usr_41.txt   Per Vim version 6.3. Ultima modifica: 2004 Mag 06

VIM USER MANUAL - di Bram Moolenaar
Traduzione di questo capitolo: Antonio Colombo

Preparare uno script Vim

Il linguaggio script di Vim è usato per il file di inizializzazione vimrc, nei file di sintassi, e molto altro. Questo capitolo spiega gli elementi che possono venir usati in uno script di Vim. Ce ne sono molti, così questo capitolo è lungo.

  Capitolo seguente: |usr_42.txt| Aggiungere nuovi menù
Capitolo precedente: |usr_40.txt| Definire nuovi comandi
             Indice: |usr_toc.txt|

==========

41.1 Introduzione *vim-script-intro*

Il vostro primo incontro con gli script di Vim è il file vimrc. Vim lo legge all'avvio e ne esegue i comandi. Potete impostare delle opzioni con il valore che preferite, ed usare al suo interno ogni comando che cominci per ":" (questi comandi sono talora designati come comandi Ex o comandi della linea di comando).
   Anche i file di sintassi sono degli script Vim. E lo sono pure i file che impostano opzioni per uno specifico tipo di file. Una macro complessa può essere definita da un file script di Vim separato. Potete pensare ad altri usi voi stessi.

Partiamo da un semplice esempio: **>**

```
        :let i = 1
        :while i < 5
        :  echo "il contatore è" i
        :  let i = i + 1
        :endwhile
<
```

        Note:
        I ":" non sono obbligatori in questo caso. Vanno necessariamente usati solo se immettete un comando direttamente. In un file di script Vim possono essere omessi. Noi li useremo comunque, per sottolineare che questi sono comandi coi ":" e per distinguerli dai comandi che si danno in Normal mode.

Il comando ":let" assegna un valore a una variabile. In forma generale: **>**

```
        :let {variabile} = {espressione}
```

In questo caso il nome della variabile è "i" e l'espressione è un valore semplice, il numero 1.
   Il comando ":while" inizia un ciclo. In forma generale: **>**

```
        :while {condizione}
        :  {comandi}
        :endwhile
```

I comandi fino ad ":endwhile" che chiude il ciclo sono eseguiti finché la condizione rimane verificata. La condizione qui usata è l'espressione "i < 5". Questa espressione è vera finché la variabile i assume valori inferiori a 5.
   Il comando ":echo" visualizza gli argomenti che gli vengono passati. In questo caso la stringa di caratteri "il contatore è" ed il valore della variabile i. Poiché i vale 1, visualizzerà:

```
        il contatore è 1 ~
```

Poi c'è un altro ":let i =". Il valore usato è l'espressione "i + 1". Questo aggiunge 1 alla variabile i ed assegna il nuovo valore alla variabile stessa.
   Il risultato del codice di esempio è:

```
        il contatore è 1 ~
        il contatore è 2 ~
        il contatore è 3 ~
        il contatore è 4 ~
```

        Note:
        Se scriveste un ciclo che duri più del previsto, potete interromperlo
        battendo CTRL-C (CTRL-Break in ambiente MS-Windows)

TRE TIPI DI NUMERI

I numeri possono essere decimali, esadecimali od ottali. Un numero esadecimale inizia con "0x" o con "0X". Ad esempio "0x1f" è 31. Un numero ottale inizia con uno 0. "017" è 15. Attenzione: non mettete uno zero davanti ad un numero decimale, per non farlo interpretare come un numero ottale!
   Il comando ":echo" stampa sempre numeri decimali. Ad es.: **>**

```
        :echo 0x7f 036
<       127 30 ~
```

Un numero diventa negativo quando è preceduto da un segno "-". Questo vale anche per numeri esadecimali ed ottali. Un segno "-" indica anche una sottrazione. Confrontate questo esempio con il precedente: **>**

```
        :echo 0x7f -036
<       97 ~
```

Gli spazi bianchi in una espressione vengono ignorati. Comunque se ne raccomanda l'uso per separare gli elementi e rendere l'espressione di più facile lettura. Per esempio, per evitare confusione con un numero negativo, mettete un spazio fra il segno "-" e il numero che lo segue: **>**

```
        :echo 0x7f - 036
```

==
41.2 Variabili

Un nome di variabile è composto di lettere ASCII, cifre ed il carattere "_". Il primo carattere del nome non può essere un numero. Nomi validi di variabile sono:

```
        contatore
        _aap3
        nome_molto_lungo_di_variabile_con_sottolineature
        LunghezzaFunz
        LUNGHEZZA
```

Nomi di variabile non validi sono "pippo+pluto" e "6var".
   Queste variabili sono globali [valgono per tutti i buffer di una sessione Vim - NdT]. Per vedere una lista di tutte le variabili correntemente definite usate questo comando: **>**

```
        :let
```

Potete usare variabili globali in qualsiasi posto. Questo implica anche che quando la variabile "contatore" è usata in un file di script, potrebbe anche essere usata in un altro file. Questo porta quanto meno a confusione, od al peggio crea problemi veri. Per evitarli, potete usare una variabile locale per un file di script, mettendogli come prefisso "s:". Ad esempio, uno script contiene questi comandi: **>**

```
        :let s:contatore = 1
        :while s:contatore < 5
        :  source altro.vim
        :  let s:contatore = s:contatore + 1
        :endwhile
```

Poiché "s:contatore" vale solo per questo script, potete essere certi del fatto che lo script "altro.vim" non ne cambierà il valore. Se "altro.vim" usa una variabile "s:contatore", si tratterà di una copia diversa, propria di quello script. Per saperne di più sulle variabili locali ad uno script si veda: |script-variable|.

Ci sono ulteriori tipi di variabili, si veda |internal-variables|. Quelle usate più spesso sono:

```
        b:name          variabile locale propria di un buffer
        w:name          variabile locale propria di una finestra (window)
        g:name          variabile globale (anche in una funzione)
        v:name          variabile predefinita da Vim
```

ANNULLARE VARIABILI

Le variabili occupano memoria e vengono visualizzate nell'output del comando ":let". Per cancellare una variabile usate il comando ":unlet". Ad es.: **>**

```
        :unlet s:contatore
```

Questo cancella la variabile di script locale "s:contatore" e libera la memoria che essa occupava. Se non siete sicuri dell'esistenza della variabile, e non volete vedere un messaggio di errore nel caso la variabile non esista, aggiungete al comando il "!" **>**

```
        :unlet! s:contatore
```

Quando uno script Vim termina, le variabili locali che ha usato non saranno annullate automaticamente. La prossima volta che lo script verrà eseguito, potrà ancora usare il vecchio valore della variabile. Ad es.: **>**

```
        :if !exists("s:contatore_chiamate")
        :  let s:contatore_chiamate = 0
        :endif
        :let s:contatore_chiamate = s:contatore_chiamate + 1
        :echo "Chiamato" s:contatore_chiamate "volte"
```

La funzione "exists()" controlla l'esistenza di una variabile. Il suo argomento è il nome della variabile di cui volete controllare l'esistenza. Non la variabile in sé! Se scriveste: **>**

```
        :if !exists(s:contatore_chiamate)
```

Allora il valore di s:contatore_chiamate sarebbe usato come nome della variabile la cui esistenza viene controllata da exists(). Questo non è ciò che volete.

   Il punto esclamativo ! nega un valore. Quando il valore della funzione era "vero" (true) diventa "falso" (false). Quando era "falso" diventa "vero". Potete leggerlo come la parola "non" (not). Ossia "if !exists()" si può leggere come "if not exists()", ovvero "se è falso il valore ritornato da exists()".

   Vim considera "vero" qualsiasi valore diverso da zero. Solo lo zero è falso.

STRINGHE VARIABILI E COSTANTI

Finora alle variabili sono stati assegnati solo valori numerici. Si possono usare come valori anche stringhe di caratteri. Numeri e stringhe sono i due soli tipi di variabile che Vim consente. Il tipo della variabile è dinamico, viene impostato ogni volta che si assegna un valore alla variabile stessa con ":let:".

   Per assegnare il valore di una stringa ad una variabile, dovete usare una costante di tipo stringa. Ne esistono due tipi. La prima è una stringa di caratteri, racchiusa in doppi apici: **>**

```
        :let nome = "pietro"
        :echo nome
<       pietro ~
```

Se volete inserire un doppio apice all'interno della vostra stringa, metteteci davanti un "\" (barra retroversa). **>**

```
        :let nome = "\"pietro\""
        :echo nome
<       "pietro" ~
```

Per evitare di usare "\", potete racchiudere la stringa fra apici: **>**

```
        :let nome = '"pietro"'
        :echo nome
```

```
<       "pietro" ~
```

All'interno di una stringa racchiusa fra apici, tutti i caratteri restano invariati. Lo svantaggio è quello di non poter inserire un apice da solo. Un "\" viene anch'esso preso alla lettera, così non lo potete usare per cambiare il significato del carattere che lo segue.
   Nelle stringhe racchiuse fra doppi apici è possibile usare i caratteri speciali. Eccone qualcuno utile:

```
        \t              <Tab>   tabulatore
        \n              <NL>    a capo
        \r              <CR>    <Invio>
        \e              <Esc>   escape
        \b              <BS>    "cancella un carattere"
        \"              "       doppi apici
        \\              \       barra retroversa
        \<Esc>          <Esc>   escape (forma alternativa)
        \<C-W>          CTRL-W carattere "iniziale" per spostarsi tra finestre
```

Gli ultimi due sono solo a titolo di esempio. La forma "\<nome_car>" si può usare per inserire il carattere speciale "nome_car".
   Si veda |expr-quote| per la lista completa degli elementi speciali di una stringa.

