

Laboratorio di Sistemi Operativi
Prima parte: la shell

Alessandro Valenti

27 gennaio 2006

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione al corso | 1 |
| 1.1 | Svolgimento delle lezioni | 1 |
| 1.2 | Obiettivi del corso | 2 |
| 1.3 | Verifica dell'apprendimento | 2 |
| 1.4 | Installazione di Linux | 3 |
| 2 | Concetti di base | 5 |
| 2.1 | Breve storia di Unix e dell'Open Source | 6 |
| 2.2 | Modularità e Processi | 7 |
| 2.3 | Gli utenti del sistema | 7 |
| 2.4 | Accesso ai comandi | 8 |
| 2.5 | Terminare la sessione | 9 |
| 2.6 | Comandi e command line | 9 |
| 2.6.1 | Argomenti ed opzioni | 9 |
| 2.6.2 | Espansione di caratteri speciali | 9 |
| 2.7 | Directory corrente | 10 |
| 2.8 | Le variabili d'ambiente | 11 |
| 2.9 | Input e output dei comandi | 12 |
| 2.10 | Spazi e doppi apici nelle command line | 12 |
| 3 | L'editor VIM | 15 |
| 4 | Il filesystem | 33 |
| 4.1 | File e Directory | 34 |
| 4.2 | Permessi associati ai file | 35 |
| 4.3 | Bit speciali | 36 |
| 4.4 | Altre informazioni sui file | 37 |
| 5 | Usiamo la shell | 39 |
| 5.1 | La ridirezione | 39 |
| 5.1.1 | Standard Error | 41 |
| 5.1.2 | Ridirezioni fra descrittori | 41 |
| 5.2 | Pipeline | 42 |
| 5.3 | Exit value di un comando/processo | 42 |
| 5.4 | Command substitution | 43 |
| 5.5 | Gli shell script | 43 |
| 5.5.1 | Eeguire uno script | 44 |
| 5.5.2 | Esportazione di variabili | 44 |
| 5.5.3 | Uso e definizione di variabili negli shell script | 45 |
| 5.5.4 | Le variabili di shell | 45 |

| | | |
|----------|--|-----------|
| 5.5.5 | Gli argomenti della linea di comando | 46 |
| 6 | Sintassi degli script | 47 |
| 6.1 | Elaborazione condizionale | 47 |
| 6.1.1 | Il comando test | 48 |
| 6.2 | Forzare un exit value | 50 |
| 6.2.1 | Il ciclo for | 50 |
| 6.3 | Il case per le selezioni multiple | 51 |
| 6.4 | Script ricorsivi | 52 |
| 6.4.1 | L'operazione di quoting | 53 |
| 7 | Manpage di alcuni comandi | 55 |
| 7.1 | Il comando cat | 55 |
| 7.1.1 | Name | 55 |
| 7.1.2 | Synopsis | 55 |
| 7.1.3 | Description | 55 |
| 7.1.4 | Author | 56 |
| 7.1.5 | Reporting Bugs | 56 |
| 7.1.6 | Copyright | 56 |
| 7.1.7 | See Also | 56 |
| 7.2 | Il comando chmod | 57 |
| 7.2.1 | Name | 57 |
| 7.2.2 | Synopsis | 57 |
| 7.2.3 | Description | 57 |
| 7.2.4 | Sticky Files | 57 |
| 7.2.5 | Sticky Directories | 58 |
| 7.2.6 | Options | 58 |
| 7.2.7 | Author | 58 |
| 7.2.8 | Reporting Bugs | 58 |
| 7.2.9 | Copyright | 58 |
| 7.2.10 | See Also | 58 |
| 7.3 | Il comando cp | 59 |
| 7.3.1 | Name | 59 |
| 7.3.2 | Synopsis | 59 |
| 7.3.3 | Description | 59 |
| 7.3.4 | Author | 60 |
| 7.3.5 | Reporting Bugs | 60 |
| 7.3.6 | Copyright | 60 |
| 7.3.7 | See Also | 60 |
| 7.4 | Il comando echo | 61 |
| 7.4.1 | Name | 61 |
| 7.4.2 | Synopsis | 61 |
| 7.4.3 | Description | 61 |
| 7.4.4 | Author | 61 |
| 7.4.5 | Reporting Bugs | 61 |
| 7.4.6 | Copyright | 61 |
| 7.4.7 | See Also | 62 |
| 7.5 | Il comando expr | 63 |
| 7.5.1 | Name | 63 |
| 7.5.2 | Synopsis | 63 |
| 7.5.3 | Description | 63 |

| | | |
|--------|-----------------------|----|
| 7.5.4 | Author | 64 |
| 7.5.5 | Reporting Bugs | 64 |
| 7.5.6 | Copyright | 64 |
| 7.5.7 | See Also | 64 |
| 7.6 | Il comando grep | 65 |
| 7.6.1 | Name | 65 |
| 7.6.2 | Synopsis | 65 |
| 7.6.3 | Description | 65 |
| 7.6.4 | Options | 65 |
| 7.6.5 | Regular Expressions | 68 |
| 7.6.6 | Environment Variables | 69 |
| 7.6.7 | Diagnostics | 70 |
| 7.6.8 | Bugs | 70 |
| 7.7 | Il comando ls | 71 |
| 7.7.1 | Name | 71 |
| 7.7.2 | Synopsis | 71 |
| 7.7.3 | Description | 71 |
| 7.7.4 | Author | 73 |
| 7.7.5 | Reporting Bugs | 73 |
| 7.7.6 | Copyright | 73 |
| 7.7.7 | See Also | 73 |
| 7.8 | Il comando mkdir | 74 |
| 7.8.1 | Name | 74 |
| 7.8.2 | Synopsis | 74 |
| 7.8.3 | Description | 74 |
| 7.8.4 | Author | 74 |
| 7.8.5 | Reporting Bugs | 74 |
| 7.8.6 | Copyright | 74 |
| 7.8.7 | See Also | 74 |
| 7.9 | Il comando pwd | 75 |
| 7.9.1 | Name | 75 |
| 7.9.2 | Synopsis | 75 |
| 7.9.3 | Description | 75 |
| 7.9.4 | Author | 75 |
| 7.9.5 | Reporting Bugs | 75 |
| 7.9.6 | Copyright | 75 |
| 7.9.7 | See Also | 75 |
| 7.10 | Il comando rm | 76 |
| 7.10.1 | Name | 76 |
| 7.10.2 | Synopsis | 76 |
| 7.10.3 | Description | 76 |
| 7.10.4 | Options | 76 |
| 7.10.5 | Author | 76 |
| 7.10.6 | Reporting Bugs | 76 |
| 7.10.7 | Copyright | 77 |
| 7.10.8 | See Also | 77 |
| 7.11 | Il comando rmdir | 78 |
| 7.11.1 | Name | 78 |
| 7.11.2 | Synopsis | 78 |
| 7.11.3 | Description | 78 |

| | | |
|---------|-------------------------------|-----|
| 7.11.4 | Author | 78 |
| 7.11.5 | Reporting Bugs | 78 |
| 7.11.6 | Copyright | 78 |
| 7.11.7 | See Also | 78 |
| 7.12 | Il comando test | 79 |
| 7.12.1 | Name | 79 |
| 7.12.2 | Synopsis | 79 |
| 7.12.3 | Description | 79 |
| 7.12.4 | Author | 80 |
| 7.12.5 | Reporting Bugs | 80 |
| 7.12.6 | Copyright | 80 |
| 7.12.7 | See Also | 80 |
| 7.13 | Il comando touch | 81 |
| 7.13.1 | Name | 81 |
| 7.13.2 | Synopsis | 81 |
| 7.13.3 | Description | 81 |
| 7.13.4 | Author | 81 |
| 7.13.5 | Reporting Bugs | 81 |
| 7.13.6 | Copyright | 81 |
| 7.13.7 | See Also | 81 |
| 7.14 | Manpage della shell bash | 82 |
| 7.14.1 | Name | 82 |
| 7.14.2 | Synopsis | 82 |
| 7.14.3 | Copyright | 82 |
| 7.14.4 | Description | 82 |
| 7.14.5 | Options | 82 |
| 7.14.6 | Arguments | 83 |
| 7.14.7 | Invocation | 84 |
| 7.14.8 | Definitions | 85 |
| 7.14.9 | Reserved Words | 85 |
| 7.14.10 | Shell Grammar | 85 |
| 7.14.11 | Comments | 88 |
| 7.14.12 | Quoting | 89 |
| 7.14.13 | Parameters | 90 |
| 7.14.14 | Expansion | 99 |
| 7.14.15 | Redirection | 105 |
| 7.14.16 | Aliases | 107 |
| 7.14.17 | Functions | 108 |
| 7.14.18 | Arithmetic Evaluation | 109 |
| 7.14.19 | Conditional Expressions | 110 |
| 7.14.20 | Simple Command Expansion | 111 |
| 7.14.21 | Command Execution | 112 |
| 7.14.22 | Command Execution Environment | 112 |
| 7.14.23 | Environment | 113 |
| 7.14.24 | Exit Status | 114 |
| 7.14.25 | Signals | 114 |
| 7.14.26 | Job Control | 114 |
| 7.14.27 | Prompting | 115 |
| 7.14.28 | Readline | 117 |
| 7.14.29 | History | 129 |

| | |
|--|-----|
| 7.14.30 History Expansion | 130 |
| 7.14.31 Shell Builtin Commands | 132 |
| 7.14.32 Restricted Shell | 152 |
| 7.14.33 See Also | 153 |
| 7.14.34 Files | 153 |
| 7.14.35 Authors | 154 |
| 7.14.36 Bug Reports | 154 |
| 7.14.37 Bugs | 154 |

Capitolo 1

Introduzione al corso

Il corso di *Laboratorio di Sistemi Operativi* è un insegnamento facoltativo previsto nei Corsi di Laurea triennali di Ingegneria Informatica, Ingegneria Elettronica ed Ingegneria delle Telecomunicazioni.

Il superamento del corso consente allo Studente di acquisire 3 crediti.

La finalità del corso è quella di presentare una serie di esercizi ed esempi di programmazione che consentano allo Studente di comprendere la struttura e l'uso di un sistema operativo. Nello specifico si farà particolare riferimento a sistemi operativi conformi allo standard IEEE POSIX, quali Linux, BSD e Solaris.

Il materiale didattico ed ulteriori informazioni sul corso possono essere ottenuti su internet all'indirizzo:

<http://www.coeing.it/didattica/labso>

Potete contattare il docente tramite posta elettronica all'indirizzo:

valenti.alessandro@unimo.it

L'orario di ricevimento Studenti è il martedì, dalle 13 alle 14 presso il laboratorio LICA previo appuntamento.

1.1 Svolgimento delle lezioni

Lo svolgimento dell'attività didattica è sincronizzato ed abbinato al corso di **Sistemi Operativi** in modo che gli studenti possano applicare in laboratorio concetti già visti e studiati teoricamente; durante l'ora di lezione in aula prevista dall'orario del corso di *Laboratorio di Sistemi Operativi*, tuttavia, ci sarà l'occasione di riprendere gli argomenti teorici di interesse per le successive esercitazioni.

Gli Studenti utilizzeranno i PC del Laboratorio di Base per le esercitazioni; in tali PC è stato installato un sistema di *dual boot* che permette di selezionare all'accensione il sistema operativo da utilizzare fra Windows e Linux.

Durante le ore di lezione al Lab. Base, verrà consentito l'accesso agli Studenti del corso di *Laboratorio di Sistemi Operativi* anche al LICA, dove si potranno utilizzare con lo stesso *account* le workstation Sun/Solaris.

L'accesso al Laboratorio di Base per le esercitazioni è regolamentato in **turni** che rispettino, per quanto possibile, le preferenze espresse dagli Studenti all'atto della **Registrazione al Corso** di *Sistemi Operativi*. Indipendentemente dal turno assegnato, ogni Studente potrà accedere alla lezione teorica in aula.

La registrazione al corso di Sistemi Operativi e Laboratorio di Sistemi Operativi è necessaria per ottenere l'account (username e password) e verrà fatta presso il Laboratorio di Base alle ore 15 di lunedì 16 gennaio 2006.

| Turno | Teoria(Aula I, Dip.Mat) | Pratica(Lab. Base) |
|-------|-------------------------|--------------------|
| 1 | lunedì 14-15 | lunedì 15-17 |
| 2 | lunedì 14-15 | martedì 9-11 |
| 3 | lunedì 14-15 | martedì 11-13 |

1.2 Obiettivi del corso

Gli Studenti che decideranno di frequentare il corso di *Laboratorio di Sistemi Operativi* avranno la possibilità di sperimentare in pratica i concetti appresi dal corso di *Sistemi Operativi* acquisendo la capacità di programmare un sistema Unix tramite il linguaggio di scripting della shell ed il linguaggio c.

Si farà, in particolare, riferimento all'evoluzione **bash** della Bourne shell ed alla versione *improved* del vi ribattezzata **vim**.

Gli obiettivi prefissati per il corso di *Laboratorio di Sistemi Operativi*, che lo studente raggiungerà seguendo le lezioni in laboratorio ed esercitandosi sui temi proposti, sono i seguenti:

- capacità d'uso di macchine unix, della bash e dell'editor vim
- conoscenza della shell di unix e delle modalità di realizzazione di shell script
- padronanza dei sistemi di programmazione concorrente orientati ai processi e conoscenza dei principali metodi di sincronizzazione e comunicazione
- capacità di realizzare programmi in c utilizzando la libreria standard
- padronanza di unix come host di sviluppo, utilizzo del tool make

1.3 Verifica dell'apprendimento

La valutazione degli Studenti viene fatta dalla commissione esaminando la soluzione ad un compito del corso di *Sistemi Operativi* proposto in laboratorio. Ogni Studente potrà pertanto ottenere la valutazione del corso di *Laboratorio di Sistemi Operativi* contestualmente a quella del corso di *Sistemi Operativi*.

Ovviamente, per effetto dei diversi criteri di valutazione, il voto di *Laboratorio di Sistemi Operativi* potrà essere sensibilmente differente da quello di *Sistemi Operativi* sul medesimo svolgimento.

La soluzione dei compiti proposti consiste in alcuni file di testo che l'Esaminando dovrà produrre con l'editor vi/vim riutilizzando a suo piacimento parti di esercitazioni o esempi che potrà consultare durante lo svolgimento.

Ogni compito si compone di due parti:

- la parte shell
- la parte c

La parte shell richiede la realizzazione di uno script interpretabile dalla Bourne shell o dalla bash. La correzione di *Laboratorio di Sistemi Operativi* consente e raccomanda l'uso dei neologismi introdotti dalla bash mentre la correzione di *Sistemi Operativi* considera l'incompatibilità con la Bourne shell come un errore. Per evitare penalizzazioni, lo Studente dovrà produrre script

interpretabili dalla shell **sh** inserendo come commento l'eventuale semplificazione ottenibile con la **bash**.

La parte **c** richiede la realizzazione di uno o più file sorgenti in linguaggio **c** corredati dal relativo **makefile** che, in seguito al processo di compilazione e **linking**, producano un file eseguibile rispondente alle specifiche del testo.

L'assenza di almeno un file richiesto o la presenza di errori sintattici negli script o nei sorgenti **c** invalidano la correzione degli elaborati.

1.4 Installazione di Linux

Gli Studenti che volessero ripetere o estendere gli esercizi proposti anche a casa possono installare un sistema operativo come Linux o BSD nel proprio PC. Per ottenere un CD di installazione è possibile eseguire il download dell'immagine ISO da masterizzare su un supporto scrivibile o acquistare qualche rivista del settore che normalmente include una o più *distribuzioni*.

La distribuzione **Debian** è una delle più complete ed affidabili e rende disponibile per il download una immagine ISO ridotta (circa 100M). Esiste una distribuzione Debian *stabile*, normalmente consigliata per i server in quanto non include software molto aggiornato ma mette a disposizione un sistema di patch molto rapido, ed una *testing* che viene consigliata per tutti gli altri usi.

Attualmente, la distribuzione Debian stabile è indicata con il nome **Sarge** mentre la testing con **Etch**. Per completare il quadro, esiste anche una Debian *unstable* molto aggiornata ma con software a volte molto instabile, denominata **Sid**.

L'immagine ISO ridotta di Debian Etch (quella che personalmente consiglio) è disponibile all'url:

http://cdimage.debian.org/cdimage/daily-builds/etch_d-i/beta1/i386/iso-cd/debian-testing-i386-netinst.iso per l'architettura i386 o alla pagina:

<http://www.debian.org/devel/debian-installer> per le altre architetture/versioni.

L'immagine ISO così ottenuta può essere masterizzata su un CD vuoto ed utilizzata per avviare l'installazione di Debian Linux. Per alcuni PC potrebbe essere necessario abilitare il boot da CD fra le opzioni del BIOS (di solito accessibile all'accensione con il tasto CANCEL o F2).

L'installazione del sistema operativo Linux richiede almeno una partizione (primaria o logica) nella quale inserire il file system di unix (root o /) mentre è caldamente consigliata una seconda partizione (primaria o logica) per lo swap. È possibile distribuire il filesystem su più partizioni e più dischi, anche se questi accorgimenti normalmente sono riservati ad un uso *professionale*; per quello che riguarda il nostro utilizzo, possiamo tranquillamente procedere con le due partizioni di base, root e swap.

Se il PC non ha spazio libero nell'hard disk è possibile ridurre la o le partizioni presenti con software particolari. Questa operazione è comunque molto pericolosa per cui conviene sempre eseguire un backup dei dati importanti prima di procedere con l'operazione. Sembra che questa ultima versione di Debian installer consenta il ridimensionamento di partizioni NTFS ma, ricordiamoci che NTFS è proprietario e che ogni utility Open Source che operi su di esso è frutto di hatching o reverse engineering.

Selezionate o create le partizioni per l'installazione, il processo prosegue con la formattazione e la copia dei pacchetti software fondamentali, chiedendo l'intervento dell'utente solo per impostazioni locali (ora, mail, accesso di rete, ecc).

Terminata questa prima fase, abbiamo a disposizione un sistema minimale ma funzionante che deve essere reso avviabile. Normalmente si lascia che l'installatore sovrascriva il Master Boot Record (MBR) del disco principale così da sostituirsi al boot loader esistente. Questo potrebbe tuttavia rendere non avviabile il sistema operativo preesistente (di solito l'installatore riconosce

tutte le versioni di Windows ma ...) per cui può essere conveniente creare un floppy di avvio ed installare il bootloader in un secondo momento.

In ogni caso, se si procede con l'installazione del bootloader nell'MBR, il bootloader originale viene salvato come file nel filesystem di Linux, per cui potrebbe essere possibile un recupero.

Capitolo 2

Concetti di base

Per quanto si rimandi al corso di *Sistemi Operativi* per ogni approfondimento sulla struttura di un sistema operativo, ritengo opportuno inserire una breve introduzione sull'argomento che potrà essere di aiuto agli Studenti nella comprensione dei lavori successivi.

Un sistema operativo è, a tutti gli effetti, una porzione di software alla quale si delegano una serie di attività legate alla gestione delle risorse di un sistema programmato. Ogni personal computer, ad esempio, viene spesso equipaggiato con un sistema operativo direttamente dal costruttore o dall'installatore in quanto la maggior parte degli utenti si limiterà ad installare dei pacchetti software aggiuntivi già pronti per l'esecuzione delle più comuni attività informatiche.

Gli utenti più esperti, invece, possono provvedere alla sostituzione o all'installazione di altri sistemi operativi o, addirittura, alla realizzazione di programmi in grado di essere eseguiti senza sistema operativo.

L'inserimento di un sistema operativo è quindi una attività discrezionale del sistemista che, tipicamente, dovrà prendere in considerazione pregi e difetti di entrambe le possibilità.

Un PC è, ad esempio, un sistema programmato discretamente complesso che richiederebbe al programmatore che decidesse di utilizzarlo senza sistema operativo una grossa attività di sviluppo per utilizzarne tutte le potenzialità garantendo, nel contempo, la compatibilità con qualsivoglia configurazione. L'entità del lavoro sarebbe invece molto minore per sistemi programmati *dedicati* o *embedded*, quali, ad esempio, sistemi di controllo industriale.

Il ruolo del sistema operativo non si limita solo alla gestione centralizzata delle periferiche, attività questa che viene spesso demandata ad altri componenti software (bios e driver), ma coinvolge anche tutte le funzioni di messa in esecuzione e sincronizzazione di altri moduli. Esistono sistemi operativi in grado di gestire, ad esempio, la coesistenza di più attività concorrenti (*task*) nella stessa CPU.

La progettazione e la realizzazione di alcuni programmi viene notevolmente semplificata se associata all'uso di sistemi operativi che consentono attività concorrenti detti sistemi operativi *multitasking*.

Il multitasking viene comunemente ottenuto alternando su base temporale l'esecuzione dei vari task; in altre parole ogni task ha a propria disposizione una porzione del tempo CPU. Una volta scaduta la porzione temporale a disposizione del task corrente, il contesto del task (memoria e stato dei registri CPU) viene congelato e sostituito con quello del nuovo task da eseguire che, a sua volta, era stato congelato precedentemente.

La determinazione dei *turni* di esecuzione è appannaggio dello **scheduler** che, di fatto, cerca di distribuire nel *miglior modo* il carico di lavoro.

Si noti, solo per completezza, che tale suddivisione delle risorse su base temporale detta *time-slicing*, non è l'unico modo di ottenere il multitasking (*preemptive*, *event-driven*, ecc.).

Nell'utilizzo *storico* di unix il multitasking assume la particolare interpretazione di multiprocesso in quanto ogni programma messo in esecuzione diviene un processo. Un processo è una

entità astratta propria del sistema operativo che ha l'*illusione* di essere l'unico programma in esecuzione ed è associato all'utente che lo ha messo in esecuzione.

Nei sistemi programmati che presuppongono una interazione con l'utente, inoltre, il sistema operativo fornisce, direttamente o tramite eseguibili esterni, una interfaccia di controllo che consenta di impartire comandi e di effettuare interrogazioni; questa interfaccia viene comunemente indicata con il termine *shell*.

La conoscenza di unix, pertanto, richiede una discreta padronanza dei concetti di utente e processo che lo studente otterrà attraverso una serie di esercitazioni relative alla programmazione, prima utilizzando la shell e, in seguito, il linguaggio c.

2.1 Breve storia di Unix e dell'Open Source

L'idea originale di UNIX nasce da alcuni programmatori della AT&T dopo aver seguito lo sviluppo di un sistema operativo chiamato MULTICS, frutto di una attività di ricerca al MIT e risalente agli anni sessanta. In quegli anni, la AT&T si defilava dal progetto MULTICS dando vita ad un proprio gruppo di sviluppo cui parteciparono personaggi dello spessore di Ritchie e Thompson. Il nome UNIX sembra derivare proprio da MULTICS del quale ne rappresenta una semplificazione di nome e di fatto.

Già da queste prime versioni di unix, inizia a delinearsi una importante distinzione fra il nucleo del sistema operativo o kernel ed una serie di utility dedicate alla gestione del file system, dell'ambiente a processi ed all'interfaccia a linea di comando. Coinvolti personalmente nello sviluppo del linguaggio C, gli autori di Unix ricodificano il kernel dall'assembler al C già nel 1973.

Questa decisione influenzerà notevolmente la diffusione di unix rappresentando un enorme passo avanti verso la **portabilità del codice** attraverso la codifica in un linguaggio ad alto livello. In poco tempo, infatti, divenne possibile portare unix su altre architetture (dal PDP-7 al PDP-11 molto diffuso all'epoca) e la diffusione si estese alle Università di tutto il mondo.

Il sistema operativo veniva tipicamente fornito a corredo dell'hardware anche in forma sorgente; questo ha stimolato lo sviluppo di versioni personalizzate fra cui la nota Berkeley Software Distribution dell'Università della California (BSD). Nei decenni seguenti la crescente disponibilità di hardware performante a costi contenuti e la disponibilità dei sorgenti hanno contribuito alla nascita di versioni di unix per quasi tutte le architetture, dai PC ai mainframe.

Il minor costo dell'hardware, tuttavia, indusse le software house a lucrare sul sistema operativo che ben presto cessò di essere distribuito unitamente ai sorgenti seguendo una politica di licensing *close source*. Le varie versioni di Unix assunsero quindi nomi commerciali (System V, XENIX, Solaris, HP-UX, ecc.) ed iniziarono ad essere distribuite indipendentemente dall'hardware e tutt'altro che gratuitamente.

L'approccio *close source* venne messo in discussione dalle idee apparentemente utopistiche di Stallman che, con la sua Free Software Foundation (1985), iniziò ad ipotizzare la realizzazione di un sistema operativo unix-like nel progetto GNU (GNU's Not Unix).

Richard M Stallman ha contribuito con le sue idee e con la sua opera (emacs ed il gcc, ad esempio, sono progetti avviati da RMS) ad una rivoluzione senza precedenti nel mondo informatico; l'idea di garantire la libertà nell'uso, nella modifica e nello sviluppo dei progetti software mediante una licenza analoga e contraria al copyright non ha precedenti. Oggi è possibile utilizzare, analizzare e contribuire al miglioramento di migliaia di progetti distribuiti a sorgenti aperti o **Open Source** e protetti dal diritto di *copyleft* sancito dalla licenza **GPL** (Gnu Public License).

Quello che mancava al progetto GNU era un kernel; fra i progetti esisteva (ed esiste ancora) un kernel avveniristico detto Hurd ma la comunità preferì appoggiarsi al kernel BSD (a sorgenti

aperti, ma con qualche limitazione alla libertà secondo Stallman) o al kernel **Linux** (distribuito da Torvalds in licenza GPL).

Esistono diverse organizzazioni e aziende che rendono disponibili, alcune gratuitamente altre a pagamento¹, delle distribuzioni di software basate sui tool GNU applicati ad un kernel Linux. Fra le distribuzioni *gratuite* troviamo, ad esempio, Debian, Slackware, Gentoo, Ubuntu mentre fra quelle che preferiscono una politica commerciale spiccano Red Hat e Suse.

L'importanza del fenomeno *open source* è ben lontana da quella che emerge in queste righe, tuttavia, almeno nell'ambito del nostro corso, possiamo per ora accontentarci di questa nota sintetica. Torneremo possibilmente sull'argomento quando prenderemo in esame la gestione dei progetti sorgenti e gli strumenti per lo sviluppo concorrente *a più mani*

Unix sarà comunque il sistema operativo di riferimento per le nostre esercitazioni; cercheremo per quanto possibile di astrarre dalle differenze che intercorrono fra le varie versioni anche se, in alcune occasioni, queste appariranno evidenti. POSIX rappresenta un tentativo di standardizzazione ma ... non è l'unico!

2.2 Modularità e Processi

L'aspetto forse più interessante di unix è l'estrema modularità che contraddistingue tutti gli oggetti software che lo compongono. Il sistema operativo è di fatto costituito da un modulo software abbastanza ridotto che ne costituisce il kernel; ogni attività collaterale viene demandata ad opportuni programmi che si interfacciano al kernel mediante *chiamate di sistema* o *system calls*

Anche una operazione apparentemente semplice come l'accesso di un utente al sistema coinvolge diversi componenti come indicato nella diapositiva in figura 2.1. A differenza di altri sistemi operativi, infatti, il kernel di unix non provvede direttamente all'interazione con l'utente ma si limita ad avviare un processo particolare di inizializzazione che rimarrà sempre attivo fino allo spegnimento o *shutdown*; questo processo viene identificato con il nome **init** e, essendo il primo processo messo in esecuzione, sarà identificato con '0'.

Il processo init, a sua volta, provvede ad avviare altri processi secondo una sequenza definita in un opportuno file di configurazione; fra questi si trovano una o più occorrenze di processi *getty* ognuno dei quali prende controllo di un *terminale*.

Attraverso l'identificazione dell'utente, il processo getty ed il successivo processo *login* consentiranno l'accesso ai comandi tramite un interprete detto **shell**.

2.3 Gli utenti del sistema

Anche prescindendo dagli aspetti (seppur importanti) della sicurezza, ogni sistema Unix/Posix richiede l'abbinamento di ogni attività o processo ad un utente. L'accesso di un utente è verificato dal sistema di login che, tipicamente, provvede a verificare la correttezza di una coppia di stringhe dette *username* e *password*.

Il sistema provvede ad abbinare un identificativo numerico (uid) ad ogni utente che contraddistinguerà anche ogni processo avviato *a nome* dell'utente stesso.

In altri termini, ogni processo è univocamente associato ad un utente. In funzione delle caratteristiche di protezione dell'utente, al processo saranno o meno concessi dei diritti di accesso su file o direttori.

Esiste un particolare utente caratterizzato dall'aver uno uid=0, detto *root*, che può vantare ogni diritto di accesso e che, pertanto, viene utilizzato solo per attività di amministrazione o

¹La GPL non impedisce di vendere prodotti basati su software licenziato secondo la GPL stessa. Questo è uno degli aspetti più interessanti di questa politica di licensing

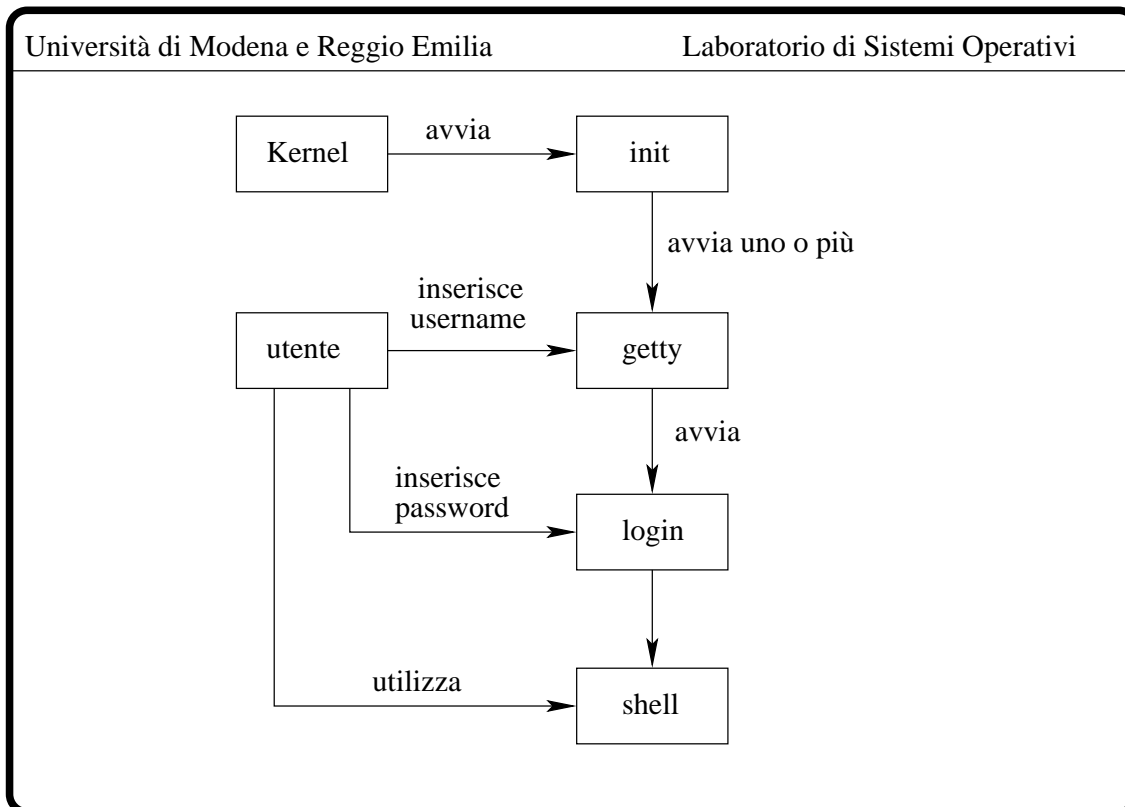


Figura 2.1: init

manutenzione. Cercate di vincere la tentazione ad utilizzare l'account di root per l'uso normale del vostro sistema.

2.4 Accesso ai comandi

L'accesso di un utente al sistema (log in) provoca l'apertura di una sessione di lavoro che, normalmente, è associata ad un processo del sistema operativo che esegue il programma responsabile dell'interazione con l'utente stesso. Questo programma è una **shell**.

La shell indica il proprio stato di *attesa comandi* con una stringa detta **prompt**.

All'apparire del prompt, l'utente avanza le proprie richieste alla shell digitando una *command line*. Questa è una sequenza di caratteri terminata da un ritorno a capo o *newline*.

Una command line può essere composta da più elementi separati da un carattere particolare che, normalmente², è lo spazio. Il primo elemento della command line corrisponde ad un comando della shell o al nome di un file eseguibile, gli elementi successivi rappresentano gli argomenti del comando.

Come primo esempio di command line, chiederemo alla shell che abbiamo ottenuto sul sistema l'esecuzione di un file eseguibile di nome **ls**; questo programma mostra la lista dei file presenti nella directory corrente³.

\$ ls

²I criteri di identificazione di una command line e degli argomenti sono qui volutamente semplificati rispetto alla realtà. Allo stato attuale dell'apprendimento è opportuno che lo Studente focalizzi la propria attenzione sugli aspetti fondamentali, rimandando ad un secondo tempo l'analisi dettagliata di ogni possibilità offerta dalla shell di Unix

³Torneremo presto sul concetto di directory corrente

I primi due caratteri della linea mostrata, ovvero il *dollaro* e lo spazio successivo, rappresenta il prompt; questo può variare da macchina a macchina ma, normalmente, viene terminato con il carattere dollaro per gli utenti comuni e con il carattere cancelletto per l'utente root.

Noi digiteremo solo i caratteri **l** e **s** seguiti da un ritorno a capo. Fatto questo, la shell provvederà ad interpretare la command line mettendo di fatto in esecuzione il programma *ls* di cui otterremo il risultato a video.

2.5 Terminare la sessione

Dal prompt, come abbiamo visto, è possibile invocare dei programmi come *ls* oppure impartire comandi direttamente alla shell. Se vogliamo terminare la sessione di lavoro iniziata al log in, possiamo digitare **exit** seguito dal ritorno a capo.

2.6 Comandi e command line

Ogni linea digitata al prompt viene letta ed interpretata dalla shell che provvede ad eseguire i *comandi* impartiti dall'utente; generalmente i comandi richiesti dall'utente sono *programmi* che la shell provvede a mettere in esecuzione per conto dell'utente stesso. In questo senso la shell rappresenta l'interfaccia fra l'utente ed il sistema operativo che, in definitiva, sarà l'oggetto software responsabile della messa in esecuzione del programma richiesto.

La shell, tuttavia, non si limita a questo. In accordo con una sintassi definita, la shell può essere utilizzata per compiti più complessi che richiedano l'espansione o la modifica di una command line. Esistono infatti dei particolari caratteri come i *wild-card* o operatori di connessione utili ad assemblare più comandi fra loro.

2.6.1 Argomenti ed opzioni

Una command line può contenere, oltre al comando richiesto, altri *gruppi di caratteri* che rappresentano gli argomenti del comando o delle opzioni se iniziano con il carattere - (meno).

```
$ ls /usr/bin -l
```

La shell processa la linea di comando digitata dall'utente identificando il comando da eseguire e passando al comando stesso gli argomenti e le opzioni tramite un vettore di stringhe. Nell'esempio precedente, la shell provvede ad identificare il comando **ls** come prima sequenza di caratteri della command line. A questo punto, la shell verifica se questo comando corrisponde ad un *comando interno* eseguendolo direttamente o, come nel caso di **ls**, avvia la ricerca del programma *esterno* corrispondente.

La ricerca del programma si traduce semplicemente nella ricerca di un *file eseguibile* in un insieme di *directory* all'interno del *file system* avente lo stesso nome del comando.

Se viene trovato *almeno* un file con tali caratteristiche, la shell richiede al sistema operativo la creazione di un nuovo processo che eseguirà il file-comando e provvede a *passare* gli argomenti e le opzioni presenti sulla linea di comando. Nell'esempio, il nuovo processo avrà gli argomenti `"/usr/bin/"` e `"-l"`. Nello specifico, il secondo argomento sarà considerato una opzione dal comando *ls* dato che inizia con il carattere `"-"`.

2.6.2 Espansione di caratteri speciali

Quando una command line contiene dei caratteri speciali o wild-card, la shell esegue una operazione indicata come *espansione*. I wild card o metacaratteri della shell sono l'asterisco `"*"` ed il punto interrogativo `"?"`.

L'asterisco viene espanso in una *qualsunque sequenza di qualsiasi carattere eccetto la barra ed il punto* mentre il punto interrogativo in un qualsiasi *singolo* carattere, sempre ad eccezione di barra e punto. Il prodotto dell'espansione deve comunque essere un nome di file.

Prendiamo, come esempio, la seguente command line:

```
$ ls *
```

La presenza del metacarattere obbliga la shell ad eseguire l'espansione; al posto del singolo asterisco la shell inserisce *tutti* i nomi di file ottenibili con l'espansione, quindi tutti i file della directory corrente⁴ privi di punti nel nome.

Se proviamo ad invocare la command line precedente in una directory vuota, come potrebbe essere la directory corrente di ogni utente al primo accesso, la shell non potrebbe espandere alcun nome di file ed in tal caso l'argomento con il quale la shell invocherebbe il comando ls sarebbe la stringa "*" (nessuna espansione).

