Linux Base - Capitolo n. 12

Edizioni ByteMan (20-01-2006)

revisione: 25/01/2008

La Shell (parte 1)

Se il **kernel** è il nucleo del sistema operativo (interprete e gestore dell'hardware), la **shell** ne è il guscio, l'interfaccia (testuale) tramite la quale l'utente può interagire con le funzionalità offerte dal kernel.
La shell è un programma che gestisce la comunicazione fra utente e sistema operativo interpretanto ed eseguendo i comandi dell'utente (viene anche chiamata *command interpreter*), e può avere diversi modi di utilizzo:

- **Uso interattivo**: il sistema attende, in una sorta di loop continuo, i comandi digitati dall'utente, che possono anche ridirezionare input ed output;
- **Configurazione**: adattamento delle modalità operative della propria sessione, con definizione/ridefinizione di variabili e parametri che vengono utilizzati in ogni interazione dell'utente con la macchina;
- **Programmazione**: utilizzando comandi di sistema e funzionalità della shell è possibile realizzare piccoli programmi (script shell) in grado di automatizzare operazioni e reagire ad eventi.

Sono state sviluppate nel tempo diverse shell, ma sulla gran parte delle distribuzioni Linux è preimpostata di default la shell **bash**. La sintassi dei comandi presentati in questo seminario è basata appunto sulla shell bash.

BASH e' l'acronimo di **B**ourne **A**gain **SH**ell, si tratta di una shell compatibile con la shell **Bourne**, che e' stata una delle piu' usate in ambiente Unix alla quale aggiunge alcune migliorie mutuate anche da altre shell Unix.
E' in grado quindi di eseguire script pensati per la shell Bourne, mettendo a disposizione al contempo costrutti e comandi piu' complessi non presenti nella Bourne shell originale.

Configurare la Shell

La shell stessa può essere configurata ed adattata alle proprie esigenze grazie a cinque files di configurazione, non sempre per la verità usati tutti, d'altra parte come vedremo non e' difficile crearsene di propri. Questi file sono:

`/etc/profile`	`/etc/bashrc`	`~/.bash_profile`	`~/.bashrc`	`~/.bash_logout`

I primi due sono dei **file globali** cioe' quelli che contengono direttive di configurazione valide per tutti gli utenti, sono quelli che si trovano infatti sotto la directory /etc.
Gli altri tre sono dei **file locali** cioè contengono direttive di configurazione valide solo per l'utente che possiede la cartella nella directory /home che li contiene. Infatti nell'elenco sono preceduti da un carattere tilde (~ che nei sistemi Unix e' un collegamento breve alla directory base dell'utente), e preceduti da un punto che li rende "invisibili" al comando ls senza l'argomento -a.

- **/etc/profile**
  E' il file di configurazione globale che determina le variabili di ambiente ed i programmi da eseguire per ogni utente che manda in esecuzione la shell. Per fare un paragone con il mondo Dos potrebbe essere qualcosa di molto simile al file autoexec.bat.
- **/etc/bashrc**
  E' un file di configurazione globale molto simile al precedente, per questo motivo spesso non e' usato, contiene alias (collegamenti brevi a comandi molto lunghi, li vedremo in seguito), e pezzi di codice nel linguaggio di scripting della shell che devono essere eseguiti alla partenza della shell. Tutto puo' essere spostato senza problema nel file /etc/profile.

- **~/.bash_profile**
  E' il corrispondente locale di */etc/profile* e risiede nella directory di ciascun utente. Il file viene letto ed eseguito successivamente a quello globale e modifica o sovrascrive variabili che riguardano esclusivamente quell'utente.

- **~/.bashrc**
  E' il corrispondente locale di */etc/bashrc* e risiede nella directory di ciascun utente. Il file viene letto ed eseguito successivamente a quello globale e modifica o sovrascrive variabili che riguardano esclusivamente quell'utente.

- **~/bash_logout**
  E' un file di configurazione locale che contiene comandi da eseguire quando l'utente esce dalla shell. I comandi influenzano solo quell'utente.

Le variabili

Il metodo piu' usato per archiviare temporaneamente uno o piu' valori consiste nell'usare le variabili, cioe' delle aree di memoria a cui viene assegnato un nome ed in cui vengono depositati i valori. Vedremo meglio in seguito l'uso delle variabili per scrivere codice eseguibile dalla bash, per il momento ci interessano solo due accezioni del termine cioe' le variabili di ambiente (environmental) e le variabili locali (local).