==
41.3 Espressioni

Vim ha un modo ricco ma semplice per gestire le espressioni. Potete leggere qui la definizione: |expression-syntax|. Qui mostreremo gli elementi più comuni.
   I numeri, le stringhe e le variabili citate, sono già espressioni. Così ovunque ci si aspetti un'espressione, potete usare un numero, una stringa od una variabile. Altri elementi fondamentali in una espressione sono:

```
        $NOME           variabile d'ambiente
        &nome           opzione
        @r              registro
```

Esempio: >

```
        :echo "Il valore di 'tabstop' è" &ts
        :echo "La tua home directory [Unix] è" $HOME
        :if @a > 5
```

La forma &nome si può usare per salvare il valore di una opzione, impostare l'opzione a un nuovo valore, fare qualcosa, e ripristinare il valore precedente. >

```
        :let salva = &ic
        :set noic
        :/L'Inizio/,$delete
        :let &ic = save_ic
```

Questo serve ad essere certi che l'espressione "L'Inizio" sia usata con l'opzione 'ignorecase' non impostata. Alla fine, prende il valore che l'utente aveva impostato.

OPERAZIONI MATEMATICHE

Diventa più interessante se combiniamo questi elementi fondamentali. Cominciamo con la operazioni matematiche.

```
        a + b           addizione
        a - b           sottrazione
        a * b           moltiplicazione
        a / b           divisione
        a % b           modulo (resto della divisione)
```

Valgono le solite regole di precedenza. Ad es.: >

```
        :echo 10 + 5 * 2
<       20 ~
```

Il raggruppamento viene fatto con le parentesi. Niente sorprese qui. Ad es.: >

```
        :echo (10 + 5) * 2
<       30 ~
```

Le stringhe si possono riunire con ".". Ad es.: >

```
        :echo "foo" . "bar"
<       foobar ~
```

Quando si danno più argomenti al comando ":echo", questi vengono separati tra loro con uno spazio. Nell'esempio l'argomento è una singola espressione, così non viene inserito alcuno spazio.

L'espressione condizionale viene presa dal linguaggio C:

```
        a ? b : c
```

Se "a" è vera, si usa "b", altrimenti si usa "c". Ad es.: >

```
        :let i = 4
        :echo i > 5 ? "i è grande" : "i è piccolo"
<       i è piccolo ~
```

Le tre parti dei costrutti vengono sempre valutate prima, per cui potreste vederle funzionare come:

```
        (a) ? (b) : (c)
```

==

41.4 Condizioni

Il comando ":if" esegue i comandi che lo seguono, fino all'":endif", solo se la condizione è verificata. La forma generica è:

```
        :if {condizione}
           {comandi}
        :endif
```

Solo quando l'espressione {condizione} viene considerata vera (valore diverso da zero) i {comandi} vengono eseguiti. Questi debbono essere comandi validi. Se contengono spazzatura, Vim non riesce a trovare l'":endif".
   Potete anche usare ":else". La forma generica è:

```
        :if {condizione}
           {comandi}
        :else
           {comandi}
        :endif
```

Il secondo {comandi} viene eseguito solo se il primo non lo è.
   Infine, abbiamo l'":elseif":

```
        :if {condizione}
           {comandi}
        :elseif {condizione}
           {comandi}
        :endif
```

Questo equivale ad usare ":else" e poi "if", ma senza la necessità di inserire un ulteriore ":endif".
   Un esempio utile per il vostro file vimrc è verificare l'opzione 'term' e fare qualcosa in funzione del suo valore: >

```
        :if &term == "xterm"
        :  " Fai qualcosa per xterm
        :elseif &term == "vt100"
        :  " Fai qualcosa per un terminale vt100
        :else
        :  " Fai qualcosa per gli altri tipi di terminale
        :endif
```

OPERAZIONI LOGICHE

Ne abbiamo già utilizzate alcune negli esempi precedenti. Quelle di più frequente utilizzo sono:

```
        a == b          uguale a
        a != b          differente da
        a >  b          maggiore di
        a >= b          maggiore o uguale a
```

```
        a <  b          minore di
        a <= b          minore o uguale a
```

Il risultato è 1 se la condizione è verificata, altrimenti è 0. Ad es.: **>**

```
        :if v:version >= 600
        :  echo "Congratulazioni"
        :else
        :  echo "State usando una vecchia, versione, passate alla nuova"
        :endif
```

Qui "v:version" è una variabile definita da Vim, che assume il valore della versione di Vim. 600 sta per la versione 6.0. La versione 6.1 ha il valore 6.01. Questo è molto utile per scrivere uno script che debba funzionare con molte versioni di Vim. |v:version|

Gli operatori logici lavorano sia con numeri che con stringhe. Confrontando due stringhe, viene usata la differenza matematica. Questa confronta il valore dei byte, ma può non andare bene per alcuni linguaggi.
   Confrontando una stringa con un numero, la stringa viene prima convertita a numero. Questo è un po' arbitrario, perché quando una stringa non appare come un numero, viene utilizzato il numero 0. Ad es.: **>**

```
        :if 0 == "uno"
        :  echo "sì"
        :endif
```

Ciò darà come risultato "sì", poiché "uno" non appare come un numero, e così viene convertito nel numero zero.

Per le stringhe ci sono due ulteriori elementi;

```
        a =~ b          corrisponde a
        a !~ b          non corrisponde a
```

L'elemento di sinistra "a" viene usato come una stringa. L'elemento di destra "b" viene usato come espressione, simile a quelle usate per la ricerca. Ad es.: **>**

```
        :if str =~ " "
        :  echo "str contiene uno spazio"
        :endif
        :if str !~ '\.$'
        :  echo "str non finisce con un punto"
        :endif
```

Notate l'uso della stringa inclusa fra apici semplici nell'espressione. Ciò è utile perché le barre retroverse devono essere scritte due volte all'interno di una stringa inclusa fra doppi apici, e le espressioni regolari tendono a contenere parecchie barre retroverse.

   L'opzione 'ignorecase' viene usata nel confronto tra stringhe. Se non volete ciò, aggiungete "#" per tener conto di maiuscole e minuscole, e "?" per non farlo. Così "==?" verifica se due stringhe siano uguali, trascurando maiuscole e minuscole. E "!~#" controlla se una espressione non corrisponde, controllando anche maiuscole e minuscole. Per la tabella completa degli operatori, si veda |expr-==|.

ALTRI CICLI

Il comando ":while" è già stato citato. Ci sono altri due comandi che si possono usare tra ":while" e ":endwhile":

```
        :continue               Ritorna all'inizio del ciclo "while"; il ciclo
                                inizia una nuova iterazione.
        :break                  Salta in avanti fino a ":endwhile"; il ciclo
                                termina l'esecuzione.
```

Ad es.: **>**

```
        :while contatore < 40
        :  call fa_qualche_controllo()
        :  if salta_un_giro
        :    continue
        :  endif
        :  if abbiamo_finito
        :    break
        :  endif
```

```
        :  sleep 50m
        :endwhile
```

Il comando ":sleep" fa fare a Vim un pisolino. "50m" significa "per cinquanta millisecondi". Un altro esempio è ":sleep 4", che fa dormire per quattro secondi.

==
41.5 Esecuzione di una espressione

Finora i comandi nello script venivano eseguiti direttamente da Vim. Il comando ":execute" permette di eseguire il risultato di una espressione. È questa una maniera molto potente per costruire comandi ed eseguirli.
   Un esempio è per saltare ad una tag, che sia contenuta in una variabile:
 >

```
        :execute "tag " . nome_tag
```

Il "." si usa per concatenare la stringa "tag " con il valore della variabile "nome_tag". Supponiamo che "tag_name" abbia come valore "get_cmd", allora il comando che verrà eseguito è: **>**

```
        :tag get_cmd
```

Il comando ":execute" può solo eseguire i comandi che iniziano con ":". Il comando ":normal" esegue i comandi che si possono dare in Normal mode. Comunque, l'argomento di ":normal" non è una espressione, ma i caratteri del comando letterale. Ad es.: **>**

```
        :normal gg=G
```

Qeusto comando salta alla prima linea e formatta tutte le linee usando l'operatore "=".
   Per far funzionare ":normal" con un'espressione, combinatelo con ":execute".
Ad es.: **>**

```
        :execute "normal " . comandi_normali
```

La variabile "comandi_normali" deve contenere comandi del Normal mode.
   Assicuratevi che l'argomento di ":normal" sia un comando completo. Altrimenti Vim raggiungerà la fine dell'argomento, ma non eseguirà il comando. Ad es. se voi iniziate in Insert mode, dovete anche uscire da Insert mode. Ad esempio: **>**

```
        :execute "normal Inuovo testo \<Esc>"
```

Ciò inserisce "nuovo testo " nella linea corrente. Si noti l'uso del tasto speciale "\<Esc>". Ciò evita di dover immettere un vero carattere <Esc> nel vostro script.