In caso contrario la shell espanderebbe l'asterisco come indicato ed il comando ls avrebbe come argomenti l'elenco dei file prodotto.

2.7 Directory corrente

La semplice esecuzione del comando ls ha sollevato la necessità di approfondire alcuni concetti fondamentali:

- come sono organizzati i file in un sistema Unix, ovvero il **filesystem**
- come vengono *collegati* i dispositivi di input e output ai processi

In prima approssimazione, il filesystem di un sistema POSIX compliant è rappresentabile da una struttura gerarchica avente una unica radice /, detta *root* o *barra*, che contiene un insieme di *file*.

Il concetto di *file* in Unix è estremamente ampio e comprende, oltre ai file *regolari*, un insieme di file speciali fra cui le *directory* che altro non sono se non dei file che contengono un insieme di riferimenti ad altri file.

Dal punto di vista dell'utente, una directory è *assimilabile* ad una suddivisione del filesystem anche se, di fatto, rappresenta solo una *lista di nomi e riferimenti*.

Ogni processo POSIX ha una informazione di stato che indica un file speciale di tipo directory all'interno del filesystem che ne rappresenta la **directory corrente**.

L'accesso ai file elencati nella directory corrente di un processo si ottiene semplicemente indicando il nome del file così come appare nella directory stessa. Questo modo di indicare un file è detto **relativo semplice**.

Supponiamo, ad esempio, che l'esecuzione del comando ls di cui sopra abbia mostrato l'esistenza di un file di nome *uno*. Questo significa che nella directory corrente è presente un riferimento di nome **uno** ad un file.

Se chiediamo alla shell di eseguire nuovamente il comando ls con l'argomento **uno**, indichiamo a ls di restringere la lista al solo file uno.

```
$ ls uno
```

Torneremo in seguito sulla definizione di *lista di nomi* che abbiamo dato alla directory; proviamo tuttavia ad aggiungere l'opzione **-i** alla precedente command line ed avremo più chiara la funzione della directory: associa il nome *uno* all'**inode** ove *fisicamente* risiede il file.

```
$ ls -i uno
```

⁴si deve trattare di una stringa senza barre, quindi un nome relativo semplice

La directory corrente del processo associato alla nostra shell può essere visualizzata con il comando *pwd*. L'output del comando identifica il nome **assoluto** della directory corrente.

Un nome *assoluto* di un file rappresenta⁵ un percorso che conduce dalla barra al file. Ogni passaggio di directory viene indicato con il carattere */*.

Ogni utente delle macchine Linux del Laboratorio Base, ad esempio, potrà verificare che la directory corrente della propria shell è:

```
$ pwd
/home/n12345
```

dove, ovviamente, 12345 è sostituito dal vero numero tessera. L'output del comando *pwd* viene letto in questo modo: la directory corrente è elencata con il nome *n12345* nella directory elencata con il nome *home* dalla directory barra.

La directory corrente può essere tipicamente⁶ modificata dall'utente mediante il comando *cd*.

Se, ad esempio, invochiamo il seguente comando:

```
$ cd /home
```

la directory corrente della shell viene posizionata in */home*; dopo questa impostazione, potremo accedere al file *uno* di cui sopra in uno qualsiasi dei seguenti modi:

- */home/n12345/uno*
- *n12345/uno*

Il secondo modo mette in evidenza un nuovo tipo di nome di file diverso sia dal *nome assoluto* che dal *nome relativo semplice* visti prima: si tratta del **nome relativo** che perde la specifica di semplice.

2.8 Le variabili d'ambiente

Una command line può essere utilizzata anche per scopi diversi da quello di invocare un comando, ad esempio per interagire direttamente con la shell; la command line seguente definisce o modifica una *variabile d'ambiente* avente per nome "variabile"

```
$ variabile=valore
```

Dopo l'esecuzione della command line, la shell avrà una variabile d'ambiente con nome *variabile* e valore *valore*. Se una variabile con lo stesso nome fosse stata presente prima dell'esecuzione della command line, il suo valore sarebbe andato perso e sovrascritto dal nuovo.

Le variabili d'ambiente sono *stringhe* e manipolate come tali dalla shell. Se si vuole utilizzare il valore di una variabile d'ambiente in una command line, sarà sufficiente inserire il nome di tale variabile preceduto dal carattere dollaro "\$".

```
$ echo $variabile
valore
```

Il comando **echo** serve ad emettere su *standard output* tutti gli argomenti con i quali viene invocato; in funzione delle varie implementazioni di shell può essere fornito dalla shell stessa (per la bash, ad esempio, si tratta di una funzione built-in) o da un comando esterno (un file eseguibile).

⁵Vedremo in seguito che potrebbe non essere l'unico

⁶a volte questo viene impedito dagli amministratori per esigenze di sicurezza, invocando la shell ristretta

2.9 Input e output dei comandi

Ogni comando eseguito dalla shell può produrre dei risultati che, normalmente, vengono visualizzati sullo schermo. Il flusso di dati generato da un processo viene riversato in un *file* detto standard output. Nell'esempio precedente, il comando `echo` ha generato un flusso di caratteri che è stato riversato sullo standard output mentre questo era *collegato* al terminale, quindi al video.

Analogamente esiste un file di ingresso indicato come standard input che rappresenta il canale predefinito dal quale i comandi possono prelevare il flusso di caratteri entrante nel terminale attraverso la tastiera.

Esiste un comando che semplicemente legge il flusso di dati dallo standard input e lo riversa su standard output. Se vogliamo verificare questo funzionamento possiamo chiedere alla shell di mettere in esecuzione questa `command line`:

```
$ cat
```

Ogni linea digitata dalla tastiera viene pedissequamente ripetuta sul terminale fino a quando non indicheremo al comando `cat` che lo standard input è terminato. Questo si fa inviando il carattere particolare di *fine file* che si ottiene premendo la combinazione di tasti **Control D**.

Il comando `cat` evidenzia la sua utilità quando si abbina ad una importante funzionalità della shell: la ridirezione. Mediante i caratteri particolari maggiore e minore, infatti, è possibile istruire la shell a collegare standard input o standard output ad un file diverso dal terminale:

```
$ cat > mioout.txt
```

La `command line` così modificata provoca l'esecuzione del comando `cat` dopo aver collegato lo standard output del processo al file `mioout.txt` che, se non esiste, viene creato. Come per la precedente invocazione, il comando `cat` riporta le linee lette da standard input su standard output ma questa volta non si tratta dello stesso terminale ma di un file reale nella directory corrente.

Terminando il comando con il `control D`, possiamo verificare che nella directory corrente è apparso il nuovo file e possiamo visualizzarne il contenuto con il comando:

```
$ cat < mioout.txt
```

2.10 Spazi e doppi apici nelle `command line`

Come abbiamo potuto osservare in queste semplici esperienze con la shell, il carattere "spazio" riveste un ruolo importante nelle `command line`: indica la separazione fra gli *elementi*, siano essi comandi, argomenti, opzioni o caratteri particolari. Invocare un comando senza inserire lo spazio fra il nome e le opzioni solleva un errore di sintassi:

```
valealex@copperbottom:~$ ls-l
bash: ls-l: command not found
```

In certe occasioni, tuttavia, potrebbe essere necessario utilizzare lo spazio come carattere *normale*; come vedremo, la necessità di inibire qualche comportamento standard della shell sui caratteri speciali è molto comune e meno banale di quanto si potrebbe inizialmente credere. L'operazione che si compie per questo fine prende il nome di **quoting**.

Prendiamo un semplice esempio: supponiamo di voler assegnare alla variabile "variabile" già vista in precedenza il valore "un valore"

Se procediamo senza nessun accorgimento, otteniamo:

```
$ variabile=un valore
bash: valore: command not found
```

La shell ha interpretato lo spazio come separatore, credendo che valore fosse il nome di un comando impartito dall'utente. Se avessimo invece racchiuso la stringa fra virgolette o doppi apici, il risultato sarebbe stato sensibilmente diverso:

```
$ variabile="un valore"
$ echo $variabile
un valore
```


Capitolo 3

L'editor VIM

L'editing di file di testo è il principale strumento per sviluppare applicazioni di shell (o script) e per produrre o modificare i sorgenti di un programma C.

L'offerta di strumenti per l'editing è estremamente vasta ma noi concentreremo l'attenzione sul programma **vi**, disponibile su qualsiasi versione di unix. Di questo editor esistono alcune versioni evolute che, pur mantenendo l'approccio *modale* del vi classico, rappresentano dei veri ed affidabili strumenti di sviluppo, integrati con il sistema di documentazione e di compilazione di unix.

Faremo uso della versione evoluta denominata **vim**, installata su tutti i PC del laboratorio di Base e disponibile in tutte le versioni di Linux e BSD.

Le ore di laboratorio a disposizione degli Studenti per la prima settimana di corso (lunedì 16/01 dopo la registrazione e fino alle 18, martedì 17/01 dalle 9 alle 13) saranno dedicate all'esercitazione compresa nel vim.

Gli Studenti accederanno al laboratorio senza poter usufruire della divisione in turni e del proprio account; gli Studenti dovranno pertanto organizzarsi fra loro per distribuirsi nel tempo a disposizione.

L'accesso senza account personalizzato avverrà utilizzando la seguente coppia di username e password:

```
login: guest
Password: prova
```

Al prompt della shell, si dovrà digitare la seguente command line:

```
$ vimtutor it
```

Di seguito si riportano le parti più interessanti della lezione che potrete eventualmente commentare con le Vostre esperienze:

```
=====
=   Benvenuto   alla   G u i d a   all'Editor   V I M   -   Versione 1.5   =
=====
```

```
Vim e' un Editor molto potente ed ha parecchi comandi, troppi per
spiegarli tutti in una guida come questa. Questa guida serve a
descrivere quei comandi che ti permettono di usare facilmente
Vim come Editor di uso generale.
```

```
Il tempo necessario per completare la guida e' circa 25-30 minuti,
a seconda di quanto tempo dedichi alla sperimentazione.
```

I comandi nelle lezioni modificano questo testo. Fai una copia di questo file per esercitarti (se hai usato "vimtutor", stai gia' usando una copia).

E' importante non scordare che questa guida vuole insegnare tramite l'uso. Questo vuol dire che devi eseguire i comandi per impararli davvero. Se leggi il testo e basta, dimenticherai presto i comandi!

Adesso, assicurati che il tasto BLOCCA-MAIUSCOLO non sia schiacciato e premi il tasto `j` tanto da muovere il cursore fino a che la Lezione 1.1 riempia completamente lo schermo.

~~~~~  
Lezione 1.1: MOVIMENTI DEL CURSORE

**\*\* Per muovere il cursore, premi i tasti h,j,k,l come indicato. \*\***

^

`k`     NOTA: Il tasto `h` e' a sinistra e muove a sinistra.

`< h l >` Il tasto `l` e' a destra e muove a destra.

`j`     Il tasto `j` ricorda una freccia in giu'.

v

1. Muovi il cursore sullo schermo finche' non ti senti a tuo agio.

2. Tieni schiacciato il tasto "giu'" (`j`) finche' non si ripete il movimento.  
---> Adesso sai come arrivare fino alla lezione successiva.

3. Usando il tasto "giu'" spostati alla Lezione 1.2.

NOTA: Quando non sei sicuro del tasto che hai premuto, premi `<ESC>` per andare in Modalita' Normale [Normal Mode]. Poi ri-immetti il comando che volevi.

NOTA: I tasti con le frecce fanno lo stesso servizio. Ma usando `hjkl` riesci a muoverti molto piu' rapidamente, una volta presa l'abitudine.

~~~~~  
Lezione 1.2: ENTRARE E USCIRE DA VIM

!! NOTA: Prima di eseguire quanto richiesto, leggi la Lezione per intero!!

1. Premi il tasto `<ESC>` (per assicurarti di essere in Modalita' Normale).

2. Batti: `:q! <INVIO>`.

---> Così' esci dall'Editor SENZA SALVARE alcuna modifica fatta.

Se vuoi uscire SALVANDO le modifiche batti:

`:wq <INVIO>`

3. Quando vedi il PROMPT della Shell, batti il comando con cui sei arrivato qui. Potrebbe essere: `vimtutor <INVIO>`

Normalmente useresti: vim tutor <INVIO>

---> 'vim' indica l'Editor vim, 'tutor' e' il nome del file che tu vuoi aprire.

4. Se hai memorizzato questi comandi e ti senti pronto, esegui i passi da 1 a 3 per uscire e rientrare nell'Editor. Poi muovi il cursore in giu' fino alla Lezione 1.3.

~~~~~  
 Lezione 1.3: EDITING DI TESTI - CANCELLAZIONE

**\*\* In Modalita' Normale premi x per cancellare il carattere sotto al cursore \*\***

1. Muovi il cursore alla linea piu' sotto, indicata da --->
2. Per correggere errori, muovi il cursore fino a posizionarlo sopra il carattere da cancellare.
3. Premi il tasto x per cancellare il carattere sbagliato.
4. Ripeti i passi da 2 a 4 finche' la frase e' corretta.

---> La mmucca salt finnoo allaa lunnna.

5. Ora che la linea e' corretta, vai alla Lezione 1.4

NOTA: Mentre segui questa guida, non cercare di imparare a memoria, ma impara facendo pratica.

~~~~~  
 Lezione 1.4: EDITING DI TESTI - INSERIMENTO

**** Quando sei in Modalita' Normale premi i per inserire testo. ****

1. Muovi il cursore alla prima linea qui sotto, indicata da --->
2. Per rendere la prima linea uguale alla seconda, muovi il cursore sopra il primo carattere DOPO la posizione in cui il testo va inserito.
3. Premi i e batti le aggiunte opportune.
4. Quando un errore e' corretto, premi <ESC> per tornare in Modalita' Normale. Ripeti i passi da 2 a 4 fino a completare la correzione della frase.

---> C'era del tsto mncnt questa .

---> C'era del testo mancante da questa linea.

5. Quando sei a tuo agio nell'inserimento di testo vai al sommario sotto.

~~~~~  
 Lezione 1 SOMMARIO

1. Il cursore si muove usando i tasti con le frecce o i tasti hjkl.  
 h (sinistra) j (giu') k (su) l (destra)
2. Per eseguire Vim (dal prompt \$) batti: vim NOMEFILE <INVIO>
3. Per uscire da Vim batti: <ESC> :q! <INVIO> per uscire senza salvare.  
 oppure batti: <ESC> :wq <INVIO> per uscire salvando modifiche.
4. Per cancellare il carattere sotto al cursore in Modalita' Normale batti: x
5. Per inserire testo subito prima del cursore in Modalita' Normale batti:  
 i batti del testo <ESC>

NOTA: premendo <ESC> ritornerai in Modalita' Normale o annullerai un comando errato che puoi aver inserito in parte.

Ora continua con la Lezione 2.

~~~~~  
 Lezione 2.1: COMANDI DI CANCELLAZIONE

** Batti dw per cancellare fino a fine parola. **

1. Premi <ESC> per accertarti di essere in Modalita' Normale.
2. Muovi il cursore fino alla linea qui sotto, indicata da --->
3. Muovi il cursore all'inizio di una parola che vuoi cancellare.
4. Batti dw per cancellare la parola.

NOTA: Le lettere dw saranno visibili sull'ultima linea dello schermo mentre le batti. Se hai battuto qualcosa di sbagliato, premi <ESC> e ricomincia.

---> Ci sono le alcune parole gioia che non c'entrano carta in questa frase.

5. Ripeti i passi 3 e 4 finche' la frase e' corretta, poi vai alla Lezione 2.2.
- ~~~~~

Lezione 2.2: ALTRI COMANDI DI CANCELLAZIONE

**** Batti d\$ per cancellare fino a fine linea. ****

1. Premi <ESC> per accertarti di essere in Modalita' Normale.
 2. Muovi il cursore fino alla linea qui sotto, indicata da --->
 3. Muovi il cursore alla fine della linea corretta (DOPO il primo .).
 4. Batti d\$ per cancellare fino a fine linea.
- > Qualcuno ha battuto la fine di questa linea due volte. linea due volte.
5. Vai alla Lezione 2.3 per capire il funzionamento di questo comando.

~~~~~

Lezione 2.3: COMANDI E OGGETTI

Il formato del comando d [delete] cancella e' il seguente:

[numero] d oggetto OPPURE d [numero] oggetto

Dove:

numero - indica quante volte va eseguito il comando (se omissso, vale 1).

d - e' il comando di cancellazione.

oggetto - indica dove il comando va applicato (lista qui sotto).

Breve lista di oggetti:

w - dal cursore alla fine della parola, incluso lo spazio.

e - dal cursore alla fine della parola, ESCLUSO lo spazio.

\$ - dal cursore fino a fine linea.

NOTA: Per amanti dell'avventura: premendo solo il tasto che indica l'oggetto mentre siete in Modalita' Normale, senza dare un comando, sposta il cursore come specificato nella "lista di oggetti" qui sopra.

~~~~~

Lezione 2.4: UNA ECCEZIONE A 'COMANDO-OGGETTO'

**** Batti dd per cancellare un'intera linea. ****

Per la frequenza con cui capita di cancellare linee intere, chi ha progettato Vi ha deciso che sarebbe stato piu' semplice battere due d consecutive per cancellare una linea.

1. Muovi il cursore alla linea 2) nella frase qui sotto.
2. Batti dd per cancellare la linea.
3. Ora spostati alla linea 4).
4. Batti 2dd (ricorda: numero-comando-oggetto) per cancellare due linee.

- 1) Le rose sono rosse,
- 2) Nel fango ci si diverte,
- 3) Le viole sono blu,
- 4) Io ho un'automobile,
- 5) Gli orologi segnano il tempo,
- 6) Il miele e' dolce,
- 7) E lo sei anche tu.

~~~~~  
 Lezione 2.5: IL COMANDO UNDO [ANNULLA]

\*\* Premi u per annullare gli ultimi comandi eseguiti. \*\*  
 \*\* Premi U per annullare le modifiche all'ultima linea. \*\*

1. Muovi il cursore fino alla linea qui sotto, indicata da ---> e posizionati sul primo errore.
2. Batti x per cancellare il primo carattere sbagliato.
3. Adesso batti u per annullare l'ultimo comando eseguito.
4. Ora invece, correggi tutti gli errori sulla linea usando il comando x .
5. Adesso batti una U Maiuscola per riportare la linea al suo stato originale.
6. Adesso batti u piu' volte per annullare la U e i comandi precedenti.
7. Adesso batti piu' volte CTRL-r (tenendo il tasto CTRL schiacciato mentre batti r) per rifare i comandi (annullare l'annullamento).

---> Correggi gli errori ssu questa linea e riimpiazzali coon "undo".

8. Questi comandi sono molto utili. Ora spostati al Sommario della Lezione 2.

~~~~~  
 Lezione 2 SOMMARIO

1. Per cancellare dal cursore fino alla fine di una parola batti: dw
2. Per cancellare dal cursore fino alla fine della linea batti: d\$
3. Per cancellare un'intera linea batti: dd

4. Il formato per un comando in Modalita' Normale e':

[numero] comando oggetto OPPURE comando [numero] oggetto

Dove:

numero - indica quante volte va eseguito il comando (se omissso, vale 1).

comando - e' il comando da eseguire, ad es. d per [delete] cancellare.

oggetto - indica dove il comando va applicato, ad es. w [word] parola,

\$ (fino alla fine della linea), etc.

5. Per annullare i comandi precedenti, batti: u (u minuscola)

Per annullare tutte le modifiche a una linea batti: U (U Maiuscola)

Per annullare l'annullamento [gli "undo"] batti: CTRL-r

~~~~~

### Lezione 3.1: IL COMANDO PUT [METTI, PONI]

**\*\* Batti p per porre [put] l'ultima cancellazione dopo il cursore. \*\***

1. Muovi il cursore alla prima linea fra quelle qui in basso.
2. Batti dd per cancellare la linea e depositarla nel buffer di Vim.
3. Muovi il cursore fino alla linea SOPRA quella dove andrebbe spostata la linea che hai appena cancellato.
4. Mentre sei in Modalita' Normale, batti p per reinserire la linea.
5. Ripeti i passi da 2 a 4 per mettere tutte le linee nel corretto ordine.

d) Riesci a impararla tu?

b) Le viole sono blu,

c) La saggezza si impara,

a) Le rose sono rosse,

~~~~~

Lezione 3.2: IL COMANDO REPLACE [RIMPIAZZA]

**** Batti r e una lettera per rimpiazzare il carattere sotto al cursore. ****

1. Muovi il cursore alla prima linea qui sotto, indicata da --->
2. Muovi il cursore fino a posizionarlo sopra il primo errore.
3. Batti r e poi il carattere che dovrebbe rimpiazzare l'errore.
4. Ripeti i passi 2 e 3 finche' la prima linea e' corretta.

---> Immattendo quetta libea, qualcuno ho predato alcuni tosti sballati!
 ---> Immettendo questa linea, qualcuno ha premuto alcuni tasti sbagliati!

5. Ora passa alla Lezione 3.2.

NOTA: Ricordati che dovrete imparare con la pratica, non solo leggendo.

~~~~~  
 Lezione 3.3: IL COMANDO CHANGE [CAMBIA]

**\*\* Per cambiare una parola in tutto o in parte, batti cw . \*\***

1. Muovi il cursore alla prima linea qui sotto, indicata da --->
2. Posiziona il cursore alla u in lubw.
3. Batti cw e la parola corretta (in questo caso, batti inea ).
4. Premi <ESC> e vai sull'errore seguente (sul primo carattere da modificare).
5. Ripeti i passi 3 e 4 finche' la prima frase e' uguale alla seconda.

---> Questa lubw ha alcune pptfd da asdert usgfk il comando CHANGE.

---> Questa linea ha alcune parole da cambiare usando il comando CHANGE.

Nota che cw non solo rimpiazza la parola, ma ti mette anche in Modalita' Inserimento [Insert Mode]

~~~~~  
 Lezione 3.4: ALTRI CAMBIAMENTI USANDO c

**** Il comando c [CHANGE] agisce sugli stessi oggetti del comando d [DELETE] ****

1. Il comando CHANGE si comporta come DELETE. Il formato e':

```
[numero] c oggetto OPPURE c [numero] oggetto
```
2. Gli oggetti sono gli stessi, ad es. w (word, parola), \$ (fine linea), etc.
3. Muovi il cursore alla prima linea qui sotto, indicata da --->
4. Posiziona il cursore al primo errore.
5. Batti c\$ per modificare il resto della linea secondo il modello della

linea successiva, e quando hai finito premi <ESC>

- > La fine di questa linea deve essere aiutata a divenire come la seguente.
 ---> La fine di questa linea deve essere corretta usando il comando c\$.

~~~~~  
 Lezione 3 SOMMARIO

1. Per reinserire testo che hai appena cancellato, batti p . Questo inserisce [pone] il testo cancellato DOPO il cursore (se era stata tolta una linea intera, questa verra' messa nella linea SOTTO il cursore).
2. Per rimpiazzare il carattere sotto il cursore, batti r e poi il carattere sostitutivo.
3. Il comando CHANGE ti permette di cambiare l'oggetto specificato dal cursore fino alla fine dell'oggetto. Ad es. Batti cw per cambiare dal cursore alla fine della parola, c\$ per cambiare fino a fine linea.
4. Il formato del comando CHANGE e':

[numero] c oggetto            OPPURE c    [numero] oggetto

Ora vai alla prossima Lezione.

~~~~~  
 Lezione 4.1: POSIZIONAMENTO E SITUAZIONE FILE

- ** Batti CTRL-g per vedere a che punto sei nel file e la situazione del file.
 Batti [numero] G per raggiungere il numero della linea [numero] nel file.
 Batti [numero] % per posizionarti alla percentuale [numero] nel file **

NOTA: Leggi l'intera Lezione prima di eseguire un qualunque comando!!

1. Tieni premuto il tasto CTRL e batti g . Una linea di situazione sara' visibile in fondo alla pagina con il NOME FILE e la linea in cui sei posizionato. Ricordati il numero della linea per il Passo 3.
2. Premi G [G Maiuscolo] per posizionarti alla fine del file.
3. Batti il numero della linea in cui ti trovavi e poi G . Questo ti riporterà fino alla linea in cui ti trovavi quando avevi battuto CTRL-g. (Mentre batti i numeri, questi NON saranno visualizzati sullo schermo.)
4. Se ti senti sicuro nel farlo, esegui i passi da 1 a 3.

~~~~~  
 Lezione 4.2: IL COMANDO SEARCH [RICERCA]

**\*\* Batti / seguito da una frase per ricercare quella frase. \*\***

1. in Modalita' Normale batti il carattere / . Nota che la "/" e il cursore sono visibili in fondo dello schermo come quando si usa il comando : .
2. Adesso batti 'erroore' <INVIO>. Questa e' la parola che vuoi ricercare.
3. Per ricercare ancora la stessa frase, batti soltanto n .  
 Per ricercare la stessa frase in direzione opposta, batti N .
4. Se vuoi ricercare una frase in direzione opposta (in su), usa il comando ? invece che / .

---> Quando la ricerca arriva a fine file, ricomincia dall'inizio del file.

"erroore" non e' il modo giusto di digitare errore; erroore e' un errore.

~~~~~  
 Lezione 4.3: RICERCA DI PARENTESI CORRISPONDENTI

**** Batti % per trovare una),], o } corripendenti. ****

1. Posiziona il cursore su un (, [, or { nella linea, indicata da --->
2. Adesso batti il carattere % .
3. Il cursore dovrebbe ora trovarsi sulla parentesi corrispondente.
4. Batti % per muovere il cursore alla parentesi di prima (corrispondente)

---> Questa (e' una linea di test con (, [] e { } al suo interno.))

NOTA: Questo e' molto utile nel "debug" di un programma con parentesi errate!

~~~~~  
 Lezione 4.4: UN MODO PER CORREGGERE GLI ERRORI

**\*\* Batti :s/vecchio/nuovo/g per sostituire 'nuovo' a 'vecchio'. \*\***

1. Muovi il cursore fino alla linea qui sotto, indicata da --->.

2. Batti :s/lla/la <INVI0> . Nota che questo comando cambia solo LA PRIMA occorrenza di "lla" sulla linea.

3. Adesso batti :s/lla/la/g dove "g" sta per "globalmente" sulla linea. Questo cambia TUTTE le occorrenze di "lla" sulla linea.

---> lla stagione migliore per lla fioritura e' lla primavera.

4. Per cambiare ogni ricorrenza di una stringa di caratteri tra due linee, batti :#,#s/vecchio/nuovo/g dove #,# sono i numeri delle due linee.  
Batti :%s/vecchio/nuovo/g per cambiare ogni occorrenza nell'intero file.

~~~~~

Lezione 4 SOMMARIO

1. CTRL-g visualizza a che punto sei nel file e la situazione del file.
G [G Maiuscolo] ti porta alla fine del file. Un numero di linea seguito da G [G Maiuscolo] ti porta a quel numero di linea nel file.
2. Battendo / seguito da una frase ricerca IN AVANTI quella frase.
Battendo ? seguito da una frase ricerca ALL'INDIETRO quella frase.
DOPO una ricerca batti n per trovare la prossima occorrenza nella stessa direzione, oppure N per cercare in direzione opposta.
3. Battendo % mentre il cursore si trova su (,)[,]{, oppure } ti posizioni sulla corrispondente parentesi.
4. Per sostituire "nuovo" al primo "vecchio" in 1 linea batti :s/vecchio/nuovo
Per sostituire "nuovo" ad ogni "vecchio" in 1 linea batti :s/vecchio/nuovo/g
Per sostituire frasi tra 2 numeri di linea [#] batti :#,#s/vecchio/nuovo/g
Per sostituire tutte le occorrenze nel file batti :%s/vecchio/nuovo/g
Per chiedere conferma ogni volta aggiungi 'c' :%s/vecchio/nuovo/gc

~~~~~

#### Lezione 5.1: COME ESEGUIRE UN COMANDO ESTERNO

**\*\* Batti :! seguito da un comando esterno per eseguire il comando. \*\***

1. Batti il comando `!` : Per posizionare il cursore in fondo allo schermo. Ci ti permette di immettere un comando.
2. Adesso batti il carattere `!` (punto esclamativo). Ci ti permette di eseguire qualsiasi comando esterno che si pu eseguire nella "shell".
3. Ad esempio batti `ls` dopo il `!` e poi premi `<INVIO>`. Questo visualizza una lista della tua directory, proprio come se fossi in una "shell". Usa `!dir` se `ls` non funziona. [Unix:ls MSDOS:dir]

---> NOTA: E' possibile in questo modo eseguire un comando a piacere.

---> NOTA: Tutti i comandi `:` devono essere terminati premendo `<INVIO>`

~~~~~  
 Lezione 5.2: ANCORA SULLA SCRITTURA DEI FILES

**** Per salvare le modifiche apportate a un file batti `:w NOMEFILE.` ****

1. Batti `!dir` or `!ls` per procurarti una lista della tua directory. Gia' sai che devi premere `<INVIO>` dopo aver scritto il comando.
2. Scegli un NOMEFILE che ancora non esista, ad es. `TEST` .
3. Adesso batti: `:w TEST` (dove `TEST` e' il NOMEFILE che hai scelto).
4. Questo salva l'intero file ("tutor.it") con il nome di `TEST`. Per una verifica batti ancora `!dir` per listare la tua directory.

---> Nota che se esci da Vim e riesegui Vim usando come NOMEFILE `TEST`, il file sara' una copia esatta di "tutor.it" al momento del salvataggio.

5. Ora cancella il file battendo: `!rm TEST` [sotto Unix] oppure `!del TEST` [sotto MSDOS]

~~~~~  
 Lezione 5.3: SCRIVERE IN MANIERA SELETTIVA

**\*\* Per salvare una porzione del file, batti `:#,# w NOMEFILE` \*\***

1. Batti ancora `!dir` o `!ls` per procurarti una lista della tua directory e scegli un NOMEFILE adatto, come ad es. `TEST` .
2. Muovi il cursore in cima a questa pagina e batti `CTRL-g` per procurarti

il numero di linea. RICORDATI QUESTO NUMERO!

3. Ora spostati in fondo alla pagina e batti CTRL-g again.  
RICORDATI ANCHE QUESTO NUMERO!
4. Per salvare SOLO una parte in un file, batti :#,# w TEST  
dove #,# sono i due numeri che hai memorizzato (cima,fondo) e TEST  
e' il tuo NOMEFILE.
5. Ancora una volta, controlla che il file esista con il comando :!dir  
ma NON CANCELLARLO.

~~~~~

Lezione 5.4: INSERIRE E RIUNIRE FILE

**** Per inserire il contenuto di un file, batti :r NOMEFILE ****

1. Battiti :!dir per accertarti che il tuo NOMEFILE TEST sia ancora presente.
2. Posiziona il cursore all'inizio di questa pagina.

NOTA: DOPO aver eseguito il Passo 3 vedrai ancora la Lezione 5.3.
Quindi spostati IN GIU' per tornare ancora a questa Lezione.

3. Ora inserisci il tuo file TEST con il comando :r TEST dove TEST e'
il nome del file.

NOTA: Il file che tu richiedi e' inserito a partire da dove si trova il cursore.

4. Per verificare che un file e' stato inserito, torna indietro col cursore
e nota che ci sono ora 2 copie della Lezione 5.3, quella originale e quella
da te inserita.

~~~~~

#### Lezione 5 SOMMARIO

1. :!comando esegue un comando esterno.

Alcuni esempi utili sono [in MSDOS]:

```

:!dir      -visualizza lista directory
:!del NOMEFILE      -cancella file NOMEFILE.
```

2. :w NOMEFILE scrive su disco il file che stai editando con nome NOMEFILE.
3. :#,#w NOMEFILE salva le linee da # a # nel file NOMEFILE.

4. `:r NOMEFILE` legge il file NOMEFILE da disco e lo inserisce nel file che stai editando, dopo il punto dove e' posizionato il cursore.

~~~~~

Lezione 6.1: IL COMANDO OPEN [APRI]

**** Batti `o` per aprire una linea sotto il cursore e per passare in Modalita' Inserimento. ****

1. Muovi il cursore fino alla linea qui sotto, indicata da `--->`.
2. Batti `o` (minuscolo) per aprire una linea sotto il cursore e per passare in Modalita' Inserimento.
3. Adesso ricopia la linea indicata da `--->` e premi `<ESC>` per uscire dalla Modalita' Inserimento.

`--->` Dopo battuto `o` il cursore e' sulla linea aperta (in Modalita' Inserimento)

4. Per aprire una linea SOPRA il cursore, batti `O` [Maiuscola], invece che una `o` minuscola. Prova sulla linea subito sotto.
- Apri una linea SOPRA questa battendo `O` quando il cursore e' su questa linea.

~~~~~

#### Lezione 6.2: IL COMANDO APPEND [AGGIUNGI]

**\*\* Batti `a` per inserire testo DOPO il cursore. \*\***

1. Muovi il cursore alla fine della prima linea qui sotto, indicata da `--->` battendo `$` mentre sei in Modalita' Normale.
2. Batti una `a` (minuscola) per aggiungere testo DOPO il carattere sotto il cursore. (A Maiuscola aggiunge alla fine della linea).

NOTA: Eviti cosi' di battere `i`, l'ultimo carattere, il testo da aggiungere, `<ESC>`, spostare il cursore a sinistra e battere `x` solo per aggiungere qualcosa alla fine della linea!

3. Adesso completa la prima linea. Nota anche che l'aggiunta funziona come la Modalita' Inserimento, tranne che per il luogo dove il testo e' inserito.

---> Questa linea ti permettera' di esercitarti  
 ---> Questa linea ti permettera' di esercitarti ad aggiungere testo a fine linea.

~~~~~

Lezione 6.3: UN'ALTRA VARIANTE DI REPLACE [RIMPIAZZA]

**** Batti una R Maiuscola per rimpiazzare piu' di un carattere. ****

1. Muovi il cursore alla prima linea qui sotto, indicata da --->.
 2. Posiziona il cursore all'inizio della prima parola differente dalla seconda linea indicata da ---> (la parola "'ultima").
 3. Adesso batti R e rimpiazza il resto del testo sulla prima linea battendo sopra il testo preesistente per rendere la prima linea uguale alla seconda.
- > Per rendere la prima linea uguale alla ultima su questa pagina usa i tasti.
 ---> Per rendere la prima linea uguale alla seconda, batti R e il nuovo testo.
4. Nota che quando premi <ESC> per uscire, ogni testo non toccato resta uguale.

~~~~~

### Lezione 6.4: SET [IMPOSTA] UN'OPZIONE

**\*\* Imposta un'opzione per ignorare maiuscole/minuscole durante la ricerca/sostituzione \*\***

1. Ricerca 'ignora' battendo:  
 /ignora  
 Ripeti la ricerca piu' volte usando il tasto n
2. Imposta l'opzione 'ic' (Ignore case, [Ignora Maiuscolo/minuscolo]) battendo:  
 :set ic
3. Adesso ricerca ancora 'ignora' premendo il tasto n  
 Ripeti la ricerca piu' volte usando il tasto n
4. Imposta le opzioni 'hlsearch' e 'incsearch' [evidenzia\_ricerca subito]  
 :set hls is
5. Adesso ribatti ancora il comando di ricerca, e guarda cosa succede:  
 /ignore

~~~~~

Lezione 6 SOMMARIO

1. Battendo `o` aggiungi una linea SOTTO il cursore ed il cursore si posiziona sulla linea appena aperta, in Modalita' Inserimento.
Battendo `O` [Maiuscola] apri la linea SOPRA la linea su cui e' il cursore.
2. Batti una `a` per inserire testo DOPO il carattere su cui e' il cursore.
Battendo `A` [Maiuscola] aggiungi testo alla fine della linea.
3. Battendo `R` [Maiuscola] entri in Modalita' Rimpiazzo [Replace mode] e ci resti finche' non premi `<ESC>` per uscirne.
4. Battendo `":set xxx"` imposti l'opzione "xxx"
Battendo `":h xxx"` vedi la documentazione [inglese] per l'opzione "xxx"

~~~~~

Lezione 7: COMANDI DI AIUTO ON-LINE

**\*\* Usa il sistema di help on-line \*\***

Vim ha un esauriente sistema di aiuto on-line. Per accedervi usa questo comando:

- batti `:help <INVIO>` OPPURE `:h <INVIO>`

Batti `:q <INVIO>` per chiudere la finestra di help.

Puoi trovare aiuto su quasi tutto, dando un argomento al comando `":help"`  
Prova questi (non dimenticare di premere `<INVIO>`):

```
:help w
:help c_<T
:help insert-index
:help user-manual
```

~~~~~

LEZIONE 8: PREPARA UNO SCRIPT INIZIALE

**** Attiva le opzioni Vim ****

Vim ha molte piu' opzioni di Vi, ma molte di esse sono predefinite inattive. Per cominciare a usare piu' opzioni, devi preparare un file "vimrc".

1. Comincia a editare il file "vimrc", a seconda del tuo sistema operativo:
`:edit ~/.vimrc` per Unix
`:edit $VIM/_vimrc` per MS-Windows

2. Ora inserisci il file "vimrc" d'esempio:
`:read $VIMRUNTIME/vimrc_example.vim`

3. Scrivi il file con:
`:write`

La prossima volta che apri Vim, vedrai in uso la colorazione sintattica. Puoi aggiungere a questo file "vimrc" tutte le tue impostazioni preferite.