Le **variabili di ambiente** sono quelle variabili create autonomamente dal sistema operativo e di norma sono definite nel file */etc/profile*, come per esempio SHELL, PS1, PS2, PATH, HOME, MAIL.

Le **variabili locali** sono quelle variabili definite dall'utente e sono generalmente definite nel file *~/.bashrc* che si trova nella /home dell'utente del quale possono influenzare l'ambiente.

La definizione di una variabile avviene in un modo intuitivo ed elementare. Consta di tre parti: il nome della variabile seguito dall'operatore di assegnazione "=" e dal valore della stessa, cioe':

```
nome_var=valore_var
```

ATTENZIONE al fatto che NON ci devono essere spazi vuoti tra i componenti dell'assegnazione.

Una volta definita la variabile, per permettere ad altri programmi (avviati dalla stessa shell) di potervi accedere, essa deve essere esportata con il comando:

```
export nome_var
```

Per accedere, infine, al valore di una variabile per usarne il valore corrente occorre premettere al nome il carattere "**$**". Per esempio per conoscere il valore della variabile **SHELL** possiamo digitare il comando:

```
echo $SHELL
```

che mostrerà sul video:

```
/bin/bash
```

Si comprende come ci stiamo avviando verso un vero e proprio linguaggio di programmazione di cui stiamo cominciando a conoscere gli elementi base. Verranno gradualmente prodotti dei file di testo chiamati **script** contenenti il necessario per eseguire tutta una serie di operazioni programmate. Se mentre si redige uno di questi file di script ci si trova con una linea di codice dagli effetti dubbi invece di cancellarla la si può commentare facendola precedere dal carattere "#". Chiaramente si puo' usare il carattere "#" anche per inserire commenti che migliorino la leggibilita' dello script. Ecco alcune righe contenenti commenti:

```
# Questa è una riga di commento
echo "testo di prova"      # Commento laterale, sulla stessa riga
#la riga seguente e' un comando commentato che non verra' eseguito
#echo "sono un testo che non verra' visto"
```

La variabile PATH

Una delle prime variabili da configurare e' sicuramente **PATH** che definisce le directory a cui possiamo accedere da qualsiasi punto del filesystem, ad esempio per avviare un eseguibile.

In Linux/Unix un eseguibile non è caratterizzato da un particolare suffisso, ma dall'avere il permesso **x** attivo.
Ad esempio: **chmod +x myprg** rende eseguibile il programma myprg.

Supponiamo di avere un eseguibile nella directory /usr/bin chiamato **myprg**, se la directory /usr/bin e' presente nella variabile PATH possiamo lanciare il programma da qualsiasi posizione nel filesystem, altrimenti saremmo costretti o a digitare l'intero percorso **/usr/bin/myprg** oppure a portarci prima nella directory dove sta l'eseguibile con **cd /usr/bin** ed a digitare quindi **./myprg**.

Chi proviene da Windows trova, in genere, strano il dover premettere i caratteri **./** per eseguire un programma che si trova nella stessa directory di lavoro. Si ricordi però che nella costruzione dei **path relativi** i caratteri "**..**" dirigono verso la directory di livello superiore, mentre il carattere "**.**" fissa l'origine a partire dalla directory attuale.

La variabile PATH e' definita in **/etc/profile** (configurazione globale), ed ha una sintassi di assegnazione particolare: ogni percorso di directory alla destra dell'operatore "**=**" è separata dal carattere "**:**", ad esempio:

```
PATH=/bin:/usr/bin:/usr/local/bin
export PATH
```

rendera' possibile mandare in esecuzione da qualsiasi punto del filesystem tutti gli eseguibili che si trovano nelle directory dichiarate nella variabile PATH. Se noi volessimo aggiungere altri valori alla variabile PATH potremmo digitare semplicemente:

```
PATH=$PATH:/usr/games
export PATH
```

il che aggiungera' al valore originale della variabile PATH il nostro nuovo valore, infatti il contenuto $PATH **/bin:/usr/bin:/usr/local/bin** diventera' **/bin:/usr/bin:/usr/local/bin:/usr/games**.