==
41.6 Utilizzo funzioni

Vim definisce parecchie funzioni e fornisce un grande numero di funzionalità in questo modo. Qualche esempio verrà dato in questa sezione. Potete trovare qui la lista completa: |functions|.

Una funzione si invoca col comando ":call". I parametri vengono passati alla funzione fra parentesi, separati da virgole. Ad es.: **>**

```
        :call search("Data: ", "W")
```

Questo invoca la funzione search(), con argomenti "Date: " e "W". La funzione search() usa il proprio primo argomento come una espressione da ricercare e il secondo come un indicatore. L'indicatore "W" significa che la ricerca non aggiri la fine del file.

Una funzione può essere invocata all'interno di una espressione. Ad es.: **>**

```
        :let line = getline(".")
        :let repl = substitute(line, '\a', "*", "g")
        :call setline(".", repl)
```

La funzione getline() ottiene una linea dal file corrente. Il suo argomento è una specificazione del numero di linea. In questo caso viene usato "." che significa la linea in cui è posizionato il cursore.
   La funzione substitute() fa qualcosa di simile al comando ":substitute". Il primo argomento è la stringa in cui effettuare la sostituzione. Il

secondo è l'espressione da cercare, il terzo argomento la stringa con cui rimpiazzarla. Gli ultimi argomenti, per finire, sono indicatori.
   La funzione setline() imposta la linea, specificata dal primo argomento, ad una nuova stringa, il secondo argomento. In questo esempio, la linea sotto il cursore viene rimpiazzata con il risultato della funzione substitute(). Così l'effetto dei tre comandi è uguale a: **>**

```
:substitute/\a/*/g
```

L'uso delle funzioni si fa più interessante quando fate più lavoro prima e dopo la invocazione di substitute().

FUNZIONI *function-list*

Ci sono parecchie funzioni. Le citiamo qui sotto, raggruppate secondo il loro impiego. Potete trovare una lista in ordine alfabetico qui: |functions|. Usate CTRL-] sul nome di funzione per saltare ad una documentazione più dettagliata su di essa.

Manipolazione di stringhe:

char2nr()	ottiene il valore ASCII di un carattere
nr2char()	ottiene un carattere dal suo valore ASCII
escape()	premette '\' a caratteri in una stringa
strtrans()	modifica una stringa rendendola stampabile
tolower()	passa una stringa a caratteri minuscoli
toupper()	passa una stringa a caratteri maiuscoli
match()	si posiziona ad una espressione in una stringa
matchend()	si posiziona a fine espressione in una stringa
matchstr()	trova una espressione in una stringa
stridx()	primo indice ad una stringa in una stringa
strridx()	ultimo indice ad una stringa in una stringa
strlen()	lunghezza di una stringa
substitute()	sostituisce una espressione in una stringa
submatch()	ottiene un dato valore in una ":substitute"
strpart()	ottiene una parte di una stringa
expand()	espande caratteri speciali
type()	restituisce il tipo di una variabile
iconv()	converte testo da una codifica a un'altra

Modificare il testo nel buffer corrente:

byte2line()	dice a che linea si trova un certo byte
line2byte()	dice posizione in byte di una data linea
col()	numero di colonna del cursore o marcatore
virtcol()	colonma sullo schermo del cursore o marcatore
line()	numero di linea del cursore o marcatore
wincol()	numero colonna del cursore nella finestra
winline()	numero linea del cursore nella finestra
cursor()	posiziona il cursore a linea/colonna
getline()	ottiene una linea dal buffer
setline()	sostituisce una linea nel buffer
append()	aggiungere {stringa} sotto linea {numero}
indent()	fa rientrare una data linea
cindent()	fa rientrare una data linea (stile C)
lispindent()	fa rientrare una data linea (stile Lisp)
nextnonblank()	trova la prossima linea non vuota
prevnonblank()	trova la linea non vuota precedente
search()	trova una espressione nel testo
searchpair()	trova l'altro capo, ad es. di un "if"

Funzioni di sistema e manipolazione di file:

browse()	visualizza un file
glob()	valorizza espressioni regolari (wildcard)
globpath()	valorizza espressione lista-di-directory
resolve()	trova nome file dato puntatore (shortcut)
fnamemodify()	modifica nome file
executable()	controlla esistenza programma eseguibile
filereadable()	controlla se possibile leggere un file
filewritable()	controlla se possibile scrivere un file
isdirectory()	controlla se una data directory esiste
getcwd()	ottiene directory di lavoro corrente
getfsize()	ottiene lunghezza di un file
getftime()	ottiene data ultima modifica di un file
localtime()	ottiene la data corrente
strftime()	converte una data in formato stampabile
tempname()	ottiene nome di un file temporaneo
delete()	cancella un file
rename()	rinomina un file

```
        system()                ottiene codice ritorno da comando shell
        hostname()              ottiene nome del computer

Buffer, finestre e lista argomenti:
        argc()                  numero argomenti nella lista argomenti
        argidx()                posizionamento corrente nella lista argomenti
        argv()                  ottiene un argomento dalla lista argomenti
        bufexists()             controlla se un buffer esiste
        buflisted()             controlla se un buffer esiste ed è listato
        bufloaded()             controlla se un buffer esiste ed è caricato
        bufname()               ottiene nome di un dato buffer
        bufnr()                 ottiene numero buffer di un dato buffer
        winnr()                 ottiene numero finestra di finestra corrente
        bufwinnr()              ottiene numero finestra di un dato buffer
        winbufnr()              ottiene numero buffer di una data finestra
        getbufvar()             ottiene valore di variabile da un dato buffer
        setbufvar()             imposta variabile in un dato buffer
        getwinvar()             ottiene valore variabile da una data finestra
        setwinvar()             imposta variabile in una data finestra

Piegature:
        foldclosed()    controlla se piegatura chiusa in una linea data
        foldclosedend()         come foldclosed() ma restituisce ultima linea
        foldlevel()             controlla livello piegatura in una linea data
        foldtext()              specifiche linea che indica piegatura chiusa

Evidenziazione sintattica:
        hlexists()              controlla se esiste gruppo evidenziazione
        hlID()                  ottiene ID di un gruppo evidenziazione
        synID()                 ottiene ID sintattico ad una data posizione
        synIDattr()             ottiene un dato attributo di un ID sintattico
        synIDtrans()            ottiene traduzione di un ID sintattico

Storia:
        histadd()               aggiunge un elemento a una storia
        histdel()               toglie un elemento a una storia
        histget()               ottiene un elemento da una storia
        histnr()                ottiene indice più alto di una storia

Interazione:
        confirm()               chiede all'utente di fare una scelta
        getchar()               ottiene un carattere dall'utente
        getcharmod()            ottiene modificatori ultimo carattere immesso
        input()                 ottiene una linea dall'utente
        inputsecret()           ottiene una linea dall'utente senza mostrarla
        inputdialog()           ottiene una linea dall'utente usando GUI
        inputresave             salva caratteri immessi e pulisce
        inputrestore()          ripristina caratteri immessi

Vim server:
        serverlist()            restituisce lista nomi server
        remote_send()           invia caratteri comando a server Vim
        remote_expr()           valuta una espressione in un server Vim
        server2client()         invia risposta a un cliente di un server Vim
        remote_peek()           controlla se esiste risposta da un server Vim
        remote_read()           legge risposta da Vim server
        foreground()            sposta finestra Vim in primo piano
        remote_foreground()     sposta finestra server Vim in primo piano

Varie:
        mode()                  ottiene modalità di edit corrente
        visualmode()            dice ultima modalità visuale utilizzata
        hasmapto()              controlla se esiste mappatura
        mapcheck()              controlla se esiste mappatura corrispondente
        maparg()                ottiene valore di una mappatura
        exists()                controlla se variabile, funzione, etc. esiste
        has()                   controlla se Vim supporta una estensione data
        cscope_connection()     controlla se esiste una connessione cscope
        did_filetype()          controlla se già usato autocomando tipo_file
        eventhandler()          controlla se invocato da un gestore eventi
        getwinposx()            ottiene posizione X della finestra GUI Vim
        getwinposy()            ottiene posizione Y della finestra GUI Vim
        winheight()             ottiene altezza di una data finestra
        winwidth()              ottiene larghezza di una data finestra
        libcall()               richiama una funzione in una libreria esterna
        libcallnr()             come sopra, la funzione restituisce un numero
        getreg()                ottiene contenuto di un registro
        getregtype()            ottiene tipo di un registro
```

```
        setreg()                imposta contenuto e tipo di un registro
```

==

41.7 Definizione funzioni

Vim vi consente di definire le proprie funzioni. La dichiarazione di una funzione comincia così: **>**

```
        :function {nome}({var1}, {var2}, ...)
        :  {corpo_della_funzione}
        :endfunction
<
```

        Note:
        I nomi di funzione devono cominciare con una lettera maiuscola.

