Tutorial de VIM para programar en C

Luis Alberto Giménez
algibe@teleline.es

2 de junio de 2005

Índice

1. Introducción

Vim es un editor de textos muy potente, nos permite hacer prácticamente cualquier cosa y es extremadamente configurable. Proviene de una versión mejorada del editor **vi**. Un punto a su favor es que está presente en alguna de sus formas en todos los UNIX que existen actualmente.

Éste *no* es un tutorial de introducción a **vim**. Aquí no se explicarán las órdenes básicas de movimiento ni de edición, ya que eso viene explicado ampliamente en otros tutoriales disponbiles en la red. En este pequeño tutorial nos centraremos en las posibilidades que nos brinda el editor de cara a la programación en el lenguaje **C**.

1.1. La ayuda de vim

Hay que remarcar aquí que todo lo explicado se ha sacado del manual de **vim**. Podremos acceder a él en cualquier momento pulsando la tecla *F1* o, desde el modo comando introducir `:help`. Los dos puntos sirven para entrar en el modo *ed* y nos ayudan a ejecutar ciertas funciones no disponibles en el modo comando.

Respecto a las palabras clave, órdenes y funciones que se comentan en este tutorial, diremos aquí que se puede acceder a su explicación utilizando la orden `:help` *`palabra`*.

En el modo *ed* funciona el completado de nombres mediante la tecla del tabulador, de modo que si escribimos:

```
:he
```

y apretamos la tecla tabulador, vim irá completando cíclicamente todas las posibilidades que empiecen por "he". Esto es muy útil si no recordamos exactamente alguna opción de la que queramos recordar para qué sirve.

Aconsejamos echar un vistazo a la ayuda de **vim** ya que es extensísima y en ella se pueden encontrar trucos que a simple vista no son evidentes. Además incluye un manual de usuario completísimo y un manual de referencia de todas sus funciones.

2. Funcionalidades internas

Vim tiene una serie de funcionalidades preestablecidas para hacernos más cómoda la programación en C. En la sección 3 veremos algunas de las funcionalidades de este editor que necesitan de algún programa externo.

2.1. Resaltado de sintaxis

El resaltado de sintaxis (o *syntax highlighting*) es una manera que tienen los editorres o visores de código para ayudarnos a identificar las palabras clave del lenguaje, las constantes, etc. **Vim** es muy bueno en esto, y no

sólo para el lenguaje C, reconociendo sintaxis y coloreando más de 300 tipos diferentes de archivos.

Para activar esta utilidad necesitamos hacerlo desde el modo *ed*:

```
:syntax on
```

Al momento se nos coloreará (o según el tipo de terminal que tengamos, aplicará estilos de negrita, subrayado o cursiva) el código. Podemos definir cómo queremos que nos coloree **vim** el código, las palabras que debe reconocer como claves, etc... en el archivo `/usr/share/vim/vim61/syntax/c.vim`, aunque viene ya con una definición del lenguaje muy buena y no es algo trivial de modificar.

A la hora de utilizar el resaltado de sintaxis no todas las terminales son iguales. **Vim** nos permite distinguir entre terminales de fondo oscuro y de fondo claro. Esto ayuda al editor a elegir los colores que más se adecuan al fondo que tengamos configurado en nuestra terminal. Para determinar el fondo de nuestra terminal utilizaremos la variable `background`:

```
:set background=color
```

donde `color` será:

dark cuando el fondo de nuestra terminal sea oscuro, para que sean utilizados colores más vivos y brillantes.

light cuando nuestra terminal sea clara, para que vim utilice unos colores más oscuros.

2.2. Indentado de código

El indentado de código es algo muy importante, ya que aumenta la legibilidad de nuestros programas y nos facilita la identificación de bloques de código. **Vim** acepta varios tipos de indentado, pero cómo no, tiene uno dedicado íntegramente a los archivos escritos con el lenguaje C.