Se si desidera modificare il path solo per l'utente desiderato basta inserire i comandi precedenti in **~/.bash_profile** (configurazione locale). Quindi ogni utente sarà libero di avere una configurazione PATH personalizzata.

Il prompt

Il prompt non e' altro che il contrassegno che indica lo stato di attesa del sistema di un input dell'utente, la sua forma classica e':

```
nome_utente@nome_computer:[#|$]
```

che oltre ad indicare il nome utente e quello del computer, nel classico formato dell'indirizzo email, ricorda se si è utenti normali (**$**) o utenti privilegiati (**#**).
Queste indicazioni dipendono dal contenuto della variabile **PS1** definita nel file **/etc/profile**, che e' di proprieta' dell'utente root. Se si volesse un prompt diverso si potrebbe però ridefinire PS1 nel file **~/.bash_profile** con effetto locale, cioè relativamente all'utente proprietario di *~/.bash_profile*.

La variabile PS1 accetta dei valori predefiniti che sono ottenuti facendo seguire alla barra rovesciata (backslash) dei caratteri speciali che sono:

```
\t      l'ora corrente nel formato HH:MM:SS
\d      la data in formato esteso es. "Tue May 18"
\n      un carattere di nuova linea
\s      il nome della shell
\w      la directory corrente
\W      il percorso completo alla directory corrente
\u      il nome dell'utente
\h      il nome della macchina
\#      il numero del processo associato al comando in esecuzione
\!      la posizione numerica nel file storico dei comandi
```

```
\$      se l'UID e' 0 mostra un "#", altrimenti un "$"
```

Quindi si può personalizzare l'aspetto del prompt inserendo opportunamente questi caratteri. Si veda la tabella seguente che riporta alcuni esempi classici:

```
PS1="[\u@\h \W]\$ "      -->     [user@computer /root]#
PS1="[\t \s]\$ "         -->     [12:18:24 bash]#
```

Oltre a questi caratteri speciali la variabile PS1 puo' contenere anche dei comandi, per esempio se si volesse far apparire la versione del kernel:

```
PS1="`uname -r` \$ "     -->     2.4.17#
```

Si noti l'uso delle particolari virgolette usate per definire il comando, non si tratta infatti delle virgolette classiche ' ma del carattere ` che viene chiamato backtick o virgoletta rovescia, e si ottiene con la combinazione di tasti AltGr+virgoletta semplice.

Oltre la variabile PS1 viene definita anche la variable PS2 che determina l'aspetto del prompt secondario che viene visualizzato quando si digitano comandi incompleti. Questo succede quando abbiamo a che fare con comandi molto lunghi, possiamo digitare il comando su piu' linee, facendo precedere il comando invio da una barra rovesciata (backslash), a quel punto la shell capira' che non abbiamo terminato e ci presentera' il prompt secondario che significa che sta attendendo il completamento del comando. Se per esempio vogliamo vedere come si presenta il prompt secondario nella nostra shell possiamo digitare un comando incompleto:

```
if PS2 then
```

in questo caso non abbiamo dovuto far precedere l'invio da una barra rovescia perche' la shell riconosce i suoi costrutti di programmazione, poi premendo invio ci accorgeremo di avere un prompt diverso, tipicamente:

```
>
```

volendo si potrebbe digitare un semplice

```
echo $PS2
```

per ottenere direttamente il valore della variabile. Inoltre tutto quello che e' stato detto per PS1 vale in linea di massima per PS2.

Gli alias

Capita a volte di dover digitare frequentemente delle linee di comando lunghe e complesse, ecco allora la possibilità di ricorrere agli **alias** che possono essere visti come dei **comandi abbreviati**. Basta eseguire un'operazione di assegnazione tra il nuovo comando ed il vecchio comando ed il gioco è fatto:

```
alias newcmd='oldcmd'                    ad esempio:
alias lss='ls -l --sort=size'
```

D'ora in poi il nuovo comando **lss** produrrà la lista della directory ordinata per dimensione dei file. Volendo conoscere quali sono gli alias già definiti basta digitare semplicemente:

```
alias
```

e si otterrà in output qualcosa del genere:

```
alias ..='cd ..'
alias cp='cp -i'
alias l='ls -a --color=auto'
alias la='ls -la --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias lss='ls -l --sort=size'
alias mv='mv -i'
```

```
alias rm='rm -i'
alias where='type -all'
alias which='type -path'
```

Alcuni alias li troviamo già predefiniti a cura della distribuzione in uso, ma noi possiamo definirne molti altri, quanti ce ne servono (senza limiti), realizzando una raccolta di *comandi in codice* personale.