Definiamo una breve funzione che restituisce il minore di due numeri. Comincia con questa linea:

```
        :function Min(num1, num2)
```

Ciò dice a Vim che la funzione si chiama "Min" ed ha due argomenti: "num1" e "num2".
La prima cosa di cui avete bisogno è di tentare di vedere quale numero sia il più piccolo:
>

```
        :  if a:num1 < a:num2
```

Il prefisso speciale "a:" dice a Vim che la variabile è un argomento della funzione. Assegniamo alla variabile "minore", il valore del numero più piccolo: **>**

```
        :  if a:num1 < a:num2
        :    let minore = a:num1
        :  else
        :    let minore = a:num2
        :  endif
```

La variabile "minore" è una variabile locale. Variabili usate all'interno di una funzione sono locali se non hanno prefissi come "g:", "a:", o "s:".

        Note:
        Per accedere ad una variabile globale dall'interno di una funzione dovete premettere "g:" alla variabile stessa. Così "g:contatore" all'interno di una funzione viene usato per la variabile globale "contatore", e "contatore" è un'altra variabile, locale per la funzione.

Usate il comando ":return" per restituire il numero più piccolo all'utente. Infine, indicate la fine della funzione: **>**

```
        :  return minore
        :endfunction
```

La definizione completa della funzione è la seguente: **>**

```
        :function Min(num1, num2)
        :  if a:num1 < a:num2
        :    let minore = a:num1
        :  else
        :    let minore = a:num2
        :  endif
        :  return minore
        :endfunction
```

Una funzione definita dall'utente viene invocata esattamente nello stesso modo con cui vengono invocate le funzioni predefinite di Vim. Solo il nome è diverso. La funzione Min si può usare così: **>**

```
        :echo Min(5, 8)
```

Solo adesso la funzione verrà eseguita, e le linee saranno interpretate da Vim. Se ci sono errori, come usare una variabile o una funzione indefinite, si otterrà un messaggio di errore. Quando si definisce una funzione, errori di questo tipo non vengono rilevati.

Quando una funzione trova ":endfunction" o se ":return" viene usato senza un argomento, la funzione restituisce zero.

Per ridefinire una funzione già esistente, usate il "!" nel comando ":function": **>**

```
:function!  Min(num1, num2, num3)
```

USARE UN INTERVALLO

Al comando ":call" si può dare un intervallo di linee su cui agire. Ciò può avere uno o due significati. Quando una funzione è stata definita con la parola chiave "range", terrà conto dell'intervallo di linee stesso.
Alla funzione verranno passate le variabili "a:firstline" and "a:lastline". Queste avranno i numeri di linea dall'intervallo in cui la funzione è stata chiamata. Ad es: **>**

```
:function Conta_parole() range
:  let n = a:firstline
:  let quante = 0
:  while n <= a:lastline
:    let quante = quante + Wordcount(getline(n))
:    let n = n + 1
:  endwhile
:  echo "trovate " . quante . " parole"
:endfunction
```

Potete chiamare questa funzione con: **>**

```
:10,30call Conta_parole()
```

Verrà eseguita una volta sola, e scriverà il numero di parole.
L'altro modo per usare un intervallo di linee consiste nel definire una funzione senza la parola chiave "range". La funzione sarà chiamata una volta per ogni linea nell'intervallo, con il cursore posizionato su quella linea.
Ad es.: **>**

```
:function  Numero()
:  echo "La linea numero " . line(".") . " contiene: " . getline(".")
:endfunction
```

Se chiamate questa funzione con: **>**

```
:10,15call Numero()
```

la funzione verrà invocata sei volte.

NUMERO VARIABILE DI ARGOMENTI

Vim vi consente di definire funzione che abbia un numero variabile di argomenti. Il comando seguente, ad es., definisce una funzione che deve avere almeno 1 argomento (inizio) e può avere più di 20 argomenti in aggiunta al primo: **>**

```
:function Mostra(inizio, ...)
```

La variabile "a:1" contiene il primo argomento facoltativo, "a:2" il secondo, e così via. La variabile "a:0" contiene il numero di argomenti aggiuntivi.
Ad es.: **>**

```
:function Mostra(inizio, ...)
:  echohl Title
:  echo "Mostra è " . a:inizio
:  echohl None
:  let indice = 1
:  while indice <= a:0
:    echo "  Arg " . indice . " è " . a:{index}
:    let indice = indice + 1
:  endwhile
:  echo ""
:endfunction
```

La funzione usa il comando ":echohl" per specificare l'evidenziazione da usare per il successivo comando ":echo" command. ":echohl None" torna alla visualizzazione normale. Il comando ":echon" si comporta come ":echo", ma non scrive il carattere per andare a capo.

LISTA DELLE FUNZIONI

Il comando ":function" lista i nomi e gli argomenti di tutte le funzioni definite dall'utente: >

```
        :function
<       function Show(start, ...) ~
        function GetVimIndent() ~
        function SetSyn(name) ~
```

Per vedere cosa fa una funzione, usate il suo nome come argomento per ":function": >

```
        :function SetSyn
<       1     if &syntax == '' ~
        2       let &syntax = a:name ~
        3     endif ~
           endfunction ~
```

DEBUGGING

Il numero linea è utile quando ricevete un messaggio di errore o in fase di debug. Si veda |debug-scripts| a proposito del debug.
Potete anche impostare l'opzione 'verbose' a 12 o più per vedere tutte le chiamate di funzione. Impostatelo a 15 o più per visualizzare ogni linea eseguita.

CANCELLARE UNA FUNZIONE

Per cancellare la funzione Mostra(): >

```
        :delfunction Mostra
```

Se la funzione non esiste, viene visualizzato un messaggio di errore.

==
41.8 Note varie

Cominciamo con un esempio: >

```
        :try
        :   read ~/templates/pascal.tmpl
        :catch /E484:/
        :   echo "Spiacente, il file_modello Pascal non è disponibile."
        :endtry
```

Il comando ":read" non può essere eseguito se il file non esiste. Invece di lasciar visualizzare un messaggio di errore, questo codice cattura l'errore e dà all'utente un messaggio più comprensibile.

Per i comandi compresi fra ":try" e ":endtry" gli errori sono trasformati in eccezioni. Una eccezione è una stringa di caratteri. Nel caso di un errore la stringa contiene il messaggio di errore. E ogni messaggio di errore ha un identificativo di messaggio. In questo caso, l'errore che intercettiamo contiene l'identificativo "E484:". Questo identificativo è una costante (il testo del messaggio può variare, ad es. può essere tradotto).

Quando il comando ":read" genera un altro errore, la stringa "E484:" non sarà contenuta nel messaggio di errore. Ragion per cui questa eccezione non sarà intercettata, e quindi riceveremo il relativo messaggio di errore.

Si potrebbe essere tentati di scrivere: >

```
        :try
        :   read ~/templates/pascal.tmpl
        :catch
        :   echo "Spiacente, il file modello Pascal non è disponibile."
        :endtry
```

Così si intercettano tutti gli errori. In questo modo però non si vedono errori "utili" tipo: "E21: Non posso fare modifiche, 'modifiable' è inibito"

Un altro meccanismo utile è il comando ":finally": >

```
        :let tmp = tempname()
        :try
        :   exe ".,$write " . tmp
```

```
        :   exe "!filter " . tmp
        :   .,$delete
        :   exe "$read " . tmp
        :finally
        :   call delete(tmp)
        :endtry
```

Questo script Vim fa passare le linee dal cursore fino alla fine del file attravrso il comando "filter", che ha come argomento un nome di file.
  Sia che il filtro funzioni, sia se qualche errore accade tra ":try" e ":finally" o l'utente cancella l'operazione di filtro premendo CTRL-C, la chiamata "call delete(tmp)" è eseguita sempre. Questo consente di essere sicuri di non lasciarsi dietro file temporanei di lavoro.

Maggiore informazione a riguardo della gestione eccezioni si trova nel manuale: |exception-handling|.