Para activar el identado podemos utilizar `:set cindent`, y para desactivarlo, `:set nocindent`. Hay que decir que todos los parámetros de indentado son configurables. Podemos también utilizar `:set autoindent`, que indenta las líneas con el indentado de las anteriores. Funciona igualmente bien (aunque utiliza internamente el indentado de C).

A la hora de cerrar bloques con el carácter '}', **vim** automáticamente quita un nivel de indentado al código y mueve este carácter a su lugar correcto, haciendo innecesario el tener que borrar el supuesto tabulador extra que proviene del indentado automático. Como vemos, con **vim** todo son facilidades.

Hay que decir que el indentado se apoya en la introducción de ciertos carácteres, como por ejemplo las llaves '{', '}', los paréntesis y algunos más.

Si nos equivocamos al poner éstos, puede ser que **vim** no nos indente correctamente el código. De hecho, como veremos en la sección 2.3, éste nos resaltará los paréntesis desemparejados.

Cuando esto ocurra, aunque corrijamos el código, el editor no corregirá automáticamente el indentado. Aquí es donde entra el comando `=`. Este operador (como prácticamente todos los de **vim**) acepta un argumento de "rango".

Veamos un ejemplo: En este caso, utilizaremos un bloque *if*. Imaginemos que olvidamos el paréntesis que cierra la expresión de comparación. Al abrir una nueva línea, nos hará un doble indentado (porque se piensa que estamos continuando la expresión, en lugar de escribiendo sentencias):

```
if (expresion
        sentencia;
```

Para corregirlo, debemos añadir el paréntesis que falta, cerrando la expresión, ir a la línea de la `sentencia`, y, desde el modo comando, presionar `==`, lo cual nos indentará correctamente la sentencia:

```
if (expresion)
    sentencia;
```

Se puede pensar que hacer esto para cada línea que tengamos mal indentada puede ser muy trabajoso. Aquí es donde entran los argumentos de rango. Para facilitar el trabajo, podemos indicar al operador `=` que reindente n líneas, todo un bloque entre '{' y '}', un bloque seleccionado en el modo visual o incluso el archivo entero fácilmente. A modo de ejemplos:

- `=2j` Corregiría el indentado de las siguientes dos líneas (j es intercambiable por el cursor abajo).
- `=a{` Reindenta todo el bloque entre { y }, ya sea una función, un bloque *if*, etc...
- `=G` Reindenta hasta el final del fichero. Podemos movernos a la primera línea de éste para hacer que el indentado sea de todo el fichero.

Dentro de esta sección de indentado, es obligatorio hablar de los tabuladores. Hay varias corrientes entre los programadores: sustituir tabuladores por espacios, utilizar un carácter tabulador, etc...

Personalmente yo utilizo un carácter tabulador, me es más cómodo y puedes configurar tu editor para visualizar el número de espacios que quieras. Además a la hora de borrar "tabuladores" es más simple que borrar "n espacios", aunqe esto sea automático con **vim** (¿quién lo dudaba?).

Para configurar el número de espacios que utiliza cada tabulador, utilizaremos la variable `tabstop`, y para definir el número de espacios que se

insertarán por cada indentado (mediante indentado automático o con órdenes a vim), utilizaremos `shiftwidth`.

Es aconsejable dejar `tabstop` y `shiftwidth` a la misma cantidad de espacios que queramos ver. Así mantendremos los tabuladores (no serán expandidos a espacios). Para mejorar la legibilidad del código es mejor que los tabuladores no estén representados por menos de 4 ó 5 espacios. Como ejemplo, mi configuración personal es:

```
:set tabstop=4
:set shiftwidth=4
```

2.3. Paréntesis y llaves

Los paréntesis y las llaves son fundamentales en la programación en C. **Vim** tiene funcionalidades para ayudarnos a no olvidar ningún paréntesis.

