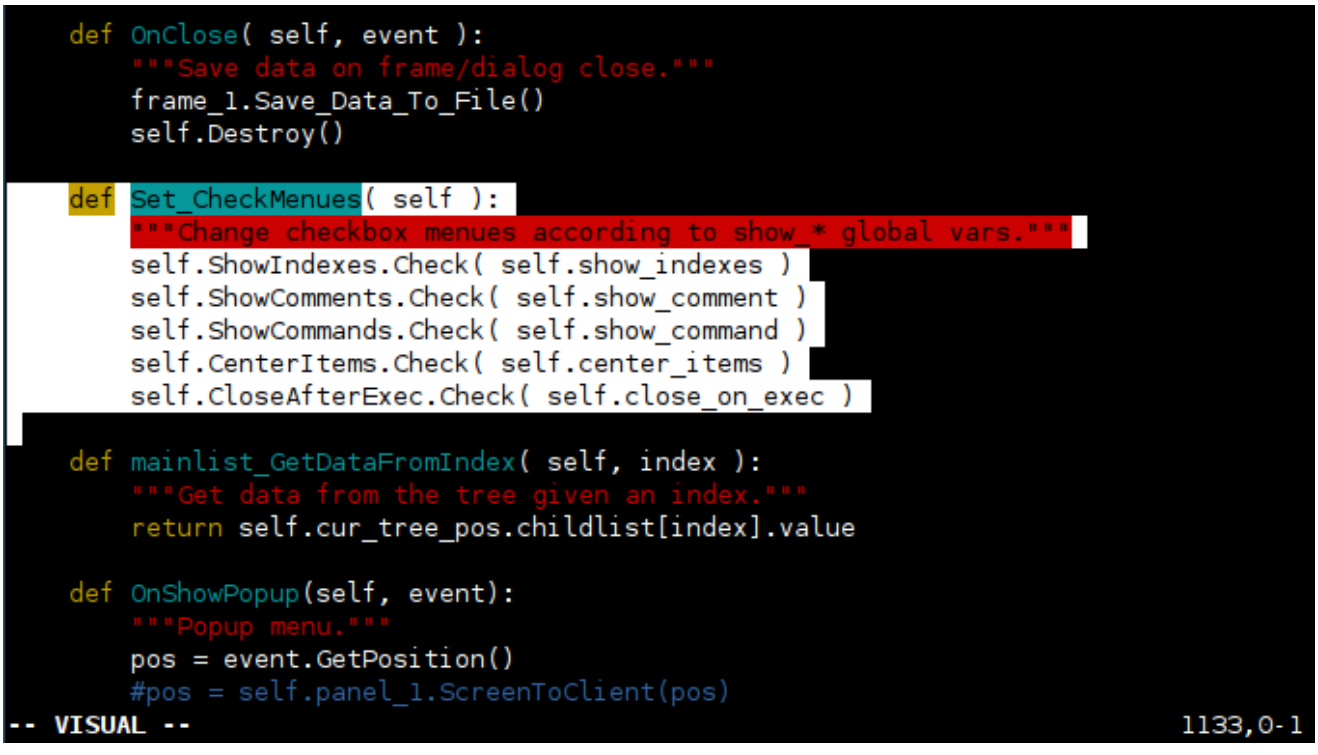
En general, **dd** y **p** (para una sóla línea) o **<NUMERO>dd** y **p** (para múltiples líneas) pueden ser utilizados para mover bloques de texto de un lugar a otro (copiándolos y pegándolos).

Finalmente, comentar que existe un comando especial llamado **]p** que a la hora de pegar tiene en cuenta la indentación del código por lo que permite pegar código en diferentes niveles de indentación del que lo hemos cortado, sin tener que reajustar todas las líneas.

Seleccionar: el modo visual

Aparte de poder pegar texto cortado con comandos, en Vim podemos seleccionar texto al estilo de lo que se puede hacer en otros editores. Si pulsamos la tecla **v** pasaremos a modo visual, donde con los cursores (o las teclas de movimiento de vim) extendemos el área de selección para después operar con ella. Nos posicionamos en la primera letra de lo que queremos seleccionar (o la última), y usamos arriba, abajo, izquierda y derecha para hacerlo, y podemos cancelar la selección en cualquier momento pulsando ESCAPE.

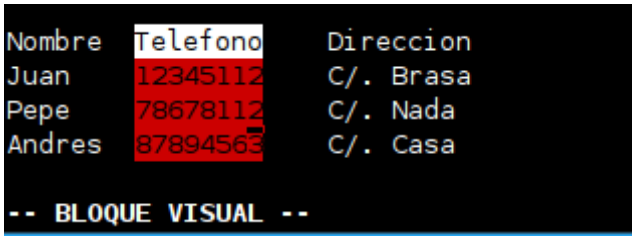
Existen 2 variantes más del modo visual para seleccionar texto: la primera es **V** (v mayúscula), que trabaja sólo con selección de líneas completas (usando las teclas de arriba y abajo). Es decir, si pulsamos **v** (minúscula) en medio de una frase, podremos mover la selección a derecha o izquierda para coger palabras sueltas (y también frases con arriba y abajo), mientras que **V** (mayúscula) sólo trabaja con frases completas.



La segunda variante son las selecciones de bloques o selecciones verticales. Supongamos que tenemos una tabla como la siguiente:

```
Nombre  Telefono    Direccion
Juan    12345112    C/. Brasa
Pepe    78678112    C/. Nada
Andres  87894563    C/. Casa
```

Pues bien, si quisieramos borrar la columna teléfono completa, en muchos editores que no disponen de selecciones verticales tendríamos que ir línea a línea borrando **Telefono**, **12345112**, **7867811** y **87894563**. En Vim podemos hacer una selección vertical de un bloque que comprenda justo esa columna y trabajar sobre ella: nos posicionamos sobre la T de **Teléfono** y pulsamos **CTRL+V**, con lo que pasamos a modo de edición de bloques. Usando los cursores o las teclas de movimiento seleccionamos la columna, y ya podemos trabajar sobre ella (cortarla, copiarla, etc).



Nótese que Ctrl+V es, en Vim para Windows, el atajo para pegar texto desde el portapapeles, por lo que en Windows puede ser necesario remapear esta tecla a otro atajo de teclado (con **map**, como veremos más adelante).

Algo muy interesante de cuando estamos en modo visual es que podemos usar las teclas de movimiento especiales. Si por ejemplo pulsamos **w**, la selección avanza una palabra completa. Si pulsamos **as** (a sentence), la selección avanza una frase entera (hasta el próximo punto o separador), y podemos repetir los comandos que deseemos y combinarlos hasta seleccionar el texto deseado.

Es decir, que lo bueno del modo visual es que podemos usar los mismos "verbos" de movimiento que usa Vim en modo comando: 0, $, G, gg, fCHAR, /string, etc. Además, podemos usar verbos espaciales para seleccionar bloques delimitados por comillas, paréntesis, corchetes, etc:

Comando	Resultado de la sección
iX	Seleccionar un bloque entero de texto basado en X, donde X puede ser: **w** o **W** palabra completa **s** ⇒ sentence (frase) **p** ⇒ párrafo **b** or **(** ⇒ bloque de paréntesis (contenido entre (y)). **B** o **{** ⇒ bloque de { llaves } **t** ⇒ contenido de un <tag> y </tag> **<** ⇒ un bloque < y > block **[** ⇒ un bloque [y] **"** o **'** ⇒ una cadena (bloque entre delimitador de cadenas) NOTA: Los caracteres de cierre también funcionan, como),], etc.

Un pequeño apunte (no muy utilizado) sobre la selección de texto: si estamos seleccionando texto y vemos que queremos modificar el INICIO de la selección, podemos pulsar **o** para cambiar entre los 2 límites de la selección y cambiar uno u otro. La versión mayúscula, **O** se utiliza para alternar entre las 4 esquinas de las selecciones verticales de CTRL+V.

Copiar, cortar y pegar

Una vez tenemos el texto seleccionado (de cualquiera de las 3 formas descritas), podemos borrarlo, cortarlo y copiarlo. Estando en modo visual, con el texto sobre el que queremos actuar marcado, podemos copiarlo pulsando **y** (de yank) y cortarlo, ya sea con **d**, **x** y **c**. La diferencia entre estos 3 modos de copiar está en que **d** y **x** se mantienen en modo comando tras cortar el texto, mientras que **c** (que recordemos que es modificar), se pasa a modo inserción tras hacerlo. Por supuesto, si nos arrepentimos del cortado podemos pulsar **u** (undo) para deshacerlo.

Recordemos que en cualquier momento podemos volver a pegar un texto copiado o borrado usando **p**.

Así pues, mediante **v**, **y**, **d** y **p** se realizan todas las operaciones de selección, copiado, borrado/cortado y pegado, respectivamente.

En el caso concreto de **y**, dado que es un operador podemos anteponerlo a otros comandos de Vim. Por ejemplo, **yw** copia una palabra completa, y **y5w** copia las siguientes 5 palabras completas en el buffer de memoria. Y, para finalizar, igual que ocurre con **d** y **dd**, duplicando la **y** como **yy** copiamos a memoria la línea actual completa (sin necesidad de seleccionarla). Utilizando multiplicadores, podemos por ejemplo copiar la línea actual y 3 más mediante **4yy**.

Pegar texto con autoindent activado

Si tenemos activado el modo de autoindentación (**set ai** o **set autoindent**), cada vez que pulsamos ENTER, el cursor se indenta de forma automática al nivel (columna) de la línea en que lo hemos pulsado. Esto, que resulta muy útil programando, puede ser problemático para pegar texto con múltiples líneas, ya que cada retorno de carro del texto pegado provocará que la línea empiece al nivel de la anterior, y que además se sumen los espacios y tabuladores de lo que estamos pegando. El resultado es el siguiente:

Copiamos el siguiente texto (con el ratón):

```
    result = dlg.ShowModal()
    dlg.Destroy()
    if result == wx.ID_YES:
        del frame_1.macros[index]
        frame_1.Save_Data_To_File()
    elif result == wx.ID_NO:
        pass
    dlg.Destroy()
```

Y lo pegamos en otra parte del documento en modo inserción con el botón central del ratón y lo que obtenemos es lo siguiente:

```
            result = dlg.ShowModal()
                dlg.Destroy()
                    if result == wx.ID_YES:
                            del frame_1.macros[index]
                                    frame_1.Save_Data_To_File()
                                        elif result == wx.ID_NO:
                                                pass
                                                    dlg.Destroy()
```

Como puede verse, cada enter recibido ha indentado la línea al nivel de la línea actual, sumando además los espacios que ya tenía el texto que se está pegando, resultando en ese horror.

Esto sólo sucede si pegamos texto "con el ratón", sin usar los comandos de copiado y pegado de Vim. Para pegar texto "con el ratón" y que esto no suceda, es recomendable desactivar el modo autoindent bien con "**set paste**" o "**set noai**".

Tuberías (pipes) para filtrar texto

Cuando tenemos texto seleccionado tanto en modo normal como en modo visual, podemos pasar ese texto a través de cualquier programa externo para filtrarlo. Por ejemplo, supongamos que tenemos un programa que acepta cualquier texto por entrada estándar y nos saca el texto modificado (ordenado alfabéticamente, cifrado, o cualquier otra operación) por la salida estándar. En ese caso, si queremos manipular un párrafo de nuestro fichero podemos seleccionarlo (con 'v' en modo visual, por ejemplo), y mientras está el párrafo seleccionado, pulsamos:

```
:!programa
```

Por ejemplo, supongamos que queremos utilizar el comando **sort** de UNIX para ordenar alfabéticamente las diferentes líneas de un párrafo. Seleccionamos el párrafo en cuestión con 'v' y los cursores y pulsamos:

```
:!sort
```

El texto seleccionado será enviado al comando **sort** por entrada estándar y será reemplazado por la salida de la ejecución de sort. De la misma forma podemos ordenar alfabéticamente el fichero entero, seleccionándolo todo:

```
1G
v
gg
:!sort
```

O, lo que es lo mismo:

- 1G = ir a la primera línea del fichero
- v = ir a modo visual
- gg = llevar al cursor al final del fichero (seleccionando todo el fichero)
- :!sort = pasar el texto seleccionado (todo el fichero) al comando sort, y reemplazarlo por la salida de la ejecución del mismo.

O, más sencillo aún:

```
:%!sort
```

(% representa a una selección del fichero completo)

El filtrado (pipe a programa externo) nos permite muchas cosas: cifrar texto (llamando a pgp/gpg), pasárselo a programas externos que lo manipulen, etc.

Otro ejemplo de uso de los filtros es el formateo y justificación de texto. Si tenemos instalado el comando **par** (un programa de Linux para formatear párrafos), podemos seleccionar texto en modo visual y filtrarlo a través de par mediante, por ejemplo:

```
:!par 72
```

La salida de **par 72** (si tenemos instalado par en el sistema, claro) consiste en justificar el texto a 72 columnas, cosa que nos puede ser útil en determinadas circunstancias de edición de textos.

Nótese que también podemos aplicar filtros a rangos de líneas y al texto delimitado por 2 marcas (posteriormente veremos cómo establecerlas):

```
:'<,'> !sort   -> Sobre la selección
:'t,'b !sort   -> Sobre el texto entre 2 marcas
:N,M !sort     -> Entre 2 rangos de líneas (N a M)
```

Insertar ficheros y salida de comandos

Podemos insertar el contenido de un fichero de texto en la posición actual del cursor mediante el comando **:r**. Tan sólo deberemos especificar el fichero a insertar (con su ruta si es necesario):

```
:r fichero
```

El comando **:r** nos permite también insertar la salida (el resultado de la ejecución) de comandos del sistema en nuestro documento. Por ejemplo, si queremos insertar la salida del comando uptime en la posición actual del cursor, pasamos a modo comando (ESC) y ejecutamos:

```
:r !uptime
```

La diferencia entre este comando y el anterior es el símbolo de admiración cerrada '!', que indica **ejecución**.

Múltiples "portapapeles": los registros de borrado

Como ya hemos visto, en Vim podemos seleccionar texto con el modo visual usando **v** para movimiento carácter a carácter o bien **V** para líneas completas. Una vez seleccionado, lo podemos copiar con **y**, cortar con **c** y borrar con **d**.

De la misma forma, fuera del modo visual (en modo comando) con **dd** borramos una línea entera, con **3dd** 3 líneas enteras, con **dw** una palabra entera, con **x** un cáracter, con **d0** desde el cursor al principio de línea, con **d$** desde el cursor al final de línea, con **d/cadena** desde el cursor hasta la aparición de "cadena", etc. Todo ellos son combinaciones de repetidores, el comando de borrado **d** y destinos (w=word, $=end of line, /cadena=primer resultado de la búsqueda, etc).

Por otra parte, los comandos básicos para pegar en Vim son **p** para pegar en justo detras de la posición del cursor, y **P** para pegar antes de la posición del cursor.

Vim autodetecta si el texto que tenemos en su buffer interno, el que estamos pegando, lo habíamos copiado carácter a carácter o bien mediante líneas completas y utiliza esto para pegar dicho texto bien a la derecha/izquierda del cursor, o bien arriba/abajo de la línea actual. Si copiamos una porción de código, la pegará "dentro" de la línea actual mientras que si copiamos líneas completas la pegará arriba/abajo de la línea actual.

Además de pegar antes o después del cursor, también podemos usar **p** cuando tenemos realizada una selección visual, de forma que nuestro texto pegado reemplazará al texto seleccionado.

Ahora bien: estos copiados y pegados tienen como destino y origen un buffer interno de Vim, y no el portapapeles del sistema que usan típicamente otras aplicaciones.

Existen **26 registros** (26 portapapeles diferentes) donde podemos copiar texto y pegar desde ellos. Sí, en lugar de tener un único "portapapeles", tenemos 26 diferentes (de la **a** a la **z**) donde podemos copiar texto para usarlo después. Esto es extremadamente útil cuando queremos cortar varias porciones de texto de un fichero para agruparlos en una posición final, ya que podríamos usar los registros a, b, c y d (por ejemplo) para almacenar 4 bloques de texto y después pegarlos juntos, evitando subir y bajar continuamente en el fichero para ir llevando cada bloque a su lugar destino.

Esto es extremadamente útil porque tenemos 26 buffers donde copiar y pegar texto y de esta forma podemos copiar varios bloques de texto sin perder copias anteriores. Cuando no indicamos un buffer específico (cuando usamos **yy** o **dd**, por ejemplo), estamos usando el Unnamed Buffer (el buffer sin nombre), pero podemos hacer

referencia a cualquiera de los 26 registros disponibles.

Los registros se referencian con las comillas dobles, de forma que tenemos desde **"a** hasta **"z**. Usando el nombre del registro como prefijo, hacemos que **y**, **yy**, **d**, **dd**, **p**, etc utilicen dicho registro:

- **"ayy** –> Copiar línea de texto actual en el registro a.
- **"add** –> Borrar línea de texto actual y poner su contenido en el registro a.
- **"ap** –> Pegar el contenido del registro a.