==
41.9 Osservazioni varie

Diamo qui una serie di osservazioni relative agli script Vim. Sono spiegate anche altrove, ma sono qui riunite per convenienza.

Il carattere di fine linea dipende dal sistema operativo. Per Unix si usa un solo carattere <NL>. Per MS-DOS, Windows, OS/2 etc. si usa, <CR><LF>.
  Questo è importante qunado si usano mappature che finiscono con <CR>.
  Si veda |:source_crnl|.

SPAZI BIANCHI

Linee bianche possono essere inserite dappertutto, e vengono ignorate.

Spazi bianchi prima di un testo (spazi e/o TAB) sono sempre ignorati. Gli spazi bianchi fra parametri (ad es. tra il 'set' e 'cpoptions' nell'esempio qui sotto) sono ridotti ad un unico spazio, che serve da separatore.
  Gli spazi bianchi dopo l'ultimo carattere (visibile) possono essere ignorati oppure no, a seconda della situazione. Si veda sotto.

Per un comando ":set" che usa il segno "=" (uguale), tipo: **>**

```
        :set cpoptions    =aABceFst
```

lo spazio bianco subito PRIMA del segno "=" è ignorato. Ma non è consentito alcuno spazio bianco DOPO il segno "="!

Per inserire uno spazio bianco nel valore di una opzione, bisogna farlo precedere da un "\" (barra retroversa) come nell'esempio seguente: **>**

```
        :set tags=il\ mio\ bel\ file
```

Lo stesso esempio scritto come **>**

```
        :set tags=Il mio bel file
```

produrrà un errore, perché viene interpretato come: **>**

```
        :set tags=il
        :set mio
        :set bel
        :set file
```

COMMENTI

Il carattere " (il doppio apice) inizia un commento. A partire da questo carattere il resto della linea è considerato un commento ed è ignorato, tranne che nei comandi che non considerano i commenti, come mostrato negli esempi qui sotto. Un commento può cominciare in qualsiasi posizione della linea.

Bisogna fare attenzione ai commenti per alcuni comandi. Ad es.: **>**

```
        :abbrev dev development         " abbreviazione
        :map <F3> o#include             " inserisci include
        :execute cmd                    " eseguilo
        :!ls *.c                        " lista i files C
```

La abbreviazione 'dev' sarè espansa come 'development   " abbreviazione'.

La mappatura di <F3> sarà costituita dall'intera linea dopo 'o# ....' compreso '" inserisci include'. Il comando "execute" darà un errore. Il comando "!" passerà tutto quel che lo segue allo shell, che darà un errore perché manca un carattere '"' che chiuda il primo '"'.
Non sono permessi commenti dopo i comandi ":map", ":abbreviate", ":execute" e "!" (ci sono alcuni altri comandi con la stessa limitazione).
Per i comandi ":map", ":abbreviate" and ":execute" si può però scrivere: **>**

```
:abbrev dev development|" abbreviazione
:map <F3> o#include|" inserisci include
:execute cmd                    |" eseguilo
```

Il carattere '|' separa un comando da quello successivo. Ed il comando successivo è solo un commento.

Si noti che non c'è alcuno spazio prima del '|' nella abbreviazione e nella mappatura. Per questi comandi, ogni carattere fino a fine linea o a '|' è compreso. Per questo motivo, non sempre è chiaro se sono presenti degli spazi dopo l'ultimo carattere visibile sulla linea; **>**

```
:map <F4> o#include
```

Per evitare questi problemi, potete attivare l'opzione 'list' mentre modificate degli script Vim.

TRABOCCHETTI

Problemi anche maggiori sono presenti nell'esempio seguente: **>**

```
:map ,ab o#include
:unmap ,ab
```

Il comando "unmap" non funzionerà, perché tenta di togliere la mappatura di ",ab " (c'è uno spazio bianco dopo il "b" − NdT). Questa mappatura non esiste. Sarà quindi visualizzato un messaggio di errore, difficile da comprendere, perché lo spazio bianco finale in ":unmap ,ab " non è visibile.

Lo stesso capita quando si usa un commento dopo un comando 'unmap': **>**

```
:unmap ,ab     " commento
```

Qui il commento verrà ignorato. Comunque, Vim tenterà di togliere la mappatura ',ab ', che non esiste. Riscrivetelo come: **>**

```
:unmap ,ab|    " commento
```

RITORNARE DOVE ERAVAMO

Talora volete fare una modifica, e ritornare al punto in cui si trovava il cursore prima della modifica. Anche un ripristino della posizione relativa dello schermo sarebbe gradita, con la stessa linea di prima visualizzata in cima alla finestra.
L'esempio che segue copia la linea sotto il cursore, la mette sopra la prima linea del file, e torna al punto di partenza: **>**

```
map ,p ma"aYHmbgg"aP`bzt`a
```

Spiegazione: **>**

```
ma"aYHmbgg"aP`bzt`a
<       ma                      marcatura "a" di posizione corrente
          "aY                   copia linea corrente nel registro "a"
             Hmb                marcatura "b" su linea in cima alla pagina
                gg              vai alla prima linea del file
                  "aP           mettici sopra la linea nel registro "a"
                      `b        vai alla linea in cima alla pagina
                        zt      posiziona testo nella finestra come prima
                          `a    torna alla posizione in cui era il cursore
```

PER LA DISTRIBUZIONE

Per evitare che i vostri nomi di funzione entrino in conflitto con funzioni che altri hanno preparato, usate questo schema:
− Premettete una vostra stringa ad ogni nume funzione. Io uso spesso una abbreviazione. Ad es., "OW_" è usato per le funzioni Opzioni Window.
− Mettete la definizione delle vostre funzioni in un unico file. Impostate una

variabile globale per indicare che la funzione è stata caricata. Quando volete caricare di nuovo il file, cancellate prima le funzioni. Ad es: **>**

```
" Questa è la funzione XXX

if exists("XXX_caricata")
  delfun XXX_uno
  delfun XXX_due
endif

function XXX_uno(a)
        ... corpo della funzione ...
endfun

function XXX_due(b)
        ... corpo della funzione ...
endfun

let XXX_caricata = 1
```

==========

41.10 Scrivere un plugin *write-plugin*

Potete scrivere uno script Vim in modo tale che chi vuole lo possa usare. Questo si chiama un "plugin". Gli utenti Vim possono copiare il vostro script Vim nella loro directory dei plugin e cominciare ad usare le sue funzioni. Si veda |add-plugin|.

Ci sono di fatto due tipi di plugin:

global plugin: Per ogni tipo di file.
filetype plugin: Solo per file di un dato tipo.

In questa sezione è spiegato solo il "global plugin". La maggior parte delle cose dette vale anche per i "filetype plugin". Le informazioni relative ai "filetype plugin" sono nella sezione seguente |write-filetype-plugin|.

NOME

innanzitutto dovete scegliere un nome per il vostro plugin. Lo scopo per cui è stato scritto il plugin dovrebbe essere riconoscibile dal nome che ha. E dovrebbe essere poco probabile che qualcun altro scriva un plugin con lo stesso nome, ma che faccia cose differenti. E per favore, limitate il nome a 8 caratteri, per evitare problemi sui vecchi sistemi Windows.

Uno script che corregge errori di battitura potrebbe essere chiamato "typecorr.vim". Lo useremo qui come esempio.

Se il plugin deve poter essere usato da tutti, deve seguire alcune linee guida. Le vediamo una alla volta. L'esempio completo del plugin è riportato alla fine.

CORPO

Cominciamo dal corpo del plugin, le linee che fanno il lavoro vero: **>**

```
 14     iabbrev li il
 15     iabbrev altor altro
 16     iabbrev voule vuole
 17     iabbrev sopratutto
 18             \ soprattutto
 19     let s:contatore = 4
```

La lista vera dovrebbe essere molto più lunga, naturalmente.

La numerazione delle linee è stata aggiunta per uso didattico, non dovete metterla nel vostro file plugin!