~~~~~  
 Qui finisce la Guida a Vim. Il suo intento e' di fornire una breve panoramica dell'Editor Vim, che ti consenta di usare l'Editor abbastanza facilmente. Questa guida e' largamente incompleta poiche' Vim ha moltissimi altri comandi. Puoi anche leggere il manuale utente: `":help user-manual"`.

Per ulteriore lettura e studio, raccomandiamo:  
 Vim - Vi Improved - di Steve Oualline      Editore: New Riders  
 Il primo libro completamente dedicato a Vim. Utile specie per principianti. Contiene molti esempi e figure.  
 Vedi <http://iccf-holland.org/click5.html>

Quest'altro libro e' piu' su Vi che su Vim, ma e' pure consigliato:  
 Learning the Vi Editor - di Linda Lamb e Arnold Robbins  
 Editore: O'Reilly & Associates Inc.  
 E' un buon libro per imparare quasi tutto ci che puoi voler fare con Vi. Ne esiste una traduzione italiana, basata su una vecchia edizione.

Questa guida e' stata scritta da Michael C. Pierce e Robert K. Ware, Colorado School of Mines, usando idee fornite da Charles Smith, Colorado State University - E-mail: [bware@mines.colorado.edu](mailto:bware@mines.colorado.edu)  
 Modificato per Vim da Bram Moolenaar.  
 Segnalare refusi ad Antonio Colombo - E-mail: [antonio.colombo@jrc.it](mailto:antonio.colombo@jrc.it)  
 ~~~~~


Capitolo 4

Il filesystem

Una delle principali risorse condivise alle quali può accedere un processo POSIX è rappresentata dall'insieme dei file presenti nel disco¹ o **filesystem** (figura 4.1).

Con il termine filesystem si intende sia il *metodo* che l'*implementazione*. Il nostro interesse attuale è quello di capire il *metodo* utilizzato dai sistemi POSIX per organizzare i file e, di conseguenza, parleremo di filesystem in tal senso.

Il secondo significato del termine si utilizza, invece, quando si vuole indicare l'oggetto filesystem presente in un disco o una partizione (esempio: *cercare un file nel filesystem*). Esistono diversi comandi fra le utility di sistema che premettono di creare, modificare o montare vari filesystem; fra questi, ad esempio, troviamo *mke2fs* che permette di creare un filesystem di tipo EXT2 in una partizione: il nome del comando è una contrazione per *make ext2 filesystem*, nella quale appare evidente l'uso del termine filesystem nel senso di *implementazione*.

Come anticipato al precedente 2.7, il filesystem ha una unica radice detta *barra* o *root*², che contiene, in forma gerarchica, altre sotto directory; in prima approssimazione³, possiamo pensare all'intera gerarchia come ad un albero avente per radice la barra.

I nomi di directory e la struttura principale dell'albero sono *discretamente* standardizzate fra i vari sistemi POSIX⁴; esistono, tuttavia, dei sistemi che preferiscono riorganizzarne completamente nomi e struttura.

Nella maggioranza dei casi, tuttavia, dovremmo individuare almeno queste sotto directory della barra:

/bin file eseguibili principali di sistema, normalmente eseguibili dagli utenti

/dev file speciali per i dispositivi (o device)

/etc file di configurazione del sistema e dei programmi

/lib librerie di sistema statiche e dinamiche

/tmp file temporanei

/usr file di uso comune per tutti gli utenti come programmi, documentazione, esempi. Spesso /usr viene *montata* in sola lettura per migliorare la velocità di accesso in lettura e la sicurezza; in tal caso la gerarchia /var viene in aiuto.

¹Un tipico sistema Unix, anche se di piccole dimensioni, utilizza normalmente più dischi che possono eventualmente essere remoti

²L'utilizzo di root come appellativo della radice può creare ambiguità con la home directory dell'utente root in /root

³In Unix/POSIX è possibile, anzi spesso necessario, utilizzare dei link fra directory. Questi oggetti introducono nel grafo degli anelli tali da impedirne la classificazione come albero

⁴Il Linux Filesystem Hierarchy Standard, FHS, ad esempio

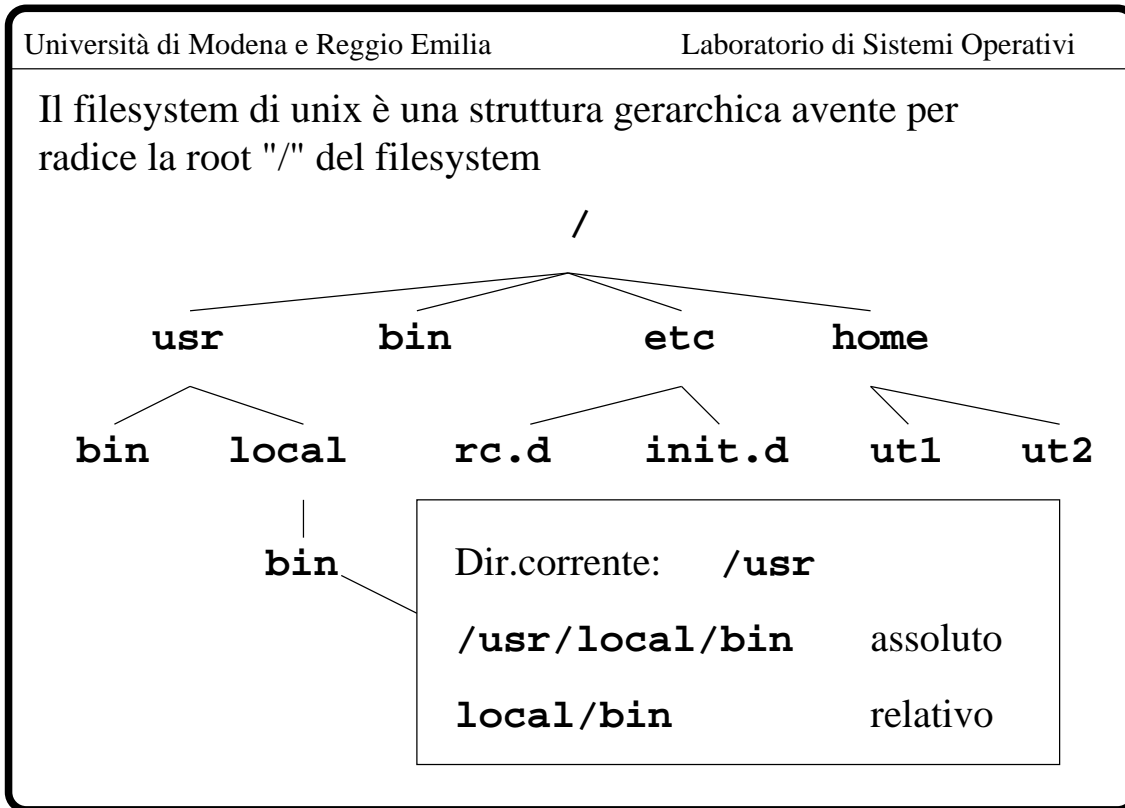


Figura 4.1: Filesystem

`/var` file di uso comune, come `/usr`, ma che richiedono l'accesso in scrittura

4.1 File e Directory

La directory è un file *particolare* o meglio, **non regolare**. In Unix, ogni oggetto che trova posto nel filesystem è un file; parleremo di file **regolari** quando questi sono meri contenitori di byte.

Un file di *tipo directory* contiene un elenco di *nomi* e la relativa associazione con un file del filesystem; questo, a sua volta, può essere di tipo directory dando luogo ad una sotto directory della precedente.

Quando invochiamo il comando `ls`, otteniamo la lista dei nomi di file elencati nella directory corrente. Specificando l'opzione `-l` otterremo la stessa lista di nomi abbinata ad una serie di informazioni:

```
-rwxr-xr-x    1 valealex users      25843 Jan 20 18:28 lso.tex
```

Vediamo in dettaglio a cosa si riferiscono:

- -
tipo di file
- `rwxr-xr-x`
diritti associati al file
- 1
numero di link

- `valealex`
identificativo del proprietario
- `users`
identificativo del gruppo che possiede il file (non necessariamente il gruppo dell'utente proprietario)
- `25843`
dimensione del file
- `Jan 20 18:28`
data dell'ultima modifica
- `lso.tex`
nome associato al file nella directory

Il primo carattere identifica il tipo di file; il segno meno contraddistingue i file regolari mentre il carattere `d` indica una directory, come nella seguente linea prodotta da `ls -l`:

```
drwxr-xr-x    3 valealex users      4096 Jan 14 14:26 old
```

4.2 Permessi associati ai file

Ogni file contiene delle informazioni aggiuntive a quelle del proprio contenuto che servono al sistema per determinare le operazioni ammissibili ad un dato utente. Come anticipato al precedente 2.2, ogni processo è associato ad un utente; quando il processo richiede al sistema un qualunque accesso ad un file, l'utente viene abbinato all'insieme dei permessi per determinare se l'accesso richiesto è ammissibile o meno.

Prendiamo ad esempio l'output di `ls -l` rappresentato in figura 4.2; la sequenza di caratteri che segue il `d` viene letto come una sequenza di informazioni binarie (bit) che viene letta come segue:

| bit | carattere | significato |
|-------|------------|--|
| 00400 | r???????? | Se 'r', accesso in lettura consentito al proprietario |
| 00200 | ?w???????? | Se 'w', accesso in scrittura consentito al proprietario |
| 00100 | ??x??????? | Se 'x', accesso in esecuzione consentito al proprietario |
| 00040 | ???r?????? | Se 'r', accesso in lettura consentito a chiunque appartenga al gruppo |
| 00020 | ???w?????? | Se 'w', accesso in scrittura consentito a chiunque appartenga al gruppo |
| 00010 | ???x?????? | Se 'x', accesso in esecuzione consentito a chiunque appartenga al gruppo |
| 00004 | ?????r??? | Se 'r', accesso in lettura consentito a tutti i rimanenti utenti |
| 00002 | ?????w??? | Se 'w', accesso in scrittura consentito a tutti i rimanenti utenti |
| 00001 | ?????x??? | Se 'x', accesso in esecuzione consentito a tutti i rimanenti utenti |

I bit relativi all'accesso in esecuzione hanno un significato differente se applicati a file regolari o a file di tipo directory: nel primo caso controllano effettivamente la possibilità di *mettere in esecuzione* il file, mentre per le directory consentono o meno l'*esplorazione*.

In altre parole, non è possibile impostare la directory corrente di un processo su una directory alla quale non si abbia accesso in esecuzione; questo non impedisce di vederne il contenuto (controllato dall'accesso in lettura) ma dei file contenuti si potrà conoscere solo il nome.

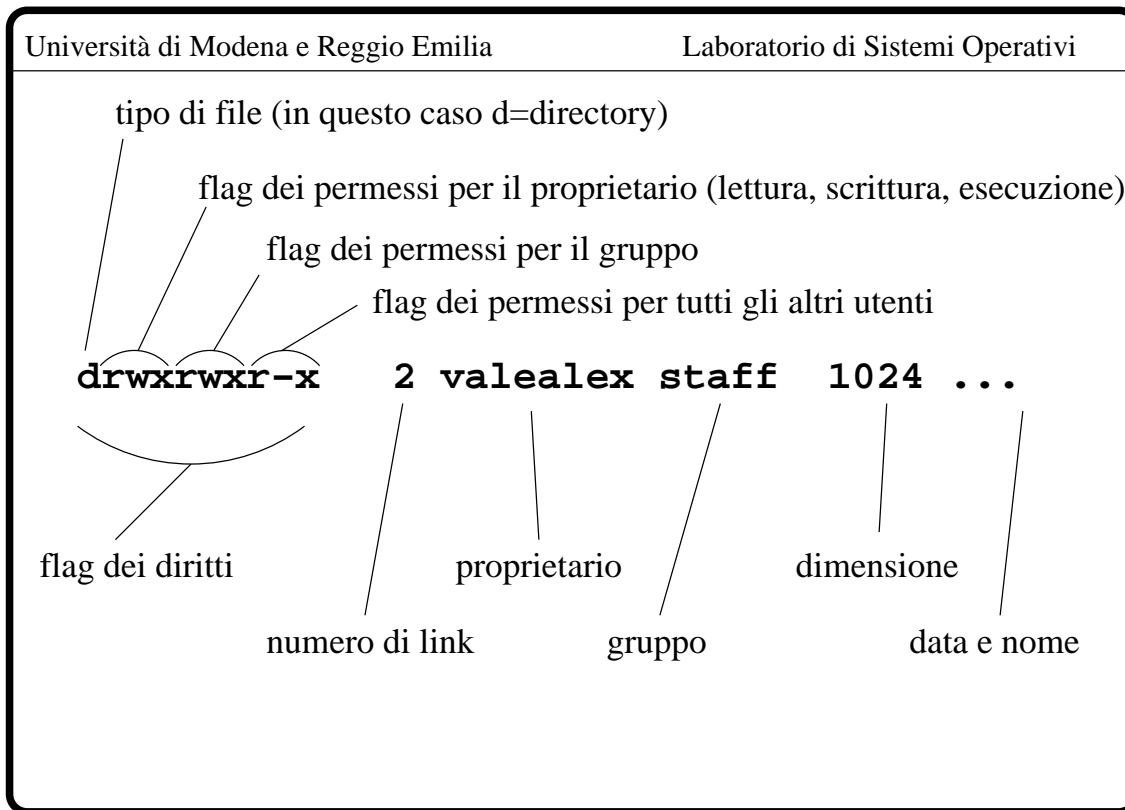


Figura 4.2: Proprietà dei file

4.3 Bit speciali

In aggiunta ai 9 bit relativi ai permessi di lettura, scrittura ed esecuzione, esistono tre bit speciali; come per i bit di esecuzione, anche i tre bit speciali cambiano lievemente significato quando applicati a file regolari o a directory.

| bit | carattere | significato |
|-------|------------|---|
| 04000 | ??s?????? | Se 's', attiva il flag suid o <i>set user id</i> |
| 02000 | ?????s??? | Se 's', attiva il flag sgid o <i>set group id</i> |
| 01000 | ??i??????t | Se 't', attiva lo sticky bit o <i>restriction deletion flag</i> |

La posizione ed il significato dei tre bit speciali è indicato nella precedente tabella. Per comprendere meglio il reale funzionamento di questi bit, dobbiamo riprendere il concetto di processo anticipato al 2.2:

- ogni processo è associato ad un utente e ad un gruppo
- ogni processo esegue un programma
- il programma è contenuto in un file eseguibile

Quando si esegue un comando, in altre parole, si legge un file eseguibile e lo si *copia* nella memoria del processo. Durante questa fase vengono esaminati i bit speciali del file eseguibile e si eseguono le eventuali *promozioni* espresse dal **suid** o dal **sgid**.

In particolare, l'esecuzione di un file eseguibile con bit suid settato provoca la modifica dell'id utente associato al processo rendendolo uguale a quello del proprietario del file; per il bit sgid avviene lo stesso ma relativamente all'identificativo del gruppo che viene posto uguale a quello del gruppo che possiede il file.

Lo sticky bit, associato ad un file eseguibile, ha più una valenza storica che pratica: esso era utilizzato da alcune versioni di Unix per impedire lo swap della memoria del processo su disco quando il gestore di memoria lo avesse ritenuto opportuno. Il deposito della memoria su disco poteva rappresentare un problema di sicurezza per alcuni eseguibili per cui era necessario impedirlo almeno per certi file critici.

I moderni Unix utilizzano delle strategie di gestione della memoria e dello swap già di per se' idonei alla protezione delle informazioni presenti nei processi soggetti a swap-out per cui, tipicamente, questo bit viene ignorato o utilizzato per scopi specifici.

Per le directory, il significato dei bit speciali *suid* e *sgid* e *sticky* è diverso. Il bit *suid* è normalmente ignorato.

Il bit *sgid* è invece utile per marcare delle directory che si vogliono mantenere accessibili ad un gruppo. Normalmente, infatti, quando un utente crea un file vi associa automaticamente il proprio user id e l'id del gruppo principale⁵ cui appartiene. Se crea un file o una directory all'interno di una directory con *sgid* attivo, invece, il gruppo del file creato sarà uguale a quello della directory superiore indipendentemente da quello dell'utente.

Il bit *sticky*, che applicato su una directory assume l'appellativo di *restriction deletion flag*, impedisce a tutti gli utenti di rimuovere o rinominare file presenti nella directory stessa a meno che non ne siano proprietari (o che l'utente non sia proprietario della directory).

Normalmente il bit *sticky* è presente in directory comuni come */tmp* per assicurare ad ogni utente l'integrità dei propri file depositati pur assicurando a tutti l'accesso in lettura e scrittura.

4.4 Altre informazioni sui file

Un file di un sistema POSIX, quindi, è più del suo semplice contenuto in quanto possiede un insieme di informazioni accessorie che comunemente vengono indicate con il termine **attributi**. Il comando `ls -l`, dal quale abbiamo iniziato la nostra analisi permette di visualizzarne alcune.

Oltre ai *permessi*, indicati anche come **modo di accesso** o **access mode**, un file possiede alcuni attributi temporali:

`atime` istante dell'ultimo accesso

`mtime` istante dell'ultima modifica del contenuto

`ctime` istante dell'ultima modifica all'i-node, ovvero agli attributi

Nell'output del comando `ls -l` viene riportato uno dei timestamp del file, che tipicamente è `mtime`.

⁵ Ogni utente può appartenere a più gruppi: il gruppo indicato nel file `/etc/passwd`

Capitolo 5

Usiamo la shell

Nelle pagine precedenti abbiamo iniziato a prendere un po' di confidenza con l'interfaccia comandi di un sistema unix senza, tuttavia, entrare troppo nel dettaglio di ogni singolo concetto.

Ora dobbiamo approfondire la nostra conoscenza di tali concetti al punto di poterli utilizzare agevolmente nella progettazione e realizzazione degli shell script; iniziamo con uno degli ultimi: la **ridirezione**

5.1 La ridirezione

Nel precedente esempio d'uso del comando *cat* abbiamo visto come sia stato agevole richiedere alla shell di *incanalare* i caratteri prodotti in un file. Questa operazione è nota come ridirezione.

Abbiamo già visto, inoltre, come un processo sia abbinato ad una serie di informazioni di *stato*, fra cui la directory corrente o l'identificativo di utente e gruppo.

Il *canale* nel quale depositare i caratteri prodotti è, ad esempio, una delle informazioni associate ad ogni processo; utilizzando una terminologia più idonea, chiameremo tale canale **standard output** o **stdout**.

Analogamente allo standard output, ogni processo è associato ad un canale di ingresso detto **standard input** o **stdin**. Senza entrare, per ora, nello specifico meccanismo di gestione dei processi, possiamo affermare che i canali sono *ereditari*, ovvero vengono passati da ogni processo agli eventuali processi discendenti.

La shell che utilizziamo normalmente è essa stessa un processo e, di conseguenza, presenta un proprio standard input ed un proprio standard output. È facile intuire come lo standard input sia in *qualche modo* riconducibile alla tastiera e lo standard output al video¹

Introduciamo ora un concetto nuovo, relativo al modo in cui un processo come quello *della* shell mette in esecuzione un **comando esterno**²: ecco cosa avviene:

- Il processo-shell crea un nuovo processo (figlio)
- Il processo-shell attende la conclusione del processo figlio
- Il processo-figlio *esegue* il file eseguibile corrispondente al comando esterno invocato
- Il processo-figlio termina
- Il processo-shell torna *attivo* e presenta il prompt

¹Con una nomenclatura un po' arcaica, tali dispositivi sono indicati come *telescriventi* o *teletype*. Le corrispondenti entry in */dev*, infatti, sono file i cui nomi iniziano o contengono i caratteri **tt**

²I comandi interni, ovvero quelli eseguiti dalla shell senza l'intervento di altri file, sono interpretati e codificati all'interno del *programma shell*

Come conseguenza a quanto appena visto, se ne deduce che ogni comando messo in esecuzione dalla shell ha standard output ed input *in comune* con la shell stessa. Un processo può, tuttavia, modificare sia il proprio standard input che il proprio standard output dirigendoli verso *file*³ differenti.

In particolare, è possibile istruire la shell indicando quali canali alterare ed in quale modo: si parla, in tal caso, di ridirezione.

La ridirezione dello standard output è già stata vista nel paragrafo ?? aggiungendo il carattere di *maggiore* fra il comando ed un nome di file:

```
cat >unoscript
```

Gli elementi presenti nella command line non sono ne' argomenti ne' opzioni del comando *cat* ma istruzioni dirette alla shell. Questo significa che **non sono necessari** spazi per separare il comando dal simbolo di ridirezione o questo dal nome di file.

La ridirezione in output appena vista distrugge l'eventuale precedente contenuto del file *unoscript*. È possibile richiedere alla shell di eseguire una ridirezione in **append**, semplicemente sostituendo il simbolo di maggiore come nella seguente linea:

```
cat >>unoscript
```

Il comando *cat*, come abbiamo visto, è in grado di *aprire* un file in input nel caso in cui venga invocato con un argomento:

```
cat README
```

In questo caso, è il comando stesso ad eseguire l'apertura del file indicato ed a visualizzarne il contenuto su stdout. Lo stesso comportamento si sarebbe potuto ottenere indicando alla shell di utilizzare *quel* file come stdin:

```
cat<README
```

Le due differenti invocazioni del comando *cat* sul file README sono funzionalmente indistinguibili ma profondamente distinte a livello di sistema. La prima utilizza una particolare funzionalità messa a disposizione dal comando *cat*, ovvero quella di leggere dati da un file diverso dal proprio stdin. La seconda, al contrario, non dipende dal particolare comando invocato ma utilizza lo strumento della ridirezione messo a disposizione dalla shell.

Le operazioni di ridirezione possono essere più di una così, ad esempio, possiamo utilizzare il comando *cat* per copiare il contenuto di un file su un altro file:

```
cat<README>copiadiREADME
```

Spaziatura degli elementi di ridirezione

I simboli di ridirezione possono essere o meno separati dai nomi di file con degli spazi; in altre parole, le seguenti linee di comando sono equivalenti:

```
cat<README>copiadiREADME
cat < README > copiadiREADME
cat <README >copiadiREADME
cat < README >copiadiREADME
cat <README > copiadiREADME
```

La seguente linea di comando, al contrario, non è lecita in quanto la shell richiede che il comando sia indicato come primo elemento della catena di ridirezione:

```
README > cat > copiadiREADME # ERRORE
```

³Con questa affermazione abbiamo ulteriormente esteso il concetto di file

5.1.1 Standard Error

Il solo stdout potrebbe non essere sufficiente a contenere tutte le informazioni prodotte da un comando; potrebbe, ad esempio, essere conveniente disporre di un canale di uscita dedicato ai messaggi diagnostici o di errore. La maggioranza dei comandi disponibili, infatti, utilizza un terzo *descrittore* di **standard error** o **stderr**.

La *nostra* shell riversa sia lo standard output che lo standard error a video. Questi canali sono tuttavia distinti e la semplice ridirezione in output vista prima devia solamente i caratteri emessi su stdout. Facciamo un esempio:

Supponiamo di *trovarci* in una directory contenente un file di nome abc: l'esecuzione del comando `ls abc`, in tale directory, lista ovviamente il file. Se utilizziamo la ridirezione in output sul file def, l'esecuzione del comando non produce nulla a video perchè tutto l'output viene trascritto in def.

```
ls abc>def
```

Ora supponiamo di eliminare il file abc e di ripetere il comando `ls abc>def`:

```
rm abc
ls abc>def
```

Il comando `ls` non può chiaramente listare il file abc e, di conseguenza, il file def che contiene l'output risulta vuoto. A video, tuttavia, appare un messaggio di errore generato dal comando `ls`, simile al seguente:

```
ls: abc: No such file or directory
```

Il descrittore stderr non è infatti stato ridiretto e, di conseguenza, rimane *collegato* al video. Proviamo a riformulare il comando nel seguente modo⁴:

```
ls abc 2>ghi
```

5.1.2 Ridirezioni fra descrittori

Finora abbiamo visto come ridirigere un descrittore su un file. In certe situazioni potrebbe essere conveniente utilizzare un altro descrittore come destinazione di una ridirezione.

Se, ad esempio, si volesse ridirigere sia stderr che stdout su un unico file, la soluzione di eseguire due ridirezioni separate sarebbe da sconsigliare⁵. Al contrario viene messa a disposizione una forma particolare di ridirezione che utilizza il medesimo descrittore per due canali:

```
ls abc >abc 2>&1
```

Questa ridirezione commuta stdout sul file abc poi commuta stderr sull'*attuale* stdout; il risultato è che sia stderr che stdout vengono connessi a abc. Una eventuale inversione fra le due ridirezioni produrrebbe un risultato differente:

```
ls abc 2>&1 >abc
```

In questo caso si commuterebbe stderr su stdout *attuale*, ovvero il video, e stdout su abc. Il risultato sarebbe quello di mantenere i messaggi d'errore a video e solo l'output su abc.

⁴ora lo spazio fra l'ultimo carattere relativo all'invocazione del comando ed il simbolo di ridirezione `2>` è necessario altrimenti la shell avrebbe potuto interpretare il comando come `ls abc2 > ...`

⁵Le scritture sarebbero operate in modo indipendente con una conseguente possibile sovrapposizione delle parti

5.2 Pipeline

La possibilità di commutare i canali di input ed output da e su file rende possibile combinare due comandi elementari per dare vita a command line più complesse; supponiamo, ad esempio, di avere a disposizione un comando che visualizzi un po' di informazioni sui processi attualmente attivi.

Questo comando esiste e si chiama **ps**. Esso ammette un numero considerevole di opzioni ma non permette di specificare il nome di un eseguibile come criterio di ricerca. Questo potrebbe sembrare un problema ed indurre qualche profano a riscrivere⁶ **ps** per includere tale criterio di selezione.

Un sistemista più accorto, al contrario, farebbe semplicemente passare l'output di **ps**, eventualmente invocato con qualche opzione, in un filtro come *grep*:

```
$ ps -A > tempfile
$ grep bash < tempfile
$ rm tempfile
```

Questa semplice sequenza di comandi visualizza le informazioni relative a tutti i processi che stanno eseguendo una copia della shell **bash**. Potremmo scriverla in uno script ed utilizzarla tutte le volte che vogliamo.

La possibilità di assemblare più comandi fra loro è una dei punti di forza dei sistemi POSIX in quanto ha permesso ai sistemisti di concentrarsi a produrre software semplice e stabile lasciando alla versatilità della shell il compito di esplicitare ogni anche più contorto obiettivo.

Tuttavia, il dover utilizzare un file di appoggio per ogni *passaggio*, è quantomeno scomodo; al suo posto è stato introdotto un meccanismo funzionalmente equivalente e migliorativo in termini di sfruttamento della macchina: la pipe

```
$ ps -A | grep bash
```

Il vero vantaggio della pipe sarà chiaro quando entreremo nel dettaglio dei processi ma, allo stato attuale, ci basta notare come questo meccanismo renda possibile la realizzazione di catene o **pipeline** complesse. Anche il più visionario dei programmatori avrebbe visto con difficoltà questo possibile uso di **ps**:

```
$ ps -A | cut -c24-,1-5 | sed 's/^ *//' | sed 's/^\([0-9]*\) \(\ \)\(.*\)$/\3 \1/'
| sort | xargs printf '%40s %d\n'
```

5.3 Exit value di un comando/processo

Ogni comando esterno viene messo in esecuzione dalla shell in un nuovo processo; all'atto della propria terminazione, il comando ha la possibilità di impostare una informazione di stato del processo che lo esegue detta *exit value*.

L'*exit value* è un numero intero compreso fra 0 e 255 ed è reso disponibile al processo che ha messo in esecuzione il comando, ovvero tipicamente alla shell. È possibile ottenere l'*exit value* dell'ultimo comando eseguito chiedendo alla shell l'espansione della variabile `$?`:

```
$ ls
$ echo $?
```

Un *exit value* uguale a zero indica, normalmente, l'esito positivo del comando stesso:

```
$ ls file_che_non_esiste
$ echo $?
```

⁶o, visto che molto probabilmente si tratta di software Open Source, a modificarne i sorgenti

5.4 Command substitution

Oltre all'exit value, un comando può produrre una sequenza di caratteri su standard output; abbiamo visto come ridirigere su un file o su una pipeline questo stream ma la shell ci consente anche di *catturarlo* ed utilizzarlo come se fosse un elemento della command line.

Questo strumento, noto come **command substitution**, viene attivato racchiudendo un comando fra due *apici inversi* 'comando'. L'elemento di command line racchiuso fra apici inversi⁷ viene trattato come se fosse una command line a se' mentre al suo posto nella command line originale viene riportato lo stream prodotto in stdout. Prendiamo come esempio il comando *hostname* che restituisce il nome dell'host:

```
$ hostname
copperbottom
$ echo hostname
hostname
$ echo `hostname`
copperbottom
```

Nella prima command line, il comando *hostname* viene eseguito ed il suo stdout viene riportato a video; nella seconda command line, il comando da eseguire è *echo* che riversa su stdout i suoi argomenti trattandoli alla stregua di comuni stringhe anche se, come nel nostro caso, si tratta di un nome di un comando.

Quello che vogliamo notare, invece, è la terza command line: in questo caso la shell trova un elemento in command substitution per cui crea un processo per l'esecuzione del comando "hostname", ne attende il completamento catturandone l'output, quindi mette in esecuzione la command line originale sostituendo ai caratteri compresi fra gli apici inversi lo standard output del processo intermedio.

5.5 Gli shell script

Saper produrre delle command line potenti è una delle capacità fondamentali del sistemista unix. Tuttavia, le sue *creazioni* avrebbero la vita di una sessione di lavoro e dovrebbero essere ricordate o trascritte ogni volta.

Fortunatamente, la shell ci consente di mettere in esecuzione comandi o sequenze di comandi contenuti in un comune file di testo che, in tal caso, assume il ruolo di script. Il linguaggio della shell rappresenta, a tutti gli effetti, un linguaggio di programmazione, con strutture di controllo e commenti.

La nostra esplorazione del linguaggio di scripting inizia, appunto, dai commenti: La shell ignora ogni linea che inizia con il carattere *cancelletto*; questo ci permette di inserire tutte le linee di commento che desideriamo semplicemente facendole iniziare con questo carattere.

```
# Questo e' un commento
```

Come abbiamo visto in precedenza, in un sistema POSIX è sufficiente settare il bit di eseguibilità ad un file per renderlo eseguibile. Non esiste, pertanto, alcun legame fra nome o estensione di un file ed il suo *tipo*.

I file eseguibili vengono tuttavia *analizzati* per determinarne il tipo, specialmente per distinguere se si tratta di file in linguaggio macchina o di file da interpretare. Questo viene di norma fatto esaminando i primi caratteri del file eseguibile. Un file da interpretare si identifica mediante la sequenza di caratteri

⁷ alcuni preferiscono indicare questi simboli con il nome di *accento grave*; in ogni caso si tratta del carattere ASCII 96 decimale ottenibile normalmente sulle tastiere italiane premendo ALT e apostrofo

`#!`

seguita dal nome assoluto dell'interprete richiesto. Per un file scritto nel linguaggio della shell **bash**, la prima linea deve essere come la seguente:

```
#!/bin/bash
```

La shell, tuttavia, tenta di interpretare ogni file che non dichiara il proprio tipo mediante un interprete di default che, normalmente, coincide con la shell stessa. Per questo motivo siamo riusciti ad eseguire il primo semplice script senza lo **sharp bang**

5.5.1 Eseguire uno script

L'esecuzione di uno script si ottiene, ovviamente, tramite una command line; è possibile chiedere alla shell di eseguire uno script *direttamente*, ovvero come se il contenuto dello script stesso fosse digitato dall'utente. Per questo si utilizza il comando **source** della shell nel seguente modo:

```
$ source file.sh
```

L'esecuzione dello script, tuttavia, potrebbe dimostrarsi troppo invasiva per la nostra shell in quanto ogni command line contenuta nello script sarebbe in grado di modificarne lo stato (directory corrente, variabili d'ambiente); anche se in alcune occasioni ricorrere questa modalità è inevitabile, preferiamo normalmente eseguire ogni script in un processo *figlio* della nostra shell, così che ogni operazione modifichi solo lo stato della sottoshell.

Il file contenente lo script, di conseguenza, deve essere impostato come eseguibile (tramite `chmod`) e messo in esecuzione creando una command line come la seguente:

```
$ ./file.sh
```

Il punto e la barra all'inizio della command line hanno una funzione specifica: indicare senza ambiguità alla shell che l'elemento `./file.sh` è un nome di file e non un nome di comando.

La shell, infatti, utilizza un particolare metodo di ricerca per mettere in esecuzione i comandi digitati sulla command line basato su una lista di directory denominata `PATH`. Forzando la shell a trattare l'elemento della command line come se fosse un nome di file, si evita la ricerca nel `PATH` del comando.

In caso contrario, e cioè digitando la seguente command line, la shell avvierebbe la ricerca di un file avente nome uguale a quello del comando nelle directory elencate dal `PATH` e, non trovandolo, emetterebbe un messaggio di errore:

```
$ file.sh
bash: file.sh: command not found
```

La lista delle directory nelle quali eseguire la ricerca viene letta da una variabile di ambiente di nome `PATH`, nella quale le varie directory sono separate dal carattere `:`. Possiamo modificare il valore di questa variabile aggiungendo la directory corrente al `PATH`:

```
PATH=.:$PATH
```

5.5.2 Esportazione di variabili

Quando la shell esegue un comando, ad esempio una sottoshell per mettere in esecuzione il nostro script, solo alcune delle variabili di ambiente vengono *trasmesse* al nuovo processo. L'azione che consente di marcare una variabile per la trasmissione ai discendenti è detta **esportazione**.

```
$ var=uno
$ echo $var
uno
```

Abbiamo creato \$var senza esportarla; avviamo un processo figlio, ad esempio una shell:

```
$ sh
$ echo $var

$ exit
exit
```

se nel processo figlio, proviamo a chiedere il valore di \$var, scopriamo che non esiste alcuna variabile con questo nome: la shell, in tal caso, la espande come stringa vuota.

Usciamo dalla sottoshell e, dalla shell originale, eseguiamo l'esportazione:

```
$ export var
```

Ora avviamo una nuova sottoshell e verifichiamo che la variabile \$var continua ad esistere:

```
$ sh
$ echo $var
uno
$ exit
exit
```

L'operazione di esportazione può essere contestuale a quella di creazione:

```
$ export var=uno
```

Non è in **alcun modo possibile**, invece, trasmettere una modifica ad una qualsiasi variabile fatta da una sottoshell alla shell *genitore*, cosiccome è impossibile trasmettere una variabile fra processi distinti.

5.5.3 Uso e definizione di variabili negli shell script

Uno shell script può utilizzare delle variabili di ambiente esportate dal processo *padre* o crearne delle proprie. Ovviamente, se lo script viene eseguito da un processo figlio, le variabili create o modificate all'interno dello shell script svaniranno all'uscita⁸.

```
#!/bin/bash
messaggio="Ciao_Mondo!"
echo $messaggio
```

La variabile "messaggio" assume il valore "Ciao Mondo!" all'interno della sottoshell che esegue lo script, indipendentemente dal fatto che una variabile con tale nome esistesse nella shell di login. Al termine dell'esecuzione dello script, inoltre, la shell di login non avrà subito alcuna modifica alle proprie variabili tanto che, se fosse esistita una variabile di nome messaggio prima dell'esecuzione dello script, allora tale variabile avrebbe mantenuto il suo valore precedente.

5.5.4 Le variabili di shell

La shell tratta in modo particolare gli elementi della linea di comando che iniziano con il carattere dollaro. In questo caso, infatti, la shell procede con la sostituzione dell'elemento con il valore della variabile d'ambiente avente il nome specificato. Esistono, tuttavia, alcune sequenze di caratteri che iniziano con il dollaro ma che non corrispondono a delle variabili d'ambiente: si tratta delle variabili di shell.

Una di tali variabili è stata introdotta quando abbiamo parlato dell'exit value. La sequenza \$?, infatti, viene sostituita dalla shell con il valore numerico restituito dall'ultimo processo figlio.

⁸invocando uno script utilizzando il comando source, al contrario, si richiede l'esecuzione dello script alla shell corrente e, di conseguenza, ogni modifica all'ambiente sarà mantenuta dopo il completamento dello script

5.5.5 Gli argomenti della linea di comando

La shell mette a disposizione alcune variabili di shell per consentire ad uno script l'accesso alla propria command line.

Ogni script, infatti, viene invocato mediante una command line (come ./script.sh) nella quale, eventualmente, possono essere inseriti degli argomenti. Il numero di argomenti presenti nella command line di invocazione è accessibile con la sequenza **\$#**.

Con l'ausilio dell'esempio seguente (args.sh), vediamo come sia possibile accedere ai singoli valori degli argomenti con **\$1**, **\$2**, ecc. o all'intera lista degli argomenti con **\$***. **\$0**, invece, consente di ottenere l'espansione dell'argomento di indice 0, ovvero il *nome* del comando stesso.