Resumiendo:

- **"x<operacion>** –> Realizar <operación> con registro x.

Esto también funciona con selecciones visuales. Si seleccionamos texto en modo visual y pulsamos **"ac**, cortaremos el texto seleccionado al registro a.

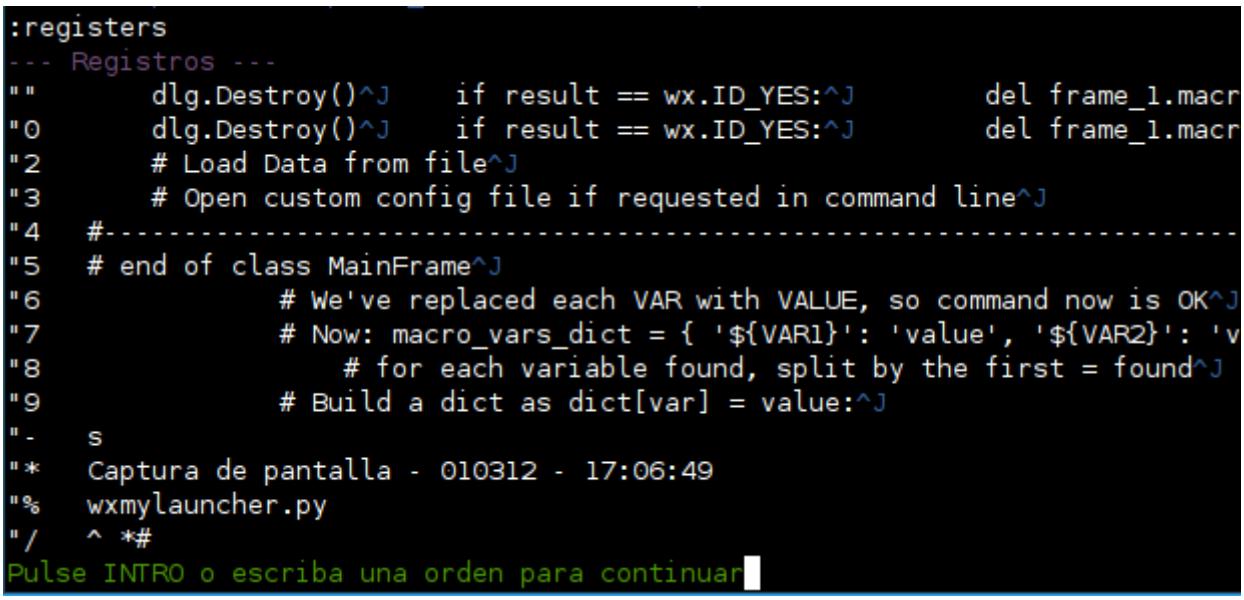
Así pues, tenemos 26 posibles registros los cuales, además, **no se pierden al cerrar la sesión de Vim** siempre y cuando tengamos Vim en modo "nocompatible" en el vimrc.

Existen también una serie de registros "especiales":

- Registro **"_** –> El registro _ (subrayado) es el registro "black hole" (agujero negro). Lo que escribamos en él no se guarda, permite borrar de verdad texto. Es muy útil porque nos permite borrar texto sin eliminar lo que tengamos en el buffer de copiado y pegado. Si tenemos algo en el portapapeles (seleccionado y copiado con **y**) y queremos borrar una línea con **dd** sin perder el contenido actual del buffer, podemos hacer **"_dd**.
- Registro **"0** –> Contiene el texto del comando **y** (yank) más reciente, de modo que **"0p** pegará el último texto copiado incluso si hemos utilizado un comando de borrado después del "y".
- Registros **"1** a **"9** –> Guardan de forma rotatoria los últimos 9 borrados (siendo el 1 el más reciente).
- Registros **"*** y **"+** –> Se refieren al portapapeles del sistema Windows (*) y X11 (+) (sí, es posible copiar y pegar en el portapapeles del S.O. desde comandos de Vim).
- Registro **""** –> El "Unnamed Register" (registro sin nombre). Es el registro en el que acaba el texto que acabamos de copiar o cortar.

Un apunte sobre el registro **"0** (y su hermano el registro **"-**): cuando copiamos, cortamos, o incluso cuando borramos (ya sean caracteres o líneas completas), Vim guarda el texto en estos 2 buffers de forma que incluso si hemos borrado después algo con **d**, seguimos teniendo acceso al "yank" anterior. Si el texto es menor de una línea, se guarda en un registro llamado **"-**. Para más de una línea, se almacena en **"0**. En otras palabras, un "delete" borrará el texto pero no afectará a **"0**.

Podemos ver el contenido de los diferentes "registros" temporales con el comando **:registers** y recuperar el contenido de cualquiera de ellos en el documento con **"<REGISTRO>p**, como: **"6p**.



De esta forma, podemos acceder a contenidos anteriormente borrados sin tener que deshacer múltiples pasos (que puede implicar deshacer también cambios correctos que hicimos entre esos pasos).

Finalmente, como si los registros de Vim no fueran ya de por si suficientemente potentes, podemos **añadir** texto al contenido de un registro usándolo en mayúsculas:

- **"Bdd** –> Delete current line and append it to the content of the register **"b**.

Es muy importante entender cómo funcionan los registros de copiado y pegado no sólo por la potencia que dan, sino porque nos podamos encontrar con situaciones extrañas (con explicación lógica, como veremos):

Supongamos que queremos copiar una línea, ir a otra parte del documento, borrar una línea y pegar el texto copiado. Supongamos que hacemos lo siguiente:

- Vamos a la línea que queremos copiar y ejecutamos **yy** para copiarla (en el Unnamed Buffer).
- Vamos a la parte del documento donde está la línea a borrar y pulsamos **dd** para borrarla.
- Pulsamos **p** para pegar la línea que copiamos.
- De repente, en vez de aparecer la línea que habíamos copiado, se pega la que acabamos de borrar.

Esto ocurre porque el comando **dd** borra la línea actual y la copia en el Unnamed Buffer, con lo que perdemos el contenido que había antes en dicho buffer (la línea que copiamos inicialmente).

Para empezar, si nos acaba de ocurrir, simplemente basta con pegar nuestra línea copiada original con **"0p**, porque como ya dijimos, el registro 0 contiene el último copiado realizado con Comandos Yank "y" (y dicho buffer 0 no es sobreescrito por el **dd**).

Una vez esto nos ha ocurrido y ya sabemos por qué es, basta para la próxima vez con borrar la línea a eliminar usando el black-hole register como destino, con **"_dd**. Así borramos la línea pero no la guardamos en el Unnamed Register (no perdemos lo que habíamos copiado).

Finalmente, destacar que podemos especificar valores para alguno de los registros tanto en el fichero `.vimrc` como en comando de Vim con:

```
let @a = "My custom buffer content"
```

E incluso podemos utilizar el contenido de los registros en operaciones de sustitución u otros comandos mediante `=@a` :

```
:%s/PATTERN/\=@a/g
```

(Cambiaría todas las apariciones de `PATTERN` por el contenido del registro `a`).

Aunque todo esto pueda parecer complicado, todo se resume en:

- Existen 26 registros que se referencian con comida doble seguido de a-z: Ej: **"a**.
- Podemos aplicar las operaciones de yank, cut, delete y paste a un registro escribiendolo delante de la operación: Ej: **"ap** o **"bdd**.
- Existen registros especiales como un agujero negro (_), la última copia realizada (0), el portapapeles del sistema (*)…
- Usar un registro en mayúsculas permite AGREGAR texto al registro, sin sobreescribirlo.

El portapapeles del sistema

En determinados sistemas existe un portapapeles del sistema que es independiente del que usa Vim internamente. Si Vim está compilado para soportar el acceso al portapapeles del sistema podemos, como ya hemos visto, copiar cosas en él y pegar cosas desde él. Los comandos son los mismos, **y** y **p**, pero anteponiendo unas comillas dobles y un símbolo asterisco: **"***. Así, quedaría **"*y** para copiar una selección de texto (o yy para la línea actual) y **"*p** para pegarlo.

En otras palabras, el portapapeles del sistema es uno de los registros internos que acabamos de ver, concretamente, **"***.

Búsquedas y reemplazos

Ahora que ya nos manejamos en la inserción y modificación de texto toca tratar el tema de las búsquedas de texto. Buscar texto en un editor de textos significa que queremos llevar el cursor desde la palabra en que estemos hacia la primera ocurrencia de una determinada cadena de texto o palabra que esté por debajo de nuestra posición actual cuando buscamos hacia adelante, y por encima de nuestra posicio actual cuando buscamos hacia atrás.

En Vim existen 2 comandos específicos para buscar hacia adelante y hacia atrás en el documento:

Comando	Resultado
/CADENA	Buscar la primera apación de CADENA por debajo de la posición actual del cursor (búsqueda hacia adelante). La barra **/** es la que aparece sobre la tecla **7** del teclado (no la barra inversa), y que se introduce mediante SHIFT+7.
? CADENA	Buscar la primera aparición de CADENA por encima de la posición actual del cursor (búsqueda hacia atrás).

Así, si ejecutamos el comando **/prueba**, el cursor se posicionará en la primera ocurrencia de prueba en el texto que aparezca tras la posición actual del cursor, mientras que si utilizamos el comando **?prueba** estaremos buscando hacia arriba en el documento.

Una vez realizamos una búsqueda, podemos repetir la misma utilizando los comandos n y N, de forma que:

Comando	Significado
n	Buscar la siguiente aparición de la cadena hacia adelante (sin tener que repetir el comando **/CADENA**).
N	Buscar la anterior aparición de la cadena hacia atrás (sin tener que repetir el comando **?CADENA**).

Una cosa muy interesante de las búsquedas en Vim es que tenemos un historial de las mismas. Si hemos ejecutado varios comandos de búsqueda, si tecleamos la / en modo comando y usamos los cursores arriba y abajo, podemos acceder a las últimas búsquedas realizadas, y volver a ejecutarlas pulsando Enter. Si además de escribir la **/** añadimos algún carácter más antes de pulsar arriba o abajo, los cursores sólo harán **historial** entre aquellas búsquedas que comiencen exactamente por los caracteres que hemos introducido. Como un ejemplo, supongamos que mientras editabamos un fichero hemos realizado las siguientes búsquedas:

```
/casa
/camión
/moto
```

Si pulsamos / y usamos la tecla de cursor arriba varias veces, iremos **scrolleando** entre /moto, /camión y /casa. Por contra, si hubieramos tecleado **/ca** antes de pulsar arriba, sólo se nos ofrecerían las opciones de /camión y /casa. De esta forma, repetir búsquedas anteriores largas o complejas es muy sencillo, no como en otros editores que sólo recuerdan la última búsqueda realizada.

Cabe decir que no sólo las búsquedas tienen un historial: los comandos que comienzan por el carácter **:** también tienen su propio historial que se maneja con los cursores arriba y abajo, y que es independiente del de búsquedas. Es más, Vim se acordará de la última búsqueda o comando **:** realizado en una sesión anterior de vim

(por ejemplo, ¡una búsqueda que realizamos ayer!).

Existe otra manera sencilla de realizar búsquedas sin tener que teclear prácticamente nada:

Comando	Significado
*	Realizar una búsqueda hacia adelante de la palabra sobre la cual está el cursor.
#	Realizar una búsqueda hacia atrás de la palabra sobre la cual está el cursor.

Diferenciar Mayúsculas y minúsculas

Vim distingue las mayúsculas de las minúsculas al realizar búsquedas. Si estamos buscando la palabra **casa** pero en el texto aparece como **Casa**, Vim no encontrará esa ocurrencia. Al igual que en otros editores y procesadores de texto, podemos decirle a Vim que ignore si las letras son mayúsculas o mínusculas en las búsquedas:

Comando	Significado
:set ignorecase	Para que se ignoren las diferencias entre mayúsculas y minúsculas.
:set_noignorecase	Para que no se ignoren las diferencias entre letras.

Si tenemos activo **set_noignorecase**, podemos usar **\c** en una búsqueda para que se ignoren mayúsculas y minúsculas en esa búsqueda:

- **/\c<string>**
- **?\c<string>**

Resaltado de las búsquedas

Cuando realizamos la búsqueda de una palabra, podemos hacer que los resultados de la búsqueda queda resaltados en un color diferente o no. Esto se hace mediante los siguientes comandos:

Comando	Significado
:set hlsearch	Todas las ocurrencias de la búsqueda se resaltan.
:set nohlsearch	Desactivar el modo de resaltado de ocurrencias.
:nohlsearch	Desactivar el resaltado de ocurrencias para la última búsqueda (pero no para las próximas que se hagan).

Asímismo, otras 2 opciones interesantes son:

Comando	Significado
:set incsearch	Búsqueda incremental: Vim irá buscando cadenas conforme la vayamos tecleando (puede sernos interesante).

Comando	Significado
:set nowrapscan	Cuando Vim busca, si llega al final del fichero continúa por el principio (y viceversa en búsquedas hacia arriba). Con esta opción le decimos a Vim que cuando llegue al final o principio del fichero pare la búsqueda.

Si alguna de estas opciones nos parece interesante como opción por defecto la podemos añadir a nuestro .vimrc personal. Por ejemplo, si nos gusta que todas las búsquedas sean resaltadas e incrementales, editamos nuestro .vimrc y le añadimos:

```
set hlsearch
set incsearch
```

Nótese como en el fichero .vimrc no es necesario poner los dos puntos **:** antes del **set**.

Expresiones regulares

Es cierto que cuando utilizamos la búsqueda con la barra o el interrogante (/ o ?) basta con escribir una palabra o frase para buscarla en el texto, pero la realidad es que Vim nos ofrece mucho más. La cadena de búsqueda que le podemos pasar a Vim es en realidad una expresión regular, es decir, permite que especifiquemos en la cadena ciertas expresiones con un significado especial diferente del literal que hemos escrito.

Por ejemplo, cuando utilizamos el carácter punto (**.**) en una cadena búsqueda (como por ejemplo **/cas.**) , para Vim dicho carácter tiene un significado especial, que en este caso es **cualquier carácter**. Así, buscando **/cas.** (barra, c, a, s, punto), vim encontrará ocurrencias como **casa**, **caso**, **case**, etc. Es decir, el punto es un comodín que significa **cualquier carácter**. Esto nos da mucho juego a la hora de buscar palabras determinadas en el texto si no sabemos exáctamente que puede haber en una posición concreta de la palabra. Obviamente, podemos poner más de un punto en nuestra expresión regular.

También podemos usar **.*** para matchear una cantidad indeterminada de caracteres, como en **/c.*a**, que matcheará tanto "casa" como "cuchara".

Veamos otro ejemplo: los caracteres `^` y `$` , que significan **principio de línea** y **fin de línea** respectivamente. Según la búsqueda que realicemos obtendremos los siguientes resultados:

Búsqueda	Resultado
/ ^ vaca	Vim encontrará todas aquellas palabras que comiencen por **vaca** y que estén al principio de la línea. Literalmente le hemos dicho a Vim que busque una cadena que sea **principio de línea seguido de la palabra vaca**.
/vaca$	Vim encontrará todas aquellas palabras que contengan la cadena **vaca** seguida de un fin de línea. Es decir. encontraría **vaca** o **Caravaca** si alguna de ellas es la última palabra de la línea.
/ ^ vaca$	Vim encontrará sólo ocurrencias en líneas que sólo contengan la palabra vaca. Es decir, una línea que sea **principio de línea, vaca, fin de línea**.

Como puede verse, `^` y `$` son muy útiles a la hora de hacer búsquedas.

Caracteres especiales a escapar

Debido a que lo que buscamos son expresiones regulares y no simples cadenas, hay caracteres que tienen un significado especial y que no pueden ser buscados directamente. Por ejemplo, el carácter **$** significa **fin de línea**, y si ejecutamos **/$**, no encontraremos la primera ocurrencia del carácter **$** sino que nos posicionará el cursor en el primer fin de línea a partir de la posición actual del cursor.

Si queremos realizar búsquedas que incluyan los caracteres especiales ". * ^ $" tenemos que escaparlos con la barra inversa (****), que para Vim es un indicador de **el próximo carácter que voy a introducir interprétalo como un caracter normal y no como un carácter especial**.

Así, para buscar la cadena **Yo tengo 1$ en el banco** hacia adelante en Vim, ejecutaremos:

```
/Yo tengo 1\$ en el banco
```

Del mismo modo, si necesitamos buscar la cadena **/prueba.** pero no queremos que Vim encuentre **/pruebas** (recordad que el punto es un carácter especial que se sustituye por cualquier carácter a la hora de buscar), sino que queremos, literalmente, encontrar las apariciones de la cadena **prueba** seguida de un punto, debemos escapar el punto, mediante: **/prueba\.** (barra prueba barra_inversa punto). La barra inversa le quita al punto el significado especial y Vim entiende que debe buscar el carácter punto.