INTESTAZIONE

Probabilmente aggiungerete nuove correzioni al vostro plugin, e vi ritroverete con diverse versioni sparse qua e là. E quando distribuirete il vostro file plugin, la gente vorrà sapere chi ha scritto questo meraviglioso plugin e a chi possono essere inviate considerazioni al riguardo. Ragion per cui,

dovreste mettere una intestazione all'inizio del vostro plugin: >

```
  1     " Plugin generale di Vim per correggere errori di battitura
  2     " Ultima Modifica:      2000 Ott 15
  3     " Manutentore:  Bram Moolenaar <Bram@vim.org>
```

A proposito di copyright e licenze: Poiché i plugin sono molto utili, e non ha gran senso restringerne la distribuzione, siete invitati a considerare l'idea di dichiarare il vostro plugin di pubblico dominio, oppure di usare la licenza Vim |license|. Una breve nota al riguardo vicino all'inizio del plugin dovrebbe essere sufficiente. Ad es.: >

```
  4     " Licenza:      Questo file è di pubblico dominio.
```

CONTINUAZIONE LINEA, AL NETTO DA EFFETTI SECONDARI   *use-cpo-save*

Nella linea 18 qui sopra, il meccanismo usato per la continuazione è descritto in |line-continuation|. Utenti che hanno attivato l'opzione 'compatible' avranno del problemi, e otterranno un messaggio di errore. Non possiamo semplicemente cambiare l'opzione 'compatible', perché questo avrebbe molti altri effetti oltre a quello desiderato. Per evitare di toccare 'compatible' imposteremo invece l'opzione 'cpoptions' al suo valore predefinito in Vim e la riporteremo poi al valore che aveva. Questo ci consente di usare le linee di continuazione e nello stesso tempo rende lo script utilizzabile da un numero molto maggiore di utenti. Si fa in questo modo: >

```
 11     let s:salva_cpo = &cpo
 12     set cpo&vim
 ..
 42     let &cpo = s:salva_cpo
```

Prima si memorizza il valore precedente di 'cpoptions' nella variabile s:salva_cpo. Alla fine del plugin questo valore viene ripristinato.

Notate l'uso di una variabile locale allo script |s:var|. Una variabile globale con lo stesso nome potrebbe essere già in uso da qualche altra parte. Usate sempre delle variabili locali allo script per azioni limitate solo all'interno dello script.

NON CARICARE

È possibile che un utente non voglia sempre caricare questo plugin. Oppure l'amministratore del sistema l'ha messo nella directory dei plugin di uso comune a tutti gli utenti, ma un utente vuole usare il suo proprio plugin.
  Quindi all'utente deve essere offerta la possibilità di evitare l'uso di questo specifico plugin. Questo si ottiene così: >

```
  6     if exists("loaded_typecorr")
  7       finish
  8     endif
  9     let loaded_typecorr = 1
```

Si evita così anche che, nel caso lo script sia caricato una seconda volta, si riceva un messaggio di errore perché la funzione erà già definita, e possano presentarsi dei problemi per degli autocomandi aggiunti più di una volta.

MAPPATURE

Per rendere il plugin più interessante, introduciamo una mappatura che aggiunga una correzione per la parola che si trova sotto il cursore. Potremmo scegliere una combinazione di tasti per questa mappatura, ma l'utente magari la usa già per fare qualcosa d'altro. Per permettere all'utente di definire la sequenza di tasti che vuole usare per una mappatura all'interno di un plugin, si può usare l'elemento <Leader>: >

```
 22       map <unique> <Leader>a  <Plug>TypecorrAdd
```

Il comando "<Plug>TypecorrAdd" rende questo possibile, come vedremo dopo.

L'utente può dare alla variabile "mapleader" la combinazione di tasti con cui vuole che cominci questa mappatura. Se l'utente ha impostato: >

```
        let mapleader = "_"
```

la mappatura definirà "_a". Se l'utente non ha fatto nulla, il valore predefinito (che è "\", barra retroversa) sarà utilizzato. In quel caso la mappatura definita sarà "\a".

Note Poiché <unique> è usato, sarà inviato un messaggio di errore nel caso la stessa mappatura sia già stata definita. |:map-<unique>|

E se l'utente volesse definire una sua sequenza di tasti? Glielo si può permettere con questo meccanismo: >

```
21      if !hasmapto('<Plug>TypecorrAdd')
22        map <unique> <Leader>a  <Plug>TypecorrAdd
23      endif
```

Qui controlliamo se una mappatura per "<Plug>TypecorrAdd" esiste già, e definiamo la mappatura con "<Leader>a" solo se non esiste. L'utente ha quindi la possibilità di mettere nel suo file vimrc: >

```
        map ,c  <Plug>TypecorrAdd
```

E quindi la sequenza di tasti mappata sarà ",c" invece che "_a" o "\a".

PARTI

Se uno script si allunga, potrebbe essere desiderabile dividere il lavoro in parti. Potete usare sia delle funzioni che delle mappature per questo. Ma non volete che queste funzioni e mappature interferiscano con altre, contenute in altri script. Per esempio, potreste definire una funzione Add(), ma un altro script potrebbe tentare di definire la stessa funzione. Per evitare questo, definiamo la funzione come locale, esistente solo all'interno del nostro script, premettendo "s:" al suo nome.

Definiano una funzione che aggiunge una nuova correzione di errore di battitura: >

```
30      function s:Add(errato, corretto)
31        let buono = input("Battere correzione per " . a:errato . ": ")
32        exe ":iabbrev " . a:errato . " " . buono
..
36      endfunction
```

Ora possiamo richiamare la funzione s:Add() dall'interno dello script. Se un altro script definisce ancora s:Add(), esisterà solo dentro quello script, e può essere chiamata solo daqlla script in cui è stata definita. Ci può anche essere una funzione globale. Ci può anche essere una funzione globale Add() (senza "s:"), che è distinta dalle altre due.

<SID> si può usare senza mappature. Genera un ID dello script, che identifica lo script in cui si trova. Nel nostro plugin di correzione errori di battitura lo usiamo in questo modo: >

```
24      noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
..
28      noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
```

Ovvero, quando un utente batte "\a", viene richiamata questa sequenza: >

```
        \a  ->  <Plug>TypecorrAdd  ->  <SID>Add  ->  :call <SID>Add()
```

Se un altro script mappasse <SID>Add, un altro script ID sarebbe generato, e quindi verrebbe definito un'altra mappatura.

Note Usiamo qui <SID>Add() invece che s:Add(). Questo perché la mappatura viene immessa dall'utente, ossia dall'esterno dello script. Il <SID> si traduce nell'ID dello script, in modo che Vim sappia in che script cercare la funzione Add().

La cosa è un po' complicata, ma serve per far sì che il plugin funzioni assieme ad altri plugin. La regola fondamentale è che voi usiate <SID>Add() nelle mappature e s:Add() da altre parti (nello script stesso, in autocomandi o in comandi utente).

Possiamo anche rendere disponibile la mappatura in un menu: >

```
26      noremenu <script> Plugin.Aggiungi\ Correzione      <SID>Add
```

Il menu "Plugin" è raccomandato per aggiungere elementi di menu a plugin. In

questo caso c'è solo un elemento. Se si aggiungono più elementi, la creazione di un sottomenu è auspicabile. Ad esempio, "Plugin.CVS" potrebbe essere usato per un plugin che offre le operazioni CVS "Plugin.CVS.checkin", "Plugin.CVS.checkout", etc. [CVS è un software per gestire sorgenti, "checkin" inserisce un sorgente, "checkout" lo preleva per aggiornarlo – NdT]

Note Nella linea 28 ":noremap" è usato per evitare che qualsiasi altra mappatura provochi problemi. Qualcuno potrebbe aver ridefinito ":call", ad esempio. Nella linea 24 usiamo anche ":noremap", ma vogliamo che "<SID>Add" sia rimappato. Per questo "<script>" è usato qui. Questo permette mappature che valgono solo all'interno dello script. |:map-<script>| La stessa cosa è fatta nella linea 26 per ":noremenu". |:menu-<script>|

<SID> E <Plug> *using-<Plug>*

Sia <SID> che <Plug> sono usati per evitare che mappature di combinazioni di tasti entrino in conflitto con altri che devono essere usati solo da altre mappature. Note Questa è la differenza fra usare <SID> e <Plug>:

<Plug> è visibile all'esterno dello script. È usato per mappature per cui l'utente voglia utilizzare una particolare sequenza di tasti. <Plug> è un codice speciale che un tasto della tastiera non è in grado di produrre.
Per rendere molto difficile che altri plugin usino la stessa sequenza di caratteri, usate questa struttura:
<Plug> nome_script nome_mappatura
Nel nostro esempio il nome_script è "Typecorr" e il nome_mappatura è "Add".
Il risultato è "<Plug>TypecorrAdd". Solo il primo carattere del nome_script e del nome_mappatura deve essere maiuscolo, in modo da poter capire dove comincia il nome_mappatura.

<SID> è l'ID dello script, un identificatore unico a quello script. Internamente Vim traduce <SID> in "<SNR>123_", dove "123" può essere un numero qualsiasi. Per cui una funzione "<SID>Add()" avrà come nome "<SNR>11_Add()" in uno script, e "<SNR>22_Add()" in un altro. Potete vederlo se usate il comando ":function" per ottenere una lista di funzioni. La traduzione di <SID> nelle mappature è esattamente la stessa, ed è questo il modo con cui potete invocare una funzione locale ad uno script da una mappatura.