Il file storico

La shell e' in grado di ricordare i comandi progressivamente immessi dall'utente, normalmente vengono salvati nel file **~/.bash_history** e possono essere richiamati premendo i tasti freccia in su' e freccia in giu'. Questo comportamento puo' essere modificato configurando le variabili:

- **HISTSIZE**: numero massimo dei comandi da memorizzare nel file storico, normalmente preimpostato a 500.
- **HISTFILE**: nome del file che deve essere usato per contenere i comandi digitati, normalmente preimpostato a *~/.bash_history*. Puo' anche non essere impostato ed allora lo storico si limitera' a ricordare i soli comandi della sessione di lavoro corrente.
- **HISTFILESIZE**: determina la grandezza fisica massima che puo' avere il file dello storico.

Ricordiamo che, per consuetudine, le variabili di ambiente vengono sempre indicate usando lettere maiuscole a differenza delle locali per cui si usano le minuscole. Queste variabili sono definite in **/etc/profile**, ma possono essere sovrascritte da quelle ridefinite in **~/.bash_profile**.

Le variabili mail

La shell utilizza le seguenti variabili che influenzano le funzioni di posta e che sono normalmente definite, globalmente, nel file */etc/profile*, oppure, localmente, nel file *~/.bash_profile*:

- **MAIL** - Quando un messaggio di posta arriva all'utente, il suo contenuto e' scritto su di un file, questo file e' definito dalla variabile MAIL che normalmente contiene il valore /var/spool /mail/nome_utente. Si puo'cosi' indicare alla shell il file da controllare per l'arrivo di nuovi messaggi.
- **MAILCHECK** - Questa variabile definisce l'intervallo di tempo che deve trascorrere prima che la casella di posta locale venga controllata. Il valore preimpostato e' 60 che significa che la shell controllera' la directory ogni minuto.
- **MAILPATH** - Questa variabile definisce il percorso per raggiungere le directory di posta normalmente contiene il valore /var/spool/mail . Puo' essere usata anche per personalizzare il messaggio che notifica l'arrivo della nuova posta, esempio: MAILPATH='/var/spool /mail/nome_utente "Hai posta....!"'
- **MAIL_WARNING** - Se questa variabile e' definita, la shell vi informera' con un messaggio, se state leggendo un messaggio gia' letto in precedenza. Visto l'uso ormai consolidato dei client di posta come pine, mutt, o altri grafici questa variabile e' ormai poco usata.

Qualitá e sviluppo del software open

Le prime osservazioni che vengono generalmente fatte sul software open source sono relative alla sua qualità ed alla sua affidabilità. Per risolvere queste perplessità vale la pena mettere a confronto i procedimenti di sviluppo del software aperto e del software chiuso.
Ciò è stato fatto, ed è ben descritto, nel celebre testo di Eric S. Raymond, tradotto in italiano da Bernardo Parrella, **La cattedrale e il bazaar**, disponibile gratuitamente nel sito della Apogeo.
In questo documento i metodi di sviluppo del software chiuso e del software aperto vengono paragonati alla realizzazione di una cattedrale ed al lavorondoliattraverso la metafora della cattedrale e del bazaar.

- Solitamente la costruzione di una cattedrale prevede un piccolo gruppo di ingegneri che si ritrovano in un edificio a progettare. Il loro lavoro viene mantenuto nascosto e pochissimi possono vedere quali sono gli sviluppi. Una volta arrivati ad un progetto preciso, la cattedrale viene finita e mostrata al grande pubblico. Eventuali modifiche vengono segnalate e un "restauro" viene fatto solo successivamente e nuovamente da un piccolo gruppo di persone che lavorano in gran segreto.
- Lo sviluppo del software open avviene in una situazione simile ad un grande mercato aperto dove tanti esperti artigiani vendono il prodotto del proprio lavoro. Ogni artigiano, in attesa che arrivi un cliente, lavora dietro alla costruzione di copie del prodotto che offre. Chiunque è libero di osservare come avviene la lavorazione e, chiunque, all'istante puó fare richieste su come l'oggetto debba essere costruito o su eventuali errori da correggere. Puó succedere che ci sia piú di un artigiano che offre un prodotto simile; il confrontarsi con il vicino permette di aumentare la qualitá di ció che si sta producendo e, in particolari casi, di riunire le forze per produrre qualcosa di qualitá superiore.