Los caracteres especiales son los siguientes: **`.*[]^/\?~$`** .

Curiosamente, otros caracteres especiales en expresiones regulares se tratan en las búsquedas y sustituciones de Vim como caracteres literales (al contrario que sucede en las expresiones regulares), por lo que es necesario escaparlos para que funcionen con su significado especial.

Un ejemplo de esto son los caracteres de inicio y fin de palabra "<" y ">". Por defecto, si los buscamos literalmente, Vim los buscará como dichos caracteres. Es decir, si buscamos "/<vaca>", buscará literalmente "<vaca>". Si los escapamos con \ (como **/\<vaca\>**) entonces buscará todas las cadenas vaca que empiecen (<) y acaben (>) como palabra completa.

Lo mismo ocurre con otros caracteres como los paréntesis, +, etc, como en la siguiente expresión que busca todos los números enteros (\d+ encerrado entre < y >):

`/\<\(\d\+\)\>`

Explicar expresiones regulares a fondo queda fuera del ámbito de este documento (donde sólo tratamos lo básico para trabajar con Vim), pero puedes aprender mucho más sobre el tema en Internet mediante cualquier buscador (por ejemplo, buscando en Google por **vim regular expressions**, en condiciones normales os debe llevar a páginas que tratan las expresiones regulares en Vim con más detalle).

URLs de interés:

- http://vim.wikia.com/wiki/Search_patterns (http://vim.wikia.com/wiki/Search_patterns)
- http://vimregex.com/ (http://vimregex.com/)

También tenemos la posibilidad de añadir al principio de una búsqueda en flag **\V** el cual trata la string a continuación como una "raw string", donde no se interpretan los caracteres especiales de las expresiones regulares:

- **/\VYo tengo 1$ en el banco**

Esto también sirve para las sustituciones:

- **:%s/\VYo tengo 1$ en el banco/No tengo dinero en el banco/g**

Búsqueda de palabras completas y OR en búsquedas

En ocasiones cuando realizamos búsquedas nos interesa encontrar palabras completas y no porciones de palabras. Por ejemplo, si buscamos **casa**, nos interesará encontrar **vaca**, pero no **vacaciones** o **Caravaca**. En ese caso podemos hacer uso de los identificadores de inicio y fin de palabra en las expresiones regulares de

Vim, que son \< y \> respectivamente (barra inversa seguida del símbolo de menor que para inicio de palabra, y barra inversa seguida del símbolo de mayor que para fin de palabra)

Así, si en un texto tenemos las 3 palabras anteriores, según la búsqueda que realicemos obtendremos unos u otros resultados:

Búsqueda	Resultado
/vaca	Encontraremos las 3 palabras: **vaca**, **vacaciones** y **Caravaca**.
/\<vaca	Encontraremos aquellas palabras que empiecen por la cadena vaca, como **vaca** y **vacaciones**.
/vaca\>	Encontraremos aquellas palabras que acaben en vaca, como **vaca** y **Caravaca**.
/\<vaca\>	Encontraremos sólo la palabra **vaca**.

Para Vim, los símbolos `\<` y `\>` se corresponden con aquellos caracteres que comienzan o acaban una palabra, como espacios, retornos de carros, comas, puntos y comas o puntos.

De la misma forma, si queremos buscar apariciones de una cadena u otra, podemos usar el operador de las expresiones regulares "|" (OR). No obstante, igual que en el caso de < y >, hay que escaparlo con \:

```
/red\|green\|blue
```

Sustituir (reemplazar) cadenas en el texto

Otra de las operaciones básicas de búsqueda es el reemplazo de cadenas, es decir, cambiar en todo el fichero (o en una parte del mismo) una cadena por otra. Esto se hace con el comando de sustitución **:s**.

Por ejemplo, para cambiar todas las apariciones de la cadena **hola** por **adios**, haremos:

```
:%s/hola/adios/g
```

Este comando viene a decirle a Vim que sustituya (s), en todo el fichero (%), la cadena **hola** por **adios**, y que si en una línea encuentra más de una aparición de **hola**, que cambie todas (g). Si quitamos la **g**, sólo cambiaremos la primera aparición de **hola** en cada frase. Si además de la `g` añadimos una `i`, se hará una comparación que no distingue mayúsculas de minúsculas.

Recordad que en el caso de usar expresiones regulares tendremos que escapar ciertos caracteres especiales, como los puntos, las barras, etc, en la parte de búsqueda. Por ejemplo:

```
:%s/hola\.hola/adios.adios/g
```

Reemplazo en texto seleccionado con 'v'

No estamos obligados a trabajar con la totalidad de un fichero, podemos realizar sustituciones también en bloques del fichero. Por ejemplo, supongamos que entramos en modo visual (v) y seleccionamos un bloque de texto. Mientras está seleccionado pulsamos ':' (dos puntos) y tecleamos:

```
s/hola/adios/g
```

En pantalla aparecerá:

```
:'<,'>s/hola/adios/g
```

Y el resultado efectivo de la sustitución será que sólo reemplazaremos la cadena **hola** por **adios** en el texto seleccionado.

Reemplazo en rangos de líneas

También podemos aplicar sólo la sustitución a las líneas situadas entre 2 líneas dadas. Si por ejemplo queremos cambiar todas las apariciones de **hola** entre la línea 100 y la línea 200, podemos hacerlo tecleando:

```
:100,200s/hola/adios/g
```

La sintaxis general es:

```
:n,Ncomando
```

Como ya hemos visto, utilizar :% equivale a poner un rango de líneas entre 1 y el máximo de líneas del fichero.

Utilizar un separador diferente entre expresión de búsqueda y sustitución

En una búsqueda como `%s/cad1/cad2/g` , el carácter separado de campos de la sustitución no tiene por qué ser "/". En algunos casos (por ejemplo, trabajando con paths de UNIX) nos puede interesar utilizar cualquier otro carácter. Vim utilizará como separador el caracter que siga a la "s":

Así, podemos usar por ejemplo "#":

Con barra (necesita escapar el path)	**Con otro carácter (#)**
:s/\/usr\/local\/\/opt\/*opt*\	:s#/usr/local/opt/#/opt/#

Uso de expresiones regulares en los reemplazos

Podemos usar expresiones regulares (con el mismo formato que las de búsqueda) en la porción de búsqueda de %s:

```
# Cambiar apariciones de red O green O blue por purple
:%s/red\|green\|blue/purple/g

# Lo mismo que lo anterior, pero sólo cuando no sean subcadenas, es decir
# cuando sean palabras completas:
:%s/\<\(red\|green\|blue\)\>/purple/g
```

Es más, podemos utilizar los paréntesis escapados `\( y \)` para reutilizar porciones de la cadena de búsqueda en los reemplazos (usando \1, \2, \N … en el orden de aparición en la búsqueda).

Por ejemplo, si queremos buscar todos los números entre signos de = paréntesis (como =123=, =11=, etc) y reemplazarlos por versiones sin paréntesis, podemos usar como búsqueda `=\(\d\+\)=` (uno o más dígitos \d+ entre iguales) y reemplazarlo por `\1` :

```
%s/=\(\d\+\)=/\1/g
```

Reutilizar la última búsqueda para un reemplazo

Si acabamos de realizar una búsqueda y queremos utilizar su resultado para un reemplazo, basta con dejar en blanco el campo de búsqueda de %s:

```
:%s//purple/g
```

Resumen sobre sustitución en Vim

El resumen del uso básico de la sustitución en Vim es:

Comando	**Significado**
:%s/cad1/cad2/	Reemplazar en cada línea del fichero la primera aparición de cad1 (sea cadena o expresión regular) por cad2.
:%s/cad1/cad2/g	Reemplazar en cada línea del fichero todas las apariciones de cad1 por cad2.
:%s/cad1/cad2/gi	Reemplazar en cada línea del fichero todas las apariciones de cad1 por cad2, sin distinguir mayúsculas y minúsculas
:%s/cad1/cad2/gc	Reemplazar en cada línea del fichero todas las apariciones de cad1 por cad2 pidiendo confirmación en cada reemplazo.
:%s/cad\(regexp\)ena/cad\1ena2/gc	Utilizar en la sustitución parte de la búsqueda por expresión regular
:n,Ncomando	Cualquiera de los 3 ejemplos anteriores, sobre un rango de líneas.
:'<,'>comando	Cualquiera de los ejemplos anteriores, sobre la selección "visual" actual. '<,'> aparece cuando tenemos un texto seleccionado y pulsamos ":".
:'a,'bs/cad1/cad2/g	Realizar la sustitución en el texto entre 2 marcas (a y b en el ejemplo).
:%s//CAD2/g	Reutilizar la última búsqueda realizada para el reemplazo (búsqueda vacía en %s)

Si investigas más acerca de las expresiones regulares y las usas a menudo, debes recordar que:

- La sintaxis de expresiones de regulares de Vim es especial, no es como la de Perl. Hay carácteres especiales cuando no los escapas (^, $, ., *) y hay otros que necesitan ser escapados para ser especiales (<, >, |, (,), +…).
- Los grupos de captura realizados con paréntesis escapados se referencian luego como \N en lugar de con $N (\1 en lugar de $1).
- Vim no soporta el "non-greedy modifier" (`?`) a continuación de `.*` . , en lugar de `.*` utiliza `.\{-}.` , como en: `%s/style="\.{-}"//g` . Es decir: `\{-}` es lo mismo que `.*?` .
- No necesitas escapar nada en la cadena de sustitución salvo las barras inversas (para que no se confunda con \1, \2 …).

Resumen de caracteres especiales y reemplazos

Veamos alguna de las tablas de vimregex.com tanto para las expresiones regulares como para los reemplazos. Primero, los caracteres especiales al ser escapados, y su significado:

```
# Matching                                                    # Matching
-------------------------------------------------------------------------------
. any character except new line
\s whitespace character                                       \S non-whitespace characte
\d digit                                                      \D non-digit
\x hex digit                                                  \X non-hex digit
\o octal digit                                                \O non-octal digit
\h head of word character (a,b,c...z,A,B,C...Z and _)         \H non-head of word charac
\p printable character                                        \P like \p, but excluding
\w word character                                             \W non-word character
\a alphabetic character                                       \A non-alphabetic characte
\l lowercase character                                        \L non-lowercase character
\u uppercase character                                        \U non-uppercase character
```

Segundo, los cuantificadores:

```
Quantifier    Description
-------------------------------------------------------------------------------
*             matches 0 or more of the preceding characters, ranges or metacharacte
.*            matches everything including empty line
\+            matches 1 or more of the preceding characters...
\=            matches 0 or 1 more of the preceding characters...
\{n,m}        matches from n to m of the preceding characters...
\{n}          matches exactly n times of the preceding characters...
\{,m}         matches at most m (from 0 to m) of the preceding characters...
\{n,}         matches at least n of of the preceding characters...
\{-}          matches 0 or more of the preceding atom, as few as possible
              (non greedy)

(where n and m are positive integers n>0 )
```

Finalmente, estos son los códigos especiales que podemos usar en las sustituciones con %s en la parte de sustitución (la segunda):

```
#   Meaning                                          #  Meaning
-------------------------------------------------------------------------------
&   the whole matched pattern                       \L  the following characters a
\0  the whole matched pattern                       \U  the following characters a
\1  the matched pattern in the 1st pair of \(\)     \X  the matched pattern in the
\r  split line in two at this point                 \l  next character made lowerc
~   the previous substitute string                  \u  next character made upperc
```

El fichero .vimrc

Antes de continuar con la parte avanzada del tutorial de vim, pasemos a examinar su principal fichero de configuración.

Ya hemos hablado del fichero **.vimrc** (o **_vimrc** en Windows). En él podemos poner nuestras configuraciones específicas y concretas, sólo para nuestro usuario (o para todos en /etc/vimrc).

En este tutorial de introducción a VIM sólo vamos a ver algunas opciones útiles e interesantes que podemos definir en el fichero .vimrc. En el manual de VIM (y en la gran cantidad de documentación que tenéis disponible en Internet) podéis encontrar muchas más opciones, variables e incluso ejemplos de código para programar (sí, programar) vuestras propias funciones para el editor.

El fichero .vimrc no sólo permite especificar parámetros y opciones de arranque para Vim: es mucho más que eso. En él podéis programar en el lenguaje interno propio de Vim (lenguaje de comandos) para realizaros vuestras propias funciones, pudiendo hacer cualquier cosa que os podáis imaginar: macros, comandos, filtros para el texto, llamadas a programas externos, etc.

Si queréis conocer la totalidad de opciones de Vim y una explicación de cada una de ellas, podéis hacerlo mediante la ayuda incluída al respecto en Vim, que se despliega tecleando **:options** (en modo comando).

Opciones

Las opciones que veremos a continuación para el fichero .vimrc no sólo están pensadas para ser utilizadas en el arranque del editor: podrán ser utilizadas en cualquier momento en modo comando durante la ejecución de VIM.

Veamos algunos ejemplos de opciones:

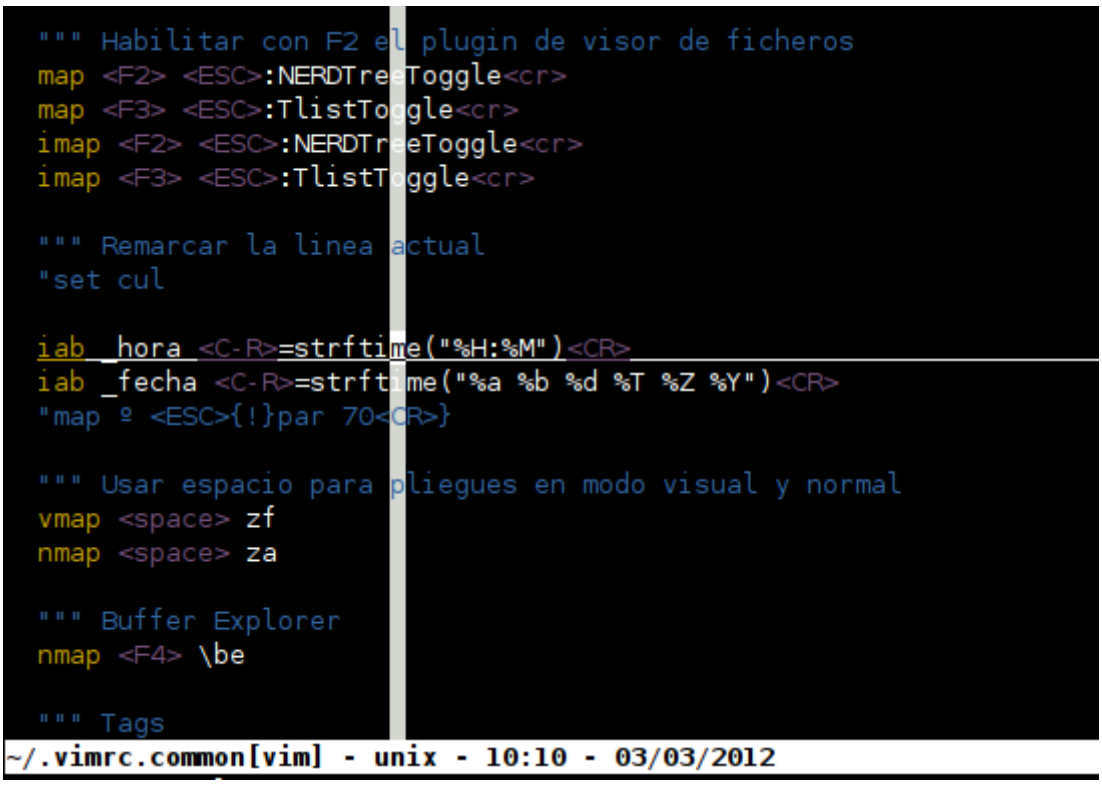
- **set nocompatible** : Añadiendo en nuestro fichero vimrc la opción **set nocompatible**, hacemos que VIM nos permita utiliza funciones extras que no están disponibles en el VI clásico y tradicional. Os recomiendo que tengáis esta opción definida en el .vimrc. Utilizar **set compatible** u omitir esta opción hará que algunas de las mejores funcionalidades de VIM no estén disponibles, para preservar la compatibilidad con VI.
- **set autoindent** : Esta función (también puede utilizarse **set ai**), hace que cuando pulsemos enter en un fichero de texto, la nueva línea que insertamos sea indentada automáticamente (es decir, se inserten espacios al principio de la misma y el cursor se posicione en una determinada posición). Esto puede servir, por ejemplo, para programar: si estamos escribiendo un bloque de código indentado a 3 espacios (por ejemplo), al pulsar enter no empezaremos en el primer carácter sino que automáticamente se nos situará el cursor en la columna 3. Literalmente, lo que hace VIM es que cuando pulsamos Enter, indenta la nueva línea a la misma profundidad que la anterior.
- **set noai** : Esta función hace lo contrario de **set autoindent**, es decir, cuando pulsemos Enter iremos directamente al primer carácter de la siguiente línea. Esta función resulta muy útil cuando estamos editando código indentado y queremos, por ejemplo, pegar texto o código desde una selección de texto externa (copiar y pegar desde un navegador, otro editor, etc.). Como el texto que pegamos ya está indentado, no necesitamos que Vim lo indente añadiendo espacios. Si lo pegáramos tal cual, veríamos como la indentación original sumada a la identación automática de Vim haría que no se respetara el indentado real del texto. Para evitar esto, podemos pulsar ESC (pasar a modo comando), y teclear **:set noai**, y pegar el texto externo (que se pegará bien). Después podemos volver al modo de identación con ESC y **:set ai**.
- **set backup** : Si está activada esta opción, cada vez que grabemos el fichero se almacenará una copia de la **versión** anterior como fichero~ (con el carácter '~' detrás).
- **set nobackup** : Esto sirve para lo contrario que **set backup**, es decir, para deshabilitar la generación de ficheros de backup.
- **set ruler**: Con **set ruler**, VIM muestra la posición X,Y actual del cursor en la barra de estado.
- **set wrap** : Activa el **cortado** de líneas largas en pantalla: si tenemos activada esta opción y una línea es más larga (de ancho) que lo que podemos ver en nuestra ventana del editor, VIM la partirá (visualmente). Si no la tenemos activada, simplemente sólo podremos ver desde el inicio de la línea hasta lo que nos permita la ventana del editor o la terminal (pero no partirá la línea).
- **set nowrap** : Las líneas que no caben en pantalla no serán visualmente partidas (lo contrario de set wrap).