COMANDI UTENTE

Aggiungiamo ora un comando utente per aggiungere una correzione: >

```
 38     if !exists(":Correzione")
 39       command -nargs=1  Correzione  :call s:Add(<q-args>, 0)
 40     endif
```

Il comando utente viene definito solo se non esiste già un comando con lo stesso nome. Altrimenti otterremmo un messaggio di errore. Sostituire il comando utente già esistente col nostro, scrivendo ":command!" non è una buona idea, in quanto l'utente si domanderebbe perché il comando che LUI ha definito non funziona. |:command|

VARIABILI NEGLI SCRIPT

Quando una variabile ha un nome che comincia con "s:" è una variabile dello script. Si può solo usare all'interno di uno script, Non esiste all'esterno dello script. Questo consente di evitare problemi quando si utilizza lo stesso nome di variabile in script differenti. La variabile sarà disponibile per tutto il tempo in cui si usa Vim. E la stessa variabile verrà riutilizzata se si ricarica lo stesso script una seconda volta. |s:var|

Il bello è che queste variabili possono anche essere usate in funzioni, autocomandi e comandi utente che sono definiti nello script. Nel nostro esempio di script possiamo aggiungere alcune linee per contare il numero di correzioni: >

```
 19     let s:contatore = 4
 ..
 30     function s:Add(from, correct)
 ..
 34       let s:contatore = s:contatore + 1
 35       echo s:contatore . " correzioni finora"
```

```
 36     endfunction
```

Dapprima s:contatore è inizializzato a 4 nello script stesso. Quando, più tardi, la funzione s:Add() viene invocata, incrementa s:count. Non importa da DOVE la funzione è stata chiamata, poiché, essendo stata definita nello script, userà solo variabili interne a questo script.

IL RISULTATO

Questo è lo script completo risultante: **>**

```
 1     " Plugin generale di Vim per correggere errori di battitura
 2     " Ultima Modifica:      2000 Ott 15
 3     " Manutentore:  Bram Moolenaar <Bram@vim.org>
 4     " Licenza:      Questo file è di pubblico dominio.
 5
 6     if exists("loaded_typecorr")
 7       finish
 8     endif
 9     let loaded_typecorr = 1
10
11     let s:salva_cpo = &cpo
12     set cpo&vim
13
14     iabbrev li il
15     iabbrev altor altro
16     iabbrev voule vuole
17     iabbrev sopratutto
18             \ soprattutto
19     let s:contatore = 4
20
21     if !hasmapto('<Plug>TypecorrAdd')
22       map <unique> <Leader>a  <Plug>TypecorrAdd
23     endif
24     noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
25
26     noremenu <script> Plugin.Aggiungi\ Correzione      <SID>Add
27
28     noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
29
30     function s:Add(errato, corretto)
31       let buono = input("Battere correzione per " . a:errato . ": ")
32       exe ":iabbrev " . a:errato . " " . buono
33       if a:corretto | exe "normal viws\<C-R>\" \b\e" | endif
34       let s:contatore = s:contatore + 1
35       echo s:contatore . " correzioni finora"
36     endfunction
37
38     if !exists(":Correzione")
39       command -nargs=1  Correzione  :call s:Add(<q-args>, 0)
40     endif
41
42     let &cpo = s:salva_cpo
```

La linea 33 non è stata ancora spiegata. Applica la nuova correzione alla parola sotto il cursore. Il comando |:normal| è usato per fare uso della nuova abbreviazione. Note mappature e abbreviazioni sono espanse qui, anche se la funzione è stata invocata da una mappatura definita come ":noremap".

L'uso di "unix" come opzione 'fileformat' è raccomandato. Gli script Vim in questo modo funzioneranno anche in tutti gli altri ambienti software. Script che abbiano 'fileformat' impostato a "dos" non funzionano sotto Unix.
  Si veda anche |:source_crnl|. Per essere sicuri di usarlo, scrivete, prima di scrivere il file che contiene lo script: **>**

```
        :set fileformat=unix
```

DOCUMENTAZIONE *write-local-help*

É una buona idea preparare anche un po' di documentazione per il vostro plugin. Questo è ancora più necessario quando il suo comportamente può essere personalizzato dall'utente. Si veda |add-local-help| per come aggiungere la vostra documentazione a quella di Vim.

Qui vedete un semplice esempio di file di aiuto, di nome "typecorr.txt": **>**

```
 1     *typecorr.txt*  Plugin per correzione di errori di battitura.
 2
 3     Se vi capita di fare errori di battitura, questo plugin li correggerà
 4     automaticamente.
 5
 6     Al momento le correzioni sono poche.  Potete aggiungerne, se volete.
 7
 8     Mappatura:
 9     <Leader>a   o   <Plug>TypecorrAdd
10             Aggiunge una correzione per la parola sotto il cursore.
11
12     Comandi:
13     :Correzione {parola}
14             Aggiunge una correzione per {parola}.
15
16                                                     *typecorr-settings*
17     Questo plugin non ha alcuna impostazione particolare.
```

La prima linea è l'unica per la quale il formato è importante. Sarà questa ad essere estratta dal file di aiuto per essere inserita nella sezione "LOCAL ADDITIONS" del file help.txt |local-additions|. Il primo "*" deve essere nella prima colonna della prima linea di testo. Dopo aver aggiunto il vostro file di aiuto date il comando ":help" per controllare che le linee in questione siano correttamente allineate.

Potete aggiungere altre tag (racchiuse fra **) nel vostro file di aiuto. Ma state attenti ad evitare nomi di tag di help già esistenti. Potreste inserire il nome del vostro plugin nei loro nome, come "typecorr-settings" nel nostro esempio.

L'uso di puntatori ad altre parti del file di help (racchiusi fra ||) è auspicabile. Questo facilita l'utente nel reperimento dell'aiuto a loro associato.

DETERMINAZIONE DEL TIPO FILE *plugin-filetype*

Se il tipo del vostro file non è già determinato da Vim, dovreste creare un piccolo script che lo determini in un file a parte. Di solito lo script consiste in un autocomando che imposta il tipo file quando il nome del file corrisponde a un dato modello.
Ad es.: **>**

```
        au BufNewFile,BufRead *.foo                     set filetype=foofoo
```

Scrivete il file contenente quest'unica linea come "ftdetect/foofoo.vim" nella prima directory che compare in 'runtimepath'. In Unix questa sarebbe "~/.vim/ftdetect/foofoo.vim". La convenzione è di usare il nome del tipo file come nome dello script.

I controlli possono essere molto più sofisticati, se volete, come ad es. una ispezione del contenuto del file per riconoscere il linguaggio.
SI veda anche |new-filetype|.

SOMMARIO *plugin-special*

Lista delle particolarità che contraddistinguono un plugin:

s:name	variabili interne allo script.
<SID>	Script-ID, usato per mappature e funczioni interne allo script.
hasmapto()	Funzione per accertare se l'utente ha già definito mappature per funzionalità contenute nello script.
<Leader>	Valore di "mapleader", che l'utente definisce come sequenza di tasti con cui iniziano le mappature del plugin.
:map <unique>	Emette un messaggio se una mappatura esiste già.
:noremap <script>	Usa solo mappature interne allo script, non mappature valide globailmente.
exists(":Cmd")	Controlla se un comando utente esiste già.

==
ftplugin
41.11 Scrivere un plugin per un tipo_file *write-filetype-plugin*

A plugin per un tipo_file è simile a un plugin globale, ma imposta opzioni e definisce mappature solo per il buffer corrente. Si veda |add-filetype-plugin| per informazioni sull'uso di questo tipo di plugin.

Leggete prima la sezione precedente sui plugin globali |41.10|. Tutto quel che si dice lì vale anche per i plugin per un tipo_file. Ci sono alcune regole in più, che sono spiegate qui di seguito. La caratteristica fondamentale è che un plugin per un tipo_file dovrebbe avere effetto solo sul buffer corrente.

DISABILITAZIONE

Se state scrivendo un plugin per un tipo_file che potrà essere usato da molta gente, dovete concedere loro la possibilità di evitare di usarlo. Mettete questo in testa al plugin: **>**

```
" Usa solo se non esiste già un latro plugin attivo
if exists("b:did_ftplugin")
  finish
endif
let b:did_ftplugin = 1
```

Occorre usare questa tecnica per evitare che lo stesso plugin sia eseguito due volte per lo stesso buffer (succede quando si usa un comando ":edit" senza fornirgli alcun argomento).

A questo punto gli utenti possono diabilitare completamente il plugin predefinito creando un plugin per un tipo_file composto solo da questa linea:

```
let b:did_ftplugin = 1
```

Per ottenere questo, la directory che contiene il vostro plugin per un tipo_file deve venire prima della directory $VIMRUNTIME nella list di directory contenuta nella opzione 'runtimepath'!

Se volete usare il plugin predefinito, ma cambiare una delle impostazioni, potete inserire l'impostazione da cambiare in uno script: **>**

```
setlocal textwidth=70
```

Poi lo scrivete nella directory "after" ["dopo"], così che venga eseguito DOPO il plugin di tipo_file predefinito (in questo caso "vim.vim").
Si veda |after-directory|. In Unix questa directory è: "~/.vim/after/ftplugin/vim.vim". Note Il plugin predefinito avrà impostato "b:did_ftplugin", ma la cosa viene qui ignorata.