En primer lugar, tenemos el comando `%`, que nos ayuda a encontrar las parejas de paréntesis o llaves. En concreto, busca el paréntesis que cierra o abre al que tenemos bajo el cursor. Por ejemplo, en esta sentencia:

```
a = (((b + c) * d) / (a + f));
```

Si colocamos el cursor sobre el primer paréntesis '(' y presionamos la tecla %, **vim** saltará el cursor hasta el último paréntesis, que es el que le cierra. Esto es útil para encontrar paréntesis desemparejados, ya que en este caso el editor emitirá un pitido y no saltará a ningún lugar.

Esta tecla es muy útil, pero a veces, a medida que introducimos código no llevamos la cuenta de los paréntesis que llevamos abiertos, o si aún no hemos cerrado el primero que abrimos.... Con el indentado que nos hace **vim**, si nuestra sentencia ocupa más de una línea nos podemos hacer una idea de si nos hemos equivocado con los paréntesis.

Hay otro método que nos proporciona vim para ayudarnos con los paréntesis. Se activa mediante `:set showmatch`. Cuando tenemos esta opción activada, cada vez que cerremos un paréntesis el cursor saltará brevemente a la posición del paréntesis al que éste cierra. Así podremos comprobar que vamos cerrando correctamente nuestros paréntesis. Adicionalmente, si introducimos un paréntesis de cierre extra, **vim** lo restaltará indicándonos que está desemparejado. ¿Quién da más?.

Cabe remarcar que en esta sección nos hemos referido a *paréntesis*, pero todo lo comentado funciona también para las llaves ('{', '}'), los corchetes ('[', ']') u otros carácteres definibles por nosotros mismos (gracias a la gran configurabilidad de **vim**).

2.4. Folding (plegado) de código

Una de las cosas que la gente echa de menos en **vim** es el hecho de poder "cerrar" el código de las funciones para mejorar la visibilidad de éste. Pues

vim lo hace, pero tenemos que decírselo explícitamente (recordad que no sólo se usa para editar programas).

Cuando pleguemos un trozo de código, será sustituido por una sóla línea resaltada con el número de líneas que contiene y el contenido de la primera de las líneas que se han plegado. Con el plegado podemos trabajar como si lo hiciéramos con todas las que contiene a la vez (cortar, copiar, ...):

```
int filter_mode (void)
+-- 98 lines: {----------------------------------------
```

En este caso hemos plegado una función que ocupa 98 líneas, y sólo se nos muestra la llave de apertura de la función (su primera línea).

Hay varios métodos para que **vim** reconozca cómo debe plegar nuestro código. Nos centraremos en los dos más fáciles de usar:

indent Utiliza los niveles de indentado en el código para determinar qué lineas plegar.

syntax Utiliza una definición similar a la del resaltado de sintaxis para decidir cómo plegar las líneas.

El mejor método es el segundo, ya que utiliza la definición de sintaxis propia que utiliza **vim** para resaltar el código. La contrapartida es que estas definiciones no son fáciles de escribir, es necesario un cierto conocimiento de los ficheros de sintaxis. Una regla sencilla para utilizar en nuestros ficheros en C podría ser:

```
:syntax region myFold start="{" end="}" transparent fold
:set foldmethod=syntax
```

Eso definiría como codigo a plegar todo aquél que se encontrara entre los carácteres '{' y '}', con lo que podríamos plegar funciones, bloques *if*, *switch*, etc. Además, activa el método de **sintaxis** de plegado.

Para utilizar el método **indent**, lo único que debemos hacer es:

```
:set foldmethod=indent
```

Y a cada nivel de indentado, nos creará un plegado. Todas las órdenes de plegado tienen en común su primer carácter: **z**. Fijaos que la *z* nos podría recordar una hoja de papel doblada vista de lado [1]. Una vez activado correctamente el plegado, las órdenes básicas para trabajar con éste son:

zo Abre un nivel (el más externo) situado bajo el cursor. Por ejemplo, teniendo el siguiente código:

[1] O eso dice el manual de Vim

```
int mifuncion(void)
{
    sentencia;
    sentencia;

    if (expresion) {
        sentencias;
        sentencias;

        if (expresion2) {
            sentencias;
            sentencias;
        }

        sentencias;
        sentencias;
    }
}
```

Si lo cerramos y desde el modo comando hacemos `zo`, obtendremos:

```
int mifuncion(void)
{
    sentencia;
    sentencia;

+--- 12 lines: if (expresion) {----------------------------------
}
```

zO Abre todos los niveles recursivamente. Por lo tanto, con esta orden obtendríamos el desplegado de toda la función.

zc Cierra el plegado correspondiente al lugar donde tengamos el cursor.

zC Cierra todos los plegados bajo el cursor recursivamente.

zR Abre todos los plegados del fichero que estemos editando.

zM Cierra todos los plegados del fichero.

2.5. Completado de código

A estas alturas, no deberíamos dudar que **vim** soporta el completado de código (y no solo de código), a pesar de ser una de las funcionalidades que más choca a la gente (que no sabe que **vim** es más que un simple editor de textos).

Para completar un nombre que tenemos comenzado, dentro del mismo modo inserción, tenemos estas dos órdenes:

CONTROL-n Completa buscando concordancias hacia adelante en el fichero (volviendo al principio si llega al final).

CONTROL-p Completa buscando hacia atrás.

El completado de código funciona también con los ficheros que hayamos incluído, y los incluídos en estos (recursivamente). Como comentario final, decir que estas dos órdenes forman parte de un submodo dentro del modo *inserción* que sirve para completar ciertos elementos. Como ejemplo, podemos decir que con:

```
CONTROL-x CONTROL-f
```

Completaríamos un nombre de fichero. Para entrar en este modo utilizamos la combinación `CONTROL-x` desde dentro del mismo modo *inserción*, y saldremos de él con cualquier tecla o combinación que no sea válida en el modo de completado.

Para ilustrar mejor el funcionamiento del completado veamos un ejemplo. Supongamos que tenemos el siguiente código:

```
int funcion1(void);
int funcion2(void);
int funcion3(void);

int funcion(void)
{
    fun
}
```

Si estamos escribiendo `fun` y apretamos la combinación `CTRL-n`, veremos que **vim** nos completa la **siguiente** coincidencia, y en la última línea nos sale un indicativo de que estamos en el modo de completado:

```
int funcion1(void);
int funcion2(void);
int funcion3(void);

int funcion(void)
{
    funcion1
}

-- Keyword completion (^N/^P) match 1
```

Si volvemos a apretar `CTRL-n`, nos saldrá `funcion2`, y así sucesivamente, hasta volver al texto original y vuelta a empezar.

Podemos completar también las macros con `CTRL-x CTRL-d`.

2.6. Identificadores

Muchas veces nos ha pasado que no recordamos de qué tipo es una variable (¿`unsigned`? ¿`long`?), o que no recordamos los argumentos que le tenemos que pasar a cierta función o el orden de éstos.

Podemos ir al fichero donde está declarado, pero podríamos perder el hilo de lo que estábamos haciendo o simplemente resultarnos incómodo. **Vim** tiene varias funcionalidades que nos ayudan en esta tarea.

En primer lugar tenemos las órdenes `gd` y `gD` (*goto declaration*), que `saltan` al lugar donde se declara una variable local o global, respectivamente. Podemos volver al lugar que estábamos editando con la orden **''**, que no se explicará aquí pero podéis buscar información en la ayuda con `:help ''`.

Estas órdenes son bastante limitadas, ya que no buscan en los ficheros incluídos. Pero **vim** tiene una orden más potente para la búsqueda de identificadores: `[I`. Cuando pulsemos esta combinación cuando estemos encima de una palabra, nos aparecerá una lista de todas las coincidencias en el archivo actual, los incluidos, etc...junto con el nombre de archivo y línea de la definición.