- **set incsearch** : Habilita la búsqueda incremental: esto implica que cuando hacemos búsquedas con el comando */*, Vim no esperará a que pulsemos ENTER para comenzar la búsqueda. VIM irá buscando las palabras conforme vayamos tecleando sus diferentes letras.
- **set hlsearch** : Habilita el coloreado de las palabras encontradas en las búsquedas, en un color diferente del color del texto.
- **set ignorecase** : No diferenciar entre mayúsculas o minúsculas en las búsquedas.
- **set tabstop**: Esta opción permite definir el tamaño (en espacios) de los tabuladores (por defecto suelen ser 8). Un ejemplo de uso sería **set tabstop=4**.
- **set sw**: Esta opción permite especificar el ancho en caracteres que se desplazará una línea a la izquierda o a la derecha cuando usemos **<<** y **>>** para indentar la línea actual o un bloque de texto seleccionado.
- **set expandtab** : Convertir todos los tabuladores en espacios: ideal para los que, como yo, odiéis los tabuladores y prefiráis los espacios para tabular. Junto a las 2 opciones anteriores, cuando pulséis **TAB** no se introducirá un carácter tabulador sino el número de espacios prefijados.
- **set list** (y **set nolist**): Si está activada esta opción, veremos de forma visual los finales de línea y tabuladores.
- **set noerrobells** : Evitar que Vim "pite" en caso de error.
- **set ff** : Permite especificar el formato del fichero que vamos a editar, para utilizar los retornos de línea adecuados, entre "=dos", "=unix" y "=mac".
- **syntax on** : Como ya hemos visto, activa el coloreado de sintaxis (si VIM entiende el formato del fichero que editamos). La orden que lo desactiva sería **syntax off**.
- **set mouse=a** : Habilitar soporte para ratón en la consola (en Vim, puesto que gVim lo soporta por defecto).
- **source fichero** : Carga el fichero especificado como fichero de opciones adicionales.
- **set guifont=** : Establecer una fuente concreta para gVim, como por ejemplo:

```
" You can also specify a different font, overriding the default font
if has('gui_gtk2')
  set guifont=Bitstream\ Vera\ Sans\ Mono\ 12
else
  set guifont=-misc-fixed-medium-r-normal--14-130-75-75-c-70-iso8859-1
endif
```

- **set cursorline** : Remarcar en pantalla la línea actual (se desactiva con **:set nocursorline**).
- **set cursorcolumn** : Remarcar en pantalla la columna actual (se desactiva con **:set nocursorcolumn**).

El efecto con ambas activadas es el siguiente (quizá la columna pueda resultar molesta a algunos, pero la marca de línea puede ser muy útil):



No obstante, podemos especificar los colores del marcador de línea y columna actuales a voluntad:

```
highlight CursorLine ctermfg=Black ctermbg=Gray guifg=Black guibg=Gray
highlight CursorColumn ctermfg=Black ctermbg=Gray guifg=Black guibg=Gray
```

Opciones aplicables a un fichero específico

Vim tiene una opción muy interesante que es la permitirnos establecer opciones específicas de tipo "**set**" que difieran de las generales para un fichero concreto. Para ello, basta con establecerlas en la última línea del documento (para ficheros en diferentes lenguajes de programación, con un comentario delante), en un formato como el siguiente:

```
// vim:tw=80:num:sw=4:ts=8
```

Al abrir dicho fichero, vim encontrará que la última línea es una selección de opciones personalizadas y las aplicará con mayor prioridad que las del fichero .vimrc.

El anterior ejemplo, con un comentario de tipo doble barra, serviría para un fichero .C o .PHP, por ejemplo. Para un fichero Python sería con el comentario de python (#) en lugar de con doble barra.

Sustituciones o Abreviaciones

Un comando muy útil para nuestro .vimrc es el comando de abreviación o sustitución. Este comando nos permite definir abreviaturas que después serán expandidas a sus versiones **largas**. Por ejemplo, supongamos que utilizamos VIM como editor para nuestro cliente de correo o de news y habitualmente tenemos que escribir la dirección de nuestra página Web:

```
http://www.sromero.org
```

Pues bien, podemos declarar lo siguiente en nuestro .vimrc:

```
iab _miweb http://www.sromero.org/
```

Con esto, cuando en cualquier momento tecleemos las letras que componen la palabra **_miweb** seguido de un espacio, automáticamente VIM expandirá la palabra **_miweb** y la reemplazará por la susodicha URL (Uniform Resource Locator). El espacio que tecleamos provoca la sustitución: sin él, podríamos seguir tecleando más letras para poder teclear, por ejemplo, **_miwebpersonal** sin que se produzca dicho reemplazo.

Así, podemos definirnos muchos y utilísimos aliases o abreviaturas en nuestro .vimrc:

```
iab _miweb http://www.sromero.org/
iab _saludos Muchas gracias y saludos.
iab _email miemail@dominio.com
iab _comment #--------------------------------

""" Correciones para errores tipicos
iab Saludso Saludos
```

Con los reemplazos se pueden utilizar cadenas especiales como **<CR>**, **<ESC>**, etc:

```
iab _firma Santiago Romero<CR>GNU/Linux<CR>sromero arroba sromero punto org
```

Incluso podemos llamar a funciones internas de vim:

```
iab _hora <C-R>=strftime("%H:%M")<CR>
iab _fecha <C-R>=strftime("%a %b %d %T %Z %Y")<CR>
```

Nótese cómo personalmente suelo anteponer un carácter _ a todas mis **abreviaturas**. Hago esto para evitar que palabras comunes (hora, fecha) sean expandidas, cuando mi objetivo es simplemente tener definidas abreviaturas como **_hora** y **_fecha**.

Podemos eliminar estando dentro de Vim una abreviatura definida mediante el comando **:unabbreviate** (por ejemplo, **:unabbreviate _hora**). Si queremos eliminar todas las abreviaturas definidas podemos usar **:abclear**.

Mapeados (Macros)

Si os pareció útil la opción **iab**, el comando **map** y sus variantes (nmap, imap, vmap) no se quedan atrás: **map** permite **mapear** teclas a acciones, de forma que cuando pulsemos una determinada tecla o combinación de teclas se ejecuten las acciones correspondientes. Veamos algunos ejemplos para el .vimrc.

Comencemos con un ejemplo sencillo: que cada vez que pulsemos la tecla F1 se inserte la cadena **prueba** en el texto, mediante la inclusión de lo siguiente en nuestro .vimrc:

```
map! <F1> <ESC>iprueba<CR>
```

Si en modo comando o inserción pulsamos F1, se insertará la cadena **prueba** dentro del texto. Lo que hace el comando **map** es sustituir la pulsación de **F1** por la serie de comandos definida.

Veamos más ejemplos:

```
map! <F2> <ESC>:r !uptime<CR>       -> Insertar uptime al pulsar F2
map <C-J> <ESC>{!}sort<CR>}         -> Odenar texto seleccionado con Ctrl+J
```

La diferencia entre map!, imap, cmap, y vmap es que imap realiza mapeados en modo inserción (el mapeado sólo será efectivo si estamos en modo inserción, y no surtirá efecto si pulsamos la tecla, por ejemplo, en modo comando o visual), vmap realiza mapeados para el modo visual (cuando hemos pulsado 'v'), cmap sólo actúal en

modo comando, mientras que **map!** se aplica tanto a modo comando como a modo inserción. Cabe destacar que podemos eliminar cualquier mapeado realizado con el comando **unmap**.

Como apunte, se recomienda utilizar para el mapeo las teclas de <F2> a <F12> y sus variantes (ej; <Shift>-<F2>) ya que no están utilizadas por Vim (salvo F1 que puede estar mapeada como :help).

.vimrc vs .gvimrc vs common

El fichero .vimrc es el fichero de configuración por defecto de vim, pero no de gvim, el cual usa su propio fichero .gvimrc para opciones específicas de este editor. Si usamos tanto vim como gvim, es posible que acabemos utilizando y mantenido opciones duplicadas en ambos ficheros.

Para evitar esto, podemos crear un fichero ~/.vimrc.common donde pongamos las opciones comunes a ambos de forma que en .vimrc y .gvimrc sólo especificaremos opciones específicas para cuando estemos usando uno u otro (por ejemplo, con gvim podemos querer usar ciertos colores o tamaño de pantalla pero con vim no).

Dentro del fichero .vimrc y .gvimrc podemos "cargar" la configuración de .vimrc.common utilizando el comando **source**. Veamos un ejemplo:

- **.vimrc**:

```
$ cat .vimrc

""" Source a global configuration file if available
if filereadable("/etc/vim/vimrc.local")
  source /etc/vim/vimrc.local
endif

""" Cargamos las opciones generales.
if filereadable("/home/sromero/.vimrc.common")
  source /home/sromero/.vimrc.common
endif

""" En el vim de terminal no quiero numeros
set nonumber
```

- **.gvimrc**:

```
sromero@compiler ~ $ cat .gvimrc
""" Source a global configuration file if available
if filereadable("/etc/vim/gvimrc.local")
  source /etc/vim/gvimrc.local
endif

""" Cargar opciones comunes a .vimrc y .gvimrc
if filereadable("/home/sromero/.vimrc.common")
  source /home/sromero/.vimrc.common
endif

""" Activamos numeros, esquema de colores y dimensiones deseadas
set number
colorscheme torte-mod
set lines=50
set columns=120
```

- **.vimrc.common**:

```
$ cat .vimrc.common
""" Common options for .vimrc and .gvimrc

filetype plugin on
set tabstop=4
set nobackup
set sw=4
set ai
set sm
set ruler
set nocompatible
set vb
set novisualbell
set noerrorbells
set ttyfast
set expandtab
set wrap
syntax on

iab _hora <C-R>=strftime("%H:%M")<CR>
iab _fecha <C-R>=strftime("%a %b %d %T %Z %Y")<CR>

""" Autoguardar los pliegues del fichero actual
"set viewoptions=folds
"autocmd BufWinLeave ?* mkview
"autocmd BufWinEnter ?* silent loadview

""" Solo mostrar las marcas a-z y A-Z en un color concreto (plugin ShowMarks):
"let g:showmarks_include="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
"highlight ShowMarksHLl guifg=#40FF40 guibg=Black
"highlight ShowMarksHLu guifg=#40FF40 guibg=Black
""" Ocultar la columna de signos de Showmarks quitando todos los simbolos
"nmap <F9> <ESC>:sign unplace *<CR>


""" Ctrl+N 2 veces seguidas alterna numeros de linea
nmap <C-N><C-n> :set invnumber<CR>
imap <C-N><C-n> :set invnumber<CR>

""" Abrir ficheros en la ultima posicion utilizada
if has("autocmd")
        autocmd BufReadPost *
        \ if line("'\"") > 0 && line("'\"") <= line("$") |
        \   exe "normal g`\"" |
        \ endif
endif

""" Usar espacio para pliegues en modo visual y normal
vmap <space> zf
nmap <space> za

""" Establecer el color para los pliegues
highlight Folded ctermfg=black ctermbg=gray guifg=black guibg=gray

""" Eliminar informacion de los pliegues (numero de lineas)
"set foldtext=getline(v:foldstart)
```

```
(etc...)
```

De esta forma, las opciones comunes a vim y gvim están en un fichero y no por duplicado en los 2 ficheros de configuración, evitandonos el tener que mantenerlas en ambos sitios.

Pestañas, Ventanas y Buffers

Vim permite trabajar de forma simultánea con múltiples ficheros. Para eso dispone de varios mecanismos:

- **Tabs** (pestañas): Si utilizamos vim bajo un entorno gráfico con gvim, podemos disponer de múltiples pestañas visibles en la parte superior de la ventana, con un fichero abierto en cada pestaña. En modo texto, desde Vim 7, también se soportan pestañas (aparecerán los nombres en la línea superior del editor).
- **Windows** (ventanas): En la ventana de vim o en una pestaña de gvim podemos dividir el área de pantalla en múltiples ventanas (con divisiones horizontales o verticales), para trabajar con múltiples ficheros y alternar entre las ventanas con atajos de teclado.
- **Buffers**: Vim puede trabajar con múltiples "pantallas" virtuales (buffers), pero en lugar de estar visibles como las ventanas, sólo uno de ellos será visible en cada momento.

Veamos un resumen con los distintos atajos de teclado que permiten la utilización de los diferentes modos de multiedición.

Tabs (pestañas)

Todos los comandos de gestión de pestañas comienzan como ":tab", y permiten crear, cerrar y movernos entre las pestañas. Estas acciones también pueden ser llevadas a cabo desde los propios menúes de Gvim.

Comando	**Significado**
:tabnew	Crea una nueva pestaña vacía.
:tabedit fichero	Abre el fichero especificado en una nueva pestaña.
:tabclose	Cerrar la pestaña actual (también con :q!).
:tabnext [n]	Saltar a la siguiente pestaña (o a la enesima siguiente).
:tabprev [n]	Saltar a la anterior pestaña (o a la enesima anterior).
gt	(En modo comando) Saltar a la siguiente pestaña. En gvim también sirve Ctrl+AvPag.
gT	(En modo comando) Saltar a la anterior pestaña. En gvim también sirve Ctrl+RePag.
Ngt	(En modo comando) Saltar a la pestaña número N.
:tabs	Mostrar una lista de pestañas abiertas.
:tabdo comando	Ejecuta el comando en todas las pestañas abiertas, abortando en caso de error.

Para abrir desde la línea de comandos múltiples ficheros en diferentes pestañas, se utiliza el flag -p:

```
vim -p *.txt
```

Los siguientes mapeos pueden ser interesantes en el .gvimrc (sólo para gVim):

```
nnoremap <C-Left> :tabprevious<CR>
nnoremap <C-Right> :tabnext<CR>
nnoremap <silent> <A-Left> :execute 'silent! tabmove ' . (tabpagenr()-2)<CR>
nnoremap <silent> <A-Right> :execute 'silent! tabmove ' . tabpagenr()<CR>
```

Con Ctrl+Izquierda y Ctrl*Derecha nos moveremos entre las pestañas y con Alt+Izquierda y Alt+Derecha cambiaremos la pestaña actual de posición.

Windows (ventanas)

Si no utilizamos GVim, lo normal será trabajar con ventanas y con buffers. Si nuestra terminal de trabajo es grande (no simplemente 80x25), podremos utilizar cómodamente ventanas para alternar entre diferentes ficheros (normalmente 2 ó 3 es un límite razonable en cuanto a tamaño resultante de ventana).

Para gestionar las ventanas existen tanto comandos ":" como atajos de teclado, todos ellos precedidos por la combinación de teclas Ctrl-W.