```
#!/bin/bash
# file: args.sh
echo Ho rilevato $# argomenti sulla command line
echo Quello con indice 0 vale $0
echo Quello con indice 1 vale $1
echo Quello con indice 2 vale $2
echo Quello con indice 3 vale $3
echo La variabile '$*' vale $*
```

Lo script sopra riportato contiene anche, sottoforma di commenti, lo standard output di una possibile invocazione con tre argomenti. La presenza della sequenza **\$*** nell'output (in modo non espanso, ovviamente) è stata ottenuta mediante **quoting** con apici singoli.

Capitolo 6

Sintassi degli script

Il contenuto di uno shell script, come abbiamo visto, viene interpretato dalla shell così come se fosse digitato interattivamente dalla tastiera. Per questo motivo, ogni comando di uno shell script deve essere una command line e sottostare alle regole di correttezza sintattica proprie della shell.

Esiste la possibilità di codificare una command line in più *linee*, ad esempio per ragioni di leggibilità o, ancora, per sottostare alla sintassi di alcuni comandi shell che richiedano un terminatore di linea fra gli elementi della command line stessa.

Rientrano in questa categoria di comandi, ad esempio, molti degli strumenti utilizzati negli shell script: proviamo ad invocare la seguente command line:

```
$ cat - <<FINE
```

Questa command line invoca, ovviamente, il comando `cat` passando come argomento la stringa `"-"`. Segue un doppio simbolo *minore* che ricorda la ridirezione ma che, in realtà, provoca il collegamento dello standard input del processo che esegue `cat` con lo standard input della shell stessa (quindi, in definitiva, la tastiera).

La sequenza di caratteri che segue il doppio minore rappresenta la *chiave* che marca la fine di questo comportamento. La shell, in tale modalità interattiva, visualizza un prompt diverso da quello standard, detto PS2¹

Ecco una possibile sequenza di input da fornire in seguito al comando precedente:

```
> Esempio di testo
> fino alla lettura della chiava
> FINE
Esempio di testo
fino alla lettura della chiava
$
```

La funzionalità della shell che abbiamo appena visto si chiama *here document*.

Molti dei comandi multilinea che introdurremo servono ad aggiungere agli shell script un *controllo di flusso*, permettendo la codifica di cicli, elaborazioni condizionali e `switch/case`.

6.1 Elaborazione condizionale

Abbiamo in precedenza introdotto il concetto di *exit value* di un processo ed abbiamo visto come la shell ci consenta di visualizzarlo tramite la variabile di shell `$?`.

¹PS2 è una variabile di ambiente; è possibile visualizzarla e modificarla...

È possibile condizionare l'esecuzione di una o più command line a risultato dell'ultimo comando eseguito dalla shell. Verifichiamo il comportamento del seguente shell script che fa uso dei comandi **true** e **false**:

```
#!/bin/bash
if true
then
    echo true restituisce vero
else
    echo true restituisce falso
fi
if false
then
    echo false restituisce vero

    echo false restituisce falso
fi
```

Il costrutto di programmazione condizionale mostrato è l'**if**. La sintassi prevede che l'**if** venga seguito dal comando da verificare e dalla keyword **then** nella linea successiva. Nel caso in cui non sia presente la keyword **else** le command line fra **then** e **fi** vengono eseguite solo se la condizione è vera. Nel caso in cui sia presente la keyword **else** le command line fra **then** e **else** vengono eseguite solo se la condizione è vera mentre quelle fra **else** e **fi** solo se la condizione è falsa.

L'**if**, di conseguenza, rappresenta un comando multilinea nel quale i fine linea assumono un ruolo sintattico (if true deve essere separato da then da un ritorno a capo, in caso contrario la shell segnalerebbe un errore). Negli shell script possiamo utilizzare il vero ritorno a capo, come indicato nell'esempio. È possibile digitare command line multilinea anche dal prompt: in tal caso possiamo usare il carattere punto e virgola al posto del ritorno a capo:

```
if true; then echo ok; fi
```

6.1.1 Il comando test

Invocare un comando che restituisce, come exit value, un valore costantemente falso o vero non è sicuramente di molta utilità. Conviene, piuttosto, disporre di uno strumento in grado di eseguire alcune semplici verifiche relative a stringhe, numeri o informazioni sul filesystem. Questo strumento è rappresentato dal comando **test** del quale troviamo alcuni esempi di invocazione nel seguente script:

```
#!/bin/bash
test 1 = 1
echo $?
test 1 = 2
echo $?
test 1 -eq 1
echo $?
test 1 -eq 2
echo $?
test uno = uno
echo $?
test uno = due
echo $?
```

```
# Il test seguente provoca un errore di sintassi
test uno -eq uno
echo $?
# Il test seguente provoca un errore di sintassi
test uno -eq due
echo $?
```

Il comando `test` può essere invocato, in alternativa, utilizzando la parentesi quadra aperta; in tal caso è convenzione terminare la command line del test stesso con una parentesi quadra chiusa:

```
#!/bin/bash
if test $# -ge 3
then
    echo Ho rilevato un numero di parametri idoneo
else
    echo Parametri insufficienti
fi
if [ $# -ge 3 -a $# -le 5 ]
then
    echo Ho rilevato un numero di parametri idoneo
else
    echo Numero di parametri non idoneo
fi
```

Il comando `test` accetta svariate opzioni, fornendo all'utente un sistema pressochè completo di valutazione di espressioni logiche e combinazioni booleane di essi. La pagina di man relativa al comando `test` rappresenta una importante fonte di informazioni sul suo uso.

Esistono altri costrutti della shell che utilizzano gli exit value; il **while**, ad esempio, permette di ripetere le command line fra **do** e **done** fintanto che l'exit value del comando che segue la keyword **while** è vera.

```
#!/bin/bash
echo Numero parametri = $#
echo Parametri: $*
while [ $# -gt 2 ]
do
    echo eseguo shift
    shift
    echo I parametri sono: $*
done
echo Ora sono solo 2: $*
```

Ecco un esempio di invocazione:

```
$ ./prova_while.sh a b c d e f
Numero parametri = 6
Parametri: a b c d e f
eseguo shift
I parametri sono: b c d e f
eseguo shift
I parametri sono: c d e f
eseguo shift
I parametri sono: d e f
```

```

eseguo shift
I parametri sono: e f
Ora sono solo 2: e f

```

Il comando **shift** della shell viene mostrato per la prima volta in questo esempio ma la sua utilità dovrebbe risultare piuttosto ovvia: in seguito all'esecuzione di un comando **shift**, la lista degli argomenti viene privata del primo elemento **e**, di conseguenza, viene diminuito di uno il valore di **\$#**.

6.2 Forzare un exit value

Il processo che esegue lo script, esattamente come ogni altro processo, restituisce all'atto della propria terminazione un exit value al processo padre. Nel caso si voglia restituire un particolare exit value, bisogna utilizzare il comando **exit** che provoca la terminazione immediata del processo ed assegna all'exit value il valore numerico rappresentato dal proprio argomento:

```

#!/bin/bash
if [ $# -lt 2 ]
then
    echo "Numero di argomenti non idoneo"
    exit 0
fi

```

6.2.1 Il ciclo for

Il **for** permette di realizzare un ciclo di ripetizione di alcune command line con un numero di iterazioni noto a priori e con una particolare sostituzione di variabile. Il concetto sarà più evidente con un esempio:

```

#!/bin/bash
for var in uno due tre
do
    echo $var
done

```

La variabile **var** assumerà, per ogni iterazione del ciclo, il valore di una delle stringhe che seguono la keyword **in**. Ovviamente, nessuno vieta di utilizzare al posto della lista di argomenti precedente una lista prodotta per noi dalla shell, ad esempio mediante espansione di un wild card:

```

#!/bin/bash
for entry in *
do
    if [ -f $entry ]
    then
        echo $entry file regolare
    fi
    if [ -d $entry ]
    then
        echo $entry directory
    fi
done

```

All'interno del ciclo `for` possono trovare spazio altri costrutti della shell, come nell'esempio, due *if*. La complessità che si può ottenere da un shell script è limitata, in pratica, solo dalla inventiva del programmatore: possiamo utilizzare gli strumenti per il controllo di flusso (`if`, `for`, `while`, ...) unitamente a command line che utilizzino la collaborazione della shell per accedere alle variabili, alla ridirezione ed alla command substitution.

Supponiamo, ad esempio, di voler contare il numero di file regolari all'interno della directory corrente:

```
#!/bin/bash
conta=0
for entry in *
do
    if [ -f $entry ]
    then
        echo $entry file regolare
        conta='expr $conta + 1'
    fi
    if [ -d $entry ]
    then
        echo $entry directory
    fi
done
exit $conta
```

La variabile `conta` viene inizializzata al valore 0 e, ad ogni occorrenza di un file regolare, viene modificata assumendo il valore prodotto dall'esecuzione del comando `expr`. Al termine dello script, la variabile `conta` viene copiata nell'exit value del processo e, di conseguenza, sarà resa disponibile al processo padre tramite la variabile `$?`. Si consideri che la coppia di apici inversi per la command substitution può essere sostituita, nella bash, dal seguente formalismo:

```
conta=$(expr $conta + 1)
```

Questo consente una maggiore leggibilità nel caso di command execution nidificate.

6.3 Il case per le selezioni multiple

Al pari di altri linguaggi di programmazione, la shell prevede un costrutto per mettere selettivamente in esecuzione una porzione di script in funzione di un caso fra più possibilità: il `case`. Ricordando che le variabili di shell sono sempre e comunque stringhe, i casi elencati in un costrutto `case` saranno sempre relativi alla rappresentazione stringa della variabile, indipendentemente dal fatto che la variabile contenga una stringa rappresentante un numero.

Per la sintassi, dovremo ricordare che ogni caso dovrà essere diverso dagli altri, terminato da un carattere di *chiusa parentesi tonda* e la lista di comandi associata dovrà essere terminata da un doppio punto e virgola.

```
#!/bin/bash
case $var in
    caso1)
        lista comandi
        ;;
    caso2)
        lista comandi
        ;;
```

```

        caso3)
            lista comandi
            ;;
esac

```

All'interno di un caso, i caratteri speciali che assumono il ruolo di wild card per la shell vengono trattati diversamente. Non si tratta, in questo caso, di una espansione in nomi del filesystem ma in caratteri speciali di una *espressione regolare*². Ecco uno script da provare:

```

#!/bin/bash
echo Argomento vale: $1
case $1 in
    abc) echo condizione abc;;
    [abc]) echo "condizione_[abc]" ;;
    ?) echo "condizione_?" ;;
    [abc]*) echo "condizione_[abc]*" ;;
    [^xyz]*) echo "condizione_[^xyz]*" ;;
    *) echo altre condizioni;;
esac

```

Il case procede all'esecuzione della lista associata al primo caso che viene soddisfatto. Se si desidera inserire una elaborazione di default, pertanto, la si dovrà inserire al termine dei casi.

6.4 Script ricorsivi

Uno script è a tutti gli effetti un comando del nostro sistema. Se operiamo l'accortezza di inserire la directory in cui si trova nel PATH, potremo addirittura metterlo in esecuzione digitando semplicemente il nome del file che lo contiene: Studiamo questa nuova evoluzione di un precedente script:

```

#!/bin/bash
echo $$ sta esplorando $(pwd)
conta=0
for entry in *
do
    if [ -f $entry ]
    then
        echo $entry file regolare
        conta=$(expr $conta + 1)
    fi
    if [ -d $entry ]
    then
        echo $entry directory
        cd $entry
        $0
        conta=$(expr $conta + $?)
        cd ..
    fi
done
exit $conta

```

Troviamo diverse modifiche rispetto alla versione precedente:

²Per approfondire, le man page di sed e grep dovrebbero essere utili

- L'uso della variabile di shell \$\$: viene espansa nel PID del processo corrente
- La command substitution di pwd per ottenere il percorso della directory corrente
- un cd \$entry per spostare la directory corrente dello script nella directory \$entry ed il relativo cd .. per risalire
- La command line \$0 per mettere in esecuzione una nuova istanza dello script che diverrà figlio del processo che esegue lo script. Nulla vieta che tale nuovo processo invochi una ulteriore istanza del processo in una forma ricorsiva.
- L'utilizzo dell'exit value \$? nell'espressione expr per sommare al valore di conta dell'istanza corrente quello restituito dall'istanza figlia

Come dovrebbe essere evidente, il valore di \$conta al termine dell'esecuzione del primo script messo in esecuzione conterrà il numero totale di file regolari trovati ricorsivamente nella gerarchia corrente.

6.4.1 L'operazione di quoting

Il carattere \$, ad esempio, ha una valenza particolare per la shell, in quanto rappresenta l'inizio di una variabile o di uno speciale simbolo da espandere. Facendo precedere il dollaro da un carattere *barra inversa* si ritrasforma il \$ in un carattere come tutti gli altri.

Questa operazione è detta **quoting** e permette di escludere la funzionalità di espansione della shell in certe occasioni.

Se, ad esempio, viene racchiuso un gruppo di caratteri della command line fra virgolette, si ottiene un quoting parziale: questo consente di raggruppare tutti i caratteri compresi all'interno di un unico elemento di command line anche se vi sono spazi, mantenendo tuttavia il ruolo dei wild card e del dollaro.

Con il termine *full quoting*, invece, si indica l'operazione di racchiudere un gruppo di caratteri fra apostrofi: in tal caso la shell evita di processare ogni carattere, anche speciale, all'interno del gruppo.

Vediamo un po' di esempi:

```
$ var="Un valore con spazi"
$ echo $var
Un valore con spazi
$ echo "$var"
Un valore con spazi
$ echo '$var'
$var
$ echo \$var
$var
$ var="Un valore con spazi e caratteri strani \'"
$ echo $var
Un valore con spazi e caratteri strani "
$ var="Un valore con spazi e caratteri strani '"
$ echo $var
Un valore con spazi e caratteri strani '
$ var="Un valore con spazi e caratteri strani *.sh"
$ echo $var
Un valore con spazi e caratteri strani ciao_mondo.sh crea_script.sh file.sh
$
```


Capitolo 7

Manpage di alcuni comandi

7.1 Il comando cat

7.1.1 Name

cat - concatenate files and print on the standard output

7.1.2 Synopsis

cat [*OPTION*] [*FILE*]...

7.1.3 Description

Concatenate FILE(s), or standard input, to standard output.

- **-A**, **-show-all** equivalent to **-vET**
- **-b**, **-number-nonblank** number nonblank output lines
- **-e** equivalent to **-vE**
- **-E**, **-show-ends** display \$ at end of each line
- **-n**, **-number** number all output lines
- **-r**, **-reversible** use \ to make the output reversible, implies **-v**
- **-s**, **-squeeze-blank** never more than one single blank line
- **-t** equivalent to **-vT**
- **-T**, **-show-tabs** display TAB characters as ^I
- **-u** (ignored)
- **-v**, **-show-nonprinting** use ^ and M- notation, except for LFD and TAB
- **-help** display this help and exit
- **-version** output version information and exit

With no FILE, or when FILE is -, read standard input.

7.1.4 Author

Written by Torbjorn Granlund and Richard M. Stallman.

7.1.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.1.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.1.7 See Also

The full documentation for **cat** is maintained as a Texinfo manual. If the **info** and **cat** programs are properly installed at your site, the command

- **info coreutils cat**

should give you access to the complete manual.

7.2 Il comando chmod

7.2.1 Name

chmod - change file access permissions

7.2.2 Synopsis

```
chmod [OPTION]... MODE[,MODE]... FILE...
chmod [OPTION]... OCTAL-MODE FILE...
chmod [OPTION]... -reference=RFILE FILE...
```

7.2.3 Description

This manual page documents the GNU version of **chmod**. **chmod** changes the permissions of each given file according to *mode*, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new permissions.

The format of a symbolic mode is '[*ugoa*...][[*+-=*][*rwXstugo*...][...][,...]'. Multiple symbolic operations can be given, separated by commas.

A combination of the letters 'ugoa' controls which users' access to the file will be changed: the user who owns it (u), other users in the file's group (g), other users not in the file's group (o), or all users (a). If none of these are given, the effect is as if 'a' were given, but bits that are set in the umask are not affected.

The operator '+' causes the permissions selected to be added to the existing permissions of each file; '-' causes them to be removed; and '=' causes them to be the only permissions that the file has.

The letters 'rwXstugo' select the new permissions for the affected users: read (r), write (w), execute (or access for directories) (x), execute only if the file is a directory or already has execute permission for some user (X), set user or group ID on execution (s), sticky (t), the permissions granted to the user who owns the file (u), the permissions granted to other users who are members of the file's group (g), and the permissions granted to users that are in neither of the two preceding categories (o).

A numeric mode is from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Any omitted digits are assumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and sticky (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.

chmod never changes the permissions of symbolic links; the **chmod** system call cannot change their permissions. This is not a problem since the permissions of symbolic links are never used. However, for each symbolic link listed on the command line, **chmod** changes the permissions of the pointed-to file. In contrast, **chmod** ignores symbolic links encountered during recursive directory traversals.

7.2.4 Sticky Files

On older Unix systems, the sticky bit caused executable files to be hoarded in swap space. This feature is not useful on modern VM systems, and the Linux kernel ignores the sticky bit on files. Other kernels may use the sticky bit on files for system-defined purposes. On some systems, only the superuser can set the sticky bit on files.

7.2.5 Sticky Directories

When the sticky bit is set on a directory, files in that directory may be unlinked or renamed only by root or their owner. Without the sticky bit, anyone able to write to the directory can delete or rename files. The sticky bit is commonly found on directories, such as /tmp, that are world-writable.

7.2.6 Options

Change the mode of each FILE to MODE.

- **-c**, **-changes** like verbose but report only when a change is made
- **-no-preserve-root** do not treat '/' specially (the default)
- **-preserve-root** fail to operate recursively on '/'
- **-f**, **-silent**, **-quiet** suppress most error messages
- **-v**, **-verbose** output a diagnostic for every file processed
- **-reference=RFILE** use RFILE's mode instead of MODE values
- **-R**, **-recursive** change files and directories recursively
- **-help** display this help and exit
- **-version** output version information and exit

Each MODE is one or more of the letters ugoa, one of the symbols += and one or more of the letters rwxXstugo.

7.2.7 Author

Written by David MacKenzie and Jim Meyering.

7.2.8 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.2.9 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.2.10 See Also

The full documentation for **chmod** is maintained as a Texinfo manual. If the **info** and **chmod** programs are properly installed at your site, the command

- **info coreutils chmod**

should give you access to the complete manual.

7.3 Il comando cp

7.3.1 Name

cp - copy files and directories

7.3.2 Synopsis

```
cp [OPTION]... SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
cp [OPTION]... -target-directory=DIRECTORY SOURCE...
```

7.3.3 Description

Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

Mandatory arguments to long options are mandatory for short options too.

- **-a**, **-archive** same as **-dpR**
- **-backup[=CONTROL]** make a backup of each existing destination file
- **-b** like **-backup** but does not accept an argument
- **-copy-contents** copy contents of special files when recursive
- **-d** same as **-no-dereference** **-preserve=link**
- **-no-dereference** never follow symbolic links
- **-f**, **-force** if an existing destination file cannot be opened, remove it and try again
- **-i**, **-interactive** prompt before overwrite
- **-H** follow command-line symbolic links
- **-l**, **-link** link files instead of copying
- **-L**, **-dereference** always follow symbolic links
- **-p** same as **-preserve=mode,ownership,timestamps**
- **-preserve[=ATTR_LIST]** preserve the specified attributes (default: mode,ownership,timestamps), if possible additional attributes: links, all
- **-no-preserve=ATTR_LIST** don't preserve the specified attributes
- **-parents** append source path to DIRECTORY
- **-P** same as **'-no-dereference'**
- **-R**, **-r**, **-recursive** copy directories recursively
- **-remove-destination** remove each existing destination file before attempting to open it (contrast with **-force**)
- **-reply={yes,no,query}** specify how to handle the prompt about an existing destination file
- **-sparse=WHEN** control creation of sparse files

- **-strip-trailing-slashes** remove any trailing slashes from each SOURCE argument
- **-s**, **-symbolic-link** make symbolic links instead of copying
- **-S**, **-suffix=***SUFFIX* override the usual backup suffix
- **-target-directory=***DIRECTORY* move all SOURCE arguments into DIRECTORY
- **-u**, **-update** copy only when the SOURCE file is newer than the destination file or when the destination file is missing
- **-v**, **-verbose** explain what is being done
- **-x**, **-one-file-system** stay on this file system
- **-help** display this help and exit
- **-version** output version information and exit

By default, sparse SOURCE files are detected by a crude heuristic and the corresponding DEST file is made sparse as well. That is the behavior selected by **-sparse=***auto*. Specify **-sparse=***always* to create a sparse DEST file whenever the SOURCE file contains a long enough sequence of zero bytes. Use **-sparse=***never* to inhibit creation of sparse files.

The backup suffix is ‘ ’, unless set with **-suffix** or SIMPLE_BACKUP_SUFFIX. The version control method may be selected via the **-backup** option or through the VERSION_CONTROL environment variable. Here are the values:

- *none*, *off* never make backups (even if **-backup** is given)
- *numbered*, *t* make numbered backups
- *existing*, *nil* numbered if numbered backups exist, *simple* otherwise
- *simple*, *never* always make simple backups

As a special case, **cp** makes a backup of SOURCE when the **force** and **backup** options are given and SOURCE and DEST are the same name for an existing, regular file.

7.3.4 Author

Written by Torbjorn Granlund, David MacKenzie, and Jim Meyering.

7.3.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.3.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.3.7 See Also

The full documentation for **cp** is maintained as a Texinfo manual. If the **info** and **cp** programs are properly installed at your site, the command

- **info coreutils cp**

should give you access to the complete manual.

7.4 Il comando echo

7.4.1 Name

echo - display a line of text

7.4.2 Synopsis

echo [*OPTION*]... [*STRING*]...

7.4.3 Description

NOTE: your shell may have its own version of echo which will supercede the version described here. Please refer to your shell's documentation for details about the options it supports.

Echo the *STRING*(s) to standard output.

- **-n** do not output the trailing newline
- **-e** enable interpretation of the backslash-escaped characters listed below
- **-help** display this help and exit
- **-version** output version information and exit

With **-e**, the following sequences are recognized and interpolated:

- **\NNN** the character whose ASCII code is NNN (octal)
- **** backslash
- **\a** alert (BEL)
- **\b** backspace
- **\c** suppress trailing newline
- **\f** form feed
- **\n** new line
- **\r** carriage return
- **\t** horizontal tab
- **\v** vertical tab

7.4.4 Author

Written by FIXME unknown.

7.4.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.4.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.4.7 See Also

The full documentation for **echo** is maintained as a Texinfo manual. If the **info** and **echo** programs are properly installed at your site, the command

- **info coreutils echo**

should give you access to the complete manual.

7.5 Il comando expr

7.5.1 Name

expr - evaluate expressions

7.5.2 Synopsis

expr *EXPRESSION*
 expr *OPTION*

7.5.3 Description

- **-help** display this help and exit
- **-version** output version information and exit

Print the value of EXPRESSION to standard output. A blank line below separates increasing precedence groups. EXPRESSION may be:

- ARG1 | ARG2 ARG1 if it is neither null nor 0, otherwise ARG2
- ARG1 & ARG2 ARG1 if neither argument is null or 0, otherwise 0
- ARG1 < ARG2 ARG1 is less than ARG2
- ARG1 <= ARG2 ARG1 is less than or equal to ARG2
- ARG1 = ARG2 ARG1 is equal to ARG2
- ARG1 != ARG2 ARG1 is unequal to ARG2
- ARG1 >= ARG2 ARG1 is greater than or equal to ARG2
- ARG1 > ARG2 ARG1 is greater than ARG2
- ARG1 + ARG2 arithmetic sum of ARG1 and ARG2
- ARG1 - ARG2 arithmetic difference of ARG1 and ARG2
- ARG1 * ARG2 arithmetic product of ARG1 and ARG2
- ARG1 / ARG2 arithmetic quotient of ARG1 divided by ARG2
- ARG1 % ARG2 arithmetic remainder of ARG1 divided by ARG2
- STRING : REGEXP anchored pattern match of REGEXP in STRING
- match STRING REGEXP same as STRING : REGEXP
- substr STRING POS LENGTH substring of STRING, POS counted from 1
- index STRING CHARS index in STRING where any CHARS is found, or 0
- length STRING length of STRING
- + TOKEN interpret TOKEN as a string, even if it is a
- keyword like 'match' or an operator like '/'

- (`EXPRESSION`) value of `EXPRESSION`

Beware that many operators need to be escaped or quoted for shells. Comparisons are arithmetic if both ARGs are numbers, else lexicographical. Pattern matches return the string matched between `\(` and `\)` or null; if `\(` and `\)` are not used, they return the number of characters matched or 0.

expr exits with:

- 0 if the expression is neither null nor 0; 1 if the expression is null or 0; or 2 for invalid expressions.

7.5.4 Author

Written by Mike Parker.

7.5.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.5.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.5.7 See Also

The full documentation for **expr** is maintained as a Texinfo manual. If the **info** and **expr** programs are properly installed at your site, the command

- **info coreutils expr**

should give you access to the complete manual.

7.6 Il comando grep

7.6.1 Name

grep, egrep, fgrep, rgrep - print lines matching a pattern

7.6.2 Synopsis