Como ejemplo veamos qué nos dice **vim** sobre nuestra `funcion2` del código de ejemplo de la página 8:

```
fold1.c
  1:    2 int funcion2(void);
```

Como vemos, nos ha encontrado la definición de `funcion2` en la línea 2 del archivo `fold1.c`.

3. Funcionalidades externas

Hay muchas utilidades dentro de **vim** que necesitan de un programa externo para poder ser utilizadas. Algunas de ellas son básicas dentro de la programación en C.

En esta sección hablaremos de las más importantes: los *tags*, la ejecución de órdenes del shell y la estrecha relación que tiene **vim** con *make*, la herramienta para compilar nuestros programas fácilmente.

3.1. Tags

Un tag es un identificador (como una etiqueta) que nos sirve para movernos a lo largo de nuestro programa. Podemos saltar de etiqueta en etiqueta, volver atrás, ...

Estas etiquetas, en lo que nos respecta, serán los nombres de funciones, variables, estructuras y demás elementos importantes del código en C que estemos editando, por ejemplo, si tenemos el código:

```
int funcion(char string[]) {
int numero;

numero = cuenta_repetidos(string);

return numero;
}
```

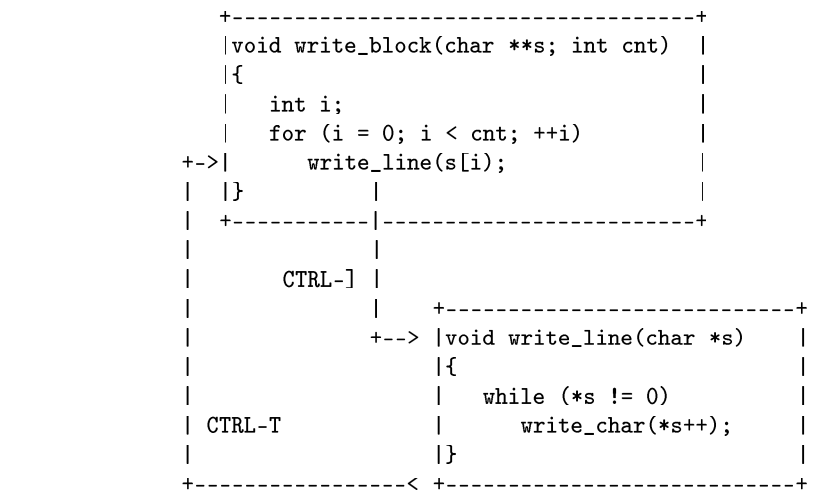
Y queremos saber cómo es la función `cuenta_repetidos`, podemos saltar a ella directamente, sin saber el fichero en la que está escrita, ni siquera salir del editor.

Para generar estas etiquetas o *tags* necesitamos un programa externo llamado *ctags* o *exuberant-ctags*, una versión mejorada del anterior. Queda fuera del ámbito de este tutorial describir cómo funciona este programa, pero debería ser suficiente la orden:

```
$ ctags -R .
```

para generar un fichero de *tags* de todos los ficheros fuente (*.c y *h) del directorio actual y los hijos, recursivamente.

Una vez creado, podemos navegar por las *tags* mediante la combinación de teclas `CTRL-]` (en modo comando). Para volver atrás tenemos la combinación `CTRL-T`. Veamos un ejemplo, sacado del mismo manual de **vim**:



Todas las tags por donde pasamos quedan guardadas en una "pila" que podemos ver en todo momento con `:tags`

Si hay varias coincidencias (por ejemplo, la definición de una función en un fichero fuente y su declaración en un fichero cabecera), **vim** así nos lo indicará y podremos utilizar las ordenes `:tnext` y `:tprevious` para pasar a la siguiente y anterior coincidencias, respectivamente.