Comando	**Atajo**	**Significado**
:split [fichero]	Ctrl-W s	Partir la pantalla horizontalmente, en blanco o con el contenido del fichero.
:vsplit [fichero]	Ctrl-W v	Partir la pantalla verticalmente, en blanco o con el contenido del fichero.
:close	Ctrl-W c	Cerrar la ventana actual.
:q!	Ctrl-W q	Salir de la ventana actual (la cierra y se pierden los buffers).
:only	Ctrl-W o	Hacer la ventana actual la única (cerrar todas las demás).
:wincmd j/k/l/h	Ctrl-W direccion	Moverse a la ventana de la izquierda/arriba/abajo/derecha. Con el atajo de teclado se pueden usar los cursores.
–	Ctrl-W w	Moverse cíclicamente entre ventanas.
—	Ctrl-W r	Rotar el orden de las ventanas.
—	Ctrl-W x	Intercambiar la ventana actual con la siguiente.
:resize +1	Ctrl-W +	Aumentar en 1 línea el tamaño de la ventana actual en las divisiones horizontales.
:resize -1	Ctrl-W -	Reducir en 1 línea el tamaño de la ventana actual en las divisiones horizontales.
—	Ctrl-W =	Igualar el tamaño de todas las ventanas horizontales
:vertical resize +1	Ctrl-W >	Aumentar en 1 línea el tamaño de la ventana actual en las divisiones verticales.
:vertical resize -1	Ctrl-W <	Reducir en 1 línea el tamaño de la ventana actual en las divisiones verticales.
–	Ctrl-W _	Maximiza la ventana actual, dejando el resto a tamaño 1.

Podemos hacer uso de las ventanas para, por ejemplo, **copiar una porción de texto de un fichero en otro**, sin el uso del ratón en UNIX ni del portapapeles del sistema en Windows. El procedimiento sería el siguiente:

- Seleccionar texto en modo visual.
- Copiar el texto con **y** (yank).
- Abrir el fichero destino con **:split fichero**.

- Pegar el texto en la posición deseada con **p**.
- Cerrar la ventana del segundo fichero con Ctrl-W + c.

Todas estas acciones se realizan de forma rápida y sin necesidad de levantar las manos del teclado o de cambiar de ventana o programa (por lo que podrían valer, perfectamente, en una consola de modo texto UNIX).

Buffers

Finalmente, tenemos los buffers. Nos permite abrir múltiples ficheros pero viendo sólo uno cada vez. Con comandos de vim de apenas 2 letras podemos saltar de uno a otro, ocultando en el cambio el que tuvieramos actualmente en pantalla. Si editamos múltiples ficheros desde línea de comandos con "vim *.c", cada fichero .c acaba en un buffer.

Comando	**Significado**
:buffers	Ver el listado de buffers junto a sus números identificativos.
:buffer N	Abrir buffer número N.
:bn[ext]	Ir al siguiente buffer.
:bp[revious]	Ir al anterior buffer.
:bf[irst]	Ir al primer buffer.
:bl[ast]	Ir al último buffer.
:bd[elete]	Cerrar el buffer actual.

Por ejemplo:

```
$ vim *.txt
:buffers
  1 %a   "documentacion.txt"            línea 1
  2      "manual-certificados.txt"      línea 0
Pulse INTRO o escriba una orden para continuar
```

El buffer activo está indicado como %a.

Marcas en el texto

Vim tiene una funcionalidad bastante útil conocida como marcas, que consiste en que podemos establecer hasta 26 marcas (desde la a hasta la z) en el texto para volver a esa posición del texto en cualquier momento. Estas marcas son invisibles, y son simplemente una referencia para nosotros.

Comando	**Significado**
m seguido de minúscula	Pulsando una tecla a-z, ponemos una marca en el documento (ej: ma).
' seguido de marca	Saltamos a la línea de la marca solicitada (ej: 'a). Es apóstrofe, no acento (la tecla a la derecha del 0).
` seguido de marca	Saltamos a la línea y columna exacta de la marca solicitada (ej: `a).

Comando	Significado
' ' (2 apóstrofes)	Volvemos a la línea anterior al salto.
`` (2 comillas invertidas)	Volvemos a la posición exacta anterior al salto. En teclado español, pulsar ` + espacio + ` + espacio.
m seguido de mayúscula	Establecer una marca global (más adelante veremos qué es).

Por ejemplo, supongamos que estamos programando y estamos modificando el bucle principal de nuestro programa, pero estamos cambiando bastante a otra función que también estamos modificando. Cuando tenemos que ir de una parte del documento a otra constántemente es bastante molesto, sobre todo con los editores convencionales, donde se hace todo a base de barra de scroll o bien de RePág/AvPág y cursores. Estar todo el rato hacia arriba y hacia abajo sólo porque tenemos que movernos entre 2 porciones del documento no es algo especialmente agradable ni cómodo.

En Vim, hasta ahora tenemos la opción de activar los números de línea (:set number), mirar y recordar los 2 números de línea en que están las 2 partes del documento entre las que vamos a ir cambiando, y cambiar entre ellos con el comando **<NUMERO>G** (por ejemplo, alternar entre **100G** y **500G**). Es cierto que esto es muchísimo más cómodo que moverse mediante teclas de movimiento, pero Vim aún puede ir más allá gracias a las marcas.

Simplemente basta con poner 2 marcas (invisibles, recordemos) en esos 2 puntos del documento, y podremos alternar entre ellos sin ninguna dificultad. No es que podamos alternar, es que podemos seguir moviéndonos libremente por el documento e ir a cualquiera de los dos puntos en cualquier momento. Y no estamos limitados a 2 marcas, sino que tenemos a nivel del fichero actual todas las letras posibles entre la a y la z para poner marcas.

Establecer y recuperar marcas

En Vim las marcas se ponen con el comando **m** seguido de una letra minúscula (a-z) identificador de la marca. Así, cuando estamos en una parte concreta del documento que nos interesa, pulsamos **ma** (letra m, letra a), y establecemos una marca en la línea actual del documento que se llamará ma. Del mismo modo, nos podemos ir a la segunda parte del documento que vamos a frecuentar y establecer una marca con **mb** (recordad que tenéis disponibles 26 marcas, de la 'a' a la 'z').

Ahora podemos ir a cualquiera de esas 2 marcas de forma inmediata con el comando ' (comilla simple, la que está a la derecha del cero en los teclados españoles) seguido de la letra de la marca a la que queremos ir. Por ejemplo, pulsando 'a iremos al principio de la línea que marcamos con **ma** y pulsando 'b iremos al principio de la línea que marcamos con **mb**. ¿Se os ocurre alguna manera más cómoda de moverse entre 2 partes diferentes del documento?

Si eres programador no hay nada más útil: una marca en el main(), otra en el bloque en que estamos trabajando, y otra por ejemplo en una función a la que estamos yendo mucho para hacer cambios, y se acabó el moverse con las teclas de movimiento de un sitio para otro.

Y si no eres programador, también: puedes poner una marca al principio del fichero y otra al final: pulsas **gg** para ir al principio del fichero, pulsas **mi** (la i de inicio, para que sea fácil de recordar), pulsas **G** para ir al final del fichero, pulsas **mf** (la f de fin), y ya tienes 2 marcas de forma que desde cualquier punto del documento puedes ir al principio o final del fichero usando marcas. O, por ejemplo, si estamos escribiendo algo y necesitamos ir a otro punto del documento a consultar algo, podemos poner una marca y desplazarmos, para después volver de forma inmediata recuperando la marca. Las posibilidades son infinitas.

Hemos dicho que la comilla simple nos devuelve a una marca posicionando el cursor al principio de la línea. Vim permite mucho más, ya que el comando ` (comilla inversa, la tecla que tenemos a la derecha de la 'p' en los teclados españoles) seguido de la letra de la marca a la que ir nos devuelve exactamente a la línea y columna en

la que realizamos la marca con el comando **m**: no sólo a la misma línea, sino en la misma posición exacta del cursor.

Marcas especiales

Cabe destacar que si nos olvidamos de las marcas que hemos puesto, podemos obtener un listado de marcas ejecutando el comando :marks seguido de Enter. Al visualizar este listado veremos que hay una serie de marcas especiales que no hemos definido nosotros, y que son:

Marca Especial	**Significado**
' (comilla simple)	Posición del cursor en el momento en que realizamos el último salto que hayamos hecho.
" (comillas dobles)	Posición del cursor la última vez que editamos el fichero. Esto quiere decir que cuando abrimos un fichero, yendo a la marca **comillas dobles** mediante comilla simple seguido de comilla doble nos posicionaremos en el lugar en que estábamos la última vez que editamos este fichero. Esto es especialmente útil a la hora de programar.
[(corchete abierto)	Posición del principio del último cambio que hayamos realizado.
] (corchete cerrado)	Posición final del último cambio que hayamos realizado.
^ (circunflejo)	Posición del último lugar en que hayamos estado en modo Inserción realizando cambios.

Merecen mención especial las 2 primeras marcas especiales de la tabla.

La comilla simple permite volver a la posición del último salto, y esto incluye los saltos realizados con **G** y **gg**. Es decir, si estamos en una posición del documento y hacemos 100G para ir a la línea 100, pulsando comilla simple seguido de comilla simple de nuevo (es decir: '') volveremos a la posición en que estábamos antes de realizar el cambio de línea. Por si fuera poco, las dos comillas simples también nos permitirán volver al punto original del salto en el caso de búsquedas, por ejemplo.

Vim guarda un historial de saltos, al cual contribuyen las búsquedas, las marcas y los cambios de línea, y podemos movernos por ese historial mediante CTRL+O (anterior) y CTRL+I (siguiente). Esto quiere decir, que podemos circular entre todas las posiciones del documento entre las que hemos saltado o buscado mediante estas 2 teclas. Yo personalmente tengo bastante con el uso de marcas y no suelo necesitar usar esta **pila de saltos**, pero es una posibilidad más que Vim nos ofrece.

Por último, las comillas dobles guardan la última posición en que estabamos la última vez que editamos el fichero: por ejemplo, si programamos y solemos salir del editor para compilar, tal vez al terminar de hacerlo nos interese recuperar la edición del fichero en el punto exacto en que estábamos y no al principio del mismo. Con la marca de comillas dobles podemos hacer esto fácilmente.

Si queremos que automáticamente se posicione el cursor en el lugar en que editamos el fichero por última vez sin necesidad de que nosotros lo hagamos manualmente podemos incluir las siguientes opciones en nuestro fichero .vimrc:

```
autocmd BufReadPost *
   \ if line("'\"") > 0 && line("'\"") <= line("$") |
   \   exe "normal g`\"" |
   \ endif
```

Este comando de Vim simplemente comprueba que existe una marca comillas dobles en el fichero y si es así la llama (puede verse en el comando exe en el que se hace un comilla invertida seguido de una comilla doble (escapada, para que Vim no la interprete), lo que nos lleva a la posición exacta de fila y columna a la que apunte la comilla doble.

Marcas globales

Cabe destacar que las letras A-Z (en mayúscula) son marcas globales, es decir, que se establecen a nivel de fichero y que nos permiten volver a la edición de un fichero que no es el actual.

Por ejemplo, supongamos que establecemos una marca "A" con **mA** en un fichero y salimos de vim. Si en otro momento editamos otro fichero diferente y saltamos a la marca A (con **'A**), vim cargará al primer fichero y saltará a la marca solicitada.

Vim guarda la información de las marcas incluso después de haber salido del documento (siempre que pueda guardar en el directorio home del usuario el archivo .viminfo) por lo que las marcas son permanentes a menos que las borremos manualmente, borremos la línea que marcan, o que las reasignemos a una nueva línea. Este funcionamiento es ideal para "marcar" aquellos puntos del documento a los que acudamos repetidamente.

Por ejemplo, si estamos programando, podemos marcar con una "v" de "variables" la zona en la que están definidas las variables y con una "t" de trabajo la zona de trabajo actual, y saltar de una zona a otra rápidamente.

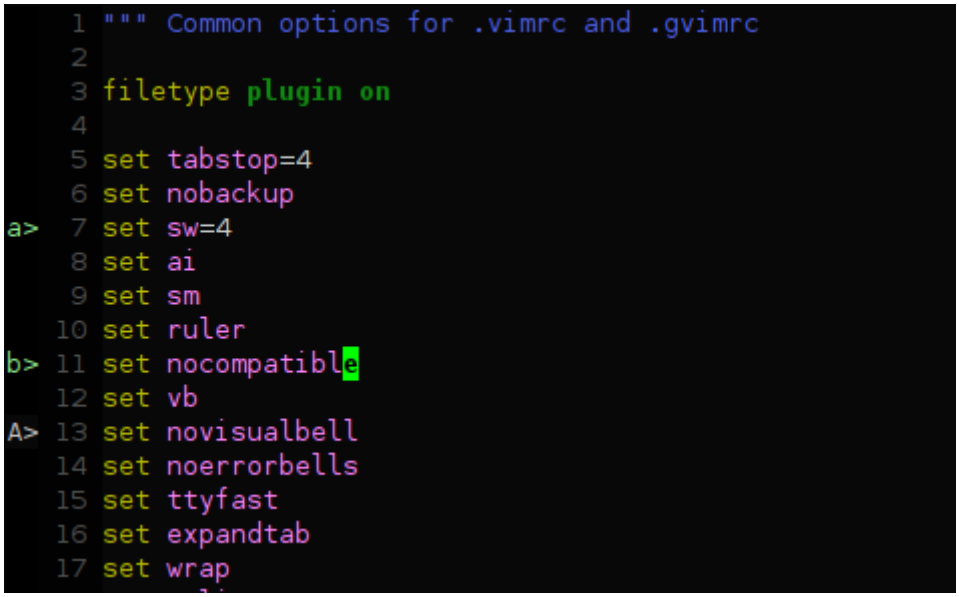
También podemos referenciar las marcas. Por ejemplo, con **d'a** borraremos todo el texto desde la posición actual del cursor hasta la marca "a" (ya estemos arriba o debajo de la misma) , poniendo dicho texto en el buffer interno para pegado.

Cómo ver las marcas

Como hemos comentado, las marcas son invisibles, pero podemos hacerlas visibles si instalamos el plugin **ShowMarks** y añadimos lo siguiente en nuestro fichero .vimrc:

```
""" Mostrar las marcas a-z y A-Z en un color concreto:
let g:showmarks_include="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

highlight ShowMarksHLl ctermfg=LightCyan ctermbg=NONE guifg=#40FF40 guibg=Black
highlight ShowMarksHLu ctermfg=LightCyan ctermbg=NONE guifg=#40FF40 guibg=Black
highlight SignColumn ctermfg=Blue ctermbg=NONE guifg=Blue guibg=Black
```

Una vez hecho esto, veremos las marcas en nuestro código en la columna especial de Vim llamada "SignColumn" (a la izquierda del texto), en el color que hemos indicado (verde clarito para gVim y LightCyan para modo texto):



Más adelante veremos el procedimiento para descargar e instalar Plugins, de forma que podremos instalar este y otros plugins interesantes.

Funcionalidades para programadores

Sin olvidarnos de las interesantísimas **Marcas** y de las **Macros** (map) y **Sustituciones** (iab), Vim ofrece una serie de funcionalidades especialmente interesantes para programadores.

Moverse entre bloques de código

Vim nos proporciona atajos para movernos al principio o al final de la función en la que estamos (o del bloque if/else, etc):

Comando	Significado
[{	Ir al inicio del bloque de código en que nos encontremos (inicio de función/método, if, etc).
]}	Ir al final del bloque de código en que nos encontremos (fin de función/método, if, etc).
[[	Ir al anterior bloque de código (anterior función/método, if(), etc).
]]	Ir al siguiente bloque de código (siguiente función/método, if(), etc).

Pongamos un ejemplo en C o PHP. Sabemos que si tenemos el cursor en la llave "{" de:

```
function my_function() {
```

Si en ese momento (sobre {) pulsamos '%', Vim moverá el cursor al carácter "}" de cierre de esta función (también vale para ifs, elses, whiles, fors, etc).

Con los 4 atajos que acabamos de ver vamos más allá, y podemos ir al inicio o fin de la función o bloque de código sin tener que estar encima de la llave que lo abre o cierra.

Recordar última posición en el fichero

Si queremos, al abrir cualquier fichero, que el cursor se coloque en la posición exacta en que estábamos en él cuando lo cerramos la última vez (muy útil para programadores o si vamos a editar repetidamente una misma zona de un fichero de configuración), podemos añadir (como ya hemos visto en el apartado sobre las Marcas) lo siguiente en nuestro .vimrc:

```
if has("autocmd")
    autocmd BufReadPost *
    \ if line("'\"") > 0 && line("'\"") <= line("$") |
    \   exe "normal g`\"" |
    \ endif
endif
```

Con el anterior "comando", cada vez que abre un fichero se verifica si existe la "marca" especial doble comilla, y si está, se salta a la línea que contiene la marca.

De forma efectiva, esto quiere decir que cuando abramos cualquier fichero, el cursor se posicionará en las coordenadas exactas en que estuviera cuando lo abandonamos (o al principio del fichero si es la primera vez que lo abrimos).

No hace falta decir lo extremadamente útil que puede ser esto.

Para que vim pueda hacer esto, es necesario que nuestro usuario del sistema tenga permisos de escritura en el home ya que guardará esta información en un fichero .viminfo (o _viminfo en los sistemas Windows). Por defecto este suele ser el caso y se genera y usa el .viminfo sin problemas.