Calando le metafore appena fatte nella realtá del software ci rendiamo conto che con entrambi i metodi si possono raggiungere ottimi risultati. Certamente il primo modo di operare parte da una situazione piú ordinata ma la finalitá di entrambi i metodi è quella di avere un prodotto di qualitá.
La sostanziale differenza fra i due è la risoluzione di errori nel programma. Nel primo caso occorre aspettare che *gli ingegneri* si riuniscano nuovamente, nel secondo caso, invece, la risoluzione di errori avviene in maniera costante: chiunque vede un errore lo puó correggere o evidenziare per una rapida correzione.

Rassicurati sul fatto che anche il software open è un prodotto di qualitá occorre tranquillizzarsi anche sulle garanzie fornite da chi sviluppa software open source.
Chi produce software open comincia il suo lavoro seguendo il motto **rilascia presto e rilascia spesso**. Rilasciare presto vuol dire mettersi subito in mostra, far vedere al mondo quale è la propria idea. La prima versione di rilascio deve essere qualcosa giá in grado di offrire un minimo di funzionalitá. In seguito a questo lo sviluppatore, se riesce a conquistare qualche utilizzatore, comincia a ricevere le prime segnalazioni di errori e a trovare collaboratori interessati allo sviluppo di quel software o ad *assorbire* quel contributo in un altro progetto simile o far *assorbire* il proprio. Rilasciando spesso il software viene ogni volta esposto alla *critica* e quindi gli è permesso di crescere rapidamente.

Metodologia e strumenti del lavoro di gruppo

Allo stesso modo di Linux che è stato fortemente sviluppato, subito dopo la prima ideazione, da una comunità di programmatori, tutto il software Open Source utilizza Internet per farsi conoscere e svilupparsi: news group, mailing list, siti specializzati, come **http://freshmeat.net**, sono gli strumenti più comuni.

L'idea - La persona che ha iniziato a sviluppare il software diventa il capoprogetto e con il tempo viene circondato da un gruppo di collaboratori sempre piú grande tra cui possono essere scelti dei *caporeparto* che si occuperanno liberamente dello sviluppo di un componente o di una funzionalitá del software.

Il sito - E' indispensabile un sito web ufficiale che funziona da collante per le varie attività di sviluppo: scaricare le versioni del programma (*stable, unstable*), seguire gli sviluppi e l'evoluzione del codice, leggere le news su quanto il movimento di pensiero sta producendo. A questo proposito le *mailing list* sono lo strumento centrale per quello che riguarda le discussioni sugli sviluppi del

software e quando i temi, a causa della complessitá del progetto, cominciano ad essere molti, si ha la necessitá di crearne piú di una.

Il server CVS - Far lavorare tante persone in gruppo non è una cosa facile, specialmente quando si manipola codice di software che interagisce con quello prodotto da altri. Per questo motivo la gestione del codice avviene attraverso server **CVS** (Concurrent Versions System). Il servizio offerto è chiamato **repository** ed è simile ad un *ripostiglio* nel quale andare per modificare/creare/cancellare il codice sorgente del programma. In questo modo non viene perso nulla del lavoro svolto dal gruppo, si tiene traccia delle modifiche fatte al codice e ognuno puó contribuire alla crescita del programma senza dover prendere ogni volta tutto ció che è stato scritto, ma soltanto gli aggiornamenti per modificare la propria copia in locale. Chi fosse interessato allo sviluppo di software open può consultare, ad esempio, il sito **http://sourceforge.net/** che offre questi servizi.

La distribuzione - La distribuzione del programma (unitamente al codice sorgente) avviene sempre attraverso il susseguirsi di due versioni:

- **unstable** (instabile): questa versione contiene tutte le caratteristiche che si vuole siano presenti nella successiva versione stabile, viene resa disponibile sia agli sviluppatori per scovarne i bug o migliorarne il codice sia agli utilizzatori che si offrono liberamente di testare il programma su cui, ovviamente, non viene data alcuna garanzia di funzionamento sicuro.
- **stable** (stabile): è la versione ripulita da ogni errore (nei limiti umani), perfettamente funzionante e che garantisce un numero di funzionalitá sul quale l'utente finale puó contare.