Hemos dicho ya anteriormente que el modo *ed* (al que se entra con los dos puntos ':') soporta también completado, lo que también nos va a ayudar con las *tags*. Con la orden `:tag` *`etiqueta`* podemos saltar a la etiqueta que queramos directamente. El completado nos puede ayudar si no recordamos el nombre completo de la etiqueta. Por ejemplo, teniendo en cuenta el código anterior, podríamos utilizar:

```
:tag write_<TAB>
```

y **vim** nos completaría la etiqueta con `write_line`.

3.2. Compilación con gcc

Desde dentro de **vim** podemos ejecutar órdenes externas mediante la shell. Esto se hace mediante `:!`*`orden`*, donde *orden* es la que queremos que sea ejecutada.

Con ayuda de esa función podemos ejecutar el compilador con nuestro archivo o con cualquiera. Por ejemplo, para compilar nuestro archivo en un ejecutable llamado `out`, podemos hacer:

```
:!gcc % -o out
```

Nótese que utilizamos el comodín `%` para referenciarnos al fichero actual.

Hay que decir que éste no es un método muy óptimo para compilar programas, sobretodo si éstos constan de varios ficheros y hay dependencias entre ellos. Aunque para pequeñas pruebas puede ser útil, es mejor utilizar lo que se explica en la siguiente sección: el uso de *make* junto con **vim**.

3.3. Compilación con make

Lo normal cuando se programa en C algo que conste de más de un par de ficheros es utilizar la herramienta *make*. Aquí no se va a explicar cómo funciona, ya que se supone que se sabe como escribir un *Makefile* para compilar un programa.

Vim se integra perfectamente con la herramienta *make*. Reconoce los posibles errores que nos devuelve la compilación y automáticamente abre el fichero en la línea que dio error. Podemos pasar también argumentos al programa (por si nos interesa compilar un objetivo en concreto).

Para llevar a cabo una compilación utilizaremos la orden `:make`. Cuidado, no hay que confundir esto, que es una orden interna de **vim** con una ejecución externa del programa *make* (`:!make`).

Una vez ejecutada la orden, veremos brevemente la salida de las órdenes correspondientes del *Makefile*, hasta que éstas acaben, con un error o bien con una compilación correcta. En caso de una compilación correcta nos devolverá al punto que estuvieramos editando. Si ha habido errores o avisos,

vim saltará a la línea del archivo que lo provocó, mostrándonos el error en la línea de estado. Veamos un ejemplo:

```
#include<stdio.h>
void escribir_mensaje(void)
{
    printf("Hola, mundo!\n");
}

int main(void)
{
    escribir_mensaje("hola");

    return 0;
}
```

En ese código, la función `escribir_mensaje` no acepta parámetros, con lo que al construir el ejecutable con `:make`, nos dará la salida de la ejecución de *make* con el error:

```
main.c: In function ‘main’:
main.c:9: too many arguments to function ‘escribir_mensaje’
make: *** [main] Error 1

Hit ENTER or type command to continue
```

Si presionamos la tecla `ENTER`, volveremos al editor justo en la línea del error, pudiendo corregirla, y el error devuelto se mostrará en la línea de estado:

```
(3 of 5): too many arguments to function ‘escribir_mensaje’
```

En ocasiones ocurre que todo el texto del error no cabe en una sóla línea, por lo que aparece cortado. Para verlo entero podemos usar la orden `:cc`.

Cuando nuestro programa tenga más de un error podemos pasar de uno a otro con `:cnext` y `:cprevious`, respectivamente para el siguiente y el anterior.

Como vemos, compilar y depurar programas desde dentro de nuestro editor favorito es muy sencillo gracias a la gran potencia de **vim**.

De hecho no sólo estamos limitados a compilar archivos en C. Podemos indicar al editor el programa que queramos que se ejecute cada vez que usamos `:make`, e incluso podemos indicarle cómo es el formato de la salida de éste para que identifique los errores y así poder tener una funcionalidad similar a la que tenemos con el compilador *gcc*, pero eso ya es otra historia...