```
grep [options] PATTERN [FILE...]
grep [options] [-e PATTERN | -f FILE] [FILE...]
```

7.6.3 Description

grep searches the named input *FILEs* (or standard input if no files are named, or the file name - is given) for lines containing a match to the given *PATTERN*. By default, **grep** prints the matching lines.

In addition, three variant programs **egrep**, **fgrep** and **rgrep** are available. **egrep** is the same as **grep-E**. **fgrep** is the same as **grep-F**. **rgrep** is the same as **grep-r**.

7.6.4 Options

- **-ANUM**, **-after-context=NUM** Print *NUM* lines of trailing context after matching lines. Places a line containing - between contiguous groups of matches.
- **-a**, **-text** Process a binary file as if it were text; this is equivalent to the **-binary-files=text** option.
- **-BNUM**, **-before-context=NUM** Print *NUM* lines of leading context before matching lines. Places a line containing - between contiguous groups of matches.
- **-b**, **-byte-offset** Print the byte offset within the input file before each line of output.
- **-binary-files=TYPE** If the first few bytes of a file indicate that the file contains binary data, assume that the file is of type *TYPE*. By default, *TYPE* is **binary**, and **grep** normally outputs either a one-line message saying that a binary file matches, or no message if there is no match. If *TYPE* is **without-match**, **grep** assumes that a binary file does not match; this is equivalent to the **-I** option. If *TYPE* is **text**, **grep** processes a binary file as if it were text; this is equivalent to the **-a** option. *Warning: grep -binary-files=text* might output binary garbage, which can have nasty side effects if the output is a terminal and if the terminal driver interprets some of it as commands.
- **-CNUM**, **-context=NUM** Print *NUM* lines of output context. Places a line containing - between contiguous groups of matches.
- **-c**, **-count** Suppress normal output; instead print a count of matching lines for each input file. With the **-v**, **-invert-match** option (see below), count non-matching lines.
- **-colour[=WHEN]**, **-color[=WHEN]** Surround the matching string with the marker find in **GREP_COLOR** environment variable. *WHEN* may be 'never', 'always', or 'auto'
- **-DACTION**, **-devices=ACTION** If an input file is a device, FIFO or socket, use *ACTION* to process it. By default, *ACTION* is **read**, which means that devices are read just as if they were ordinary files. If *ACTION* is **skip**, devices are silently skipped.

- **-d** *ACTION*, **-directories=***ACTION* If an input file is a directory, use *ACTION* to process it. By default, *ACTION* is **read**, which means that directories are read just as if they were ordinary files. If *ACTION* is **skip**, directories are silently skipped. If *ACTION* is **recurse**, **grep** reads all files under each directory, recursively; this is equivalent to the **-r** option.
- **-E**, **-extended-regexp** Interpret *PATTERN* as an extended regular expression (see below).
- **-e** *PATTERN*, **-regexp=***PATTERN* Use *PATTERN* as the pattern; useful to protect patterns beginning with **-**.
- **-F**, **-fixed-strings** Interpret *PATTERN* as a list of fixed strings, separated by newlines, any of which is to be matched.
- **-f** *FILE*, **-file=***FILE* Obtain patterns from *FILE*, one per line. The empty file contains zero patterns, and therefore matches nothing.
- **-G**, **-basic-regexp** Interpret *PATTERN* as a basic regular expression (see below). This is the default.
- **-H**, **-with-filename** Print the filename for each match.
- **-h**, **-no-filename** Suppress the prefixing of filenames on output when multiple files are searched.
- **-help** Output a brief help message.
- **-I** Process a binary file as if it did not contain matching data; this is equivalent to the **-binary-files=without-match** option.
- **-i**, **-ignore-case** Ignore case distinctions in both the *PATTERN* and the input files.
- **-L**, **-files-without-match** Suppress normal output; instead print the name of each input file from which no output would normally have been printed. The scanning will stop on the first match.
- **-l**, **-files-with-matches** Suppress normal output; instead print the name of each input file from which output would normally have been printed. The scanning will stop on the first match.
- **-label=***LABEL* Displays input actually coming from standard input as input coming from file *LABEL*. This is especially useful for tools like **zgrep**, e.g. **gzip -cd foo.gz |grep -label=foo something**
- **-line-buffered** Use line buffering, it can be a performance penalty.
- **-m** *NUM*, **-max-count=***NUM* Stop reading a file after *NUM* matching lines. If the input is standard input from a regular file, and *NUM* matching lines are output, **grep** ensures that the standard input is positioned to just after the last matching line before exiting, regardless of the presence of trailing context lines. This enables a calling process to resume a search. When **grep** stops after *NUM* matching lines, it outputs any trailing context lines. When the **-c** or **-count** option is also used, **grep** does not output a count greater than *NUM*. When the **-v** or **-invert-match** option is also used, **grep** stops after outputting *NUM* non-matching lines.

- **-mmap** If possible, use the **mmap(2)** system call to read input, instead of the default **read(2)** system call. In some situations, **-mmap** yields better performance. However, **-mmap** can cause undefined behavior (including core dumps) if an input file shrinks while **grep** is operating, or if an I/O error occurs.
- **-n, -line-number** Prefix each line of output with the line number within its input file.
- **-o, -only-matching** Show only the part of a matching line that matches *PATTERN*.
- **-P, -perl-regexp** Interpret *PATTERN* as a Perl regular expression.
- **-q, -quiet, -silent** Quiet; do not write anything to standard output. Exit immediately with zero status if any match is found, even if an error was detected. Also see the **-s** or **-no-messages** option.
- **-R, -r, -recursive** Read all files under each directory, recursively; this is equivalent to the **-d recurse** option.
- **-include=PATTERN** Recurse in directories only searching file matching *PATTERN*.
- **-exclude=PATTERN** Recurse in directories skip file matching *PATTERN*.
- **-s, -no-messages** Suppress error messages about nonexistent or unreadable files. Portability note: unlike GNU **grep**, traditional **grep** did not conform to POSIX.2, because traditional **grep** lacked a **-q** option and its **-s** option behaved like GNU **grep**'s **-q** option. Shell scripts intended to be portable to traditional **grep** should avoid both **-q** and **-s** and should redirect output to `/dev/null` instead.
- **-U, -binary** Treat the file(s) as binary. By default, under MS-DOS and MS-Windows, **grep** guesses the file type by looking at the contents of the first 32KB read from the file. If **grep** decides the file is a text file, it strips the CR characters from the original file contents (to make regular expressions with `^` and `$` work correctly). Specifying **-U** overrules this guesswork, causing all files to be read and passed to the matching mechanism verbatim; if the file is a text file with CR/LF pairs at the end of each line, this will cause some regular expressions to fail. This option has no effect on platforms other than MS-DOS and MS-Windows.
- **-u, -unix-byte-offsets** Report Unix-style byte offsets. This switch causes **grep** to report byte offsets as if the file were Unix-style text file, i.e. with CR characters stripped off. This will produce results identical to running **grep** on a Unix machine. This option has no effect unless **-b** option is also used; it has no effect on platforms other than MS-DOS and MS-Windows.
- **-V, -version** Print the version number of **grep** to standard error. This version number should be included in all bug reports (see below).
- **-v, -invert-match** Invert the sense of matching, to select non-matching lines.
- **-w, -word-regexp** Select only those lines containing matches that form whole words. The test is that the matching substring must either be at the beginning of the line, or preceded by a non-word constituent character. Similarly, it must be either at the end of the line or followed by a non-word constituent character. Word-constituent characters are letters, digits, and the underscore.
- **-x, -line-regexp** Select only those matches that exactly match the whole line.

- **-y** Obsolete synonym for **-i**.
- **-Z**, **-null** Output a zero byte (the ASCII **NUL** character) instead of the character that normally follows a file name. For example, **grep -IZ** outputs a zero byte after each file name instead of the usual newline. This option makes the output unambiguous, even in the presence of file names containing unusual characters like newlines. This option can be used with commands like **find -print0**, **perl -0**, **sort -z**, and **xargs -0** to process arbitrary file names, even those that contain newline characters.
- **-z**, **-null-data** Treat the input as a set of lines, each terminated by a zero byte (the ASCII **NUL** character) instead of a newline. Like the **-Z** or **-null** option, this option can be used with commands like **sort -z** to process arbitrary file names.

7.6.5 Regular Expressions

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

grep understands three different versions of regular expression syntax: ‘basic,’ ‘extended,’ and ‘perl.’ In GNU**grep**, there is no difference in available functionality using either of the first two syntaxes. In other implementations, basic regular expressions are less powerful. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards. Perl regular expressions add additional functionality, but the implementation used here is undocumented and is not compatible with other **grep** implementations.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash.

A *bracket expression* is a list of characters enclosed by [and]. It matches any single character in that list; if the first character of the list is the caret ^ then it matches any character *not* in the list. For example, the regular expression **[0123456789]** matches any single digit.

Within a bracket expression, a *range expression* consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive, using the locale’s collating sequence and character set. For example, in the default C locale, **[a-d]** is equivalent to **[abcd]**. Many locales sort characters in dictionary order, and in these locales **[a-d]** is typically not equivalent to **[abcd]**; it might be equivalent to **[aBbCcDd]**, for example. To obtain the traditional interpretation of bracket expressions, you can use the C locale by setting the **LC_ALL** environment variable to the value **C**.

Finally, certain named classes of characters are predefined within bracket expressions, as follows. Their names are self explanatory, and they are **[:alnum:]**, **[:alpha:]**, **[:cntrl:]**, **[:digit:]**, **[:graph:]**, **[:lower:]**, **[:print:]**, **[:punct:]**, **[:space:]**, **[:upper:]**, and **[:xdigit:]**. For example, **[:alnum:]** means **[0-9A-Za-z]**, except the latter form depends upon the C locale and the ASCII character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal] place it first in the list. Similarly, to include a literal ^ place it anywhere but first. Finally, to include a literal - place it last.

The period . matches any single character. The symbol **\w** is a synonym for **[:alnum:]** and **\W** is a synonym for **[^[:alnum:]]**.

The caret ^ and the dollar sign \$ are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols **\<** and **\>** respectively match the empty

string at the beginning and end of a word. The symbol `\b` matches the empty string at the edge of a word, and `\B` matches the empty string provided it's *not* at the edge of a word.

A regular expression may be followed by one of several repetition operators:

- `?` The preceding item is optional and matched at most once.
- `*` The preceding item will be matched zero or more times.
- `+` The preceding item will be matched one or more times.
- `{n}` The preceding item is matched exactly n times.
- `{n,}` The preceding item is matched n or more times.
- `{n,m}` The preceding item is matched at least n times, but not more than m times.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator `|`; the resulting regular expression matches any string matching either subexpression.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference `\n`, where n is a single digit, matches the substring previously matched by the n th parenthesized subexpression of the regular expression.

In basic regular expressions the metacharacters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslashed versions `\?`, `\+`, `\{`, `\|`, `\(`, and `\)`.

Traditional **egrep** did not support the `{` metacharacter, and some **egrep** implementations support `\{` instead, so portable scripts should avoid `{` in **egrep** patterns and should use `[{]` to match a literal `{`.

GNU **egrep** attempts to support traditional usage by assuming that `{` is not special if it would be the start of an invalid interval specification. For example, the shell command **egrep** `'{1'` searches for the two-character string `{1` instead of reporting a syntax error in the regular expression. POSIX.2 allows this behavior as an extension, but portable scripts should avoid it.

7.6.6 Environment Variables

`grep`'s behavior is affected by the following environment variables.

A locale `LC_foo` is specified by examining the three environment variables `LC_ALL`, `LC_foo`, `LANG`, in that order. The first of these variables that is set specifies the locale. For example, if `LC_ALL` is not set, but `LC_MESSAGES` is set to `pt_BR`, then Brazilian Portuguese is used for the `LC_MESSAGES` locale. The C locale is used if none of these environment variables are set, or if the locale catalog is not installed, or if **grep** was not compiled with national language support (NLS).

- **GREP_OPTIONS** This variable specifies default options to be placed in front of any explicit options. For example, if **GREP_OPTIONS** is `'-binary-files=without-match -directories=skip'`, **grep** behaves as if the two options `-binary-files=without-match` and `-directories=skip` had been specified before any explicit options. Option specifications are separated by whitespace. A backslash escapes the next character, so it can be used to specify an option containing whitespace or a backslash.
- **GREP_COLOR** Specifies the marker for highlighting.

- **LC_ALL, LC_COLLATE, LANG** These variables specify the **LC_COLLATE** locale, which determines the collating sequence used to interpret range expressions like **[a-z]**.
- **LC_ALL, LC_CTYPE, LANG** These variables specify the **LC_CTYPE** locale, which determines the type of characters, e.g., which characters are whitespace.
- **LC_ALL, LC_MESSAGES, LANG** These variables specify the **LC_MESSAGES** locale, which determines the language that **grep** uses for messages. The default C locale uses American English messages.
- **POSIXLY_CORRECT** If set, **grep** behaves as POSIX.2 requires; otherwise, **grep** behaves more like other GNU programs. POSIX.2 requires that options that follow file names must be treated as file names; by default, such options are permuted to the front of the operand list and are treated as options. Also, POSIX.2 requires that unrecognized options be diagnosed as ‘illegal’, but since they are not really against the law the default is to diagnose them as ‘invalid’. **POSIXLY_CORRECT** also disables **_GNU_nonoption_argv_flags_**, described below.
- **_GNU_nonoption_argv_flags_** (Here *N* is **grep**’s numeric process ID.) If the *i*th character of this environment variable’s value is **1**, do not consider the *i*th operand of **grep** to be an option, even if it appears to be one. A shell can put this variable in the environment for each command it runs, specifying which operands are the results of file name wildcard expansion and therefore should not be treated as options. This behavior is available only with the GNU C library, and only when **POSIXLY_CORRECT** is not set.

7.6.7 Diagnostics

Normally, exit status is 0 if selected lines are found and 1 otherwise. But the exit status is 2 if an error occurred, unless the **-q** or **-quiet** or **-silent** option is used and a selected line is found.

7.6.8 Bugs

Email bug reports to **bug-gnu-utils@gnu.org**. Be sure to include the word ‘grep’ somewhere in the ‘Subject:’ field.

Large repetition counts in the $\{n,m\}$ construct may cause **grep** to use lots of memory. In addition, certain other obscure regular expressions require exponential time and space, and may cause **grep** to run out of memory.

Backreferences are very slow, and may require exponential time.

7.7 Il comando ls

7.7.1 Name

ls - list directory contents

7.7.2 Synopsis

ls [*OPTION*]... [*FILE*]...

7.7.3 Description

List information about the FILEs (the current directory by default). Sort entries alphabetically if none of **-cftuSUX** nor **-sort**.

Mandatory arguments to long options are mandatory for short options too.

- **-a**, **-all** do not hide entries starting with **.**
- **-A**, **-almost-all** do not list implied **.** and **..**
- **-author** print the author of each file
- **-b**, **-escape** print octal escapes for nongraphic characters
- **-block-size=SIZE** use SIZE-byte blocks
- **-B**, **-ignore-backups** do not list implied entries ending with **~**
- **-c** with **-lt**: sort by, and show, ctime (time of last modification of file status information) with **-l**: show ctime and sort by name otherwise: sort by ctime
- **-C** list entries by columns
- **-color[=WHEN]** control whether color is used to distinguish file types. WHEN may be 'never', 'always', or 'auto'
- **-d**, **-directory** list directory entries instead of contents, and do not dereference symbolic links
- **-D**, **-dired** generate output designed for Emacs' dired mode
- **-f** do not sort, enable **-aU**, disable **-lst**
- **-F**, **-classify** append indicator (one of */=@|) to entries
- **-format=WORD** across **-x**, commas **-m**, horizontal **-x**, long **-l**, single-column **-1**, verbose **-l**, vertical **-C**
- **-full-time** like **-l** **-time-style=full-iso**
- **-g** like **-l**, but do not list owner
- **-G**, **-no-group** inhibit display of group information
- **-h**, **-human-readable** print sizes in human readable format (e.g., 1K 234M 2G)
- **-si** likewise, but use powers of 1000 not 1024
- **-H**, **-dereference-command-line** follow symbolic links listed on the command line

- **-dereference-command-line-symlink-to-dir** follow each command line symbolic link
- that points to a directory
- **-indicator-style=WORD** append indicator with style WORD to entry names: none (default), classify (-F), file-type (-p)
- **-i, -inode** print index number of each file
- **-I, -ignore=PATTERN** do not list implied entries matching shell PATTERN
- **-k** like **-block-size=1K**
- **-l** use a long listing format
- **-L, -dereference** when showing file information for a symbolic link, show information for the file the link references rather than for the link itself
- **-m** fill width with a comma separated list of entries
- **-n, -numeric-uid-gid** like **-l**, but list numeric UIDs and GIDs
- **-N, -literal** print raw entry names (don't treat e.g. control characters specially)
- **-o** like **-l**, but do not list group information
- **-p, -file-type** append indicator (one of /=@|) to entries
- **-q, -hide-control-chars** print ? instead of non graphic characters
- **-show-control-chars** show non graphic characters as-is (default unless program is 'ls' and output is a terminal)
- **-Q, -quote-name** enclose entry names in double quotes
- **-quoting-style=WORD** use quoting style WORD for entry names: literal, locale, shell, shell-always, c, escape
- **-r, -reverse** reverse order while sorting
- **-R, -recursive** list subdirectories recursively
- **-s, -size** print size of each file, in blocks
- **-S** sort by file size
- **-sort=WORD** extension **-X**, none **-U**, size **-S**, time **-t**, version **-v**
- status **-c**, time **-t**, atime **-u**, access **-u**, use **-u**
- **-time=WORD** show time as WORD instead of modification time: atime, access, use, ctime or status; use specified time as sort key if **-sort=time**
- **-time-style=STYLE** show times using style STYLE: full-iso, long-iso, iso, locale, +FORMAT
- FORMAT is interpreted like 'date'; if FORMAT is FORMAT1<newline>FORMAT2, FORMAT1 applies to non-recent files and FORMAT2 to recent files; if STYLE is prefixed with 'posix-', STYLE takes effect only outside the POSIX locale

- **-t** sort by modification time
- **-T**, **-tabsize=COLS** assume tab stops at each COLS instead of 8
- **-u** with **-lt**: sort by, and show, access time with **-l**: show access time and sort by name otherwise: sort by access time
- **-U** do not sort; list entries in directory order
- **-v** sort by version
- **-w**, **-width=COLS** assume screen width instead of current value
- **-x** list entries by lines instead of by columns
- **-X** sort alphabetically by entry extension
- **-1** list one file per line
- **-help** display this help and exit
- **-version** output version information and exit

SIZE may be (or may be an integer optionally followed by) one of following: kB 1000, K 1024, MB 1000*1000, M 1024*1024, and so on for G, T, P, E, Z, Y.

By default, color is not used to distinguish types of files. That is equivalent to using **-color=none**. Using the **-color** option without the optional WHEN argument is equivalent to using **-color=always**. With **-color=auto**, color codes are output only if standard output is connected to a terminal (tty).

7.7.4 Author

Written by Richard Stallman and David MacKenzie.

7.7.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.7.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.7.7 See Also

The full documentation for **ls** is maintained as a Texinfo manual. If the **info** and **ls** programs are properly installed at your site, the command

- **info coreutils ls**

should give you access to the complete manual.

7.8 Il comando `mkdir`

7.8.1 Name

`mkdir` - make directories

7.8.2 Synopsis

`mkdir` [*OPTION*] *DIRECTORY*...

7.8.3 Description

Create the *DIRECTORY*(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

- **-m**, **-mode=MODE** set permission mode (as in `chmod`), not `rw-rw-rw-r` - `umask`
- **-p**, **-parents** no error if existing, make parent directories as needed
- **-v**, **-verbose** print a message for each created directory
- **-help** display this help and exit
- **-version** output version information and exit

7.8.4 Author

Written by David MacKenzie.

7.8.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.8.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.8.7 See Also

The full documentation for `mkdir` is maintained as a Texinfo manual. If the `info` and `mkdir` programs are properly installed at your site, the command

- `info coreutils mkdir`

should give you access to the complete manual.

7.9 Il comando pwd

7.9.1 Name

pwd - print name of current/working directory

7.9.2 Synopsis

pwd [*OPTION*]

7.9.3 Description

NOTE: your shell may have its own version of pwd which will supercede the version described here. Please refer to your shell's documentation for details about the options it supports.

Print the full filename of the current working directory.

- **-help** display this help and exit
- **-version** output version information and exit

7.9.4 Author

Written by Jim Meyering.

7.9.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.9.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.9.7 See Also

The full documentation for **pwd** is maintained as a Texinfo manual. If the **info** and **pwd** programs are properly installed at your site, the command

- **info coreutils pwd**

should give you access to the complete manual.

7.10 Il comando `rm`

7.10.1 Name

`rm` - remove files or directories

7.10.2 Synopsis

`rm` [*OPTION*]... *FILE*...

7.10.3 Description

This manual page documents the GNU version of `rm`. `rm` removes each specified file. By default, it does not remove directories.

If a file is unwritable, the standard input is a tty, and the `-f` or `-force` option is not given, `rm` prompts the user for whether to remove the file. If the response does not begin with 'y' or 'Y', the file is skipped.

7.10.4 Options

Remove (unlink) the *FILE*(s).

- `-d`, `-directory` unlink *FILE*, even if it is a non-empty directory (super-user only; this works only if your system supports 'unlink' for nonempty directories)
- `-f`, `-force` ignore nonexistent files, never prompt
- `-i`, `-interactive` prompt before any removal
- `-no-preserve-root` do not treat '/' specially (the default)
- `-preserve-root` fail to operate recursively on '/'
- `-r`, `-R`, `-recursive` remove the contents of directories recursively
- `-v`, `-verbose` explain what is being done
- `-help` display this help and exit
- `-version` output version information and exit

To remove a file whose name starts with a '-', for example '-foo', use one of these commands:

- `rm -- -foo`
- `rm ./-foo`

Note that if you use `rm` to remove a file, it is usually possible to recover the contents of that file. If you want more assurance that the contents are truly unrecoverable, consider using `shred`.

7.10.5 Author

Written by Paul Rubin, David MacKenzie, Richard Stallman, and Jim Meyering.

7.10.6 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.10.7 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.10.8 See Also

`chattr(1)`, `shred(1)`

The full documentation for **rm** is maintained as a Texinfo manual. If the **info** and **rm** programs are properly installed at your site, the command

- **info coreutils rm**

should give you access to the complete manual.

7.11 Il comando `rmdir`

7.11.1 Name

`rmdir` - remove empty directories

7.11.2 Synopsis

`rmdir` [*OPTION*]... *DIRECTORY*...

7.11.3 Description

Remove the *DIRECTORY*(ies), if they are empty.

- **-ignore-fail-on-non-empty**
- ignore each failure that is solely because a directory is non-empty
- **-p**, **-parents** remove *DIRECTORY*, then try to remove each directory component of that path name. E.g., '`rmdir -p a/b/c`' is similar to '`rmdir a/b/c a/b a`'.
- **-v**, **-verbose** output a diagnostic for every directory processed
- **-help** display this help and exit
- **-version** output version information and exit

7.11.4 Author

Written by David MacKenzie.

7.11.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.11.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.11.7 See Also

The full documentation for **`rmdir`** is maintained as a Texinfo manual. If the **`info`** and **`rmdir`** programs are properly installed at your site, the command

- **`info coreutils rmdir`**

should give you access to the complete manual.

7.12 Il comando test

7.12.1 Name

test - check file types and compare values

7.12.2 Synopsis

```
test EXPRESSION
  [ EXPRESSION ]
  [ OPTION ]
```

7.12.3 Description

Exit with the status determined by *EXPRESSION*.

- **-help** display this help and exit
- **-version** output version information and exit

EXPRESSION is true or false and sets exit status. It is one of:

- (*EXPRESSION*) *EXPRESSION* is true
- ! *EXPRESSION* *EXPRESSION* is false
- *EXPRESSION*₁ **-a** *EXPRESSION*₂ both *EXPRESSION*₁ and *EXPRESSION*₂ are true
- *EXPRESSION*₁ **-o** *EXPRESSION*₂ either *EXPRESSION*₁ or *EXPRESSION*₂ is true

-n *STRING* the length of *STRING* is nonzero

- **-z** *STRING* the length of *STRING* is zero
- *STRING*₁ = *STRING*₂ the strings are equal
- *STRING*₁ != *STRING*₂ the strings are not equal
- *INTEGER*₁ **-eq** *INTEGER*₂ *INTEGER*₁ is equal to *INTEGER*₂
- *INTEGER*₁ **-ge** *INTEGER*₂ *INTEGER*₁ is greater than or equal to *INTEGER*₂
- *INTEGER*₁ **-gt** *INTEGER*₂ *INTEGER*₁ is greater than *INTEGER*₂
- *INTEGER*₁ **-le** *INTEGER*₂ *INTEGER*₁ is less than or equal to *INTEGER*₂
- *INTEGER*₁ **-lt** *INTEGER*₂ *INTEGER*₁ is less than *INTEGER*₂
- *INTEGER*₁ **-ne** *INTEGER*₂ *INTEGER*₁ is not equal to *INTEGER*₂
- *FILE*₁ **-ef** *FILE*₂ *FILE*₁ and *FILE*₂ have the same device and inode numbers
- *FILE*₁ **-nt** *FILE*₂ *FILE*₁ is newer (modification date) than *FILE*₂
- *FILE*₁ **-ot** *FILE*₂ *FILE*₁ is older than *FILE*₂
- **-b** *FILE* *FILE* exists and is block special
- **-c** *FILE* *FILE* exists and is character special

- **-d** FILE FILE exists and is a directory
- **-e** FILE FILE exists
- **-f** FILE FILE exists and is a regular file
- **-g** FILE FILE exists and is set-group-ID
- **-h** FILE FILE exists and is a symbolic link (same as **-L**)
- **-G** FILE FILE exists and is owned by the effective group ID
- **-k** FILE FILE exists and has its sticky bit set
- **-L** FILE FILE exists and is a symbolic link (same as **-h**)
- **-O** FILE FILE exists and is owned by the effective user ID
- **-p** FILE FILE exists and is a named pipe
- **-r** FILE FILE exists and is readable
- **-s** FILE FILE exists and has a size greater than zero
- **-S** FILE FILE exists and is a socket
- **-t** FD file descriptor FD is opened on a terminal
- **-u** FILE FILE exists and its set-user-ID bit is set
- **-w** FILE FILE exists and is writable
- **-x** FILE FILE exists and is executable

Beware that parentheses need to be escaped (e.g., by backslashes) for shells. `INTEGER` may also be `-l STRING`, which evaluates to the length of `STRING`.

7.12.4 Author

Written by Kevin Braunsdorf and Matthew Bradburn.

7.12.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.12.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.12.7 See Also

The full documentation for `[` is maintained as a Texinfo manual. If the `info` and `[` programs are properly installed at your site, the command

- **info coreutils test**

should give you access to the complete manual.

7.13 Il comando touch

7.13.1 Name

touch - change file timestamps

7.13.2 Synopsis

touch [*OPTION*]... *FILE*...

7.13.3 Description

Update the access and modification times of each *FILE* to the current time.

Mandatory arguments to long options are mandatory for short options too.

- **-a** change only the access time
- **-c**, **-no-create** do not create any files
- **-d**, **-date=STRING** parse *STRING* and use it instead of current time
- **-f** (ignored)
- **-m** change only the modification time
- **-r**, **-reference=FILE** use this file's times instead of current time
- **-t STAMP** use [[*CC*]*YY*]*MMDD**hhmm*[.*ss*] instead of current time
- **-time=WORD** set time given by *WORD*: access atime use (same as **-a**) modify mtime (same as **-m**)
- **-help** display this help and exit
- **-version** output version information and exit

Note that the **-d** and **-t** options accept different time-date formats.

7.13.4 Author

Written by Paul Rubin, Arnold Robbins, Jim Kingdon, David MacKenzie, and Randy Smith.

7.13.5 Reporting Bugs

Report bugs to <bug-coreutils@gnu.org>.

7.13.6 Copyright

Copyright ©2004 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

7.13.7 See Also

The full documentation for **touch** is maintained as a Texinfo manual. If the **info** and **touch** programs are properly installed at your site, the command

- **info coreutils touch**

should give you access to the complete manual.

7.14 Manpage della shell bash

7.14.1 Name

bash - GNU Bourne-Again SHell

7.14.2 Synopsis

bash [options] [file]

7.14.3 Copyright

Bash is Copyright (C) 1989-2004 by the Free Software Foundation, Inc.

7.14.4 Description

Bash is an **sh**-compatible command language interpreter that executes commands read from the standard input or from a file. **Bash** also incorporates useful features from the *Korn* and *C* shells (**ksh** and **csh**).

Bash is intended to be a conformant implementation of the IEEE POSIX Shell and Tools specification (IEEE Working Group 1003.2).

7.14.5 Options

In addition to the single-character shell options documented in the description of the **set** builtin command, **bash** interprets the following options when it is invoked:

- **-cstring** If the **-c** option is present, then commands are read from *string*. If there are arguments after the *string*, they are assigned to the positional parameters, starting with **\$0**.
- **-i** If the **-i** option is present, the shell is *interactive*.
- **-l** Make **bash** act as if it had been invoked as a login shell (see **INVOCATION** below).
- **-r** If the **-r** option is present, the shell becomes *restricted* (see **RESTRICTED SHELL** below).
- **-s** If the **-s** option is present, or if no arguments remain after option processing, then commands are read from the standard input. This option allows the positional parameters to be set when invoking an interactive shell.
- **-D** A list of all double-quoted strings preceded by **\$** is printed on the standard output. These are the strings that are subject to language translation when the current locale is not **C** or **POSIX**. This implies the **-n** option; no commands will be executed.
- **[-+]O [shopt_option]** *shopt_option* is one of the shell options accepted by the **shopt** builtin (see **SHELL BUILTIN COMMANDS** below). If *shopt_option* is present, **-O** sets the value of that option; **+O** unsets it. If *shopt_option* is not supplied, the names and values of the shell options accepted by **shopt** are printed on the standard output. If the invocation option is **+O**, the output is displayed in a format that may be reused as input.
- **-** A **-** signals the end of options and disables further option processing. Any arguments after the **-** are treated as filenames and arguments. An argument of **-** is equivalent to **-**.

Bash also interprets a number of multi-character options. These options must appear on the command line before the single-character options to be recognized.

- **-debugger** Arrange for the debugger profile to be executed before the shell starts. Turns on extended debugging mode (see the description of the **extdebug** option to the **shopt** builtin below) and shell function tracing (see the description of the **-o functrace** option to the **set** builtin below).
- **-dump-po-strings** Equivalent to **-D**, but the output is in the GNU *gettext* **po** (portable object) file format.
- **-dump-strings** Equivalent to **-D**.
- **-help** Display a usage message on standard output and exit successfully.
- **-init-file** *file*
- **-rcfile** *file* Execute commands from *file* instead of the system wide initialization file */etc/bash.bashrc* and the standard personal initialization file *./bashrc* if the shell is interactive (see **INVOCATION** below).
- **-login** Equivalent to **-l**.
- **-noediting** Do not use the GNU **readline** library to read command lines when the shell is interactive.
- **-noprofile** Do not read either the system-wide startup file */etc/profile* or any of the personal initialization files *./bash_profile*, *./bash_login*, or *./profile*. By default, **bash** reads these files when it is invoked as a login shell (see **INVOCATION** below).
- **-norc** Do not read and execute the system wide initialization file */etc/bash.bashrc* and the personal initialization file *./bashrc* if the shell is interactive. This option is on by default if the shell is invoked as **sh**.
- **-posix** Change the behavior of **bash** where the default operation differs from the POSIX 1003.2 standard to match the standard (*posix mode*).
- **-restricted** The shell becomes restricted (see **RESTRICTED SHELL** below).
- **-verbose** Equivalent to **-v**.
- **-version** Show version information for this instance of **bash** on the standard output and exit successfully.

7.14.6 Arguments

If arguments remain after option processing, and neither the **-c** nor the **-s** option has been supplied, the first argument is assumed to be the name of a file containing shell commands. If **bash** is invoked in this fashion, **\$0** is set to the name of the file, and the positional parameters are set to the remaining arguments. **Bash** reads and executes commands from this file, then exits. **Bash**'s exit status is the exit status of the last command executed in the script. If no commands are executed, the exit status is 0. An attempt is first made to open the file in the current directory, and, if no file is found, then the shell searches the directories in **PATH** for the script.

7.14.7 Invocation

A *login shell* is one whose first character of argument zero is a -, or one started with the **-login** option.

An *interactive shell* is one started without non-option arguments and without the **-c** option whose standard input and error are both connected to terminals (as determined by *isatty(3)*), or one started with the **-i** option. **PS1** is set and **\$-** includes **i** if **bash** is interactive, allowing a shell script or a startup file to test this state.

The following paragraphs describe how **bash** executes its startup files. If any of the files exist but cannot be read, **bash** reports an error. Tildes are expanded in file names as described below under **Tilde Expansion** in the **EXPANSION** section.

When **bash** is invoked as an interactive login shell, or as a non-interactive shell with the **-login** option, it first reads and executes commands from the file */etc/profile*, if that file exists. After reading that file, it looks for *./bash_profile*, *./bash_login*, and *./profile*, in that order, and reads and executes commands from the first one that exists and is readable. The **-nopprofile** option may be used when the shell is started to inhibit this behavior.

When a login shell exits, **bash** reads and executes commands from the file *./bash_logout*, if it exists.

When an interactive shell that is not a login shell is started, **bash** reads and executes commands from */etc/bash.bashrc* and *./bashrc*, if these files exist. This may be inhibited by using the **-norc** option. The **-rcfile file** option will force **bash** to read and execute commands from *file* instead of */etc/bash.bashrc* and *./bashrc*.

When **bash** is started non-interactively, to run a shell script, for example, it looks for the variable **BASH_ENV** in the environment, expands its value if it appears there, and uses the expanded value as the name of a file to read and execute. **Bash** behaves as if the following command were executed:

```
if [ -n $BASH_ENV ]; then . $BASH_ENV; fi
but the value of the PATH variable is not used to search for the file name.
```

If **bash** is invoked with the name **sh**, it tries to mimic the startup behavior of historical versions of **sh** as closely as possible, while conforming to the POSIX standard as well. When invoked as an interactive login shell, or a non-interactive shell with the **-login** option, it first attempts to read and execute commands from */etc/profile* and *./profile*, in that order. The **-nopprofile** option may be used to inhibit this behavior. When invoked as an interactive shell with the name **sh**, **bash** looks for the variable **ENV**, expands its value if it is defined, and uses the expanded value as the name of a file to read and execute. Since a shell invoked as **sh** does not attempt to read and execute commands from any other startup files, the **-rcfile** option has no effect. A non-interactive shell invoked with the name **sh** does not attempt to read any other startup files. When invoked as **sh**, **bash** enters *posix* mode after the startup files are read.

When **bash** is started in *posix* mode, as with the **-posix** command line option, it follows the POSIX standard for startup files. In this mode, interactive shells expand the **ENV** variable and commands are read and executed from the file whose name is the expanded value. No other startup files are read.

Bash attempts to determine when it is being run by the remote shell daemon, usually *rshd*. If **bash** determines it is being run by *rshd*, it reads and executes commands from */etc/bash.bashrc* and *./bashrc*, if these files exist and are readable. It will not do this if invoked as **sh**. The **-norc** option may be used to inhibit this behavior, and the **-rcfile** option may be used to force another file to be read, but *rshd* does not generally invoke the shell with those options or allow them to be specified.

If the shell is started with the effective user (group) id not equal to the real user (group) id, and the **-p** option is not supplied, no startup files are read, shell functions are not inherited from the environment, the **SHELLOPTS** variable, if it appears in the environment, is ignored,

and the effective user id is set to the real user id. If the **-p** option is supplied at invocation, the startup behavior is the same, but the effective user id is not reset.

7.14.8 Definitions

The following definitions are used throughout the rest of this document.

- **blank** A space or tab.
- **word** A sequence of characters considered as a single unit by the shell. Also known as a **token**.
- **name** A *word* consisting only of alphanumeric characters and underscores, and beginning with an alphabetic character or an underscore. Also referred to as an **identifier**.
- **metacharacter** A character that, when unquoted, separates words. One of the following:
| & ; () < > space tab
- **control operator** A *token* that performs a control function. It is one of the following symbols:

|| & && ; ;; () | <newline>

7.14.9 Reserved Words

Reserved words are words that have a special meaning to the shell. The following words are recognized as reserved when unquoted and either the first word of a simple command (see **SHELL GRAMMAR** below) or the third word of a **case** or **for** command:

! case do done elif else esac fi for function if in select then until while { } time [[]]

7.14.10 Shell Grammar

Simple Commands

A *simple command* is a sequence of optional variable assignments followed by **blank**-separated words and redirections, and terminated by a *control operator*. The first word specifies the command to be executed, and is passed as argument zero. The remaining words are passed as arguments to the invoked command.

The return value of a *simple command* is its exit status, or $128+n$ if the command is terminated by signal n .

Pipelines

A *pipeline* is a sequence of one or more commands separated by the character |. The format for a pipeline is:

```
[time [-p]] [ ! ] command [ | command2 ... ]
```

The standard output of *command* is connected via a pipe to the standard input of *command2*. This connection is performed before any redirections specified by the command (see **REDIRECTION** below).

The return status of a pipeline is the exit status of the last command, unless the **pipefail** option is enabled. If **pipefail** is enabled, the pipeline's return status is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands exit successfully. If the reserved word ! precedes a pipeline, the exit status of that pipeline is the logical negation of

the exit status as described above. The shell waits for all commands in the pipeline to terminate before returning a value.

If the **time** reserved word precedes a pipeline, the elapsed as well as user and system time consumed by its execution are reported when the pipeline terminates. The **-p** option changes the output format to that specified by POSIX. The **TIMEFORMAT** variable may be set to a format string that specifies how the timing information should be displayed; see the description of **TIMEFORMAT** under **Shell Variables** below.

Each command in a pipeline is executed as a separate process (i.e., in a subshell).

Lists

A *list* is a sequence of one or more pipelines separated by one of the operators **;**, **&**, **&&**, or **||**, and optionally terminated by one of **;**, **&**, or **<newline>**.

Of these list operators, **&&** and **||** have equal precedence, followed by **;** and **&**, which have equal precedence.

A sequence of one or more newlines may appear in a *list* instead of a semicolon to delimit commands.

If a command is terminated by the control operator **&**, the shell executes the command in the *background* in a subshell. The shell does not wait for the command to finish, and the return status is 0. Commands separated by a **;** are executed sequentially; the shell waits for each command to terminate in turn. The return status is the exit status of the last command executed.

The control operators **&&** and **||** denote AND lists and OR lists, respectively. An AND list has the form

```
command1 && command2
```

command2 is executed if, and only if, *command1* returns an exit status of zero.

An OR list has the form

```
command1 || command2
```

command2 is executed if and only if *command1* returns a non-zero exit status. The return status of AND and OR lists is the exit status of the last command executed in the list.

Compound Commands

A *compound command* is one of the following:

- (*list*) *list* is executed in a subshell environment (see **COMMAND EXECUTION ENVIRONMENT** below). Variable assignments and builtin commands that affect the shell's environment do not remain in effect after the command completes. The return status is the exit status of *list*.
- { *list*; } *list* is simply executed in the current shell environment. *list* must be terminated with a newline or semicolon. This is known as a *group command*. The return status is the exit status of *list*. Note that unlike the metacharacters (and), { and } are *reserved words* and must occur where a reserved word is permitted to be recognized. Since they do not cause a word break, they must be separated from *list* by whitespace.
- ((*expression*)) The *expression* is evaluated according to the rules described below under **ARITHMETIC EVALUATION**. If the value of the expression is non-zero, the return status is 0; otherwise the return status is 1. This is exactly equivalent to **let** *expression*.
- [[*expression*]] Return a status of 0 or 1 depending on the evaluation of the conditional expression *expression*. Expressions are composed of the primaries described below under **CONDITIONAL EXPRESSIONS**. Word splitting and pathname expansion are not

performed on the words between the `[[` and `]]`; tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal are performed. Conditional operators such as `-f` must be unquoted to be recognized as primaries.

When the `==` and `!=` operators are used, the string to the right of the operator is considered a pattern and matched according to the rules described below under **Pattern Matching**. The return value is 0 if the string matches or does not match the pattern, respectively, and 1 otherwise. Any part of the pattern may be quoted to force it to be matched as a string.

An additional binary operator, `=`, is available, with the same precedence as `==` and `!=`. When it is used, the string to the right of the operator is considered an extended regular expression and matched accordingly (as in `regex(3)`). The return value is 0 if the string matches the pattern, and 1 otherwise. If the regular expression is syntactically incorrect, the conditional expression's return value is 2. If the shell option **nocaseglob** is enabled, the match is performed without regard to the case of alphabetic characters. Substrings matched by parenthesized subexpressions within the regular expression are saved in the array variable **BASH_REMATCH**. The element of **BASH_REMATCH** with index 0 is the portion of the string matching the entire regular expression. The element of **BASH_REMATCH** with index *n* is the portion of the string matching the *n*th parenthesized subexpression.

Expressions may be combined using the following operators, listed in decreasing order of precedence:

- `(expression)` Returns the value of *expression*. This may be used to override the normal precedence of operators.
- `! expression` True if *expression* is false.
- `expression1 && expression2` True if both *expression1* and *expression2* are true.
- `expression1 || expression2` True if either *expression1* or *expression2* is true.

The `&&` and `||` operators do not evaluate *expression2* if the value of *expression1* is sufficient to determine the return value of the entire conditional expression.

- **for** *name* [**in** *word*] ; **do** *list* ; **done** The list of words following **in** is expanded, generating a list of items. The variable *name* is set to each element of this list in turn, and *list* is executed each time. If the **in** *word* is omitted, the **for** command executes *list* once for each positional parameter that is set (see **PARAMETERS** below). The return status is the exit status of the last command that executes. If the expansion of the items following **in** results in an empty list, no commands are executed, and the return status is 0.
- **for** ((*expr1* ; *expr2* ; *expr3*)) ; **do** *list* ; **done** First, the arithmetic expression *expr1* is evaluated according to the rules described below under **ARITHMETIC EVALUATION**. The arithmetic expression *expr2* is then evaluated repeatedly until it evaluates to zero. Each time *expr2* evaluates to a non-zero value, *list* is executed and the arithmetic expression *expr3* is evaluated. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in *list* that is executed, or false if any of the expressions is invalid.
- **select** *name* [**in** *word*] ; **do** *list* ; **done** The list of words following **in** is expanded, generating a list of items. The set of expanded words is printed on the standard error,

each preceded by a number. If the **in word** is omitted, the positional parameters are printed (see **PARAMETERS** below). The **PS3** prompt is then displayed and a line read from the standard input. If the line consists of a number corresponding to one of the displayed words, then the value of *name* is set to that word. If the line is empty, the words and prompt are displayed again. If EOF is read, the command completes. Any other value read causes *name* to be set to null. The line read is saved in the variable **REPLY**. The *list* is executed after each selection until a **break** command is executed. The exit status of **select** is the exit status of the last command executed in *list*, or zero if no commands were executed.

- **case word in** [*(| pattern [| pattern]*] A **case** command first expands *word*, and tries to match it against each *pattern* in turn, using the same matching rules as for pathname expansion (see **Pathname Expansion** below). When a match is found, the corresponding *list* is executed. After the first match, no subsequent matches are attempted. The exit status is zero if no pattern matches. Otherwise, it is the exit status of the last command executed in *list*.
- **if list; then list; [elif list; then list;] ... else list;] fi** The **if list** is executed. If its exit status is zero, the **then list** is executed. Otherwise, each **elif list** is executed in turn, and if its exit status is zero, the corresponding **then list** is executed and the command completes. Otherwise, the **else list** is executed, if present. The exit status is the exit status of the last command executed, or zero if no condition tested true.
- **while list; do list; done**
- **until list; do list; done** The **while** command continuously executes the **do list** as long as the last command in *list* returns an exit status of zero. The **until** command is identical to the **while** command, except that the test is negated; the **do list** is executed as long as the last command in *list* returns a non-zero exit status. The exit status of the **while** and **until** commands is the exit status of the last **do list** command executed, or zero if none was executed.

Shell Function Definitions

A shell function is an object that is called like a simple command and executes a compound command with a new set of positional parameters. Shell functions are declared as follows:

function *name* () *compound-command* [*redirection*] This defines a function named *name*. The reserved word **function** is optional. If the **function** reserved word is supplied, the parentheses are optional. The *body* of the function is the compound command *compound-command* (see **Compound Commands** above). That command is usually a *list* of commands between { and }, but may be any command listed under **Compound Commands** above. *compound-command* is executed whenever *name* is specified as the name of a simple command. Any redirections (see **REDIRECTION** below) specified when a function is defined are performed when the function is executed. The exit status of a function definition is zero unless a syntax error occurs or a readonly function with the same name already exists. When executed, the exit status of a function is the exit status of the last command executed in the body. (See **FUNCTIONS** below.)

7.14.11 Comments

In a non-interactive shell, or an interactive shell in which the **interactive_comments** option to the **shopt** builtin is enabled (see **SHELL BUILTIN COMMANDS** below), a word beginning with **#** causes that word and all remaining characters on that line to be ignored. An

interactive shell without the **interactive_comments** option enabled does not allow comments. The **interactive_comments** option is on by default in interactive shells.

7.14.12 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion.

Each of the *metacharacters* listed above under **DEFINITIONS** has special meaning to the shell and must be quoted if it is to represent itself.

When the command history expansion facilities are being used, the *history expansion* character, usually **!**, must be quoted to prevent history expansion.

There are three quoting mechanisms: the *escape character*, single quotes, and double quotes.

A non-quoted backslash (****) is the *escape character*. It preserves the literal value of the next character that follows, with the exception of **<newline>**. If a **\<newline>** pair appears, and the backslash is not itself quoted, the **\<newline>** is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

Enclosing characters in single quotes preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

Enclosing characters in double quotes preserves the literal value of all characters within the quotes, with the exception of **\$**, **'**, and ****. The characters **\$** and **'** retain their special meaning within double quotes. The backslash retains its special meaning only when followed by one of the following characters: **\$**, **'**, **,**, ****, or **<newline>**. A double quote may be quoted within double quotes by preceding it with a backslash. When command history is being used, the double quote may not be used to quote the history expansion character.

The special parameters ***** and **@** have special meaning when in double quotes (see **PARAMETERS** below).

Words of the form **\$'string'** are treated specially. The word expands to *string*, with backslash-escaped characters replaced as specified by the ANSI C standard. Backslash escape sequences, if present, are decoded as follows:

- **\a** alert (bell)
- **\b** backspace
- **\e** an escape character
- **\f** form feed
- **\n** new line
- **\r** carriage return
- **\t** horizontal tab
- **\v** vertical tab
- **** backslash
- **\'** single quote
- **\nnn** the eight-bit character whose value is the octal value *nnn* (one to three digits)
- **\xHH** the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)

- `\cx` a control-*x* character

The expanded result is single-quoted, as if the dollar sign had not been present.

A double-quoted string preceded by a dollar sign (\$) will cause the string to be translated according to the current locale. If the current locale is **C** or **POSIX**, the dollar sign is ignored. If the string is translated and replaced, the replacement is double-quoted.

7.14.13 Parameters

A *parameter* is an entity that stores values. It can be a *name*, a number, or one of the special characters listed below under **Special Parameters**. A *variable* is a parameter denoted by a *name*. A variable has a *value* and zero or more *attributes*. Attributes are assigned using the **declare** builtin command (see **declare** below in **SHELL BUILTIN COMMANDS**).

A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the **unset** builtin command (see **SHELL BUILTIN COMMANDS** below).

A *variable* may be assigned to by a statement of the form