Expansión de tabuladores

Por norma general, los programadores huímos de los tabuladores, especialmente en lenguajes donde la indentación es importante como en python. Esto es así porque un fichero generado con tabuladores puede acabar causando problemas cuando éstos (invisibles) se mezclan con espacios.

Para evitar esto, podemos hacer que vim expanda los tabuladores como "espacios consecutivos" (espacios reales). Eso implica que cuando pulsemos tabulador, éste no se insertará sino que se insertarán N espacios.

Las opciones adecuadas del .vimrc son las siguientes:

```
""" Establecer ancho de tabulador y de indentación a 4
set tabstop=4
set shiftwidth=4

""" Expandir los tabuladores
set expandtab

""" Mostrar los tabuladores y fines de línea (no es imprescindible pero ayuda):
set list
```

Tampoco olvidemos que cuando editamos en Vim un fichero que ha sido creado con otros editores, podemos "convertir" todos sus "tabuladores" a nuestra configuración de tabulado con `:retab`.

Conversión mayúsculas/minúsculas

Para los programadores puede ser necesario, en ocasiones, cambiar el "case" de un texto a minúsculas, mayúsculas, o alternarlo. Esto lo podemos hacer (una vez seleccionado el texto en modo visual) con:

Comando	Significado
~	Cambia el caso del caracter sobre el cursor en modo normal, o del texto seleccionado en modo visual.
u	En modo visual, pasa todo el texto seleccionado a minúsculas.
U	En modo visual, pasa todo el texto seleccionado a mayúsculas.

Un apunte: el comando "~" en los teclados españoles se introduce pulsando AltGr + 4, al menos en Linux. Además de cambiar el "case", avanza hasta e lsiguiente carácter.

Borrar el contenido de un string, bloque, etc

Vim nos proporciona herramientas tan útiles como la posibilidad de "vaciar" una cadena o de eliminar todos los parámetros de una función (lo que hay entre sus paréntesis) con un simple comando:

- **di'** o **ci'**: borrar o cambiar el contenido de un string delimitado por ' con el cursor dentro de ella.
- **di"** o **ci"**: borrar o cambiar el contenido de un string delimitado por " con el cursor dentro de ella.
- **di(** o **ci(**: borrar o cambiar el contenido entre (y) estando con el cursor entre ambos caracteres.
- **di[** o **ci]**: borrar o cambiar el contenido entre [y] estando con el cursor entre ambos caracteres.
- **di<** o **ci<**: borrar o cambiar el contenido entre < y > estando con el cursor entre ambos caracteres.
- **di{** o **ci}**: borrar o cambiar el contenido entre { y } (o un párrafo si no es código) estando con el cursor entre ambos caracteres.
- **dit** o **cit**: borrar o cambiar el contenido entre tags (y , <li> y , etc) estando con el cursor entre inicio y fin del tag.
- **diw** o **ci'**: borrar o cambiar el contenido de una palabra con el cursor dentro de ella.

Eso quiere decir que en el siguiente ejemplo:

```
void my_function( char arg1, int arg2, char arg3, int arg4, float arg5 ) {
```

Teniendo el cursor en cualquier posición entre (y) (dentro del listado de parámetros), bastará pulsar **di(** para dejar el texto así:

```
void my_function( ) {
```

Si en vez de **di(** usamos **ci(**, cambiaremos además a modo inserción para directamente empezar a teclear parámetros diferentes. También se puede usar **di)** (cerrar paréntesis) si se desea.

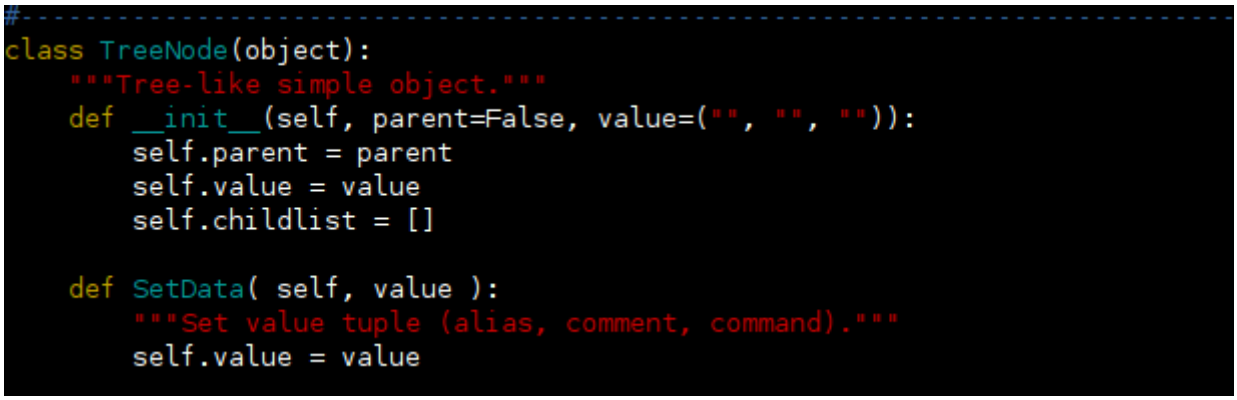
El mismo concepto sirve para eliminar el interior de un string con **di'** o **di"**.

Coloreado de sintaxis

Vim soporta coloreado de sintaxis, que quiere decir que puede resaltar con diferentes colores palabras claves del fichero que estemos utilizando. Así, si estamos programando y Vim tiene instalado un fichero de sintaxis para el lenguaje de programación que estamos usando, las palabras clave aparecerán en un color, los literales en otro, los números en otro, etc. Esto clarifica enormemente la edición de ficheros y permite encontrar errores más fácilmente. No sólo sirve para programar, porque gran parte de los ficheros de configuración típicos de UNIX aparecerán también con resaltado de sintaxis para evitarnos errores.

Si nuestra terminal de texto soporta colores y tenemos bien definida la variable $TERM en el sistema, podemos activar el coloreado de sintaxis mediante el comando **:syntax on** en el editor, o añadiendo **syntax on** en nuestro .vimrc. Si tras hacer esto el fichero que estamos editando no aparece coloreado, puede ser bien porque Vim no

ha sabido determinar el formato del fichero que estamos editando (cosa que le podríamos especificar con, por ejemplo, **:set filetype=python** en el caso de un fichero en python), o también puede ser que el fichero que estamos editando sea de un lenguaje o tipo del cual Vim no tiene una definición del lenguaje.



```
#--------------------------------------------------------------------------
class TreeNode(object):
    """Tree-like simple object."""
    def __init__(self, parent=False, value=("", "", "")):
        self.parent = parent
        self.value = value
        self.childlist = []

    def SetData( self, value ):
        """Set value tuple (alias, comment, command)."""
        self.value = value
```

En mi caso, los ficheros de sintaxis se guardan en /usr/share/vim/syntax, y como podréis ver en ese directorio, entiende cientos de **lenguajes** y **formatos**.

Si Vim no entiende el tipo de lenguaje que estamos usando, siempre podemos crear un fichero de sintaxis para él e introducirlo en ese directorio o en $HOME/.vim/syntax/. Ese fichero debe de contener reglas para decidir qué es una palabra clave, qué un literal y qué un comentario, por ejemplo. Podemos incluso modificar las reglas de detección de sintaxis existentes en nuestro propio fichero de nombre "lenguaje_a_modificar.vim" ubicándolo en $HOME/.vim/syntax/, donde tendrá prioridad sobre el fichero de /usr/share/vim/syntax.

Al activar el coloreado de sintaxis, vim utiliza el fichero de sintaxis para decidir qué es una palabra clave, qué un literal o qué un comentario (entre otras cosas) y utiliza un color diferente para cada cosa.

Los colores que utilizará Vim los define el **esquema de colores** actual. Podemos cambiar entre diferentes esquemas de colores de los existentes en /usr/share/vim/colors/ o $HOME/.vim/colors/ poniendo en nuestro .vimrc o .gvimrc el comando **colorscheme fichero-de-color**.

Un fichero de esquema de colores tiene un formato como el siguiente:

```
" Console
highlight Normal     ctermfg=LightGrey  ctermbg=Black
highlight Search     ctermfg=Black      ctermbg=Red     cterm=NONE
highlight Visual                                        cterm=reverse
highlight Cursor     ctermfg=Black      ctermbg=Green   cterm=bold
highlight Special    ctermfg=Brown
highlight Comment    ctermfg=Blue
highlight StatusLine ctermfg=Blue       ctermbg=White
highlight Statement  ctermfg=Yellow                     cterm=NONE
highlight Type                                          cterm=NONE
highlight ShowMarksHLl ctermfg=Green ctermbg=Black
highlight SignColumn ctermfg=Blue       ctermbg=Black

" GUI
highlight Normal     guifg=Grey80       guibg=#080808
highlight Search     guifg=Black        guibg=Red       gui=bold
highlight Visual     guifg=#404040                      gui=bold
highlight Cursor     guifg=Black        guibg=Green     gui=bold
highlight Special    guifg=Orange
highlight Constant   guifg=Red
highlight Comment    guifg=#5060ee
highlight StatusLine guifg=blue         guibg=red
highlight Statement  guifg=Yellow                       gui=NONE
highlight Type           guifg=#118811
highlight LineNr     guifg=#505050 guibg=#020202
highlight ShowMarksHLl guifg=LightGreen guibg=Black
highlight SignColumn guifg=Blue guibg=Black
```

El formato de un fichero de esquema de colores no es muy complicado aunque no lo trataremos aquí. Basta decir que si por ejemplo queremos crear un nuevo esquema de colores a partir de otro ya existente cambiando el color de los comentarios de rojo (por ejemplo) a cyan, podemos copiar el fichero con otro nombre:

```
mkdir ~/.vim/colors
cp /usr/share/vim/colors/torte.vim ~/vim/colors/test.vim
```

Después lo editamos y cambiamos:

```
SynColor Comment term=bold cterm=NONE ctermfg=DarkRed guifg=DarkRed (etc...)
```

por

```
SynColor Comment term=bold cterm=NONE ctermfg=Cyan guifg=Cyan (etc...)
```

Los posibles campos a cambiar son:

Campo	Significado
ctermfg	Color del texto en terminal (vim)
ctermbg	Color de fondo del texto en terminal (vim)
guifg	Color del texto en GUIs (gvim)
guibg	Color de fondo del texto en GUIs (gvim)

Cabe destacar que Vim tiene 2 juegos de colores diferentes el mismo fichero, según si la terminar que utilizamos tiene un fondo claro o un fondo oscuro. Podemos cambiar el juego de colores utilizados indicando el tipo de fondo de terminal que usamos, entre **:set background=dark** y **:set background=light**.

Finalmente, tened en cuenta que a veces nos puede dar la impresión de que el coloreado de sintaxis no se realiza bien cuando estamos scrolleando. Esto es así porque Vim, para ahorrar tiempo, no colorea todo el fichero, sino sólo lo que vemos por pantalla, y conforme lo vamos viendo. Si el scroll hace alguna palabra especial se corte, Vim puede no entenderla como una palabra clave y no ponerle el color apropiado. Pulsando CTRL+L, que redibuja la pantalla, podemos solucionarlo (si es que llega a sucedernos).

En cualquier momento podemos desactivar el coloreado de sintaxis con **:syntax off**.

Expandiendo el coloreado de sintaxis

Podemos ampliar el fichero de sintaxis con reglas propias que haga match en algún tipo de patrón concreto, o añadir alguna regla adicional en nuestro .vimrc.

Por ejemplo, supongamos las siguientes entradas en .vimrc (o con : delante, directamente dentro de Vim):

```
highlight MyPattern ctermbg=grey ctermfg=red guibg=grey guifg=red
match MyPattern /pattern/
```

Donde *pattern* es una expresión regular, de forma que si queremos resaltar por ejemplo una palabra concreta para nuestra lista TO-DO, podemos hacer:

```
highlight Pendiente ctermbg=grey ctermfg=red guibg=grey guifg=red
match Pendiente /\cPENDIENTE/
```

Así, cualquier aparición de la palabra PENDIENTE (ya sea en minúsculas o mayúsculas, debido al \c) lo hará en el color indicado.

Indentado de código

Podemos indentar texto (tanto la línea actual como una selección de texto realizada en modo visual) usando **<<** y **>>** (es decir, pulsando en modo comando o visual 2 veces "menor-que" y "mayor-que", indicando la dirección en la que queremos indentar el código).

El código o texto se indentará N espacios, siendo N el valor que tengamos especificado en **set shiftwidth=** o **set sw=**.

Por supuesto, es posible usar modificadores para indentar más niveles el código. Si sw está especificado a 4, ejecutando **3>>** lo indentaremos 3 niveles (12 caracteres).

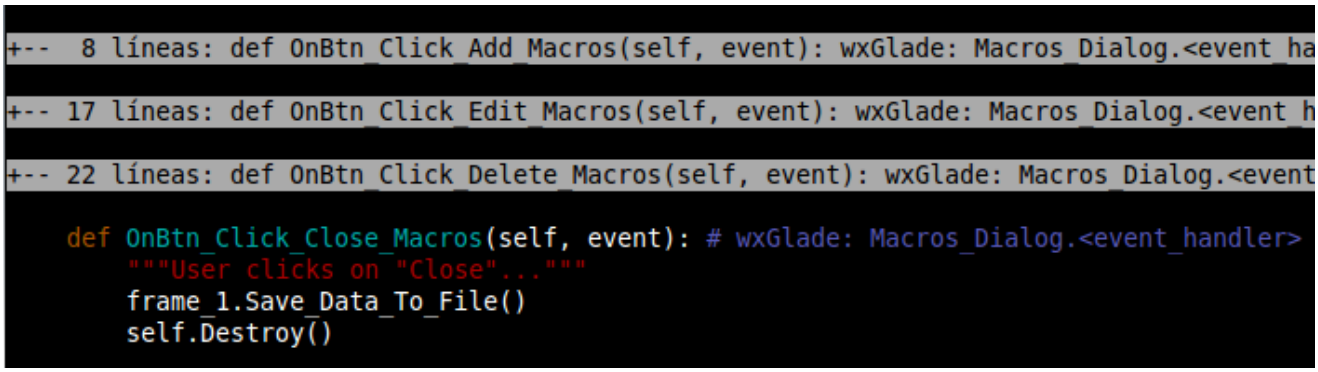
Un apunte importante sobre copiar y pegar y la indentación: Vim trata de intendar el texto de forma inteligente si estamos en modo "autoindent" (**:set ai**), de modo que para pegar texto copiado con el ratón podríamos necesitar cambiar al modo "no automático" (**:set noai** o bien **:set paste**).

Plegado (folding) de texto

Vim nos permite "plegar" texto (agruparlo) en una sóla línea con un comando, de forma que ese texto no nos moleste a la hora de trabajar con el documento. Por ejemplo, podemos plegar (fold) una función o clase completa con la que ya hayamos acabado para eliminar su visibilidad del documento y que no sea necesario scrollear a través de ella para moverse por el código.

Basta con seleccionar un bloque de texto en modo visual y pulsar **zf** (z es el comando para plegados, porque la z parece un pliegue, seguido de f de fold).

El aspecto de los bloques plegados es similar al siguiente:



Veamos un ejemplo de plegado. En el código siguiente, seleccionamos en modo visual todo el método "OnBtn_Click_Add_Macros" y pulsamos zf:

El aspecto en pantalla pasará de:

```
        return item

    def OnBtn_Click_Add_Macros(self, event):
        """User clicks on "Add"... (macro)."""
        dlg = AddEdit_Macro(None, -1, "")
        dlg.action = "add_macro"
        dlg.index = -1
        dlg.ShowModal()
        dlg.Destroy()
        self.Populate_macros()

    def OnBtn_Click_Edit_Macros(self, event):
```

a:

```
       return item

+--  8 líneas: def OnBtn_Click_Add_Macros(self, event):----

    def OnBtn_Click_Edit_Macros(self, event):
```

De esta forma, esa función sobre la que no pretendemos trabajar se reduce a nuestra vista y no molesta para el resto de edición del fichero. El pliegue es sólo visual, no a nivel del contenido del fichero. Si grabamos el fichero, se estarán grabando la totalidad de las líneas del mismo, y no la línea de plegado.

Podemos desplegar de nuevo el código si nos situamos sobre la línea de plegado y pulsamos **za** (a de alternar). Al hacerlo, desaparece la línea de plegado (la que empieza por +–) y aparece el código de nuevo. Y podemos volver a plegarlo poniéndonos encima de cualquiera de las líneas de plegado y pulsando otra vez **za**.

Estos son los comandos para plegado:

Comando	Significado
zf	Plegar el texto seleccionado
za	Abrir / Cerrar (alternar) un grupo de líneas plegadas.

Comando	Significado
zR	Desplegar todas las líneas plegadas de un fichero.
zd	Eliminar un pliegue (se recupera el contenido y ya no se puede alternar con za).
zE	Eliminar todos los pliegues del fichero.
zf/cadena	Plegar el texto que va desde la línea actual a la siguiente aparición de "cadena".
:n,M fold	Plegar el texto desde la línea n a la M.
zFNj	Crear un pliegue desde la línea actual hasta N líneas adelante.
zM	Vista de pájaro del fichero, cuando tenemos pliegues.
zc	Cerrar un pliegue.