La distribuzione del software segue poi degli standard, ovviamente uno sviluppatore non è obbligato a seguire queste regole, ma quando il progetto diventa sempre piú grande e importante è lo stesso team che impone delle regole di distribuzione. A titolo di esempio si riporta il protocollo (*regole da seguire*) di distribuzione dei pacchetti Gnu/Gnome. Un pacchetto (*un unico file compresso con tutti i file necessari*) di un programma open deve contenere:

- i file del codice sorgente del programma
- un file di testo **Readme** con informazioni sul programma
- un file di testo **Install** con le istruzioni per l'installazione
- un file eseguibile **configure** che prepara il codice sorgente per la compilazione
- un file di testo **Copyright** con le informazioni sulla licenza d'uso e sugli autori
- un file di testo **Changelog** con le informazioni sulle differenze con la precedente versione
- un file eseguibile **install** per la procedura di installazione automatica
- un file eseguibile **clean** con la procedura inversa agli effetti di **install**
- il programma **gettext** che serve per l'internazionalizzazione del programma

Poichè l'inglese è la lingua piú usata in questo settore tutti i programmi vengono generalmente progettati in inglese, e in un secondo tempo vengono adattati (*localizzati* alle varie lingue. Può quindi capitare che nei gruppi di sviluppo si avvicinino anche persone che, pur non sapendo programmare, decidono di dare un contributo facendo la traduzione del software e occupandosi della creazione di documentazione. Piú un software diventa di interesse internazionale e maggiori saranno le persone che si occuperanno della traduzione.

Il compilatore GCC

Il compilatore **GCC** (**G**NU **C**ompiler **C**ollection) è un compilatore di sorgenti scritto in linguaggio C, è stato realizzato dallo statunitense Richard Stallman nel corso dell'ambizioso progetto per lo sviluppo del Sistema GNU, un sistema operativo libero e compatibile con Unix.
E' ormai diventato parte integrante delle distribuzioni Linux, l'esistenza di gcc e dell'insieme delle varie utilities (make, autoconf, etc.) è ciò che rende possibile la distribuzione in codice sorgente sia del kernel Linux sia di una gran varietà di applicazioni ed utilities.

Già il nome stesso **Compiler Collection** ricorda che è in grado di compilare sorgenti di diversi linguaggi, tra cui: **C**, **C++**, **Objective C**, **Java**, **Fortran** e **ADA** su oltre 40 diverse famiglie di chip (tra cui: **x86** Intel, **x86** a 64 bit, **AMD** a 64 bit, **PowerPC**, **s390** IBM, **Sparc** Sun Microsystems), ed è corredato delle librerie per questi linguaggi.
Il vantaggio principale di questa collezione di compilatori è la portabilità, gira su parecchie piattaforme e generalmente si trova installato di default sui principali sistemi di tipo unix per PC (GNU/Linux, *BSD, ma è contenuto anche nei dischi di installazione di MacOSX). Oggi anche in ambiente Windows è possibile installare gratuitamente gcc, basta collegarsi al sito della **CigWin** e scaricare l'emulatore Unix per Windows, completo di compilatore gcc.
Data la portabilità accade che su alcune piattaforme non è proprio il compilatore che ottimizza maggiormente, ma sui computer moderni e per la maggior parte delle applicazioni la cosa non influisce eccessivamente sulle prestazioni.
E' l'ideale per chi abbia bisogno di un compilatore per installare facilmente programmi open source, ma anche per chi abbia bisogno di scrivere programmi per diverse piattaforme.

In ambiente Linux sono disponibili i comandi **gcc** e **g++**, rispettivamente per compilare in **C** oppure in **C++**. In realtà g++ e' uno script che chiama gcc con opzioni specifiche per riconoscere il C++.