```
name=[value]
```

If *value* is not given, the variable is assigned the null string. All *values* undergo tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal (see **EXPANSION** below). If the variable has its **integer** attribute set, then *value* is evaluated as an arithmetic expression even if the `$((...))` expansion is not used (see **Arithmetic Expansion** below). Word splitting is not performed, with the exception of `$@` as explained below under **Special Parameters**. Pathname expansion is not performed. Assignment statements may also appear as arguments to the **alias**, **declare**, **typeset**, **export**, **readonly**, and **local** builtin commands.

Positional Parameters

A *positional* parameter is a parameter denoted by one or more digits, other than the single digit 0. Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the **set** builtin command. Positional parameters may not be assigned to with assignment statements. The positional parameters are temporarily replaced when a shell function is executed (see **FUNCTIONS** below).

When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces (see **EXPANSION** below).

Special Parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

- ***** Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the **IFS** special variable. That is, `$*` is equivalent to `$1c$2c...`, where *c* is the first character of the value of the **IFS** variable. If **IFS** is unset, the parameters are separated by spaces. If **IFS** is null, the parameters are joined without intervening separators.
- **@** Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word. That is, `$@` is equivalent to `$1 $2 ...`. When there are no positional parameters, `$@` and `$*` expand to nothing (i.e., they are removed).

- **#** Expands to the number of positional parameters in decimal.
- **?** Expands to the status of the most recently executed foreground pipeline.
- **-** Expands to the current option flags as specified upon invocation, by the **set** builtin command, or those set by the shell itself (such as the **-i** option).
- **\$** Expands to the process ID of the shell. In a **()** subshell, it expands to the process ID of the current shell, not the subshell.
- **!** Expands to the process ID of the most recently executed background (asynchronous) command.
- **0** Expands to the name of the shell or shell script. This is set at shell initialization. If **bash** is invoked with a file of commands, **\$0** is set to the name of that file. If **bash** is started with the **-c** option, then **\$0** is set to the first argument after the string to be executed, if one is present. Otherwise, it is set to the file name used to invoke **bash**, as given by argument zero.
- **_** At shell startup, set to the absolute file name of the shell or shell script being executed as passed in the argument list. Subsequently, expands to the last argument to the previous command, after expansion. Also set to the full file name of each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file currently being checked.

Shell Variables

The following variables are set by the shell:

- **BASH** Expands to the full file name used to invoke this instance of **bash**.
- **BASH_ARGC** An array variable whose values are the number of parameters in each frame of the current bash execution call stack. The number of parameters to the current subroutine (shell function or script executed with **.** or **source**) is at the top of the stack. When a subroutine is executed, the number of parameters passed is pushed onto **BASH_ARGC**.
- **BASH_ARGV** An array variable containing all of the parameters in the current bash execution call stack. The final parameter of the last subroutine call is at the top of the stack; the first parameter of the initial call is at the bottom. When a subroutine is executed, the parameters supplied are pushed onto **BASH_ARGV**.
- **BASH_COMMAND** The command currently being executed or about to be executed, unless the shell is executing a command as the result of a trap, in which case it is the command executing at the time of the trap.
- **BASH_EXECUTION_STRING** The command argument to the **-c** invocation option.
- **BASH_LINENO** An array variable whose members are the line numbers in source files corresponding to each member of **@var{FUNCNAME}**. **\${BASH_LINENO[\$i]}** is the line number in the source file where **\${FUNCNAME[\$i + 1]}** was called. The corresponding source file name is **\${BASH_SOURCE[\$i + 1]}**. Use **LINENO** to obtain the current line number.

- **BASH_REMATCH** An array variable whose members are assigned by the `=` binary operator to the `[]` conditional command. The element with index 0 is the portion of the string matching the entire regular expression. The element with index n is the portion of the string matching the n th parenthesized subexpression. This variable is read-only.
- **BASH_SOURCE** An array variable whose members are the source filenames corresponding to the elements in the **FUNCNAME** array variable.
- **BASH_SUBSHELL** Incremented by one each time a subshell or subshell environment is spawned. The initial value is 0.
- **BASH_VERSINFO** A readonly array variable whose members hold version information for this instance of **bash**. The values assigned to the array members are as follows:
 - **BASH_VERSINFO[0]** The major version number (the *release*).
 - **BASH_VERSINFO[1]** The minor version number (the *version*).
 - **BASH_VERSINFO[2]** The patch level.
 - **BASH_VERSINFO[3]** The build version.
 - **BASH_VERSINFO[4]** The release status (e.g., *beta1*).
 - **BASH_VERSINFO[5]** The value of **MACHTYPE**.
- **BASH_VERSION** Expands to a string describing the version of this instance of **bash**.
- **COMP_CWORD** An index into `${COMP_WORDS}` of the word containing the current cursor position. This variable is available only in shell functions invoked by the programmable completion facilities (see **Programmable Completion** below).
- **COMP_LINE** The current command line. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see **Programmable Completion** below).
- **COMP_POINT** The index of the current cursor position relative to the beginning of the current command. If the current cursor position is at the end of the current command, the value of this variable is equal to `${#COMP_LINE}`. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see **Programmable Completion** below).
- **COMP_WORDBREAKS** The set of characters that the Readline library treats as word separators when performing word completion. If **COMP_WORDBREAKS** is unset, it loses its special properties, even if it is subsequently reset.
- **COMP_WORDS** An array variable (see **Arrays** below) consisting of the individual words in the current command line. This variable is available only in shell functions invoked by the programmable completion facilities (see **Programmable Completion** below).
- **DIRSTACK** An array variable (see **Arrays** below) containing the current contents of the directory stack. Directories appear in the stack in the order they are displayed by the **dirs** builtin. Assigning to members of this array variable may be used to modify directories already in the stack, but the **pushd** and **popd** builtins must be used to add and remove directories. Assignment to this variable will not change the current directory. If **DIRSTACK** is unset, it loses its special properties, even if it is subsequently reset.

- **EUID** Expands to the effective user ID of the current user, initialized at shell startup. This variable is readonly.
- **FUNCNAME** An array variable containing the names of all shell functions currently in the execution call stack. The element with index 0 is the name of any currently-executing shell function. The bottom-most element is `main`. This variable exists only when a shell function is executing. Assignments to **FUNCNAME** have no effect and return an error status. If **FUNCNAME** is unset, it loses its special properties, even if it is subsequently reset.
- **GROUPS** An array variable containing the list of groups of which the current user is a member. Assignments to **GROUPS** have no effect and return an error status. If **GROUPS** is unset, it loses its special properties, even if it is subsequently reset.
- **HISTCMD** The history number, or index in the history list, of the current command. If **HISTCMD** is unset, it loses its special properties, even if it is subsequently reset.
- **HOSTNAME** Automatically set to the name of the current host.
- **HOSTTYPE** Automatically set to a string that uniquely describes the type of machine on which `bash` is executing. The default is system-dependent.
- **LINENO** Each time this parameter is referenced, the shell substitutes a decimal number representing the current sequential line number (starting with 1) within a script or function. When not in a script or function, the value substituted is not guaranteed to be meaningful. If **LINENO** is unset, it loses its special properties, even if it is subsequently reset.
- **MACHTYPE** Automatically set to a string that fully describes the system type on which `bash` is executing, in the standard GNU *cpu-company-system* format. The default is system-dependent.
- **OLDPWD** The previous working directory as set by the `cd` command.
- **OPTARG** The value of the last option argument processed by the `getopts` builtin command (see **SHELL BUILTIN COMMANDS** below).
- **OPTIND** The index of the next argument to be processed by the `getopts` builtin command (see **SHELL BUILTIN COMMANDS** below).
- **OSTYPE** Automatically set to a string that describes the operating system on which `bash` is executing. The default is system-dependent.
- **PIPESTATUS** An array variable (see **Arrays** below) containing a list of exit status values from the processes in the most-recently-executed foreground pipeline (which may contain only a single command).
- **PPID** The process ID of the shell's parent. This variable is readonly.
- **PWD** The current working directory as set by the `cd` command.
- **RANDOM** Each time this parameter is referenced, a random integer between 0 and 32767 is generated. The sequence of random numbers may be initialized by assigning a value to **RANDOM**. If **RANDOM** is unset, it loses its special properties, even if it is subsequently reset.
- **REPLY** Set to the line of input read by the `read` builtin command when no arguments are supplied.

- **SECONDS** Each time this parameter is referenced, the number of seconds since shell invocation is returned. If a value is assigned to **SECONDS**, the value returned upon subsequent references is the number of seconds since the assignment plus the value assigned. If **SECONDS** is unset, it loses its special properties, even if it is subsequently reset.
- **SHELLOPTS** A colon-separated list of enabled shell options. Each word in the list is a valid argument for the **-o** option to the **set** builtin command (see **SHELL BUILTIN COMMANDS** below). The options appearing in **SHELLOPTS** are those reported as *on* by **set -o**. If this variable is in the environment when **bash** starts up, each shell option in the list will be enabled before reading any startup files. This variable is read-only.
- **SHLVL** Incremented by one each time an instance of **bash** is started.
- **UID** Expands to the user ID of the current user, initialized at shell startup. This variable is readonly.

The following variables are used by the shell. In some cases, **bash** assigns a default value to a variable; these cases are noted below.

- **BASH_ENV** If this parameter is set when **bash** is executing a shell script, its value is interpreted as a filename containing commands to initialize the shell, as in */.bashrc*. The value of **BASH_ENV** is subjected to parameter expansion, command substitution, and arithmetic expansion before being interpreted as a file name. **PATH** is not used to search for the resultant file name.
- **CDPATH** The search path for the **cd** command. This is a colon-separated list of directories in which the shell looks for destination directories specified by the **cd** command. A sample value is *./:/usr*.
- **COLUMNS** Used by the **select** builtin command to determine the terminal width when printing selection lists. Automatically set upon receipt of a SIGWINCH.
- **COMPREPLY** An array variable from which **bash** reads the possible completions generated by a shell function invoked by the programmable completion facility (see **Programmable Completion** below).
- **EMACS** If **bash** finds this variable in the environment when the shell starts with value *t*, it assumes that the shell is running in an emacs shell buffer and disables line editing.
- **FCEDIT** The default editor for the **fc** builtin command.
- **FIGIGNORE** A colon-separated list of suffixes to ignore when performing filename completion (see **READLINE** below). A filename whose suffix matches one of the entries in **FIGIGNORE** is excluded from the list of matched filenames. A sample value is *.o:* (Quoting is needed when assigning a value to this variable, which contains tildes).
- **GLOBIGNORE** A colon-separated list of patterns defining the set of filenames to be ignored by pathname expansion. If a filename matched by a pathname expansion pattern also matches one of the patterns in **GLOBIGNORE**, it is removed from the list of matches.
- **HISTCONTROL** A colon-separated list of values controlling how commands are saved on the history list. If the list of values includes *ignorespace*, lines which begin with a **space** character are not saved in the history list. A value of *ignoredups* causes lines matching the previous history entry to not be saved. A value of *ignoreboth* is shorthand for *ignorespace*

and *ignoredups*. A value of *erasedups* causes all previous lines matching the current line to be removed from the history list before that line is saved. Any value not in the above list is ignored. If **HISTCONTROL** is unset, or does not include a valid value, all lines read by the shell parser are saved on the history list, subject to the value of **HISTIGNORE**. The second and subsequent lines of a multi-line compound command are not tested, and are added to the history regardless of the value of **HISTCONTROL**.

- **HISTFILE** The name of the file in which command history is saved (see **HISTORY** below). The default value is */.bash_history*. If unset, the command history is not saved when an interactive shell exits.
- **HISTFILESIZE** The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated, if necessary, to contain no more than that number of lines. The default value is 500. The history file is also truncated to this size after writing it when an interactive shell exits.
- **HISTIGNORE** A colon-separated list of patterns used to decide which command lines should be saved on the history list. Each pattern is anchored at the beginning of the line and must match the complete line (no implicit ***** is appended). Each pattern is tested against the line after the checks specified by **HISTCONTROL** are applied. In addition to the normal shell pattern matching characters, **&** matches the previous history line. **&** may be escaped using a backslash; the backslash is removed before attempting a match. The second and subsequent lines of a multi-line compound command are not tested, and are added to the history regardless of the value of **HISTIGNORE**.
- **HISTSIZE** The number of commands to remember in the command history (see **HISTORY** below). The default value is 500.
- **HISTTIMEFORMAT** If this variable is set and not null, its value is used as a format string for *strftime(3)* to print the time stamp associated with each history entry displayed by the **history** builtin. If this variable is set, time stamps are written to the history file so they may be preserved across shell sessions.
- **HOME** The home directory of the current user; the default argument for the **cd** builtin command. The value of this variable is also used when performing tilde expansion.
- **HOSTFILE** Contains the name of a file in the same format as */etc/hosts* that should be read when the shell needs to complete a hostname. The list of possible hostname completions may be changed while the shell is running; the next time hostname completion is attempted after the value is changed, **bash** adds the contents of the new file to the existing list. If **HOSTFILE** is set, but has no value, **bash** attempts to read */etc/hosts* to obtain the list of possible hostname completions. When **HOSTFILE** is unset, the hostname list is cleared.
- **IFS** The *Internal Field Separator* that is used for word splitting after expansion and to split lines into words with the **read** builtin command. The default value is “<space><tab><newline>”.
- **IGNOREEOF** Controls the action of an interactive shell on receipt of an **EOF** character as the sole input. If set, the value is the number of consecutive **EOF** characters which must be typed as the first characters on an input line before **bash** exits. If the variable exists but does not have a numeric value, or has no value, the default value is 10. If it does not exist, **EOF** signifies the end of input to the shell.
- **INPUTRC** The filename for the **readline** startup file, overriding the default of */.inputrc* (see **READLINE** below).

- **LANG** Used to determine the locale category for any category not specifically selected with a variable starting with **LC_**.
- **LC_ALL** This variable overrides the value of **LANG** and any other **LC_** variable specifying a locale category.
- **LC_COLLATE** This variable determines the collation order used when sorting the results of pathname expansion, and determines the behavior of range expressions, equivalence classes, and collating sequences within pathname expansion and pattern matching.
- **LC_CTYPE** This variable determines the interpretation of characters and the behavior of character classes within pathname expansion and pattern matching.
- **LC_MESSAGES** This variable determines the locale used to translate double-quoted strings preceded by a **\$**.
- **LC_NUMERIC** This variable determines the locale category used for number formatting.
- **LINES** Used by the **select** builtin command to determine the column length for printing selection lists. Automatically set upon receipt of a SIGWINCH.
- **MAIL** If this parameter is set to a file name and the **MAILPATH** variable is not set, **bash** informs the user of the arrival of mail in the specified file.
- **MAILCHECK** Specifies how often (in seconds) **bash** checks for mail. The default is 60 seconds. When it is time to check for mail, the shell does so before displaying the primary prompt. If this variable is unset, or set to a value that is not a number greater than or equal to zero, the shell disables mail checking.
- **MAILPATH** A colon-separated list of file names to be checked for mail. The message to be printed when mail arrives in a particular file may be specified by separating the file name from the message with a **'?'**. When used in the text of the message, **\$_** expands to the name of the current mailfile. Example:

```
MAILPATH='/var/mail/bfox?You have mail: /shell-mail?$_ has mail'
```

Bash supplies a default value for this variable, but the location of the user mail files that it uses is system dependent (e.g., **/var/mail/\$USER**).

- **OPTERR** If set to the value 1, **bash** displays error messages generated by the **getopts** builtin command (see **SHELL BUILTIN COMMANDS** below). **OPTERR** is initialized to 1 each time the shell is invoked or a shell script is executed.
- **PATH** The search path for commands. It is a colon-separated list of directories in which the shell looks for commands (see **COMMAND EXECUTION** below). A zero-length (null) directory name in the value of **PATH** indicates the current directory. A null directory name may appear as two adjacent colons, or as an initial or trailing colon. The default path is system-dependent, and is set by the administrator who installs **bash**. A common value is **"/usr/gnu/bin:/usr/local/bin:/usr/ucb/bin:/usr/bin"**.
- **POSIXLY_CORRECT** If this variable is in the environment when **bash** starts, the shell enters *posix mode* before reading the startup files, as if the **-posix** invocation option had been supplied. If it is set while the shell is running, **bash** enables *posix mode*, as if the command **set -o posix** had been executed.
- **PROMPT_COMMAND** If set, the value is executed as a command prior to issuing each primary prompt.

- **PS1** The value of this parameter is expanded (see **PROMPTING** below) and used as the primary prompt string. The default value is “\s-\v\\$ ”.
- **PS2** The value of this parameter is expanded as with **PS1** and used as the secondary prompt string. The default is “> ”.
- **PS3** The value of this parameter is used as the prompt for the **select** command (see **SHELL GRAMMAR** above).
- **PS4** The value of this parameter is expanded as with **PS1** and the value is printed before each command **bash** displays during an execution trace. The first character of **PS4** is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is “+ ”.
- **SHELL** The full pathname to the shell is kept in this environment variable. If it is not set when the shell starts, **bash** assigns to it the full pathname of the current user’s login shell.
- **TIMEFORMAT** The value of this parameter is used as a format string specifying how the timing information for pipelines prefixed with the **time** reserved word should be displayed. The % character introduces an escape sequence that is expanded to a time value or other information. The escape sequences and their meanings are as follows; the braces denote optional portions.
 - %% A literal %.
 - %[p][l]R The elapsed time in seconds.
 - %[p][l]U The number of CPU seconds spent in user mode.
 - %[p][l]S The number of CPU seconds spent in system mode.
 - %P The CPU percentage, computed as (%U + %S) / %R.
- The optional *p* is a digit specifying the *precision*, the number of fractional digits after a decimal point. A value of 0 causes no decimal point or fraction to be output. At most three places after the decimal point may be specified; values of *p* greater than 3 are changed to 3. If *p* is not specified, the value 3 is used.
- The optional *l* specifies a longer format, including minutes, of the form *MMm.SS.FFs*. The value of *p* determines whether or not the fraction is included.
- If this variable is not set, **bash** acts as if it had the value \$'\nreal\t%3lR\nuser\t%3lU\nsyst%3lS'. If the value is null, no timing information is displayed. A trailing newline is added when the format string is displayed.
- **TMOU** If set to a value greater than zero, **TMOU** is treated as the default timeout for the **read** builtin. The **select** command terminates if input does not arrive after **TMOU** seconds when input is coming from a terminal. In an interactive shell, the value is interpreted as the number of seconds to wait for input after issuing the primary prompt. **Bash** terminates after waiting for that number of seconds if input does not arrive.
- **auto_resume** This variable controls how the shell interacts with the user and job control. If this variable is set, single word simple commands without redirections are treated as candidates for resumption of an existing stopped job. There is no ambiguity allowed; if there is more than one job beginning with the string typed, the job most recently accessed

is selected. The *name* of a stopped job, in this context, is the command line used to start it. If set to the value *exact*, the string supplied must match the name of a stopped job exactly; if set to *substring*, the string supplied needs to match a substring of the name of a stopped job. The *substring* value provides functionality analogous to the `%?` job identifier (see **JOB CONTROL** below). If set to any other value, the supplied string must be a prefix of a stopped job's name; this provides functionality analogous to the `%` job identifier.

- **command_not_found_handle** The name of a shell function to be called if a command cannot be found. The return value of this function should be 0, if the command is available after execution of the function, otherwise 127 (EX_NOTFOUND). Enabled only in interactive, non POSIX mode shells. This is a Debian extension.
- **histchars** The two or three characters which control history expansion and tokenization (see **HISTORY EXPANSION** below). The first character is the *history expansion* character, the character which signals the start of a history expansion, normally '!'. The second character is the *quick substitution* character, which is used as shorthand for re-running the previous command entered, substituting one string for another in the command. The default is '^'. The optional third character is the character which indicates that the remainder of the line is a comment when found as the first character of a word, normally '#'. The history comment character causes history substitution to be skipped for the remaining words on the line. It does not necessarily cause the shell parser to treat the rest of the line as a comment.

Arrays

Bash provides one-dimensional array variables. Any variable may be used as an array; the **declare** builtin will explicitly declare an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously. Arrays are indexed using integers and are zero-based.

An array is created automatically if any variable is assigned to using the syntax `name[subscript]=value`. The *subscript* is treated as an arithmetic expression that must evaluate to a number greater than or equal to zero. To explicitly declare an array, use **declare -a name** (see **SHELL BUILTIN COMMANDS** below). **declare -a name[subscript]** is also accepted; the *subscript* is ignored. Attributes may be specified for an array variable using the **declare** and **readonly** builtins. Each attribute applies to all members of an array.

Arrays are assigned to using compound assignments of the form `name=(value1 ... valuen)`, where each *value* is of the form `[subscript]=string`. Only *string* is required. If the optional brackets and subscript are supplied, that index is assigned to; otherwise the index of the element assigned is the last index assigned to by the statement plus one. Indexing starts at zero. This syntax is also accepted by the **declare** builtin. Individual array elements may be assigned to using the `name[subscript]=value` syntax introduced above.

Any element of an array may be referenced using `${name[subscript]}`. The braces are required to avoid conflicts with pathname expansion. If *subscript* is `@` or `*`, the word expands to all members of *name*. These subscripts differ only when the word appears within double quotes. If the word is double-quoted, `${name[*]}` expands to a single word with the value of each array member separated by the first character of the **IFS** special variable, and `${name[@]}` expands each element of *name* to a separate word. When there are no array members, `${name[@]}` expands to nothing. This is analogous to the expansion of the special parameters `*` and `@` (see **Special Parameters** above). `${#name[subscript]}` expands to the length of `${name[subscript]}`. If *subscript* is `*` or `@`, the expansion is the number of elements in the array. Referencing an array variable without a subscript is equivalent to referencing element zero.

The **unset** builtin is used to destroy arrays. **unset** *name*[*subscript*] destroys the array element at index *subscript*. **unset** *name*, where *name* is an array, or **unset** *name*[*subscript*], where *subscript* is ***** or **@**, removes the entire array.

The **declare**, **local**, and **readonly** builtins each accept a **-a** option to specify an array. The **read** builtin accepts a **-a** option to assign a list of words read from the standard input to an array. The **set** and **declare** builtins display array values in a way that allows them to be reused as assignments.

7.14.14 Expansion

Expansion is performed on the command line after it has been split into words. There are seven kinds of expansion performed: *brace expansion*, *tilde expansion*, *parameter and variable expansion*, *command substitution*, *arithmetic expansion*, *word splitting*, and *pathname expansion*.

The order of expansions is: brace expansion, tilde expansion, parameter, variable and arithmetic expansion and command substitution (done in a left-to-right fashion), word splitting, and pathname expansion.

On systems that can support it, there is an additional expansion available: *process substitution*.

Only brace expansion, word splitting, and pathname expansion can change the number of words of the expansion; other expansions expand a single word to a single word. The only exceptions to this are the expansions of **\$@** and **\${name[@]}** as explained above (see **PARAMETERS**).

Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to *pathname expansion*, but the filenames generated need not exist. Patterns to be brace expanded take the form of an optional *preamble*, followed by either a series of comma-separated strings or a sequence expression between a pair of braces, followed by an optional *postscript*. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example, **a{d,c,b}e** expands into ‘ade ace abe’.

A sequence expression takes the form **{x.y}**, where *x* and *y* are either integers or single characters. When integers are supplied, the expression expands to each number between *x* and *y*, inclusive. When characters are supplied, the expression expands to each character lexicographically between *x* and *y*, inclusive. Note that both *x* and *y* must be of the same type.

Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. It is strictly textual. **Bash** does not apply any syntactic interpretation to the context of the expansion or the text between the braces.

A correctly-formed brace expansion must contain unquoted opening and closing braces, and at least one unquoted comma or a valid sequence expression. Any incorrectly formed brace expansion is left unchanged. A **{** or **,** may be quoted with a backslash to prevent its being considered part of a brace expression. To avoid conflicts with parameter expansion, the string **\${** is not considered eligible for brace expansion.

This construct is typically used as shorthand when the common prefix of the strings to be generated is longer than in the above example:

```
mkdir /usr/local/src/bash/{old,new,dist,bugs} or chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

Brace expansion introduces a slight incompatibility with historical versions of **sh**. **sh** does not treat opening or closing braces specially when they appear as part of a word, and preserves them in the output. **Bash** removes braces from words as a consequence of brace expansion. For

example, a word entered to **sh** as *file{1,2}* appears identically in the output. The same word is output as *file1 file2* after expansion by **bash**. If strict compatibility with **sh** is desired, start **bash** with the **+B** option or disable brace expansion with the **+B** option to the **set** command (see **SHELL BUILTIN COMMANDS** below).

Tilde Expansion

If a word begins with an unquoted tilde character (‘~’), all of the characters preceding the first unquoted slash (or all characters, if there is no unquoted slash) are considered a *tilde-prefix*. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible *login name*. If this login name is the null string, the tilde is replaced with the value of the shell parameter **HOME**. If **HOME** is unset, the home directory of the user executing the shell is substituted instead. Otherwise, the tilde-prefix is replaced with the home directory associated with the specified login name.

If the tilde-prefix is a ‘+’, the value of the shell variable **PWD** replaces the tilde-prefix. If the tilde-prefix is a ‘-’, the value of the shell variable **OLDPWD**, if it is set, is substituted. If the characters following the tilde in the tilde-prefix consist of a number *N*, optionally prefixed by a ‘+’ or a ‘-’, the tilde-prefix is replaced with the corresponding element from the directory stack, as it would be displayed by the **dirs** builtin invoked with the tilde-prefix as an argument. If the characters following the tilde in the tilde-prefix consist of a number without a leading ‘+’ or ‘-’, ‘+’ is assumed.

If the login name is invalid, or the tilde expansion fails, the word is unchanged.

Each variable assignment is checked for unquoted tilde-prefixes immediately following a **:** or **=**. In these cases, tilde expansion is also performed. Consequently, one may use file names with tildes in assignments to **PATH**, **MAILPATH**, and **CDPATH**, and the shell assigns the expanded value.

Parameter Expansion

The ‘\$’ character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

When braces are used, the matching ending brace is the first ‘}’ not escaped by a backslash or within a quoted string, and not within an embedded arithmetic expansion, command substitution, or parameter expansion.

- $\${parameter}$ The value of *parameter* is substituted. The braces are required when *parameter* is a positional parameter with more than one digit, or when *parameter* is followed by a character which is not to be interpreted as part of its name.

If the first character of *parameter* is an exclamation point, a level of variable indirection is introduced. **Bash** uses the value of the variable formed from the rest of *parameter* as the name of the variable; this variable is then expanded and that value is used in the rest of the substitution, rather than the value of *parameter* itself. This is known as *indirect expansion*. The exceptions to this are the expansions of $\${!prefix*}$ and $\${!name[@]}$ described below. The exclamation point must immediately follow the left brace in order to introduce indirection.

In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. When not performing substring expansion, **bash** tests for a parameter that is unset or null; omitting the colon results in a test only for a parameter that is unset.

- $\${parameter:-word}$ **Use Default Values.** If *parameter* is unset or null, the expansion of *word* is substituted. Otherwise, the value of *parameter* is substituted.
- $\${parameter:=word}$ **Assign Default Values.** If *parameter* is unset or null, the expansion of *word* is assigned to *parameter*. The value of *parameter* is then substituted. Positional parameters and special parameters may not be assigned to in this way.
- $\${parameter:?word}$ **Display Error if Null or Unset.** If *parameter* is null or unset, the expansion of *word* (or a message to that effect if *word* is not present) is written to the standard error and the shell, if it is not interactive, exits. Otherwise, the value of *parameter* is substituted.
- $\${parameter:+word}$ **Use Alternate Value.** If *parameter* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted.
- $\${parameter:offset}$
- $\${parameter:offset:length}$ **Substring Expansion.** Expands to up to *length* characters of *parameter* starting at the character specified by *offset*. If *length* is omitted, expands to the substring of *parameter* starting at the character specified by *offset*. *length* and *offset* are arithmetic expressions (see ARITHMETIC EVALUATION below). *length* must evaluate to a number greater than or equal to zero. If *offset* evaluates to a number less than zero, the value is used as an offset from the end of the value of *parameter*. Arithmetic expressions starting with a - must be separated by whitespace from the preceding : to be distinguished from the **Use Default Values** expansion. If *parameter* is @, the result is *length* positional parameters beginning at *offset*. If *parameter* is an array name indexed by @ or *, the result is the *length* members of the array beginning with $\${parameter[offset]}$. Substring indexing is zero-based unless the positional parameters are used, in which case the indexing starts at 1.
- $\${!prefix*}$
- $\${!prefix@}$ Expands to the names of variables whose names begin with *prefix*, separated by the first character of the **IFS** special variable.
- $\${!name[@]}$
- $\${!name[*]}$ If *name* is an array variable, expands to the list of array indices (keys) assigned in *name*. If *name* is not an array, expands to 0 if *name* is set and null otherwise. When @ is used and the expansion appears within double quotes, each key expands to a separate word.
- $\${#parameter}$ The length in characters of the value of *parameter* is substituted. If *parameter* is * or @, the value substituted is the number of positional parameters. If *parameter* is an array name subscripted by * or @, the value substituted is the number of elements in the array.
- $\${parameter#word}$
- $\${parameter##word}$ The *word* is expanded to produce a pattern just as in pathname expansion. If the pattern matches the beginning of the value of *parameter*, then the result of the expansion is the expanded value of *parameter* with the shortest matching pattern (the “#” case) or the longest matching pattern (the “##” case) deleted. If *parameter* is @ or *, the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with

@ or *, the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

- $\${parameter}\%word$
- $\${parameter}\%\%word$ The *word* is expanded to produce a pattern just as in pathname expansion. If the pattern matches a trailing portion of the expanded value of *parameter*, then the result of the expansion is the expanded value of *parameter* with the shortest matching pattern (the “%” case) or the longest matching pattern (the “%%” case) deleted. If *parameter* is @ or *, the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with @ or *, the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.
- $\${parameter}/pattern/string$
- $\${parameter}/pattern/string$ The *pattern* is expanded to produce a pattern just as in pathname expansion. *Parameter* is expanded and the longest match of *pattern* against its value is replaced with *string*. In the first form, only the first match is replaced. The second form causes all matches of *pattern* to be replaced with *string*. If *pattern* begins with #, it must match at the beginning of the expanded value of *parameter*. If *pattern* begins with %, it must match at the end of the expanded value of *parameter*. If *string* is null, matches of *pattern* are deleted and the / following *pattern* may be omitted. If *parameter* is @ or *, the substitution operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with @ or *, the substitution operation is applied to each member of the array in turn, and the expansion is the resultant list.

Command Substitution

Command substitution allows the output of a command to replace the command name. There are two forms:

$\$(command)$ or ‘*command*’

Bash performs the expansion by executing *command* and replacing the command substitution with the standard output of the command, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed during word splitting. The command substitution $\$(cat\ file)$ can be replaced by the equivalent but faster $\$(<\ file)$.

When the old-style backquote form of substitution is used, backslash retains its literal meaning except when followed by \$, ‘, or \. The first backquote not preceded by a backslash terminates the command substitution. When using the $\$(command)$ form, all characters between the parentheses make up the command; none are treated specially.

Command substitutions may be nested. To nest when using the backquoted form, escape the inner backquotes with backslashes.

If the substitution appears within double quotes, word splitting and pathname expansion are not performed on the results.

Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

$\$((expression))$

The old format $\$[expression]$ is deprecated and will be removed in upcoming versions of bash.

The *expression* is treated as if it were within double quotes, but a double quote inside the parentheses is not treated specially. All tokens in the expression undergo parameter expansion, string expansion, command substitution, and quote removal. Arithmetic expansions may be nested.

The evaluation is performed according to the rules listed below under **ARITHMETIC EVALUATION**. If *expression* is invalid, **bash** prints a message indicating failure and no substitution occurs.

Process Substitution

Process substitution is supported on systems that support named pipes (*FIFOs*) or the `/dev/fd` method of naming open files. It takes the form of `<(list)` or `>(list)`. The process *list* is run with its input or output connected to a *FIFO* or some file in `/dev/fd`. The name of this file is passed as an argument to the current command as the result of the expansion. If the `>(list)` form is used, writing to the file will provide input for *list*. If the `<(list)` form is used, the file passed as an argument should be read to obtain the output of *list*.

When available, process substitution is performed simultaneously with parameter and variable expansion, command substitution, and arithmetic expansion.

Word Splitting

The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for *word splitting*.

The shell treats each character of **IFS** as a delimiter, and splits the results of the other expansions into words on these characters. If **IFS** is unset, or its value is exactly `<space><tab><newline>`, the default, then any sequence of **IFS** characters serves to delimit words. If **IFS** has a value other than the default, then sequences of the whitespace characters **space** and **tab** are ignored at the beginning and end of the word, as long as the whitespace character is in the value of **IFS** (an **IFS** whitespace character). Any character in **IFS** that is not **IFS** whitespace, along with any adjacent **IFS** whitespace characters, delimits a field. A sequence of **IFS** whitespace characters is also treated as a delimiter. If the value of **IFS** is null, no word splitting occurs.

Explicit null arguments (“ or ”) are retained. Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed. If a parameter with no value is expanded within double quotes, a null argument results and is retained.

Note that if no expansion occurs, no splitting is performed.

Pathname Expansion

After word splitting, unless the **-f** option has been set, **bash** scans each word for the characters *****, **?**, and **[**. If one of these characters appears, then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of file names matching the pattern. If no matching file names are found, and the shell option **nullglob** is disabled, the word is left unchanged. If the **nullglob** option is set, and no matches are found, the word is removed. If the **failglob** shell option is set, and no matches are found, an error message is printed and the command is not executed. If the shell option **nocaseglob** is enabled, the match is performed without regard to the case of alphabetic characters. Note that when using range expressions like `[a-z]` (see below), letters of the other case may be included, depending on the setting of **LC_COLLATE**. When a pattern is used for pathname expansion, the character “.” at the start of a name or immediately following a slash must be matched explicitly, unless the shell option **dotglob** is set. When matching a pathname, the slash character must always be matched explicitly. In other cases, the “.” character is not treated specially. See the description of **shopt** below under

SHELL BUILTIN COMMANDS for a description of the **nocaseglob**, **nullglob**, **failglob**, and **dotglob** shell options.

The **GLOBIGNORE** shell variable may be used to restrict the set of file names matching a *pattern*. If **GLOBIGNORE** is set, each matching file name that also matches one of the patterns in **GLOBIGNORE** is removed from the list of matches. The file names “.” and “..” are always ignored when **GLOBIGNORE** is set and not null. However, setting **GLOBIGNORE** to a non-null value has the effect of enabling the **dotglob** shell option, so all other file names beginning with a “.” will match. To get the old behavior of ignoring file names beginning with a “.”, make “.*” one of the patterns in **GLOBIGNORE**. The **dotglob** option is disabled when **GLOBIGNORE** is unset.

Pattern Matching

Any character that appears in a pattern, other than the special pattern characters described below, matches itself. The NUL character may not occur in a pattern. A backslash escapes the following character; the escaping backslash is discarded when matching. The special pattern characters must be quoted if they are to be matched literally.

The special pattern characters have the following meanings:

- * Matches any string, including the null string.
- ? Matches any single character.
- [...] Matches any one of the enclosed characters. A pair of characters separated by a hyphen denotes a *range expression*; any character that sorts between those two characters, inclusive, using the current locale’s collating sequence and character set, is matched. If the first character following the [is a ! or a ^ then any character not enclosed is matched. The sorting order of characters in range expressions is determined by the current locale and the value of the **LC_COLLATE** shell variable, if set. A - may be matched by including it as the first or last character in the set. A] may be matched by including it as the first character in the set.

Within [and], *character classes* can be specified using the syntax **[:class:]**, where *class* is one of the following classes defined in the POSIX.2 standard:

alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit

A character class matches any character belonging to that class. The **word** character class matches letters, digits, and the character _.

Within [and], an *equivalence class* can be specified using the syntax **[=c=]**, which matches all characters with the same collation weight (as defined by the current locale) as the character *c*.

Within [and], the syntax **[.symbol.]** matches the collating symbol *symbol*.

If the **extglob** shell option is enabled using the **shopt** builtin, several extended pattern matching operators are recognized. In the following description, a *pattern-list* is a list of one or more patterns separated by a |. Composite patterns may be formed using one or more of the following sub-patterns:

- **?(pattern-list)** Matches zero or one occurrence of the given patterns
- ***(pattern-list)** Matches zero or more occurrences of the given patterns
- **+(pattern-list)** Matches one or more occurrences of the given patterns
- **@(pattern-list)** Matches exactly one of the given patterns
- **!(pattern-list)**

Matches anything except one of the given patterns

Quote Removal

After the preceding expansions, all unquoted occurrences of the characters `\`, `'`, and ``` that did not result from one of the above expansions are removed.

7.14.15 Redirection

Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment. The following redirection operators may precede or appear anywhere within a *simple* command or may follow a *command*. Redirections are processed in the order they appear, from left to right.

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is `<`, the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is `>`, the redirection refers to the standard output (file descriptor 1).

The word following the redirection operator in the following descriptions, unless otherwise noted, is subjected to brace expansion, tilde expansion, parameter expansion, command substitution, arithmetic expansion, quote removal, pathname expansion, and word splitting. If it expands to more than one word, **bash** reports an error.

Note that the order of redirections is significant. For example, the command

```
ls > dirlist 2>&1
```

directs both standard output and standard error to the file *dirlist*, while the command

```
ls 2>&1 > dirlist
```

directs only the standard output to file *dirlist*, because the standard error was duplicated as standard output before the standard output was redirected to *dirlist*.

Bash handles several filenames specially when they are used in redirections, as described in the following table:

- **/dev/fd/fd** If *fd* is a valid integer, file descriptor *fd* is duplicated.
- **/dev/stdin** File descriptor 0 is duplicated.
- **/dev/stdout** File descriptor 1 is duplicated.
- **/dev/stderr** File descriptor 2 is duplicated.
- **/dev/tcp/host/port** If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, **bash** attempts to open a TCP connection to the corresponding socket.
- **/dev/udp/host/port** If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, **bash** attempts to open a UDP connection to the corresponding socket.