Los pliegues pueden anidarse (de hecho, existen comandos para tratar los pliegues recursivamente), aunque por simplificar no he comentado los atajos de teclado para ello.

Finalmente, sabed que podemos asignar una tecla en nuestro vimrc para realizar el plegado de forma automática. Por ejemplo, si queremos que la tecla **espacio** cree un pliegue con el texto seleccionado cuando estamos en modo visual y que además permita abrir y cerrar pliegues cuando estamos en modo normal, podemos poner lo siguiente en nuestro fichero .vimrc:

```
vmap <space> zf
nmap <space> za
```

De esta forma, podemos alternar el estado de un pliegue pulsando espacio en modo comando, y crear un pliegue nuevo a partir de un texto seleccionado en modo visual pulsando espacio al acabar la selección. Y todo ello sin necesidad de recordar los comandos. Sólo necesitamos recordar que el espacio pliega un texto seleccionado y que también sirve para desplegar un pliegue.

Podemos cambiar el color con el que aparecen los pliegues tanto en el fichero .vimrc como en un fichero de colores/syntaxis mediante:

```
highlight Folded ctermfg=yellow ctermbg=gray guifg=yellow guibg=gray
```

Finalmente, podemos eliminar de pantalla la información relativa al número de líneas del pliegue (y dejar sólo la primera línea, con el color de texto y fondo de los pliegues) mediante:

```
set foldtext=getline(v:foldstart)
```

Quiero hacer notar que los pliegues no tienen por qué partir de selecciones de texto, de rangos de líneas o de búsquedas de cadenas. Vim tiene un modo de pliegue automático que entiende la sintaxis de múltiples lenguajes y que pliega la estructura sintáctica completa simplemente solicitando el pliegue en la primera línea. Este modo se activa con **set foldmethod**, pudiendo asignarse a esta variable los valores **manual** (a mano), **indent** (se pliegan las líneas hijas de la indentación actual), **syntax** según indique el coloreado de sintaxis (toda la función, toda la clase, etc). Yo, particularmente, prefiero la granularidad del modo manual.

Persistencia de los pliegues

Al contrario que las marcas, cuando salimos de un fichero se pierden los pliegues que hemos creado para él. Para evitar esto, podemos utilizar unos comandos en el fichero .vimrc que automáticamente salvan la sesión de "pliegues" cuando salimos de Vim, y la cargan al editar de nuevo el mismo fichero.

```
""" Save only fold in viewoptions
set foldmethod=manual
set viewoptions=folds

""" Save folds on exit and load them on edit
autocmd BufWinLeave ?* mkview
autocmd BufWinEnter ?* silent loadview
```

Es necesario crear el directorio ~/.vim/view (con mkdir) previamente para que vim pueda grabar las "vistas" en él.

Nótese que con esta opción activa, para cada fichero que editemos (tenga o no folds) se generará un pequeño fichero de 300 bytes en $HOME/.vim/view/ conteniendo la información de "vista" de dicho fichero. Aunque 300 bytes requeriría 10.000 ficheros editados para ocupar apenas 3 MB (Megabyte) de información, podemos querer borrar regularmente el contenido de dicho directorio o tan vez los ficheros de vista más antiguos de, por ejemplo, 3 meses (con un **find ~/.vim/view/ -type f -name "*" -mtime +90 -exec rm -f {} \;**).

Incluso es posible que sólo tengamos intención de guardar los folds (crear vistas) para ficheros de programación, por lo que si programamos en C y Python, podemos cambiar la configuración de "?*" por:

```
autocmd BufWinLeave ?*.py mkview
autocmd BufWinEnter ?*.py silent loadview
autocmd BufWinLeave ?*.[ch] mkview
autocmd BufWinEnter ?*.[ch] silent loadview
```

Eso guardará vistas en $HOME/.vim/view sólo para ficheros .py, .c y .h (aunque podemos alegremente dejarlo activado para ?* y borrar regularmente los ficheros más antiguos de N días sin problemas).

Etiquetas (tags)

Una de las habilidades de cualquier editor moderno es la de "etiquetar" internamente las variables, funciones, clases, métodos y atributos de nuestro programa de forma que podamos saltar, desde cualquier punto del documento, a su definición/declaración, y poder volver al punto inicial del salto.

De esta forma, si tenemos duda acerca de un método o función, podemos saltar a su definición para ver sus argumentos de entrada o su salida, y volver al punto donde estábamos (una vez resuelta la duda) para continuar con la inserción de código.

Las etiquetas también pueden permitirnos estudiar código ya escrito saltando a las declaraciones o definiciones de lo que estamos estudiando en caso de duda.

Vim permite utilizar un sistema de etiquetas denominado **ctags**, que debemos instalar como:

Sistema	**Comando**
CentOS / RedHat / Fedora	yum install ctags
Ubuntu / Debian / Mint	apt-get install exuberant-ctags

Una vez instalado, debemos generar los tags para nuestro programa con el binario de ctags, con alguno de los siguientes comandos:

```
ctags *.c *.h    -> Todos los .c o .h del directorio
ctags *.py       -> Todos los .py del directorio.
ctags -R *.py    -> Todos los ficheros si el proyecto tiene subdirectorios
```

Una vez hecho esto, se generará en el directorio actual un fichero llamado "tags" el cual contiene la información que Vim utilizará como etiquetas (se puede cambiar el path de la ubicación de este fichero, si queremos uno genérico, con **:set tags**, aunque personalmente prefiero un fichero de tags por proyecto). Ctags soporta más de 30 lenguajes diferentes de programación.

Si ahora editamos un fichero de código, podremos utilizar los siguientes comandos:

Comando	**Significado**
:tag subrutina_o_variable	Salta a la definición de la variable. Tiene autocompletado con Tabulador. En gVim podemos usar Ctrl + Click izquierdo del ratón sobre la palabra.
:stag subrutina_o_variableTAB	Salta a la definición de la variable en una nueva ventana. Tiene autocompletado con Tabulador.
:tags o :ts	Muestra la pila / histórico de tags.
:tn[ext]	Cuando hemos hecho una búsqueda parcial, saltar al siguiente tag que la cumple.
:tp[revious]	Cuando hemos hecho una búsqueda parcial, saltar al anterior tag que la cumple.
:tf[irst]	Cuando hemos hecho una búsqueda parcial, saltar al primer tag que la cumple.
:tl[ast]	Cuando hemos hecho una búsqueda parcial, saltar al último tag que la cumple.
Ctrl+]	Saltar al tag de la palabra bajo el cursor. En el teclado español, para pulsar] se necesita pulsar Ctrl+AltGr+].
Ctrl+t o :pop	Volver al lugar desde el que se hizo el salto con Ctrl+]. En gVim, podemos usar Ctrl + Click derecho del ratón.
Ctrl+W+]	Saltar al tag de la palabra bajo el cursor en una ventana nueva.
:ptag subrutina_o_variable	Muestra la definición del tab en una ventana de preview. Para múltiples matches, podemos movernos con :ptnext y :ptprevious o sus abreviaturas :ptn y :ptp
Ctrl+W+}	Mostrar el tag en una ventana de preview.
:pc	Cerrar la ventana de preview.

Adicionalmente, podemos hacer que vim busque automáticamente el fichero adecuado con:

```
vim -t tag_a_buscar
```

Nótese que las combinaciones de teclas para saltar al tag bajo el cursor (Ctrl+]) y para volver (Ctrl+T) son bastante problemáticas… la primera, en el teclado españo, implica pulsar Ctrl+AltGr+], y la segunda puede provocar, en algunos gestores de terminales, el lanzar una nueva pestaña.

Para facilitar el uso de los tags, podemos establecer unas macros para modo comando en nuestro .vimrc:

```
""" Tags
nmap <F5> <C-]>
nmap <F6> <C-T>
```

Nótese que Vim utiliza el fichero "**tags**", creado al ejecutar ctags, como información para los saltos y etiquetas. Esta información es estática, es decir, no se actualiza hasta que volvemos a ejecutar ctags. Esto implica que si estamos analizando un programa ya finalizado, bastará con la ejecución inicial de ctags pero para programas "en crecimiento" deberemos lanzar ctags regularmente de forma que el fichero de tags esté actualizado.

En programas compilables, se puede poner la llamada a ctags en el Makefile aunque lo más cómodo es mapear (map) una tecla contra una llamada del tipo **!ctags miprograma.py** o similar. Existen también plugins (extensiones) dedicados a mantener los ctags (como **autotags**, **easytags**, **indexer**, etc).

Existen otros plugins de Vim como **taglist** o **tagbar** que regeneran los tags de forma automática y hacen uso de ellos para proporcionarnos una ventana de navegación vertical con la estructura del código, para movernos rápidamente entre variables, métodos, clases y funciones. Incluso podremos mapear una tecla del teclado para mostrar u ocultar la ventana de tags cuando la necesitemos.

Más adelante veremos cómo instalar plugins para hacer uso de estas interesantes funcionalidades.

Personalizando la barra de estado

La barra de estado de Vim proporciona información sobre el fichero que estamos editando, el modo de trabajo actual y, normalmente, la fila y columna en que está posicionada el cursor.

Esta barra de estado puede ser personalizada con múltiples parámetros, como por ejemplo:

```
""" Formato de la barra de estado, y posicion en la penultima linea
set statusline=%t\ %y\ format:\ %{&ff};\ [%c,%l]
set laststatus=2
```

La línea de estado tendría el siguiente aspecto:

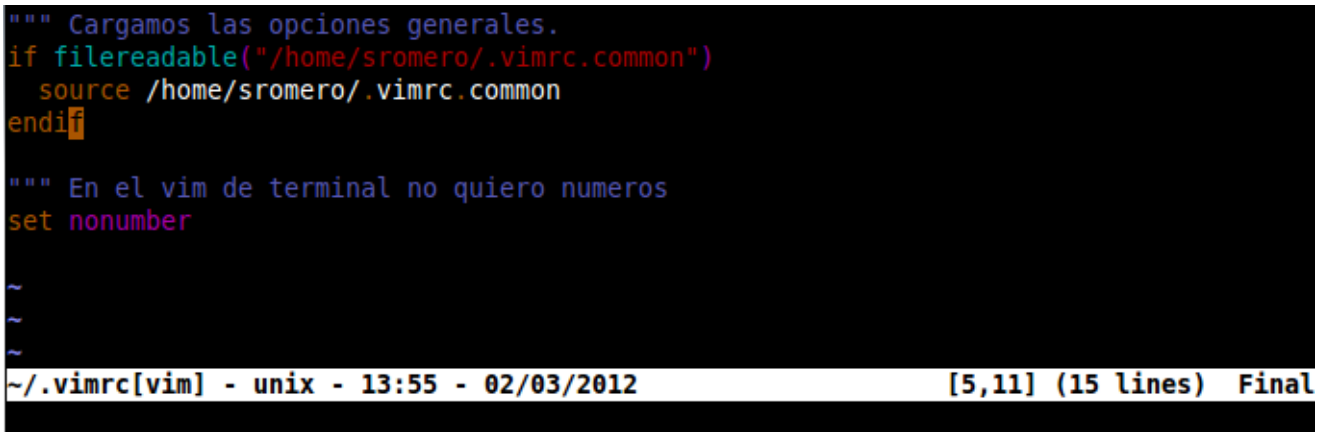
```
.vimrc.common [vim]      format: unix;        [133,22]
```

Los diferentes parámetros a utilizar para statusline los podemos obtener dentro de Vim con **:help statusline**.

Dentro de statusline podemos incluso hacer uso de funciones complejas como (ojo, línea partida pero debe ir toda en una sóla línea en el vimrc):

```
set statusline=%<%F%h%m%r%h%w%y\ \-\ %{&ff}\ \-\ %{strftime
(\"%H:%M\ \-\ %d/%m/%Y\",getftime(expand(\"%:p\")))}%
=\ [%c\,%l]\ \(%L\ lines\)\ \ %P
```

Y el resultado es:



Opciones de arranque de Vim

Veamos algunas opciones interesantes para lanzar el binario "vim" desde la línea de comandos:

Usando el operador "+", podemos lanzar Vim yendo directamente a una determinada línea, al final del fichero, o a la primera aparición de un patrón. También podemos usar `-u NONE` para lanzar Vim sin cargar nuestro `.vimrc` (vim "vanilla").

```
# Abrir fichero y posicionar el cursor en la línea 25:
$ vim fichero +25

# Abrir fichero y posicionar el cursor en la última línea:
$ vim fichero +

# Abrir fichero y posicionar el cursor en la primera aparición de PATTERN
$ vim fichero +/PATTERN

# Editar en Vim la salida de un comando (luego guardarlo con ":w out.txt").
$ command | vim -

# Lanzar vim sin leer nuestro .vimrc
$ vim -u NONE
```

Ejemplos de comandos útiles varios

A continuación mostramos una serie de ejemplos basicos útiles que no tienen cabida en otra sección pero que pueden resultar interesantes si el lector no los ha deducido ya para realizar una tarea concreta:

- **:%s/*$/** → Eliminar espacios en blanco al final de línea en todo el documento.
- **:%s/TABULADOR*$//** → Eliminar espacios en blanco y tabuladores al final de línea en todo el documento.
- **:g/cadena/d** → Eliminar todas aquellas líneas que contienen una cadena o patrón.
- **:g/^ *#/d** → Eliminar todas las líneas de comentarios de python (que desde el principo de línea haya 0 o más espacios y una almohadilla). Aplicación del ejemplo anterior.
- **:g!/cadena/d** → Eliminar todas aquellas líneas que no cumplen un patrón indicado.
- **:1,$ s/^V^M//** → Eliminar retornos de carro de MSDOS (las teclas ^V y ^M se corresponden con Ctrl+V y Ctrl+M y hay que pulsarlas primero una y después la otra, no teclear su texto).
- **vim sftp://usuario@destino.com/path/fichero.txt** → Edición remota de ficheros por SCP (u otros protocolos soportados).
- **gf** → Abrir el fichero cuyo nombre está ubicado bajo el cursor (especialmente útil para #includes, imports, etc). El fichero debe de estar en el mismo directorio que el directorio de trabajo. No obstante, podemos forzar a que Vim lo busque recursivamente en los directorios hijos o padres añadiendo en nuestro .vimrc la opción: "**set path +=./****" .
- **imap ii <Esc>** → Volver al modo comando desde el modo inserción al teclear 2 íes seguídas. Es complicado encontrar palabras y expresiones que tengan 2 íes seguidas (si no, podemos usar "jj"). Este mapeado permite cambiar entre comando e inserción siempre con la "i" ("ii" o "i").
- Abrir la URL (Uniform Resource Locator) presente en la línea actual en el navegador:

```
function! Browser ()
   let line = getline (".")
   let line = matchstr (line, "http[^   ]*")
   exec "!firefox -new-tab ".line
endfunction

map <F8> :call Browser ()<CR>
```

- Insertar una cadena al principio de múltiples líneas (por ejemplo, un comentario, un símbolo " +"…).
    - En la primera línea de las que queremos modificar, en la columna donde queremos agregar el texto (por ejemplo, la primera), pulsar CTRL+V para entrar en modo selección de bloque.
    - Pulsar abajo para bajar hasta la última línea (no importa el ancho de la selección, puede ser de 1 simple bloque).
    - Pulsar I (mayúsculas + i = insertar al principio de la línea, en este caso de la selección, aunque esté a mediados de la línea completa).
    - Escribir el texto.
    - Pulsar ESC. Pasado 1 segundo veremos cómo se añade el texto a todas las líneas justo en la columna donde agregamos el texto manualmente.
- Insertar una cadena al final de múltiples líneas.
    - En la primera línea de las que queremos modificar, al principio de la línea, pulsar CTRL+V para entrar en modo selección de bloque.
    - Pulsar abajo para bajar hasta la última línea (no importa el ancho de la selección, puede ser de 1 simple bloque).
    - Pulsar $ (fin de línea) para que se seleccione hasta el final de línea en todas las líneas.
    - Pulsar A (mayúsculas + a = insertar al final de la línea, en este caso de la selección).
    - Escribir el texto.
    - Pulsar ESC. Pasado 1 segundo veremos cómo se añade el texto a todas las líneas justo al final de cada línea, sin importar que no tengan todas el mismo ancho.
- Macros: Pulsa **q[a-z]** para empezar a grabar una macro en uno de las 26 macros (de la a a la z). Pulsa **q** en modo comando para parar la grabación. Después, se puede reproducr la macro con @**[a-z]**. La última macro ejecutada se puede lanzar de nuevo con @@ e incluso repetir varias veces con multiplicadores como **3**@@.
- Hacer que los cursores no funcionen para obligarnos a usar las teclas de movimiento h, j, k, l:

```
noremap <Up> <nop>
noremap <Down> <nop>
noremap <Left> <nop>
noremap <Right> <nop>
```

Este último ejemplo es recomendado por los puristas de Vim para enseñar a los usuarios a no utilizar los cursores y acostumbrarse a las teclas de movimiento h, j, k ,l. Estas teclas, en el centro del teclado, requieren cierto tiempo de aprendizaje pero permiten trabajar más rápido con Vim al no tener que llevar las manos hasta los cursores para el movimiento.