Si puo' determinare la versione del compilatore invocando:

```
gcc -v
```

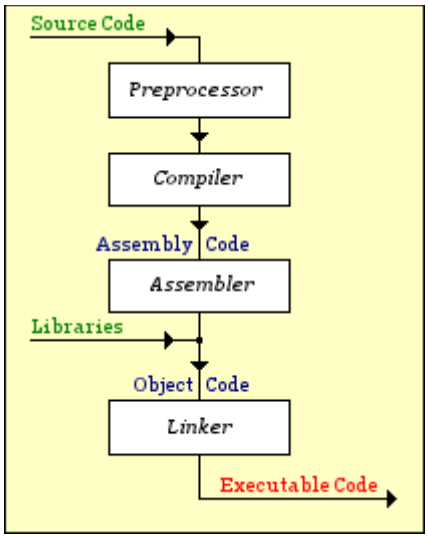
ottenendo in risposta qualcosa del genere:

```
Using built-in specs.
Target: i486-linux-gnu
Configured with: ..............
..............................
Thread model: posix
gcc version 4.0.3 20060115 (prerelease) (Debian 4.0.2-7)
```

Le quattro fasi della compilazione

Sia **gcc** che **g++** processano i file di input attraverso una o piu' delle seguenti quattro fasi:

1. **preprocessing**
   - rimozione dei commenti
   - interpretazioni di speciali direttive per il preprocessore come:
     #include (include il contenuto di un determinato file)
     #define (definisce un nome simbolico o una variabile)
2. **compilation**
   - traduzione del codice sorgente ricevuto dal preprocessore in codice assembly
3. **assembly**
   - creazione del codice oggetto
4. **linking**
   - combinazione delle funzioni definite in altri file sorgenti o definite in librerie con la funzione main() per creare il file eseguibile



Di solito vengono utilizzati dei suffissi standard per distinguere i vari file utilizzati nelle varie fasi del processo di compilazione, ecco i più utilizzati:

```
.c            modulo sorgente C da preprocessare, compilare e assemblare
.cc o .cpp    modulo sorgente C++ da preprocessare, compilare e assemblare
.h            modulo per il preprocessore, di solito non nominato nella riga di commando
.o            modulo oggetto da passare al linker
.a            libreria statica
.so           libreria dinamica
```

Un esempio in C

A semplice titolo di esempio consideriamo il seguente codice sorgente C:

```
/* programma prova1.c */
#include <stdio.h>
int main() {
puts("Ciao Linux!");
return 0;  }
```

Per effettuare la compilazione occorrerà digitare:

```
gcc prova1.c
```

In questo caso l'output di default sarà direttamente l'eseguibile **a.out**. Ma di solito si specifica il nome del file di output desiderato utilizzando l'opzione **-o**, come nella riga seguente:

```
gcc prova1.c -o prova1
```

I due eseguibili ottenuti potranno essere lanciati usando semplicemente uno dei due comandi:

```
./a.out
./prova1
```

Ottenendo in uscita, sul video, in entrambi i casi:

```
Ciao Linux!
```

Occorre fare 2 annotazioni:

1) Gli eseguibili di Linux non sono caratterizzati da un particolare suffisso, ma si riconoscono solo dal permesso **x** di esecuzione attivato.

2) Usare "**./**" puo' sembrare superfluo. In realta' si dimostra molto utile sia per evitare di lanciare involontariamente un programma omonimo presente sul *path*, sia per obbligare Linux ad eseguire proprio il file presente nella propria directory di lavoro.

Ed uno in C++

Consideriamo adesso il codice sorgente analogo scritto per C++:

```
/* programma prova2.cpp */
#include
int main() {
   cout<<"Ciao Linux!"<<'\n';
return 0;  }
```

Per effettuare la compilazione questa volta digiteremo:

```
g++ prova2.cpp -o prova2
```

L'eseguibile ottenuto potrà, al solito, essere lanciato con:

```
./prova2
```

Ottenendo in uscita, sul video:

```
Ciao Linux!
```

Se immediatamente dopo digitiamo il seguente comando:

```
echo $?
```

Verrà visualizzato il valore restituito dal programma al sistema operativo tramite l'ultima istruzione **return 0**. A titolo sperimentale provare a ricompilare con un valore diverso da 0.

Continuazione

E' evidente che le informazioni fornite in questa breve presentazione introduttiva sono insufficienti a dare una panoramica sulle potenzialità di **gcc**. Internet è ricchissima di tutorial e manuali, ormai molti anche in lingua italiana.
Nella sezione **Schede & Guide**, a solo titolo indicativo, viene riportato un link che può servire sicuramente come primo approccio alla programmazione in C/C++ sotto Linux.
Anche la consultazione delle pagine **man** relative a gcc ed a g++ possono tornare utili.