A failure to open or create a file causes the redirection to fail.

Redirecting Input

Redirection of input causes the file whose name results from the expansion of *word* to be opened for reading on file descriptor *n*, or the standard input (file descriptor 0) if *n* is not specified.

The general format for redirecting input is:

```
[n]<word
```

Redirecting Output

Redirection of output causes the file whose name results from the expansion of *word* to be opened for writing on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size.

The general format for redirecting output is:

```
[n]>word
```

If the redirection operator is `>`, and the **noclobber** option to the **set** builtin has been enabled, the redirection will fail if the file whose name results from the expansion of *word* exists and is a regular file. If the redirection operator is `>|`, or the redirection operator is `>` and the **noclobber** option to the **set** builtin command is not enabled, the redirection is attempted even if the file named by *word* exists.

Appending Redirected Output

Redirection of output in this fashion causes the file whose name results from the expansion of *word* to be opened for appending on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created.

The general format for appending output is:

```
[n]>>word
```

Redirecting Standard Output and Standard Error

Bash allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be redirected to the file whose name is the expansion of *word* with this construct.

There are two formats for redirecting standard output and standard error:

```
&&>word and >&word
```

Of the two forms, the first is preferred. This is semantically equivalent to `>word 2>&1`

Here Documents

This type of redirection instructs the shell to read input from the current source until a line containing only *word* (with no trailing blanks) is seen. All of the lines read up to that point are then used as the standard input for a command.

The format of here-documents is:

```
<<[-]word
here-document
delimiter
```

No parameter expansion, command substitution, arithmetic expansion, or pathname expansion is performed on *word*. If any characters in *word* are quoted, the *delimiter* is the result of quote removal on *word*, and the lines in the here-document are not expanded. If *word* is unquoted, all lines of the here-document are subjected to parameter expansion, command substitution, and arithmetic expansion. In the latter case, the character sequence `\<newline>` is ignored, and `\` must be used to quote the characters `\`, `$`, and `'`.

If the redirection operator is `<<-`, then all leading tab characters are stripped from input lines and the line containing *delimiter*. This allows here-documents within shell scripts to be indented in a natural fashion.

Here Strings

A variant of here documents, the format is:

<<<*word*

The *word* is expanded and supplied to the command on its standard input.

Duplicating File Descriptors

The redirection operator

`[n]<&word`

is used to duplicate input file descriptors. If *word* expands to one or more digits, the file descriptor denoted by *n* is made to be a copy of that file descriptor. If the digits in *word* do not specify a file descriptor open for input, a redirection error occurs. If *word* evaluates to -, file descriptor *n* is closed. If *n* is not specified, the standard input (file descriptor 0) is used.

The operator

`[n]>&word`

is used similarly to duplicate output file descriptors. If *n* is not specified, the standard output (file descriptor 1) is used. If the digits in *word* do not specify a file descriptor open for output, a redirection error occurs. As a special case, if *n* is omitted, and *word* does not expand to one or more digits, the standard output and standard error are redirected as described previously.

Moving File Descriptors

The redirection operator

`[n]<&digit-`

moves the file descriptor *digit* to file descriptor *n*, or the standard input (file descriptor 0) if *n* is not specified. *digit* is closed after being duplicated to *n*.

Similarly, the redirection operator

`[n]>&digit-`

moves the file descriptor *digit* to file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified.

Opening File Descriptors for Reading and Writing

The redirection operator

`[n]<>word`

causes the file whose name is the expansion of *word* to be opened for both reading and writing on file descriptor *n*, or on file descriptor 0 if *n* is not specified. If the file does not exist, it is created.

7.14.16 Aliases

Aliases allow a string to be substituted for a word when it is used as the first word of a simple command. The shell maintains a list of aliases that may be set and unset with the **alias** and **unalias** builtin commands (see **SHELL BUILTIN COMMANDS** below). The first word of each simple command, if unquoted, is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The characters /, \$, ', and = and any of the shell *metacharacters* or quoting characters listed above may not appear in an alias name. The replacement text may contain any valid shell input, including shell metacharacters. The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias **ls** to **ls -F**, for instance, and **bash** does not try to recursively expand the replacement text. If the last character of the alias value is a *blank*, then the next command word following the alias is also checked for alias expansion.

Aliases are created and listed with the **alias** command, and removed with the **unalias** command.

There is no mechanism for using arguments in the replacement text. If arguments are needed, a shell function should be used (see **FUNCTIONS** below).

Aliases are not expanded when the shell is not interactive, unless the **expand_aliases** shell option is set using **shopt** (see the description of **shopt** under **SHELL BUILTIN COMMANDS** below).

The rules concerning the definition and use of aliases are somewhat confusing. **Bash** always reads at least one complete line of input before executing any of the commands on that line. Aliases are expanded when a command is read, not when it is executed. Therefore, an alias definition appearing on the same line as another command does not take effect until the next line of input is read. The commands following the alias definition on that line are not affected by the new alias. This behavior is also an issue when functions are executed. Aliases are expanded when a function definition is read, not when the function is executed, because a function definition is itself a compound command. As a consequence, aliases defined in a function are not available until after that function is executed. To be safe, always put alias definitions on a separate line, and do not use **alias** in compound commands.

For almost every purpose, aliases are superseded by shell functions.

7.14.17 Functions

A shell function, defined as described above under **SHELL GRAMMAR**, stores a series of commands for later execution. When the name of a shell function is used as a simple command name, the list of commands associated with that function name is executed. Functions are executed in the context of the current shell; no new process is created to interpret them (contrast this with the execution of a shell script). When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter **#** is updated to reflect the change. Special parameter **0** is unchanged. The first element of the **FUNCNAME** variable is set to the name of the function while the function is executing. All other aspects of the shell execution environment are identical between a function and its caller with the exception that the **DEBUG** trap (see the description of the **trap** builtin under **SHELL BUILTIN COMMANDS** below) is not inherited unless the function has been given the **trace** attribute (see the description of the **declare** builtin below) or the **-o functrace** shell option has been enabled with the **set** builtin (in which case all functions inherit the **DEBUG** trap).

Variables local to the function may be declared with the **local** builtin command. Ordinarily, variables and their values are shared between the function and its caller.

If the builtin command **return** is executed in a function, the function completes and execution resumes with the next command after the function call. Any command associated with the **RETURN** trap is executed before execution resumes. When a function completes, the values of the positional parameters and the special parameter **#** are restored to the values they had prior to the function's execution.

Function names and definitions may be listed with the **-f** option to the **declare** or **typeset** builtin commands. The **-F** option to **declare** or **typeset** will list the function names only (and optionally the source file and line number, if the **extdebug** shell option is enabled). Functions may be exported so that subshells automatically have them defined with the **-f** option to the **export** builtin. Note that shell functions and variables with the same name may result in multiple identically-named entries in the environment passed to the shell's children. Care should be taken in cases where this may cause a problem.

Functions may be recursive. No limit is imposed on the number of recursive calls.

7.14.18 Arithmetic Evaluation

The shell allows arithmetic expressions to be evaluated, under certain circumstances (see the **let** and **declare** builtin commands and **Arithmetic Expansion**). Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

- *id*++ *id*-- variable post-increment and post-decrement
- ++*id* --*id* variable pre-increment and pre-decrement
- - + unary minus and plus
- ! logical and bitwise negation
- ** exponentiation
- * / % multiplication, division, remainder
- + - addition, subtraction
- << >> left and right bitwise shifts
- <= >= < > comparison
- == != equality and inequality
- & bitwise AND
- ^ bitwise exclusive OR
- | bitwise OR
- && logical AND
- || logical OR
- *expr*?*expr*:*expr* conditional operator
- = *= /= %= += -= <<= >>= &= ^= |= assignment
- *expr1* , *expr2* comma

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. A shell variable that is null or unset evaluates to 0 when referenced by name without using the parameter expansion syntax. The value of a variable is evaluated as an arithmetic expression when it is referenced, or when a variable which has been given the *integer* attribute using **declare -i** is assigned a value. A null value evaluates to 0. A shell variable need not have its integer attribute turned on to be used in an expression.

Constants with a leading 0 are interpreted as octal numbers. A leading 0x or 0X denotes hexadecimal. Otherwise, numbers take the form [*base*#]n, where *base* is a decimal number between 2 and 64 representing the arithmetic base, and *n* is a number in that base. If *base*# is omitted, then base 10 is used. The digits greater than 9 are represented by the lowercase letters, the uppercase letters, @, and _, in that order. If *base* is less than or equal to 36, lowercase and uppercase letters may be used interchangeably to represent numbers between 10 and 35.

Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

7.14.19 Conditional Expressions

Conditional expressions are used by the `[[` compound command and the `test` and `[` builtin commands to test file attributes and perform string and arithmetic comparisons. Expressions are formed from the following unary or binary primaries. If any *file* argument to one of the primaries is of the form `/dev/fd/n`, then file descriptor *n* is checked. If the *file* argument to one of the primaries is one of `/dev/stdin`, `/dev/stdout`, or `/dev/stderr`, file descriptor 0, 1, or 2, respectively, is checked. All expressions taking a *file* argument except `-h` and `-L` are acting on the target of the symbolic link, not on the symlink itself, if *file* is a symbolic link.

See the description of the `test` builtin command (section SHELL BUILTIN COMMANDS below) for the handling of parameters (i.e. missing parameters).

- `-a file` True if *file* exists.
- `-b file` True if *file* exists and is a block special file.
- `-c file` True if *file* exists and is a character special file.
- `-d file` True if *file* exists and is a directory.
- `-e file` True if *file* exists.
- `-f file` True if *file* exists and is a regular file.
- `-g file` True if *file* exists and is set-group-id.
- `-h file` True if *file* exists and is a symbolic link.
- `-k file` True if *file* exists and its “sticky” bit is set.
- `-p file` True if *file* exists and is a named pipe (FIFO).
- `-r file` True if *file* exists and is readable.
- `-s file` True if *file* exists and has a size greater than zero.
- `-t fd` True if file descriptor *fd* is open and refers to a terminal.
- `-u file` True if *file* exists and its set-user-id bit is set.
- `-w file` True if *file* exists and is writable.
- `-x file` True if *file* exists and is executable.
- `-O file` True if *file* exists and is owned by the effective user id.
- `-G file` True if *file* exists and is owned by the effective group id.
- `-L file` True if *file* exists and is a symbolic link.
- `-S file` True if *file* exists and is a socket.
- `-N file` True if *file* exists and has been modified since it was last read.
- `file1 -nt file2` True if *file1* is newer (according to modification date) than *file2*, or if *file1* exists and *file2* does not.
- `file1 -ot file2` True if *file1* is older than *file2*, or if *file2* exists and *file1* does not.

- *file1 -ef file2* True if *file1* and *file2* refer to the same device and inode numbers.
- **-o** *optname* True if shell option *optname* is enabled. See the list of options under the description of the **-o** option to the **set** builtin below.
- **-z** *string* True if the length of *string* is zero.
- *string*
- **-n** *string* True if the length of *string* is non-zero.
- *string1 == string2* True if the strings are equal. **=** may be used in place of **==** for strict POSIX compliance.
- *string1 != string2* True if the strings are not equal.
- *string1 < string2* True if *string1* sorts before *string2* lexicographically in the current locale.
- *string1 > string2* True if *string1* sorts after *string2* lexicographically in the current locale.
- *arg1 OP arg2 OP* is one of **-eq**, **-ne**, **-lt**, **-le**, **-gt**, or **-ge**. These arithmetic binary operators return true if *arg1* is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to *arg2*, respectively. *Arg1* and *arg2* may be positive or negative integers.

7.14.20 Simple Command Expansion

When a simple command is executed, the shell performs the following expansions, assignments, and redirections, from left to right.

1. The words that the parser has marked as variable assignments (those preceding the command name) and redirections are saved for later processing.
2. The words that are not variable assignments or redirections are expanded. If any words remain after expansion, the first word is taken to be the name of the command and the remaining words are the arguments.
3. Redirections are performed as described above under **REDIRECTION**.
4. The text after the **=** in each variable assignment undergoes tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal before being assigned to the variable.

If no command name results, the variable assignments affect the current shell environment. Otherwise, the variables are added to the environment of the executed command and do not affect the current shell environment. If any of the assignments attempts to assign a value to a readonly variable, an error occurs, and the command exits with a non-zero status.

If no command name results, redirections are performed, but do not affect the current shell environment. A redirection error causes the command to exit with a non-zero status.

If there is a command name left after expansion, execution proceeds as described below. Otherwise, the command exits. If one of the expansions contained a command substitution, the exit status of the command is the exit status of the last command substitution performed. If there were no command substitutions, the command exits with a status of zero.

7.14.21 Command Execution

After a command has been split into words, if it results in a simple command and an optional list of arguments, the following actions are taken.

If the command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, that function is invoked as described above in **FUNCTIONS**. If the name does not match a function, the shell searches for it in the list of shell builtins. If a match is found, that builtin is invoked.

If the name is neither a shell function nor a builtin, and contains no slashes, **bash** searches each element of the **PATH** for a directory containing an executable file by that name. **Bash** uses a hash table to remember the full pathnames of executable files (see **hash** under **SHELL BUILTIN COMMANDS** below). A full search of the directories in **PATH** is performed only if the command is not found in the hash table. If the search is unsuccessful, the shell prints an error message and returns an exit status of 127.

If the search is successful, or if the command name contains one or more slashes, the shell executes the named program in a separate execution environment. Argument 0 is set to the name given, and the remaining arguments to the command are set to the arguments given, if any.

If this execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a *shell script*, a file containing shell commands. A subshell is spawned to execute it. This subshell reinitializes itself, so that the effect is as if a new shell had been invoked to handle the script, with the exception that the locations of commands remembered by the parent (see **hash** below under **SHELL BUILTIN COMMANDS**) are retained by the child.

If the program is a file beginning with **#!**, the remainder of the first line specifies an interpreter for the program. The shell executes the specified interpreter on operating systems that do not handle this executable format themselves. The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the program, followed by the name of the program, followed by the command arguments, if any.

7.14.22 Command Execution Environment

The shell has an *execution environment*, which consists of the following:

- • open files inherited by the shell at invocation, as modified by redirections supplied to the **exec** builtin
- • the current working directory as set by **cd**, **pushd**, or **popd**, or inherited by the shell at invocation
- • the file creation mode mask as set by **umask** or inherited from the shell's parent
- • current traps set by **trap**
- • shell parameters that are set by variable assignment or with **set** or inherited from the shell's parent in the environment
- • shell functions defined during execution or inherited from the shell's parent in the environment
- • options enabled at invocation (either by default or with command-line arguments) or by **set**
- • options enabled by **shopt**

- • shell aliases defined with **alias**
- • various process IDs, including those of background jobs, the value of **\$\$**, and the value of **\$PPID**

When a simple command other than a builtin or shell function is to be executed, it is invoked in a separate execution environment that consists of the following. Unless otherwise noted, the values are inherited from the shell.

- • the shell's open files, plus any modifications and additions specified by redirections to the command
- • the current working directory
- • the file creation mode mask
- • shell variables and functions marked for export, along with variables exported for the command, passed in the environment
- • traps caught by the shell are reset to the values inherited from the shell's parent, and traps ignored by the shell are ignored

A command invoked in this separate environment cannot affect the shell's execution environment.

Command substitution, commands grouped with parentheses, and asynchronous commands are invoked in a subshell environment that is a duplicate of the shell environment, except that traps caught by the shell are reset to the values that the shell inherited from its parent at invocation. Builtin commands that are invoked as part of a pipeline are also executed in a subshell environment. Changes made to the subshell environment cannot affect the shell's execution environment.

If a command is followed by a **&** and job control is not active, the default standard input for the command is the empty file */dev/null*. Otherwise, the invoked command inherits the file descriptors of the calling shell as modified by redirections.

7.14.23 Environment

When a program is invoked it is given an array of strings called the *environment*. This is a list of *name-value* pairs, of the form *name=value*.

The shell provides several ways to manipulate the environment. On invocation, the shell scans its own environment and creates a parameter for each name found, automatically marking it for *export* to child processes. Executed commands inherit the environment. The **export** and **declare -x** commands allow parameters and functions to be added to and deleted from the environment. If the value of a parameter in the environment is modified, the new value becomes part of the environment, replacing the old. The environment inherited by any executed command consists of the shell's initial environment, whose values may be modified in the shell, less any pairs removed by the **unset** command, plus any additions via the **export** and **declare -x** commands.

The environment for any *simple* command or function may be augmented temporarily by prefixing it with parameter assignments, as described above in **PARAMETERS**. These assignment statements affect only the environment seen by that command.

If the **-k** option is set (see the **set** builtin command below), then *all* parameter assignments are placed in the environment for a command, not just those that precede the command name.

When **bash** invokes an external command, the variable **_** is set to the full file name of the command and passed to that command in its environment.

7.14.24 Exit Status

For the shell's purposes, a command which exits with a zero exit status has succeeded. An exit status of zero indicates success. A non-zero exit status indicates failure. When a command terminates on a fatal signal N , **bash** uses the value of $128+N$ as the exit status.

If a command is not found, the child process created to execute it returns a status of 127. If a command is found but is not executable, the return status is 126.

If a command fails because of an error during expansion or redirection, the exit status is greater than zero.

Shell builtin commands return a status of 0 (*true*) if successful, and non-zero (*false*) if an error occurs while they execute. All builtins return an exit status of 2 to indicate incorrect usage.

Bash itself returns the exit status of the last command executed, unless a syntax error occurs, in which case it exits with a non-zero value. See also the **exit** builtin command below.

7.14.25 Signals

When **bash** is interactive, in the absence of any traps, it ignores **SIGTERM** (so that **kill 0** does not kill an interactive shell), and **SIGINT** is caught and handled (so that the **wait** builtin is interruptible). In all cases, **bash** ignores **SIGQUIT**. If job control is in effect, **bash** ignores **SIGTTIN**, **SIGTTOU**, and **SIGTSTP**.

Non-builtin commands run by **bash** have signal handlers set to the values inherited by the shell from its parent. When job control is not in effect, asynchronous commands ignore **SIGINT** and **SIGQUIT** in addition to these inherited handlers. Commands run as a result of command substitution ignore the keyboard-generated job control signals **SIGTTIN**, **SIGTTOU**, and **SIGTSTP**.

The shell exits by default upon receipt of a **SIGHUP**. Before exiting, an interactive shell resends the **SIGHUP** to all jobs, running or stopped. Stopped jobs are sent **SIGCONT** to ensure that they receive the **SIGHUP**. To prevent the shell from sending the signal to a particular job, it should be removed from the jobs table with the **disown** builtin (see **SHELL BUILTIN COMMANDS** below) or marked to not receive **SIGHUP** using **disown -h**.

If the **huponexit** shell option has been set with **shopt**, **bash** sends a **SIGHUP** to all jobs when an interactive login shell exits.

If **Bbash** is waiting for a command to complete and receives a signal for which a trap has been set, the trap will not be executed until the command completes. When **bash** is waiting for an asynchronous command via the **wait** builtin, the reception of a signal for which a trap has been set will cause the **wait** builtin to return immediately with an exit status greater than 128, immediately after which the trap is executed.

7.14.26 Job Control

Job control refers to the ability to selectively stop (*suspend*) the execution of processes and continue (*resume*) their execution at a later point. A user typically employs this facility via an interactive interface supplied jointly by the system's terminal driver and **bash**.

The shell associates a *job* with each pipeline. It keeps a table of currently executing jobs, which may be listed with the **jobs** command. When **bash** starts a job asynchronously (in the *background*), it prints a line that looks like:

```
[1] 25647
```

indicating that this job is job number 1 and that the process ID of the last process in the pipeline associated with this job is 25647. All of the processes in a single pipeline are members of the same job. **Bash** uses the *job* abstraction as the basis for job control.

To facilitate the implementation of the user interface to job control, the operating system maintains the notion of a *current terminal process group ID*. Members of this process group (processes whose process group ID is equal to the current terminal process group ID) receive keyboard-generated signals such as **SIGINT**. These processes are said to be in the *foreground*. *Background* processes are those whose process group ID differs from the terminal's; such processes are immune to keyboard-generated signals. Only foreground processes are allowed to read from or write to the terminal. Background processes which attempt to read from (write to) the terminal are sent a **SIGTTIN (SIGTTOU)** signal by the terminal driver, which, unless caught, suspends the process.

If the operating system on which **bash** is running supports job control, **bash** contains facilities to use it. Typing the *suspend* character (typically **^Z**, Control-Z) while a process is running causes that process to be stopped and returns control to **bash**. Typing the *delayed suspend* character (typically **^Y**, Control-Y) causes the process to be stopped when it attempts to read input from the terminal, and control to be returned to **bash**. The user may then manipulate the state of this job, using the **bg** command to continue it in the background, the **fg** command to continue it in the foreground, or the **kill** command to kill it. A **^Z** takes effect immediately, and has the additional side effect of causing pending output and typeahead to be discarded.

There are a number of ways to refer to a job in the shell. The character **%** introduces a job name. Job number *n* may be referred to as **%n**. A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, **%ce** refers to a stopped **ce** job. If a prefix matches more than one job, **bash** reports an error. Using **??ce**, on the other hand, refers to any job containing the string **ce** in its command line. If the substring matches more than one job, **bash** reports an error. The symbols **%%** and **%+** refer to the shell's notion of the *current job*, which is the last job stopped while it was in the foreground or started in the background. The *previous job* may be referenced using **%-**. In output pertaining to jobs (e.g., the output of the **jobs** command), the current job is always flagged with a **+**, and the previous job with a **-**.

Simply naming a job can be used to bring it into the foreground: **%1** is a synonym for **“fg %1”**, bringing job 1 from the background into the foreground. Similarly, **“%1 &”** resumes job 1 in the background, equivalent to **“bg %1”**.

The shell learns immediately whenever a job changes state. Normally, **bash** waits until it is about to print a prompt before reporting changes in a job's status so as to not interrupt any other output. If the **-b** option to the **set** builtin command is enabled, **bash** reports such changes immediately. Any trap on **SIGCHLD** is executed for each child that exits.

If an attempt to exit **bash** is made while jobs are stopped, the shell prints a warning message. The **jobs** command may then be used to inspect their status. If a second attempt to exit is made without an intervening command, the shell does not print another warning, and the stopped jobs are terminated.

7.14.27 Prompting

When executing interactively, **bash** displays the primary prompt **PS1** when it is ready to read a command, and the secondary prompt **PS2** when it needs more input to complete a command. **Bash** allows these prompt strings to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

- **\a** an ASCII bell character (07)
- **\d** the date in Weekday Month Date format (e.g., Tue May 26)

- `\D{format}` the *format* is passed to `strftime(3)` and the result is inserted into the prompt string; an empty *format* results in a locale-specific time representation. The braces are required
- `\e` an ASCII escape character (033)
- `\h` the hostname up to the first ‘.’
- `\H` the hostname
- `\j` the number of jobs currently managed by the shell
- `\l` the basename of the shell’s terminal device name
- `\n` newline
- `\r` carriage return
- `\s` the name of the shell, the basename of `$0` (the portion following the final slash)
- `\t` the current time in 24-hour HH:MM:SS format
- `\T` the current time in 12-hour HH:MM:SS format
- `\@` the current time in 12-hour am/pm format
- `\A` the current time in 24-hour HH:MM format
- `\u` the username of the current user
- `\v` the version of **bash** (e.g., 2.00)
- `\V` the release of **bash**, version + patch level (e.g., 2.00.0)
- `\w` the current working directory, with `$HOME` abbreviated with a tilde
- `\W` the basename of the current working directory, with `$HOME` abbreviated with a tilde
- `\!` the history number of this command
- `\#` the command number of this command
- `\$` if the effective UID is 0, a `#`, otherwise a `$`
- `\nnn` the character corresponding to the octal number *nnn*
- `\` a backslash
- `\[` begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt
- `\]` end a sequence of non-printing characters

The command number and the history number are usually different: the history number of a command is its position in the history list, which may include commands restored from the history file (see **HISTORY** below), while the command number is the position in the sequence of commands executed during the current shell session. After the string is decoded, it is expanded via parameter expansion, command substitution, arithmetic expansion, and quote removal, subject to the value of the **promptvars** shell option (see the description of the **shopt** command under **SHELL BUILTIN COMMANDS** below).

7.14.28 Readline

This is the library that handles reading input when using an interactive shell, unless the `-noediting` option is given at shell invocation. By default, the line editing commands are similar to those of emacs. A vi-style line editing interface is also available. To turn off line editing after the shell is running, use the `+o emacs` or `+o vi` options to the `set` builtin (see **SHELL BUILTIN COMMANDS** below).

Readline Notation

In this section, the emacs-style notation is used to denote keystrokes. Control keys are denoted by *C-key*, e.g., C-n means Control-N. Similarly, *meta* keys are denoted by *M-key*, so M-x means Meta-X. (On keyboards without a *meta* key, M-x means ESC *x*, i.e., press the Escape key then the *x* key. This makes ESC the *meta prefix*. The combination M-C-*x* means ESC-Control-*x*, or press the Escape key then hold the Control key while pressing the *x* key.)

Readline commands may be given numeric *arguments*, which normally act as a repeat count. Sometimes, however, it is the sign of the argument that is significant. Passing a negative argument to a command that acts in the forward direction (e.g., **kill-line**) causes that command to act in a backward direction. Commands whose behavior with arguments deviates from this are noted below.

When a command is described as *killing* text, the text deleted is saved for possible future retrieval (*yanking*). The killed text is saved in a *kill ring*. Consecutive kills cause the text to be accumulated into one unit, which can be yanked all at once. Commands which do not kill text separate the chunks of text on the kill ring.

Readline Initialization

Readline is customized by putting commands in an initialization file (the *inputrc* file). The name of this file is taken from the value of the **INPUTRC** variable. If that variable is unset, the default is `/.inputrc`. When a program which uses the readline library starts up, the initialization file is read, and the key bindings and variables are set. There are only a few basic constructs allowed in the readline initialization file. Blank lines are ignored. Lines beginning with a `#` are comments. Lines beginning with a `$` indicate conditional constructs. Other lines denote key bindings and variable settings.

The default key-bindings may be changed with an *inputrc* file. Other programs that use this library may add their own commands and bindings.

For example, placing

M-Control-u: universal-argument or C-Meta-u: universal-argument into the *inputrc* would make M-C-u execute the readline command *universal-argument*.

The following symbolic character names are recognized: *RUBOUT*, *DEL*, *ESC*, *LFD*, *NEW-LINE*, *RET*, *RETURN*, *SPC*, *SPACE*, and *TAB*.

In addition to command names, readline allows keys to be bound to a string that is inserted when the key is pressed (a *macro*).

Readline Key Bindings

The syntax for controlling key bindings in the *inputrc* file is simple. All that is required is the name of the command or the text of a macro and a key sequence to which it should be bound. The name may be specified in one of two ways: as a symbolic key name, possibly with *Meta-* or *Control-* prefixes, or as a key sequence.

When using the form **keyname**:*function-name* or *macro*, *keyname* is the name of a key spelled out in English. For example:

Control-u: universal-argument

Meta-Rubout: backward-kill-word

Control-o: > output

In the above example, *C-u* is bound to the function **universal-argument**, *M-DEL* is bound to the function **backward-kill-word**, and *C-o* is bound to run the macro expressed on the right hand side (that is, to insert the text “> output” into the line).

In the second form, **keyseq**:*function-name* or *macro*, **keyseq** differs from **keyname** above in that strings denoting an entire key sequence may be specified by placing the sequence within double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the symbolic character names are not recognized.

\C-u: universal-argument

\C-x\C-r: re-read-init-file

\e[11 : Function Key 1

In this example, *C-u* is again bound to the function **universal-argument**. *C-x C-r* is bound to the function **re-read-init-file**, and *ESC [1 1* is bound to insert the text “Function Key 1”.

The full set of GNU Emacs style escape sequences is

- \C- control prefix
- \M- meta prefix
- \e an escape character
- \\ backslash
- \ literal
- \' literal ’

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available:

- \a alert (bell)
- \b backspace
- \d delete
- \f form feed
- \n newline
- \r carriage return
- \t horizontal tab
- \v vertical tab
- \nnn the eight-bit character whose value is the octal value *nnn* (one to three digits)
- \xHH the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)

When entering the text of a macro, single or double quotes must be used to indicate a macro definition. Unquoted text is assumed to be a function name. In the macro body, the backslash escapes described above are expanded. Backslash will quote any other character in the macro text, including and ’.

Bash allows the current readline key bindings to be displayed or modified with the **bind** builtin command. The editing mode may be switched during interactive use by using the **-o** option to the **set** builtin command (see **SHELL BUILTIN COMMANDS** below).

Readline Variables

Readline has variables that can be used to further customize its behavior. A variable may be set in the *inputrc* file with a statement of the form

```
set variable-name value
```

Except where noted, readline variables can take the values **On** or **Off**. The variables and their default values are:

- **bell-style (audible)** Controls what happens when readline wants to ring the terminal bell. If set to **none**, readline never rings the bell. If set to **visible**, readline uses a visible bell if one is available. If set to **audible**, readline attempts to ring the terminal's bell.
- **comment-begin (“#”)** The string that is inserted when the readline **insert-comment** command is executed. This command is bound to **M-#** in emacs mode and to **#** in vi command mode.
- **completion-ignore-case (Off)** If set to **On**, readline performs filename matching and completion in a case-insensitive fashion.
- **completion-query-items (100)** This determines when the user is queried about viewing the number of possible completions generated by the **possible-completions** command. It may be set to any integer value greater than or equal to zero. If the number of possible completions is greater than or equal to the value of this variable, the user is asked whether or not he wishes to view them; otherwise they are simply listed on the terminal.
- **convert-meta (On)** If set to **On**, readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prefixing an escape character (in effect, using escape as the *meta prefix*).
- **disable-completion (Off)** If set to **On**, readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to **self-insert**.
- **editing-mode (emacs)** Controls whether readline begins with a set of key bindings similar to *emacs* or *vi*. **editing-mode** can be set to either **emacs** or **vi**.
- **enable-keypad (Off)** When set to **On**, readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys.
- **expand-tilde (Off)** If set to **on**, tilde expansion is performed when readline attempts word completion.
- **history-preserve-point** If set to **on**, the history code attempts to place point at the same location on each history line retrieved with **previous-history** or **next-history**.
- **horizontal-scroll-mode (Off)** When set to **On**, makes readline use a single line for display, scrolling the input horizontally on a single screen line when it becomes longer than the screen width rather than wrapping to a new line.
- **input-meta (Off)** If set to **On**, readline will enable eight-bit input (that is, it will not strip the high bit from the characters it reads), regardless of what the terminal claims it can support. The name **meta-flag** is a synonym for this variable.
- **isearch-terminators (“C-[C-J”)** The string of characters that should terminate an incremental search without subsequently executing the character as a command. If this variable has not been given a value, the characters *ESC* and *C-J* will terminate an incremental search.

- **keymap (emacs)** Set the current readline keymap. The set of valid keymap names is *emacs*, *emacs-standard*, *emacs-meta*, *emacs-ctlx*, *vi*, *vi-command*, and *vi-insert*. *vi* is equivalent to *vi-command*; *emacs* is equivalent to *emacs-standard*. The default value is *emacs*; the value of **editing-mode** also affects the default keymap.
- **mark-directories (On)** If set to **On**, completed directory names have a slash appended.
- **mark-modified-lines (Off)** If set to **On**, history lines that have been modified are displayed with a preceding asterisk (*).
- **mark-symlinked-directories (Off)** If set to **On**, completed names which are symbolic links to directories have a slash appended (subject to the value of **mark-directories**).
- **match-hidden-files (On)** This variable, when set to **On**, causes readline to match files whose names begin with a '.' (hidden files) when performing filename completion, unless the leading '.' is supplied by the user in the filename to be completed.
- **output-meta (Off)** If set to **On**, readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence.
- **page-completions (On)** If set to **On**, readline uses an internal *more*-like pager to display a screenful of possible completions at a time.
- **print-completions-horizontally (Off)** If set to **On**, readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen.
- **show-all-if-ambiguous (Off)** This alters the default behavior of the completion functions. If set to **on**, words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell.
- **show-all-if-unmodified (Off)** This alters the default behavior of the completion functions in a fashion similar to **show-all-if-ambiguous**. If set to **on**, words which have more than one possible completion without any possible partial completion (the possible completions don't share a common prefix) cause the matches to be listed immediately instead of ringing the bell.
- **visible-stats (Off)** If set to **On**, a character denoting a file's type as reported by *stat(2)* is appended to the filename when listing possible completions.

Readline Conditional Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are four parser directives used.

- **\$if** The **\$if** construct allows bindings to be made based on the editing mode, the terminal being used, or the application using readline. The text of the test extends to the end of the line; no characters are required to isolate it.
- **mode** The **mode=** form of the **\$if** directive is used to test whether readline is in emacs or vi mode. This may be used in conjunction with the **set keymap** command, for instance, to set bindings in the *emacs-standard* and *emacs-ctlx* keymaps only if readline is starting out in emacs mode.

- **term** The **term=** form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the = is tested against the both full name of the terminal and the portion of the terminal name before the first -. This allows *sun* to match both *sun* and *sun-cmd*, for instance.
- **application** The **application** construct is used to include application-specific settings. Each program using the readline library sets the *application name*, and an initialization file can test for a particular value. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

```
$if Bash
# Quote the current or previous word
\C-xq: \eb\|ef\“
$endif
```

- **\$endif** This command, as seen in the previous example, terminates an **\$if** command.
- **\$else** Commands in this branch of the **\$if** directive are executed if the test fails.
- **\$include** This directive takes a single filename as an argument and reads commands and bindings from that file. For example, the following directive would read */etc/inputrc*:

```
$include /etc/inputrc
```

Searching

Readline provides commands for searching through the command history (see **HISTORY** below) for lines containing a specified string. There are two search modes: *incremental* and *non-incremental*.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, readline displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. The characters present in the value of the **isearch-terminators** variable are used to terminate an incremental search. If that variable has not been assigned a value the Escape and Control-J characters will terminate an incremental search. Control-G will abort an incremental search and restore the original line. When the search is terminated, the history entry containing the search string becomes the current line.

To find other matching entries in the history list, type Control-S or Control-R as appropriate. This will search backward or forward in the history for the next entry matching the search string typed so far. Any other key sequence bound to a readline command will terminate the search and execute that command. For instance, a *newline* will terminate the search and accept the line, thereby executing the command from the history list.

Readline remembers the last incremental search string. If two Control-Rs are typed without any intervening characters defining a new search string, any remembered search string is used.

Non-incremental searches read the entire search string before starting to search for matching history lines. The search string may be typed by the user or be part of the contents of the current line.

Readline Command Names

The following is a list of the names of the commands and the default key sequences to which they are bound. Command names without an accompanying key sequence are unbound by default. In the following descriptions, *point* refers to the current cursor position, and *mark* refers to a cursor position saved by the **set-mark** command. The text between the point and mark is referred to as the *region*.

Commands for Moving

- **beginning-of-line (C-a)** Move to the start of the current line.
- **end-of-line (C-e)** Move to the end of the line.
- **forward-char (C-f)** Move forward a character.
- **backward-char (C-b)** Move back a character.
- **forward-word (M-f)** Move forward to the end of the next word. Words are composed of alphanumeric characters (letters and digits).
- **backward-word (M-b)** Move back to the start of the current or previous word. Words are composed of alphanumeric characters (letters and digits).
- **clear-screen (C-l)** Clear the screen leaving the current line at the top of the screen. With an argument, refresh the current line without clearing the screen.
- **redraw-current-line** Refresh the current line.

Commands for Manipulating the History

- **accept-line (Newline, Return)** Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list according to the state of the **HISTCONTROL** variable. If the line is a modified history line, then restore the history line to its original state.
- **previous-history (C-p)** Fetch the previous command from the history list, moving back in the list.
- **next-history (C-n)** Fetch the next command from the history list, moving forward in the list.
- **beginning-of-history (M-<)** Move to the first line in the history.
- **end-of-history (M->)** Move to the end of the input history, i.e., the line currently being entered.
- **reverse-search-history (C-r)** Search backward starting at the current line and moving ‘up’ through the history as necessary. This is an incremental search.
- **forward-search-history (C-s)** Search forward starting at the current line and moving ‘down’ through the history as necessary. This is an incremental search.
- **non-incremental-reverse-search-history (M-p)** Search backward through the history starting at the current line using a non-incremental search for a string supplied by the user.