4. El archivo de configuración

Durante gran parte de este tutorial se han visto órdenes de **vim** que modifican su comportamiento como por ejemplo `:syntax on`. Posiblemente

muchas de estas opciones las queremos utilizar a diario y nos es incómodo tener que introducirlas cada vez que abramos el editor.

Para guardar estas opciones por defecto tenemos el archivo `.vimrc` de nuestro directorio *home*, donde las guardaremos tal y como las escribimos dentro del mismo **vim**, pero sin los dos puntos. Por ejemplo, el siguiente fichero `.vimrc`:

```
syntax on
set showmatch
set cindent
```

Haría que cada vez que abríeramos el editor se activaran las opciones de resaltado de sintaxis, emparejado de paréntesis y el indentado especial de código C.

5. Resumen

Como hemos visto, con **vim** tenemos muchas ayudas y características bastante desconocidas que hacen de éste un editor muy potente, flexible, y muy superior a muchos otros disponibles. En resumen, **vim** es más que un editor de textos. Es El Editor.

De hecho, quedan muchísimas cosas en el tintero, pero se ha intentado reflejar en este tutorial las características más importantes que tiene **vim** de cara a la programación en el lenguaje C. Como se ha comentado en la sección 1.1, podemos utilizar la muy extensa documentación que trae el programa incorporada, que consta desde un manual de usuario (navegable mediante *tags*) hasta una completa referencia de todas las órdenes, opciones, variables, etc... De obligada consulta si queréis dominar un poco más este fantástico editor y descubrir funcionalidades que no se pueden ni imaginar otros programas similares.

A. Referencia

A.1. Opciones de configuración

Orden	Significado
`:syntax on`	Activa el resaltado de sintaxis
`:set background=`*`color`*	Especifica el conjunto de colores a usar respecto del fondo de nuestra terminal
`:set cindent/nocindent`	Activa/desactiva el indentado de C
`:set autoindent`	Activa el indentado automático (sigue el de la anterior línea)
`:set tabstop=4`	Los tabuladores serán de 4 espacios.
`:set shiftwidth=4`	Los indentados serán de 4 espacios.
`:set showmatch`	Activa el mostrado de los paréntesis emparejados
`:set foldmethod=syntax`	Activa el plegado de código según sintaxis
`:set foldmethod=indent`	Activa el plegado de código según indentado

A.2. Órdenes internas

Orden	Significado
`:help`	Abre una nueva ventana con la ayuda de **vim**
`==`	Corrige el indentado de la línea actual
`=a{`	Corrige el indentado de todo un bloque de código
`%`	Busca la pareja del paréntesis o llave actual.
`zo`	Despliega un bloque
`zO`	Despliega los bloques recursivamente
`zc`	Pliega un bloque
`zC`	Pliega los bloques recursivamente
`zR`	Despliega todos los bloques del fichero actual
`zM`	Pliega todos los bloques del fichero actual
`CTRL-n`	Completa la palabra buscando hacia delante
`CTRL-p`	Completa la palabra buscando hacia atrás
`CTRL-x CTRL-f`	Completa un nombre de fichero
`CTRL-x CTRL-d`	Completa una macro
`gd`	Salta a la definición local del identificador
`gD`	Salta a la definición global del identificador
`[I`	Muestra la definición del identificador (busca en ficheros incluídos).

A.3. Órdenes externas

Orden	Significado
`CTRL-]`	Saltar a la etiqueta bajo el cursor
`CTRL-T`	Volver de un salto a etiqueta
`:tags`	Muestra la pila de etiquetas
`:tag` *etiqueta*	Salta a una etiqueta determinada
`:tnext`	Salta a la siguiente etiqueta coincidente
`:tprevious`	Salta a la anterior etiqueta coincidente
`:!`*orden*	Ejecuta *orden* en una shell
`:make`	Ejecuta la herramienta *make* analizando la salida
`:cc`	Ver un mensaje de error de compilación entero (si no cabe en la línea de estado)
`:cnext`	Salta al siguiente mensaje de error
`:cprevious`	Salta al anterior mensaje de error