OPZIONI

Per far sì che il plugin per un tipo_file agisca solo sul buffer corrente usate il comando:

```
:setlocal
```

per impostare le opzioni. Ed impostate solo le opzioni che agiscono solo sul buffer corrente (si veda l'aiuto relativo a ogni singola opzione per controllare quali lo facciano). Quando si usa |:setlocal| per opzioni globali o per opzioni che riguardano una particolare finestra, il valore potrebbe cambiare in numerosi buffer, e non è questo l'obiettivo che si dovrebbe avere in un plugin per un tipo_file.

Quando una opzione ha un valore che è una lista di indicatori o elementi, potete utilizzare la sintassi "+=" e "-=" per non modificare i valori già impostati. Tenete presente che l'utente possa aver già cambiato il valore di una opzione. Ripristinare il valore predefinito e poi cambiarlo come serve è spesso una buona idea. Ad es.: **>**

```
:setlocal formatoptions& formatoptions+=ro
```

MAPPATURE

Per far sì che le mappature siano applicate solo nel buffer corrente usate il

comando: >

```
:map <buffer>
```

Questo va combinato con la mappatura in due passi spiegata più sopra.
  Ecco un esempio di come si definisce una funzionalità in un plugin per un tipo_file: >

```
if !hasmapto('<Plug>JavaImport')
  map <buffer> <unique> <LocalLeader>i <Plug>JavaImport
endif
noremap <buffer> <unique> <Plug>JavaImport oimport ""<Left><Esc>
```

|hasmapto()| è usato per controllare se l'utente ha già definito una mappatura per <Plug>JavaImport. In caso negativo, il plugin per un tipo_file definisce la mappatura predefinita. Questa inizia con |<LocalLeader>|, che permette all'utente di scegliere il tasto (o i tasti) con cui vuole che inizino le mappature dei plugin per un tipo_file. Il valore predefinito è la barra rovesciata ("\").
  "<unique>" è usato per mandare un messaggio di errore se la mappatura esiste già o entra in conflitto con una mappatura esistente.
  |:noremap| è usato per evitare che ogni altra mappatura che l'utente abbia definito possa interferire. Potreste voler usare ":noremap <script>" per far sì che si possano ri-mappare mappature definite in questo script, che comincino con <SID>.

L'utente deve avere la possibilità di disabilitare una mappatura in un plugin per un tipo_file, senza disabilitarlo completamente. Ecco un esempio di come si può fare questo per il tipo_file "mail" (messaggi di posta elettronica): >

```
" Aggiungi mappature, a meno che l'utente preferisca evitarlo.
if !exists("no_plugin_maps") && !exists("no_mail_maps")
  " Citate un testo inserendo "> "
  if !hasmapto('<Plug>MailQuote')
    vmap <buffer> <LocalLeader>q <Plug>MailQuote
    nmap <buffer> <LocalLeader>q <Plug>MailQuote
  endif
  vnoremap <buffer> <Plug>MailQuote :s/^/> /<CR>
  nnoremap <buffer> <Plug>MailQuote :.,$s/^/> /<CR>
endif
```

Due variabili globali sono usate:
no_plugin_maps   disabilita mappature per ogni plugin di un tipo_file
no_mail_maps   disabilita mappature per un dato tipo_file

COMANDI UTENTE

Per aggiungere un comando utente per un dato tipo_file, in modo che possa solo essere usato in un buffer, usate l'argomento "-buffer" di |:command|.
  Ad es.: >

```
:command -buffer  Make  make %:r.s
```

VARIABILI

Un plugin per un tipo_file sarà utilizzato in ogni buffer contenente un file di quel tipo. La variabile interna di script |s:var| sarà la stessa per tutti i buffer in cui il plugin viene utilizzato. Usate variabili interne al buffer |b:var| se volete che una variabile sia usata sono all'interno di un unico buffer.

FUNZIONI

Quando si definisce una funzione, questa va definita soltanto una volta. Ma il plugin di tipo_file verrà invocato ogni volta che verrà aperto un file con quel dato tipo file. La tecnica seguente fa sì che la funzione venga definita una volta sola. >

```
:if !exists("*s:Func")
:  function s:Func(arg)
:    ...
:  endfunction
:endif
```
<

ANNULLARE *undo_ftplugin*

Quando l'utente imposta ":setfiletype xyz" l'effetto del tipo_file "precedente" dovrebbe essere annullato. Impostate la variabile b:undo_ftplugin con i comandi che annulleranno le impostazioni di un vostro plugin per un tipo_file. Ad es.: >

```
let b:undo_ftplugin = "setlocal fo< com< tw< commentstring<"
        \ . "| unlet b:match_ignorecase b:match_words b:match_skip"
```

L'uso di ":setlocal" con "<" subito dopo il nome dell'opzione reimposta per quell'opzione il suo valore globale. Nella maggior parte dei casi è questa la maniera migliore di ripristinare i valori "precedenti" delle opzioni.

Occorre togliere la indicazione "C" da 'cpoptions' per consentire la continuazione della linea, come detto prima |use-cpo-save|.

NOME FILE

Il tipo_file deve far parte del nome del plugin |ftplugin-name|. Usate una di queste tre forme:

```
.../ftplugin/roba.vim
.../ftplugin/roba_mia.vim
.../ftplugin/roba/tua.vim
```

"roba" è il tipo_file, "mia" e "tua" sono nomi a piacere.

SOMMARIO *ftplugin-special*

Lista delle particolarità che contraddistinguono un plugin di tipo_file:

<LocalLeader>	Valore di "maplocalleader", che l'utente definisce come la combinazione di tasti con cui iniziano le mappatur per un plugin di tipo_file.
:map <buffer>	Definisce una mappatura che vale solo all'interno del buffer.
:noremap <script>	Ri-mappa solo mappature definite in questo script e che cominciano con <SID>.
:setlocal	Imposta una opzione solo per il buffer corrente.
:command -buffer	Definisce un comando utente valido solo in questo buffer.
exists("*s:Func")	Controlla se una funzione è stata già definita.

Si veda anche |plugin-special|, che descrive le proprietà specifiche dei plugin.

==

41.12 Scrivere un plugin per un compilatore *write-compiler-plugin*

Un plugin per un compilatore imposta le opzioni da usare con un dato compilatore. L'utente può invocarlo con il comando |:compiler|. L'uso principale che se ne fa è per impostare le opzioni 'errorformat' e 'makeprg'.

Un esempio è la maniera più semplice di introdurlo. Questo comando modificherà tutti i plugin predefiniti per i compilatori: >

```
:next $VIMRUNTIME/compiler/*.vim
```

Usate |:next| per passare al file di plugin successivo.

Ci sono due caratteristiche notevoli di questi file. La prima è un meccanismo che permette a un utente di fare modifiche o aggiunte al file di plugin predefinito. I file predefiniti cominciano con: >

```
:if exists("current_compiler")
:  finish
:endif
:let current_compiler = "mine"
```

Quando scrivete un file per un compilatore e lo mettere nella vostra propria
directory di files di esecuzione ("runtime directory") (ad es.,
~/.vim/compiler in Unix), voi potete impostare la variabile "current_compiler"
in modo che il file predefinito non esegua (in seguito) le impostazioni
che contiene.

Il secondo metodo consiste nell'usare ":set" per ":compiler!" e ":setlocal"
per ":compiler". Vim fornisce il comando utente ":CompilerSet" a questo
scopo. D'altra parte, precedenti versioni di Vim non ne sono dotate, e per
questo motivo un vostro plugin dovrebbe definirla al suo interno. Ecco
un esempio: **>**

```
  if exists(":CompilerSet") != 2
    command -nargs=* CompilerSet setlocal <args>
  endif
  CompilerSet errorformat&              " usa 'errorformat' predefinito
  CompilerSet makeprg=nmake
```

<

When you write a compiler plugin for the Vim distribution or for a system-wide

Quando scrivete un plugin per un compilatore da distribuire con Vim o da
mettere in una directory di esecuzione ("untime directory") a livello di
sistema, usate il meccanismo delineato più sopra. Se "current_compiler" era
già stato impostato da un plugin dell'utente, questo verrà ritenuto valido
e non sostituito in alcun modo.

Qaundo scrivete un plugin di compilatore per modificare impostazioni contenute
nel plugin predefinito, non controllate la variabile "current_compiler".
Un plugin di questo tipo dovrebbe venire eseguito per ultimo, e quindi
dovrebbe trovarsi in una directory in fondo alla lista di directory
specificata in 'runtimepath'. In Unix il suo nome potrebbe essere
~/.vim/after/compiler.

==

Capitolo seguente: |usr_42.txt| Aggiungere nuovi menù

Segnalare refusi a Bartolomeo Ravera - E-mail: barrav at libero.it