Atajos de teclado con la tecla "leader"

En Vim, existe una tecla especial llamada "leader", la cual es una especie de prefijo para ejecutar atajos de teclado personalizados. Por defecto, la tecla <Leader> (que es como se la referencia en el fichero rc) es la barra invertida \ .

Ahora que ya conocemos la existencia de <Leader>, podemos crear atajos de teclado en nuestro .vimrc para utilizarla:

```
" Leader + C = Introducir una línea de comentario y bajar a la siguiente linea

noremap <Leader>C i#-----------------------------------<ESC>j
```

Ahora, si en modo comando pulsamos "barra invertida", la soltamos, y antes de 1000 milisegundos (1 segundo) pulsamos C, se insertará la línea de comentario con los guiones, se volverá a modo inserción (con <ESC>) y se bajará a la siguiente línea (con "j").

La tecla \ es una tecla muy cómoda de usar en los teclados USA, pero a los Europeos nos resultará probablemente poco cómoda. Lo que podemos hacer es modificarla en nuestro vimrc para que <Leader> sea, por ejemplo, la coma (',') o el guion ('-'):

```
let mapleader = ","
```

Ahora, usando coma como Leader, podemos definirnos "macros" o atajos para cubrir nuestras necesidades. Por ejemplo, veamos uno para guardar el fichero actual como root (por ejemplo, si hemos editado un fichero como usuario y después de hacer los cambios, no podemos guardar):

```
" Leader + W = grabar el fichero actual como root

noremap <Leader>W :w !sudo tee % > /dev/null
```

Si por algún motivo necesitamos modificar el tiempo de 1000 milisegundos que tenemos para pulsar la segunda tecla tras <Leader>, podemos hacerlo con ":set timeoutlen".

Extendiendo Vim con plugins

Aunque ya de por sí Vim es un editor muy completo, resulta que podemos extender sus funcionalidades para hacer cualquier cosa que se nos pueda ocurrir dentro de las posibilidades del "lenguaje interno" que utiliza para su configuración y de los comandos que soporta.

En la web de vim podemos encontrar una sección denominada **scripts** que contiene plugins que extienden Vim con capacidades especiales. La URL (Uniform Resource Locator) concreta es http://www.vim.org/scripts/ (http://www.vim.org/scripts/) y este es el aspecto que tiene:

Searched scripts for "" Showing **1** to **20** of **3915** results

Script	Type	Rating	Down loads	Summary
taglist.vim	utility	9874	182734	Source code browser (supports C/C++, java, perl, python, tcl, sql, php, etc)
The NERD tree	utility	5682	102419	A tree explorer plugin for navigating the filesystem
rails.vim	utility	5205	65046	Ruby on Rails: easy file navigation, enhanced syntax highlighting, and more
vimwiki	utility	5038	9756	Personal Wiki for Vim
snipMate	utility	4908	49922	TextMate-style snippets for Vim
molokai	color scheme	4625	25118	A port of the monokai scheme for TextMate
c.vim	utility	4332	69729	C/C++ IDE -- Write and run programs. Insert statements, idioms, comments etc.
AutoComplPop	utility	3902	30652	Automatically opens popup menu for completions
minibufexpl.vim	utility	3450	74382	Elegant buffer explorer - takes very little screen space
vcscommand.vim	utility	3368	46702	CVS/SVN/SVK/git/hg/bzr integration plugin

Si pulsamos en [more] (abajo de la página) y hacemos una búsqueda por "Rating", podemos observar algunos de los scripts más populares (mejor puntuados), como por ejemplo:

- **snipMate** → Implementa en Vim los populares "snippets" del editor TextMate. Permite definir en ficheros específicos de cada lenguaje de programación pequeñas porciones de código que serán insertadas en el fichero cuando las "lancemos" mediante palabras clave seguidas del tabulador. Por ejemplo, escribiendo en modo inserción "class" seguido del tabulador, se insertará en el documento el snippet (toda una clase con sus comentarios, métodos de inicio, etc).
- **tagbar** → Utiliza ctags para generar una ventana vertical (que podemos mostrar u ocultar con un atajo de teclado) con la estructura del código fuente del proyecto y que permite saltar entre los diferentes métodos, funciones y ficheros de una forma sencilla.
- **supertab** → Cuando comenzamos a teclear una palabra en modo inserción, Vim saca un menu contextual con opciones de autocompletado que podemos seleccionar con los cursores y enter, o descartar al continuar escribiendo.
- **The NERD tree** → Este plugin habilita una ventana vertical (que podemos mostrar u ocultar con un atajo de teclado) con la estructura de directorios actual, para poder movernos entre ellos fácilmente.

Para mostrar cómo se instalan y cómo funcionan los plugins vamos a instalar **snipMate**.

Instalando snipMate manualmente

Primero, descargamos el plugin de esta URL (Uniform Resource Locator):

- http://www.vim.org/scripts/script.php?script_id=2540 (http://www.vim.org/scripts/script.php?script_id=2540)

Normalmente, los plugins son simples ficheros .vim a copiar dentro del directorio $HOME/.vim/plugins. También hay ficheros en formato .zip o .tar.gz que contienen más ficheros y que se deben desempaquetar dentro de $HOME/.vim/ .

```
$ cd $HOME/.vim/
$ unzip $HOME/snipMate.zip
Archive:  /home/sromero/Escritorio/snipMate.zip
  inflating: after/plugin/snipMate.vim
  inflating: autoload/snipMate.vim
  inflating: doc/snipMate.txt
  inflating: ftplugin/html_snip_helper.vim
  inflating: plugin/snipMate.vim
  inflating: snippets/_.snippets
  inflating: snippets/autoit.snippets
  inflating: snippets/c.snippets
  inflating: snippets/cpp.snippets
  inflating: snippets/html.snippets
  (etc...)
  inflating: snippets/vim.snippets
  inflating: snippets/zsh.snippets
  inflating: syntax/snippet.vim
```

Al desempaquetar el plugin, se crean una serie de ficheros: documentación, el fichero del plugin en sí mismo, y un directorio con los snippets personalizables.

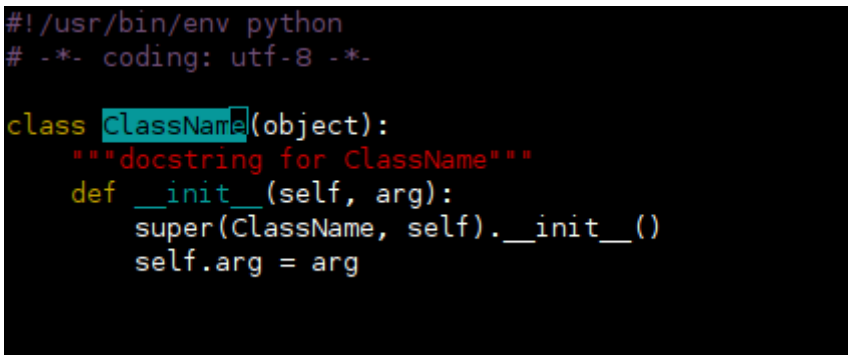
Estos ficheros, dentro de $HOME/.vim/snippets/ contienen (para cada lenguaje) los snippets que podemos utilizar. Por ejemplo, el fichero python.snippets contiene porciones como:

```
snippet docs
        '''
        File: ${1:`Filename('$1.py', 'foo.py')`}
        Author: ${2:`g:snips_author`}
        Description: ${3}
        '''

snippet for
        for ${1:needle} in ${2:haystack}:
                ${3:# code...}

snippet cl
        class ${1:ClassName}(${2:object}):
                """${3:docstring for $1}"""
                def __init__(self, ${4:arg}):
                        ${5:super($1, self).__init__()}
                        self.$4 = $4
                        ${6}
```

Para asegurarnos el correcto funcionamiento de este plugin, necesitamos tener esta línea en el .vimrc:

```
filetype plugin on
```

Si ahora editamos un fichero en python, y tecleamos "cl" y pulsamos el tabulador, aparecerá esto en el fichero:



Además, el campo ClassName queda seleccionado con el cursor encima, y si tecleamos modificaremos dicho campo, actualizándose automáticamente el nombre de la clase en la llamada a super(), más abajo. Usando Tabulador, podemos movernos hacia los siguientes parámetros y reemplazarlos.

Editando y personalizando los ficheros de snippets, podemos hacernos una excelente biblioteca que introduzca por nosotros funciones, clases, getters/setters, docstrings, cabeceras de ficheros estándar con sus licencias, esqueletos de documentos de marcado (HTML (HyperText Markup Language)/XML/etc), etc.

Como véis, es algo parecido a **iab** pero con las siguientes características avanzadas:

- Soporta parámetros y éstos se pueden referenciar dentro del snippet, podemos movernos entre ellos con el Tabulador y son autoactualizados conforme los cambiamos.
- Los "disparadores" de los snippets pueden ser idénticos en 2 lenguajes diferentes. Por ejemplo, el snippet **class** puede estar en el fichero de configuración python.snippets y c.snippets y provocará la inserción de un snippet diferente según el tipo de fichero de código que estamos editando. Es decir, las definiciones de los snippets dependen del lenguaje que estemos utilizando.
- Por supuesto, podemos editar los ficheros de snippets para extender los snippets de un determinado lenguaje según nos interese.
- Podemos poner en un mismo fichero varios snippets con el mismo nombre, y al lanzarlo aparecerá un menú para seleccionar cuál de las definiciones queremos. En python.snippets tenemos un ejemplo con 4 definiciones de **try** que permite insertar, vía menu de teclado, claúsulas try/except, try/except/else, try/except/finally y try/except/else/finally.

Las posibilidades son ilimitadas.

Gestión de plugins con Vundle

Instalar y mantener plugins a mano es algo que no se suele realizar en Vim. En su lugar, se utilizan herramientas de gestión de Plugins que facilitan mucho la tarea. Con estas herramientas (Pathogen, Vundle, NeoBundle…), no tenemos que descargar manualmente ningún plugin, ni desempaquetarlo en ~/.vim/, ni estar pendientes de sus actualizaciones.

Vamos a ver cómo se puede usar Vundle para gestionar nuestros plugins. Lo haremos con un vimrc de ejemplo y con los pasos necesarios para instalar Vundle y los plugins. Vamos a asumir un .vim sin plugins descargados (si tenemos alguno instalado manualmente, podemos mantenerlo, o bien borrarlo y agregarlo después a los plugins gestionados por Vundle).

Lo primero que haremos será instalar Vundle, que es un plugin en sí mismo:

```
# cd ~/.vim
# mkdir -p bundle
# git clone https://github.com/gmarik/Vundle.vim.git bundle/Vundle.vim
```

Una vez instalado Vundle (el plugin), editamos nuestro .vimrc e incluímos las líneas necesarias para inicializar Vundle, para decirle a Vundle qué plugins queremos, y para cerrar la inicialización:

```
""" Principio del fichero .vimrc, sólo necesitamos que contenga:
set nocompatible

""" Inicializamos Vundle (el filetype off es obligatorio
filetype off
set rtp+=~/.vim/bundle/vundle
call vundle#begin()

""" Declaramos los plugins deseados
Plugin 'bling/vim-airline'
Plugin 'majutsushi/tagbar'
Plugin 'godlygeek/tabular'
Plugin 'scrooloose/nerdcommenter'
Plugin 'mileszs/ack.vim'
Plugin 'vim-scripts/Command-T'
Plugin 'kien/ctrlp.vim'
Plugin 'scrooloose/nerdtree'
Plugin 'ervandew/supertab'
Plugin 'vim-scripts/TaskList.vim'
Plugin 'vim-scripts/ShowMarks'
Plugin 'garbas/vim-snipmate'
" tlib_vim and vim-addon-mw-utils are required by vim-snipmate:
Plugin 'tomtom/tlib_vim'
Plugin 'MarcWeber/vim-addon-mw-utils'
" Vundle gestiona vundle!
Plugin 'gmarik/Vundle.vim'

" Colour schemes to install
Plugin 'altercation/vim-colors-solarized'
Plugin 'tomasr/molokai'
Plugin 'benjaminwhite/Benokai'
Plugin 'sickill/vim-monokai'

""" Cierre de la inicialización de Bundle y activamos el filetype
call vundle#end()
filetype plugin indent on
let mapleader=","

""" Ahora, definir nuestras opciones genericas del VIM,
""" asi como las funciones de configuracion especificas
""" de los plugins declarados con Plugin.
set nobackup
set ruler
set novisualbell
set noerrorbells
set ttyfast
set nowrap
set autoindent
set expandtab
set tabstop=4
""" (etc etc etc...)
```

A la hora de definir los plugins que queremos instalar, las cadenas tras la keyword "Plugin", del tipo "A/B" se corresponden con "repositorio/NombrePlugin" de github, de forma que:

Plugin 'scrooloose/nerdcommenter'

Se refiere a https://github.com/scrooloose/nerdcommenter (https://github.com/scrooloose/nerdcommenter) .

Ahora ya le podemos decir a Vundle que descargue e instale los plugins, saliendo a la terminal y ejecutando:

```
vim +PluginInstall +qall
```

Al hacer esto, veremos cómo Vundle se conecta a todos los repositorios de github indicados y descarga todos los plugins dentro de ~/.vim/Bundle/ :

```
" Installing plugins to /root/.vim/bundle                              |  1
. Plugin 'bling/vim-airline'                                           |~
. Plugin 'majutsushi/tagbar'                                           |~
. Plugin 'godlygeek/tabular'                                           |~
(...)
```

Ahora ya podemos abrir Vim y veremos cómo los diferentes plugins están instalados. Para configurarlos, editamos nuestro fichero .vimrc y debajo de la configuración "genérica" que teníamos podemos incluir configuraciones específicas para los plugins:

```
" Nerd Tree: abrir en modo comando con ,n   (coma seguido de n):
let g:NERDTreeWinPos = "left"
let g:NERDTreeWinSize = 25
nmap <leader>n :NERDTreeToggle<CR>
map <F1> <ESC>:NERDTreeToggle<CR>

" Cerrar Vim si NerdTree es la unica ventana abierta
autocmd bufenter * if (winnr("$") == 1 && exists("b:NERDTreeType") &&
        \ b:NERDTreeType == "primary") | q! | endif


" Cargar el esquema de colores molokai (instalado con Vundle tambien)
colorscheme molokai

" etc...
```

Modo vi en bash

Si nos hemos hecho expertos en Vim y hemos cogido habilidad con sus atajos de teclado, podemos exprimir esta habilidad para mejorar nuestro uso de la terminal **bash**. Bash (por medio de readline) funciona por defecto en modo "emacs", pero podemos pasarla a "modo vi" añadiendo lo siguiente a nuestro .bashrc:

```
export EDITOR="vim"
export VISUAL="vim"
set -o vi
```

(también podemos ejecutar los anteriores comandos en una terminal ya abierta para probarlo, y volver a modo emacs después con **set -o emacs**).

Una vez en modo vi, podemos utilizar las teclas y atajos de Vim tal y como se explica en esta minireceta en inglés:

- Vi mode for bash readline [EN] (/wiki/linux/aplicaciones/bash_tips?&#vi_mode_for_bash_readline).

En resumen

En este sencillo tutorial hemos tratado los fundamentos básicos de este espléndido editor. La cantidad de opciones y funciones disponibles en Vim nos permitirá realizar una edición de cualquier tipo de fichero de texto o programa mucho más rápida y eficiente que con cualquier otro editor de textos.

Espero que este tutorial haya cubierto vuestras necesidades básicas de manejo de este imprescindible editor. En cualquier caso, si habéis llegado hasta aquí seguramente no será necesario que os recuerde la potencia de Vim: a estas alturas de tutorial la conoceréis de sobra. Por si fuera poco, ya sabéis que Vim puede ser extendido mediante el uso de plugins, por lo que sus posibilidades no quedan limitadas al programa "original".

Para finalizar, sólo recordaros que la documentación incluída con Vim es todo un libro en sí misma, y que la podéis completar con todos los recursos disponibles en Internet. Y para recordar todo el rosario de comandos existentes, lo mejor es practicar usando el editor, que (como me sucedió a mí) seguramente se os acabará convirtiendo en una herramienta imprescindible.

SANTIAGO ROMERO
http://www.sromero.org/ (http://www.sromero.org/)
Santiago Romero 2012-03-01
Updated on 2014-03-30

<Volver a la sección de GNU/Linux (/wiki/linux/indice)>