- **non-incremental-forward-search-history (M-n)** Search forward through the history using a non-incremental search for a string supplied by the user.
- **history-search-forward** Search forward through the history for the string of characters between the start of the current line and the point. This is a non-incremental search.
- **history-search-backward** Search backward through the history for the string of characters between the start of the current line and the point. This is a non-incremental search.
- **yank-nth-arg (M-C-y)** Insert the first argument to the previous command (usually the second word on the previous line) at point. With an argument *n*, insert the *n*th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command.
- **yank-last-arg (M-., M-.)** Insert the last argument to the previous command (the last word of the previous history entry). With an argument, behave exactly like **yank-nth-arg**. Successive calls to **yank-last-arg** move back through the history list, inserting the last argument of each line in turn.
- **shell-expand-line (M-C-e)** Expand the line as the shell does. This performs alias and history expansion as well as all of the shell word expansions. See **HISTORY EXPANSION** below for a description of history expansion.
- **history-expand-line (M-^)** Perform history expansion on the current line. See **HISTORY EXPANSION** below for a description of history expansion.
- **magic-space** Perform history expansion on the current line and insert a space. See **HISTORY EXPANSION** below for a description of history expansion.
- **alias-expand-line** Perform alias expansion on the current line. See **ALIASES** above for a description of alias expansion.
- **history-and-alias-expand-line** Perform history and alias expansion on the current line.
- **insert-last-argument (M-., M-.)** A synonym for **yank-last-arg**.
- **operate-and-get-next (C-o)** Accept the current line for execution and fetch the next line relative to the current line from the history for editing. Any argument is ignored.
- **edit-and-execute-command (C-xC-e)** Invoke an editor on the current command line, and execute the result as shell commands. **Bash** attempts to invoke **\$FCEDIT**, **\$EDITOR**, and *emacs* as the editor, in that order.

Commands for Changing Text

- **delete-char (C-d)** Delete the character at point. If point is at the beginning of the line, there are no characters in the line, and the last character typed was not bound to **delete-char**, then return **EOF**.
- **backward-delete-char (Rubout)** Delete the character behind the cursor. When given a numeric argument, save the deleted text on the kill ring.
- **forward-backward-delete-char** Delete the character under the cursor, unless the cursor is at the end of the line, in which case the character behind the cursor is deleted.

- **quoted-insert (C-q, C-v)** Add the next character typed to the line verbatim. This is how to insert characters like **C-q**, for example.
- **tab-insert (C-v TAB)** Insert a tab character.
- **self-insert (a,b,A,1,!,...)** Insert the character typed.
- **transpose-chars (C-t)** Drag the character before point forward over the character at point, moving point forward as well. If point is at the end of the line, then this transposes the two characters before point. Negative arguments have no effect.
- **transpose-words (M-t)** Drag the word before point past the word after point, moving point over that word as well. If point is at the end of the line, this transposes the last two words on the line.
- **upcase-word (M-u)** Uppercase the current (or following) word. With a negative argument, uppercase the previous word, but do not move point.
- **downcase-word (M-l)** Lowercase the current (or following) word. With a negative argument, lowercase the previous word, but do not move point.
- **capitalize-word (M-c)** Capitalize the current (or following) word. With a negative argument, capitalize the previous word, but do not move point.
- **overwrite-mode** Toggle overwrite mode. With an explicit positive numeric argument, switches to overwrite mode. With an explicit non-positive numeric argument, switches to insert mode. This command affects only **emacs** mode; **vi** mode does overwrite differently. Each call to *readline()* starts in insert mode. In overwrite mode, characters bound to **self-insert** replace the text at point rather than pushing the text to the right. Characters bound to **backward-delete-char** replace the character before point with a space. By default, this command is unbound.

Killing and Yanking

- **kill-line (C-k)** Kill the text from point to the end of the line.
- **backward-kill-line (C-x Rubout)** Kill backward to the beginning of the line.
- **unix-line-discard (C-u)** Kill backward from point to the beginning of the line. The killed text is saved on the kill-ring.
- **kill-whole-line** Kill all characters on the current line, no matter where point is.
- **kill-word (M-d)** Kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as those used by **forward-word**.
- **backward-kill-word (M-Rubout)** Kill the word behind point. Word boundaries are the same as those used by **backward-word**.
- **unix-word-rubout (C-w)** Kill the word behind point, using white space as a word boundary. The killed text is saved on the kill-ring.
- **unix-filename-rubout** Kill the word behind point, using white space and the slash character as the word boundaries. The killed text is saved on the kill-ring.
- **delete-horizontal-space (M-\)** Delete all spaces and tabs around point.

- **kill-region** Kill the text in the current region.
- **copy-region-as-kill** Copy the text in the region to the kill buffer.
- **copy-backward-word** Copy the word before point to the kill buffer. The word boundaries are the same as **backward-word**.
- **copy-forward-word** Copy the word following point to the kill buffer. The word boundaries are the same as **forward-word**.
- **yank (C-y)** Yank the top of the kill ring into the buffer at point.
- **yank-pop (M-y)** Rotate the kill ring, and yank the new top. Only works following **yank** or **yank-pop**.

Numeric Arguments

- **digit-argument (M-0, M-1, ..., M-)** Add this digit to the argument already accumulating, or start a new argument. M- starts a negative argument.
- **universal-argument** This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing **universal-argument** again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit or minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on.

Completing

- **complete (TAB)** Attempt to perform completion on the text before point. **Bash** attempts completion treating the text as a variable (if the text begins with **\$**), username (if the text begins with **_**), hostname (if the text begins with **@**), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted.
- **possible-completions (M-?)** List the possible completions of the text before point.
- **insert-completions (M-*)** Insert all completions of the text before point that would have been generated by **possible-completions**.
- **menu-complete** Similar to **complete**, but replaces the word to be completed with a single match from the list of possible completions. Repeated execution of **menu-complete** steps through the list of possible completions, inserting each match in turn. At the end of the list of completions, the bell is rung (subject to the setting of **bell-style**) and the original text is restored. An argument of *n* moves *n* positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to **TAB**, but is unbound by default.
- **delete-char-or-list** Deletes the character under the cursor if not at the beginning or end of the line (like **delete-char**). If at the end of the line, behaves identically to **possible-completions**. This command is unbound by default.
- **complete-filename (M-/)** Attempt filename completion on the text before point.

- **possible-filename-completions** (**C-x /**) List the possible completions of the text before point, treating it as a filename.
- **complete-username** (**M-**) Attempt completion on the text before point, treating it as a username.
- **possible-username-completions** (**C-x**) List the possible completions of the text before point, treating it as a username.
- **complete-variable** (**M-\$**) Attempt completion on the text before point, treating it as a shell variable.
- **possible-variable-completions** (**C-x \$**) List the possible completions of the text before point, treating it as a shell variable.
- **complete-hostname** (**M-@**) Attempt completion on the text before point, treating it as a hostname.
- **possible-hostname-completions** (**C-x @**) List the possible completions of the text before point, treating it as a hostname.
- **complete-command** (**M-!**) Attempt completion on the text before point, treating it as a command name. Command completion attempts to match the text against aliases, reserved words, shell functions, shell builtins, and finally executable filenames, in that order.
- **possible-command-completions** (**C-x !**) List the possible completions of the text before point, treating it as a command name.
- **dynamic-complete-history** (**M-TAB**) Attempt completion on the text before point, comparing the text against lines from the history list for possible completion matches.
- **complete-into-braces** (**M-{}**) Perform filename completion and insert the list of possible completions enclosed within braces so the list is available to the shell (see **Brace Expansion** above).

Keyboard Macros

- **start-kbd-macro** (**C-x ()**) Begin saving the characters typed into the current keyboard macro.
- **end-kbd-macro** (**C-x))** Stop saving the characters typed into the current keyboard macro and store the definition.
- **call-last-kbd-macro** (**C-x e**) Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

Miscellaneous

- **re-read-init-file** (**C-x C-r**) Read in the contents of the *inputrc* file, and incorporate any bindings or variable assignments found there.
- **abort** (**C-g**) Abort the current editing command and ring the terminal's bell (subject to the setting of **bell-style**).
- **do-uppercase-version** (**M-a, M-b, M-x, ...**) If the metafiled character *x* is lowercase, run the command that is bound to the corresponding uppercase character.

- **prefix-meta (ESC)** Metafy the next character typed. **ESC f** is equivalent to **Meta-f**.
- **undo (C-_, C-x C-u)** Incremental undo, separately remembered for each line.
- **revert-line (M-r)** Undo all changes made to this line. This is like executing the **undo** command enough times to return the line to its initial state.
- **tilde-expand (M-&)** Perform tilde expansion on the current word.
- **set-mark (C-@, M-<space>)** Set the mark to the point. If a numeric argument is supplied, the mark is set to that position.
- **exchange-point-and-mark (C-x C-x)** Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.
- **character-search (C-])** A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occurrences.
- **character-search-backward (M-C-])** A character is read and point is moved to the previous occurrence of that character. A negative count searches for subsequent occurrences.
- **insert-comment (M-#)** Without a numeric argument, the value of the readline **comment-begin** variable is inserted at the beginning of the current line. If a numeric argument is supplied, this command acts as a toggle: if the characters at the beginning of the line do not match the value of **comment-begin**, the value is inserted, otherwise the characters in **comment-begin** are deleted from the beginning of the line. In either case, the line is accepted as if a newline had been typed. The default value of **comment-begin** causes this command to make the current line a shell comment. If a numeric argument causes the comment character to be removed, the line will be executed by the shell.
- **glob-complete-word (M-g)** The word before point is treated as a pattern for pathname expansion, with an asterisk implicitly appended. This pattern is used to generate a list of matching file names for possible completions.
- **glob-expand-word (C-x *)** The word before point is treated as a pattern for pathname expansion, and the list of matching file names is inserted, replacing the word. If a numeric argument is supplied, an asterisk is appended before pathname expansion.
- **glob-list-expansions (C-x g)** The list of expansions that would have been generated by **glob-expand-word** is displayed, and the line is redrawn. If a numeric argument is supplied, an asterisk is appended before pathname expansion.
- **dump-functions** Print all of the functions and their key bindings to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.
- **dump-variables** Print all of the settable readline variables and their values to the readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.
- **dump-macros** Print all of the readline key sequences bound to macros and the strings they output. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file.
- **display-shell-version (C-x C-v)** Display version information about the current instance of **bash**.

Programmable Completion

When word completion is attempted for an argument to a command for which a completion specification (a *compspec*) has been defined using the **complete** builtin (see **SHELL BUILTIN COMMANDS** below), the programmable completion facilities are invoked.

First, the command name is identified. If a *compspec* has been defined for that command, the *compspec* is used to generate the list of possible completions for the word. If the command word is a full pathname, a *compspec* for the full pathname is searched for first. If no *compspec* is found for the full pathname, an attempt is made to find a *compspec* for the portion following the final slash.

Once a *compspec* has been found, it is used to generate the list of matching words. If a *compspec* is not found, the default **bash** completion as described above under **Completing** is performed.

First, the actions specified by the *compspec* are used. Only matches which are prefixed by the word being completed are returned. When the **-f** or **-d** option is used for filename or directory name completion, the shell variable **FIGNORE** is used to filter the matches.

Any completions specified by a filename expansion pattern to the **-G** option are generated next. The words generated by the pattern need not match the word being completed. The **GLOBIGNORE** shell variable is not used to filter the matches, but the **FIGNORE** variable is used.

Next, the string specified as the argument to the **-W** option is considered. The string is first split using the characters in the **IFS** special variable as delimiters. Shell quoting is honored. Each word is then expanded using brace expansion, tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and pathname expansion, as described above under **EXPANSION**. The results are split using the rules described above under **Word Splitting**. The results of the expansion are prefix-matched against the word being completed, and the matching words become the possible completions.

After these matches have been generated, any shell function or command specified with the **-F** and **-C** options is invoked. When the command or function is invoked, the **COMP_LINE** and **COMP_POINT** variables are assigned values as described above under **Shell Variables**. If a shell function is being invoked, the **COMP_WORDS** and **COMP_CWORD** variables are also set. When the function or command is invoked, the first argument is the name of the command whose arguments are being completed, the second argument is the word being completed, and the third argument is the word preceding the word being completed on the current command line. No filtering of the generated completions against the word being completed is performed; the function or command has complete freedom in generating the matches.

Any function specified with **-F** is invoked first. The function may use any of the shell facilities, including the **compgen** builtin described below, to generate the matches. It must put the possible completions in the **COMP_REPLY** array variable.

Next, any command specified with the **-C** option is invoked in an environment equivalent to command substitution. It should print a list of completions, one per line, to the standard output. Backslash may be used to escape a newline, if necessary.

After all of the possible completions are generated, any filter specified with the **-X** option is applied to the list. The filter is a pattern as used for pathname expansion; a **&** in the pattern is replaced with the text of the word being completed. A literal **&** may be escaped with a backslash; the backslash is removed before attempting a match. Any completion that matches the pattern will be removed from the list. A leading **!** negates the pattern; in this case any completion not matching the pattern will be removed.

Finally, any prefix and suffix specified with the **-P** and **-S** options are added to each member of the completion list, and the result is returned to the readline completion code as the list of possible completions.

If the previously-applied actions do not generate any matches, and the **-o dirnames** option was supplied to **complete** when the compspec was defined, directory name completion is attempted.

If the **-o plusdirs** option was supplied to **complete** when the compspec was defined, directory name completion is attempted and any matches are added to the results of the other actions.

By default, if a compspec is found, whatever it generates is returned to the completion code as the full set of possible completions. The default **bash** completions are not attempted, and the readline default of filename completion is disabled. If the **-o bashdefault** option was supplied to **complete** when the compspec was defined, the **bash** default completions are attempted if the compspec generates no matches. If the **-o default** option was supplied to **complete** when the compspec was defined, readline's default completion will be performed if the compspec (and, if attempted, the default **bash** completions) generate no matches.

When a compspec indicates that directory name completion is desired, the programmable completion functions force readline to append a slash to completed names which are symbolic links to directories, subject to the value of the **mark-directories** readline variable, regardless of the setting of the **mark-symlinked-directories** readline variable.

7.14.29 History

When the **-o history** option to the **set** builtin is enabled, the shell provides access to the *command history*, the list of commands previously typed. The value of the **HISTSIZE** variable is used as the number of commands to save in a history list. The text of the last **HISTSIZE** commands (default 500) is saved. The shell stores each command in the history list prior to parameter and variable expansion (see **EXPANSION** above) but after history expansion is performed, subject to the values of the shell variables **HISTIGNORE** and **HISTCONTROL**.

On startup, the history is initialized from the file named by the variable **HISTFILE** (default *./bash_history*). The file named by the value of **HISTFILE** is truncated, if necessary, to contain no more than the number of lines specified by the value of **HISTFILESIZE**. When an interactive shell exits, the last **\$HISTSIZE** lines are copied from the history list to **\$HISTFILE**. If the **histappend** shell option is enabled (see the description of **shopt** under **SHELL BUILTIN COMMANDS** below), the lines are appended to the history file, otherwise the history file is overwritten. If **HISTFILE** is unset, or if the history file is unwritable, the history is not saved. After saving the history, the history file is truncated to contain no more than **HISTFILESIZE** lines. If **HISTFILESIZE** is not set, no truncation is performed.

The builtin command **fc** (see **SHELL BUILTIN COMMANDS** below) may be used to list or edit and re-execute a portion of the history list. The **history** builtin may be used to display or modify the history list and manipulate the history file. When using command-line editing, search commands are available in each editing mode that provide access to the history list.

The shell allows control over which commands are saved on the history list. The **HISTCONTROL** and **HISTIGNORE** variables may be set to cause the shell to save only a subset of the commands entered. The **cmdhist** shell option, if enabled, causes the shell to attempt to save each line of a multi-line command in the same history entry, adding semicolons where necessary to preserve syntactic correctness. The **lithist** shell option causes the shell to save the command with embedded newlines instead of semicolons. See the description of the **shopt** builtin below under **SHELL BUILTIN COMMANDS** for information on setting and unsetting shell options.

7.14.30 History Expansion

The shell supports a history expansion feature that is similar to the history expansion in **cs**. This section describes what syntax features are available. This feature is enabled by default for interactive shells, and can be disabled using the **+H** option to the **set** builtin command (see **SHELL BUILTIN COMMANDS** below). Non-interactive shells do not perform history expansion by default.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, insert the arguments to a previous command into the current input line, or fix errors in previous commands quickly.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words. It takes place in two parts. The first is to determine which line from the history list to use during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the history is the *event*, and the portions of that line that are acted upon are *words*. Various *modifiers* are available to manipulate the selected words. The line is broken into words in the same fashion as when reading input, so that several *metacharacter*-separated words surrounded by quotes are considered one word. History expansions are introduced by the appearance of the history expansion character, which is **!** by default. Only backslash (****) and single quotes can quote the history expansion character.

Several characters inhibit history expansion if found immediately following the history expansion character, even if it is unquoted: space, tab, newline, carriage return, and **=**. If the **extglob** shell option is enabled, **(** will also inhibit expansion.

Several shell options settable with the **shopt** builtin may be used to tailor the behavior of history expansion. If the **histverify** shell option is enabled (see the description of the **shopt** builtin), and **readline** is being used, history substitutions are not immediately passed to the shell parser. Instead, the expanded line is reloaded into the **readline** editing buffer for further modification. If **readline** is being used, and the **histreedit** shell option is enabled, a failed history substitution will be reloaded into the **readline** editing buffer for correction. The **-p** option to the **history** builtin command may be used to see what a history expansion will do before using it. The **-s** option to the **history** builtin may be used to add commands to the end of the history list without actually executing them, so that they are available for subsequent recall.

The shell allows control of the various characters used by the history expansion mechanism (see the description of **histchars** above under **Shell Variables**).

Event Designators

An event designator is a reference to a command line entry in the history list.

- **!** Start a history substitution, except when followed by a **blank**, newline, carriage return, **=** or **(** (when the **extglob** shell option is enabled using the **shopt** builtin).
- **!n** Refer to command line *n*.
- **!-n** Refer to the current command line minus *n*.
- **!!** Refer to the previous command. This is a synonym for **!-1**.
- **!string** Refer to the most recent command starting with *string*.
- **!?string[?]** Refer to the most recent command containing *string*. The trailing **?** may be omitted if *string* is followed immediately by a newline.

- **d^{ustring1}d^{ustring2}d^u** Quick substitution. Repeat the last command, replacing *string1* with *string2*. Equivalent to “`!!:s/string1/string2/`” (see **Modifiers** below).
- **!#** The entire command line typed so far.

Word Designators

Word designators are used to select desired words from the event. A **:** separates the event specification from the word designator. It may be omitted if the word designator begins with a **^**, **\$**, *****, **-**, or **%**. Words are numbered from the beginning of the line, with the first word being denoted by 0 (zero). Words are inserted into the current line separated by single spaces.

- **0 (zero)** The zeroth word. For the shell, this is the command word.
- **n** The *n*th word.
- **^** The first argument. That is, word 1.
- **\$** The last argument.
- **%** The word matched by the most recent ‘*?string?*’ search.
- **x-y** A range of words; ‘-y’ abbreviates ‘0-y’.
- ***** All of the words but the zeroth. This is a synonym for ‘1-\$’. It is not an error to use ***** if there is just one word in the event; the empty string is returned in that case.
- **x*** Abbreviates *x-\$*.
- **x-** Abbreviates *x-\$* like **x***, but omits the last word.

If a word designator is supplied without an event specification, the previous command is used as the event.

Modifiers

After the optional word designator, there may appear a sequence of one or more of the following modifiers, each preceded by a ‘:’.

- **h** Remove a trailing file name component, leaving only the head.
- **t** Remove all leading file name components, leaving the tail.
- **r** Remove a trailing suffix of the form *.xxx*, leaving the basename.
- **e** Remove all but the trailing suffix.
- **p** Print the new command but do not execute it.
- **q** Quote the substituted words, escaping further substitutions.
- **x** Quote the substituted words as with **q**, but break into words at **blanks** and newlines.
- **s/old/new/** Substitute *new* for the first occurrence of *old* in the event line. Any delimiter can be used in place of **/**. The final delimiter is optional if it is the last character of the event line. The delimiter may be quoted in *old* and *new* with a single backslash. If **&** appears in *new*, it is replaced by *old*. A single backslash will quote the **&**. If *old* is null, it is set to the last *old* substituted, or, if no previous history substitutions took place, the last *string* in a `!string[?]` search.

- **&** Repeat the previous substitution.
- **g** Cause changes to be applied over the entire event line. This is used in conjunction with **:s** (e.g., **:gs/old/new/**) or **:&**. If used with **:s**, any delimiter can be used in place of **/**, and the final delimiter is optional if it is the last character of the event line. An **a** may be used as a synonym for **g**.
- **G** Apply the following **s** modifier once to each word in the event line.

7.14.31 Shell Builtin Commands

Unless otherwise noted, each builtin command documented in this section as accepting options preceded by **-** accepts **-** to signify the end of the options.

- **:** [*arguments*] No effect; the command does nothing beyond expanding *arguments* and performing any specified redirections. A zero exit code is returned.
- **.** *filename* [*arguments*]
- **source** *filename* [*arguments*] Read and execute commands from *filename* in the current shell environment and return the exit status of the last command executed from *filename*. If *filename* does not contain a slash, file names in **PATH** are used to find the directory containing *filename*. The file searched for in **PATH** need not be executable. When **bash** is not in *posix mode*, the current directory is searched if no file is found in **PATH**. If the **sourcepath** option to the **shopt** builtin command is turned off, the **PATH** is not searched. If any *arguments* are supplied, they become the positional parameters when *filename* is executed. Otherwise the positional parameters are unchanged. The return status is the status of the last command exited within the script (0 if no commands are executed), and false if *filename* is not found or cannot be read.
- **alias** [**-p**] [*name*[=*value*] ...] **Alias** with no arguments or with the **-p** option prints the list of aliases in the form **alias name=value** on standard output. When arguments are supplied, an alias is defined for each *name* whose *value* is given. A trailing space in *value* causes the next word to be checked for alias substitution when the alias is expanded. For each *name* in the argument list for which no *value* is supplied, the name and value of the alias is printed. **Alias** returns true unless a *name* is given for which no alias has been defined.
- **bg** [*jobspec*] Resume the suspended job *jobspec* in the background, as if it had been started with **&**. If *jobspec* is not present, the shell's notion of the *current job* is used. **bg jobspec** returns 0 unless run when job control is disabled or, when run with job control enabled, if *jobspec* was not found or started without job control.
- **bind** [**-m** *keymap*] [**-lpsvPSV**]
- **bind** [**-m** *keymap*] [**-q** *function*] [**-u** *function*] [**-r** *keyseq*]
- **bind** [**-m** *keymap*] **-f** *filename*
- **bind** [**-m** *keymap*] **-x** *keyseq:shell-command*
- **bind** [**-m** *keymap*] *keyseq:function-name*

- **bind** *readline-command* Display current **readline** key and function bindings, bind a key sequence to a **readline** function or macro, or set a **readline** variable. Each non-option argument is a command as it would appear in *.inputrc*, but each binding or command must be passed as a separate argument; e.g., '\C-x\C-r: re-read-init-file'. Options, if supplied, have the following meanings:
- **-m** *keymap* Use *keymap* as the keymap to be affected by the subsequent bindings. Acceptable *keymap* names are *emacs*, *emacs-standard*, *emacs-meta*, *emacs-ctlx*, *vi*, *vi-move*, *vi-command*, and *vi-insert*. *vi* is equivalent to *vi-command*; *emacs* is equivalent to *emacs-standard*.
- **-l** List the names of all **readline** functions.
- **-p** Display **readline** function names and bindings in such a way that they can be re-read.
- **-P** List current **readline** function names and bindings.
- **-v** Display **readline** variable names and values in such a way that they can be re-read.
- **-V** List current **readline** variable names and values.
- **-s** Display **readline** key sequences bound to macros and the strings they output in such a way that they can be re-read.
- **-S** Display **readline** key sequences bound to macros and the strings they output.
- **-f** *filename* Read key bindings from *filename*.
- **-q** *function* Query about which keys invoke the named *function*.
- **-u** *function* Unbind all keys bound to the named *function*.
- **-r** *keyseq* Remove any current binding for *keyseq*.
- **-x** *keyseq:shell-command* Cause *shell-command* to be executed whenever *keyseq* is entered.

The return value is 0 unless an unrecognized option is given or an error occurred.

- **break** [*n*] Exit from within a **for**, **while**, **until**, or **select** loop. If *n* is specified, break *n* levels. *n* must be ≥ 1 . If *n* is greater than the number of enclosing loops, all enclosing loops are exited. The return value is 0 unless the shell is not executing a loop when **break** is executed.
- **builtin** *shell-builtin* [*arguments*] Execute the specified shell builtin, passing it *arguments*, and return its exit status. This is useful when defining a function whose name is the same as a shell builtin, retaining the functionality of the builtin within the function. The **cd** builtin is commonly redefined this way. The return status is false if *shell-builtin* is not a shell builtin command.
- **cd** [-L|-P] [*dir*] Change the current directory to *dir*. The variable **HOME** is the default *dir*. The variable **CDPATH** defines the search path for the directory containing *dir*. Alternative directory names in **CDPATH** are separated by a colon (:). A null directory name in **CDPATH** is the same as the current directory, i.e., ".". If *dir* begins with a slash (/), then **CDPATH** is not used. The **-P** option says to use the physical directory structure instead of following symbolic links (see also the **-P** option to the **set** builtin command); the **-L** option forces symbolic links to be followed. An argument of - is equivalent to **\$OLDPWD**. If a non-empty directory name from **CDPATH** is used, or if - is

the first argument, and the directory change is successful, the absolute pathname of the new working directory is written to the standard output. The return value is true if the directory was successfully changed; false otherwise.

- **caller** [*expr*] Returns the context of any active subroutine call (a shell function or a script executed with the `.` or **source** builtins. Without *expr*, **caller** displays the line number and source filename of the current subroutine call. If a non-negative integer is supplied as *expr*, **caller** displays the line number, subroutine name, and source file corresponding to that position in the current execution call stack. This extra information may be used, for example, to print a stack trace. The current frame is frame 0. The return value is 0 unless the shell is not executing a subroutine call or *expr* does not correspond to a valid position in the call stack.
- **command** [-pVv] *command* [*arg* ...] Run *command* with *args* suppressing the normal shell function lookup. Only builtin commands or commands found in the **PATH** are executed. If the **-p** option is given, the search for *command* is performed using a default value for **PATH** that is guaranteed to find all of the standard utilities. If either the **-V** or **-v** option is supplied, a description of *command* is printed. The **-v** option causes a single word indicating the command or file name used to invoke *command* to be displayed; the **-V** option produces a more verbose description. If the **-V** or **-v** option is supplied, the exit status is 0 if *command* was found, and 1 if not. If neither option is supplied and an error occurred or *command* cannot be found, the exit status is 127. Otherwise, the exit status of the **command** builtin is the exit status of *command*.
- **compgen** [*option*] [*word*] Generate possible completion matches for *word* according to the *options*, which may be any option accepted by the **complete** builtin with the exception of **-p** and **-r**, and write the matches to the standard output. When using the **-F** or **-C** options, the various shell variables set by the programmable completion facilities, while available, will not have useful values.

The matches will be generated in the same way as if the programmable completion code had generated them directly from a completion specification with the same flags. If *word* is specified, only those completions matching *word* will be displayed.

The return value is true unless an invalid option is supplied, or no matches were generated.

- **complete** [-abcdefgjkusv] [-o *comp-option*] [-A *action*] [-G *globpat*] [-W *wordlist*] [-P *prefix*] [-S *suffix*]
[-X *filterpat*] [-F *function*] [-C *command*] *name* [*name* ...]
- **complete -pr** [*name* ...] Specify how arguments to each *name* should be completed. If the **-p** option is supplied, or if no options are supplied, existing completion specifications are printed in a way that allows them to be reused as input. The **-r** option removes a completion specification for each *name*, or, if no *names* are supplied, all completion specifications.

The process of applying these completion specifications when word completion is attempted is described above under **Programmable Completion**.

Other options, if specified, have the following meanings. The arguments to the **-G**, **-W**, and **-X** options (and, if necessary, the **-P** and **-S** options) should be quoted to protect them from expansion before the **complete** builtin is invoked.

- **-o** *comp-option* The *comp-option* controls several aspects of the `compspec`'s behavior beyond the simple generation of completions. *comp-option* may be one of:

- **bashdefault** Perform the rest of the default **bash** completions if the **compspec** generates no matches.
- **default** Use readline's default filename completion if the **compspec** generates no matches.
- **dirnames** Perform directory name completion if the **compspec** generates no matches.
- **filenames** Tell readline that the **compspec** generates filenames, so it can perform any filename-specific processing (like adding a slash to directory names or suppressing trailing spaces). Intended to be used with shell functions.
- **nospace** Tell readline not to append a space (the default) to words completed at the end of the line.
- **-A action** The *action* may be one of the following to generate a list of possible completions:
 - **alias** Alias names. May also be specified as **-a**.
 - **arrayvar** Array variable names.
 - **binding** **Readline** key binding names.
 - **builtin** Names of shell builtin commands. May also be specified as **-b**.
 - **command** Command names. May also be specified as **-c**.
 - **directory** Directory names. May also be specified as **-d**.
 - **disabled** Names of disabled shell builtins.
 - **enabled** Names of enabled shell builtins.
 - **export** Names of exported shell variables. May also be specified as **-e**.
 - **file** File names. May also be specified as **-f**.
 - **function** Names of shell functions.
 - **group** Group names. May also be specified as **-g**.
 - **helptopic** Help topics as accepted by the **help** builtin.
 - **hostname** Hostnames, as taken from the file specified by the **HOSTFILE** shell variable.
 - **job** Job names, if job control is active. May also be specified as **-j**.
 - **keyword** Shell reserved words. May also be specified as **-k**.
 - **running** Names of running jobs, if job control is active.
 - **service** Service names. May also be specified as **-s**.
 - **setopt** Valid arguments for the **-o** option to the **set** builtin.
 - **shopt** Shell option names as accepted by the **shopt** builtin.
 - **signal** Signal names.
 - **stopped** Names of stopped jobs, if job control is active.

- **user** User names. May also be specified as **-u**.
- **variable** Names of all shell variables. May also be specified as **-v**.
- **-G globpat** The filename expansion pattern *globpat* is expanded to generate the possible completions.
- **-W wordlist** The *wordlist* is split using the characters in the **IFS** special variable as delimiters, and each resultant word is expanded. The possible completions are the members of the resultant list which match the word being completed.
- **-C command** *command* is executed in a subshell environment, and its output is used as the possible completions.
- **-F function** The shell function *function* is executed in the current shell environment. When it finishes, the possible completions are retrieved from the value of the **COMP_REPLY** array variable.
- **-X filterpat** *filterpat* is a pattern as used for filename expansion. It is applied to the list of possible completions generated by the preceding options and arguments, and each completion matching *filterpat* is removed from the list. A leading **!** in *filterpat* negates the pattern; in this case, any completion not matching *filterpat* is removed.
- **-P prefix** *prefix* is added at the beginning of each possible completion after all other options have been applied.
- **-S suffix** *suffix* is appended to each possible completion after all other options have been applied.

The return value is true unless an invalid option is supplied, an option other than **-p** or **-r** is supplied without a *name* argument, an attempt is made to remove a completion specification for a *name* for which no specification exists, or an error occurs adding a completion specification.

- **continue** [*n*] Resume the next iteration of the enclosing **for**, **while**, **until**, or **select** loop. If *n* is specified, resume at the *n*th enclosing loop. *n* must be ≥ 1 . If *n* is greater than the number of enclosing loops, the last enclosing loop (the “top-level” loop) is resumed. The return value is 0 unless the shell is not executing a loop when **continue** is executed.
- **declare** [**-afFirtx**] [**-p**] [*name*[=*value*] ...]
- **typeset** [**-afFirtx**] [**-p**] [*name*[=*value*] ...] Declare variables and/or give them attributes. If no *names* are given then display the values of variables. The **-p** option will display the attributes and values of each *name*. When **-p** is used, additional options are ignored. The **-F** option inhibits the display of function definitions; only the function name and attributes are printed. If the **extdebug** shell option is enabled using **shopt**, the source file name and line number where the function is defined are displayed as well. The **-F** option implies **-f**. The following options can be used to restrict output to variables with the specified attribute or to give variables attributes:
 - **-a** Each *name* is an array variable (see **Arrays** above).
 - **-f** Use function names only.
 - **-i** The variable is treated as an integer; arithmetic evaluation (see **ARITHMETIC EVALUATION**) is performed when the variable is assigned a value.

- **-r** Make *names* readonly. These names cannot then be assigned values by subsequent assignment statements or unset.
- **-t** Give each *name* the *trace* attribute. Traced functions inherit the **DEBUG** trap from the calling shell. The trace attribute has no special meaning for variables.
- **-x** Mark *names* for export to subsequent commands via the environment.

Using '+' instead of '-' turns off the attribute instead, with the exception that **+a** may not be used to destroy an array variable. When used in a function, makes each *name* local, as with the **local** command. If a variable name is followed by *=value*, the value of the variable is set to *value*. The return value is 0 unless an invalid option is encountered, an attempt is made to define a function using "-f foo=bar", an attempt is made to assign a value to a readonly variable, an attempt is made to assign a value to an array variable without using the compound assignment syntax (see **Arrays** above), one of the *names* is not a valid shell variable name, an attempt is made to turn off readonly status for a readonly variable, an attempt is made to turn off array status for an array variable, or an attempt is made to display a non-existent function with **-f**.

- **dirs [-clpv] [+n] [-n]** Without options, displays the list of currently remembered directories. The default display is on a single line with directory names separated by spaces. Directories are added to the list with the **pushd** command; the **popd** command removes entries from the list.
- **+n** Displays the *n*th entry counting from the left of the list shown by **dirs** when invoked without options, starting with zero.
- **-n** Displays the *n*th entry counting from the right of the list shown by **dirs** when invoked without options, starting with zero.
- **-c** Clears the directory stack by deleting all of the entries.
- **-l** Produces a longer listing; the default listing format uses a tilde to denote the home directory.
- **-p** Print the directory stack with one entry per line.
- **-v** Print the directory stack with one entry per line, prefixing each entry with its index in the stack.

The return value is 0 unless an invalid option is supplied or *n* indexes beyond the end of the directory stack.

- **disown [-ar] [-h] [jobspec ...]** Without options, each *jobspec* is removed from the table of active jobs. If the **-h** option is given, each *jobspec* is not removed from the table, but is marked so that **SIGHUP** is not sent to the job if the shell receives a **SIGHUP**. If no *jobspec* is present, and neither the **-a** nor the **-r** option is supplied, the *current job* is used. If no *jobspec* is supplied, the **-a** option means to remove or mark all jobs; the **-r** option without a *jobspec* argument restricts operation to running jobs. The return value is 0 unless a *jobspec* does not specify a valid job.
- **echo [-neE] [arg ...]** Output the *args*, separated by spaces, followed by a newline. The return status is always 0. If **-n** is specified, the trailing newline is suppressed. If the **-e** option is given, interpretation of the following backslash-escaped characters is enabled. The **-E** option disables the interpretation of these escape characters, even on systems where they are interpreted by default. The **xpg_echo** shell option may be used to dynamically determine whether or not **echo** expands these escape characters by default. **echo** does not interpret **-** to mean the end of options. **echo** interprets the following escape sequences:

- `\a` alert (bell)
- `\b` backspace
- `\c` suppress trailing newline
- `\e` an escape character
- `\f` form feed
- `\n` new line
- `\r` carriage return
- `\t` horizontal tab
- `\v` vertical tab
- `\\` backslash
- `\0nnn` the eight-bit character whose value is the octal value *nnn* (zero to three octal digits)
- `\nnn` the eight-bit character whose value is the octal value *nnn* (one to three octal digits)
- `\xHH` the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)
- **enable** [-adnps] [-f *filename*] [*name* ...] Enable and disable builtin shell commands. Disabling a builtin allows a disk command which has the same name as a shell builtin to be executed without specifying a full pathname, even though the shell normally searches for builtins before disk commands. If **-n** is used, each *name* is disabled; otherwise, *names* are enabled. For example, to use the **test** binary found via the **PATH** instead of the shell builtin version, run “enable -n test”. The **-f** option means to load the new builtin command *name* from shared object *filename*, on systems that support dynamic loading. The **-d** option will delete a builtin previously loaded with **-f**. If no *name* arguments are given, or if the **-p** option is supplied, a list of shell builtins is printed. With no other option arguments, the list consists of all enabled shell builtins. If **-n** is supplied, only disabled builtins are printed. If **-a** is supplied, the list printed includes all builtins, with an indication of whether or not each is enabled. If **-s** is supplied, the output is restricted to the POSIX *special* builtins. The return value is 0 unless a *name* is not a shell builtin or there is an error loading a new builtin from a shared object.
- **eval** [*arg* ...] The *args* are read and concatenated together into a single command. This command is then read and executed by the shell, and its exit status is returned as the value of **eval**. If there are no *args*, or only null arguments, **eval** returns 0.
- **exec** [-cl] [-a *name*] [*command* [*arguments*]] If *command* is specified, it replaces the shell. No new process is created. The *arguments* become the arguments to *command*. If the **-l** option is supplied, the shell places a dash at the beginning of the zeroth arg passed to *command*. This is what *login(1)* does. The **-c** option causes *command* to be executed with an empty environment. If **-a** is supplied, the shell passes *name* as the zeroth argument to the executed command. If *command* cannot be executed for some reason, a non-interactive shell exits, unless the shell option **execfail** is enabled, in which case it returns failure. An interactive shell returns failure if the file cannot be executed. If *command* is not specified, any redirections take effect in the current shell, and the return status is 0. If there is a redirection error, the return status is 1.

- **exit** [*n*] Cause the shell to exit with a status of *n*. If *n* is omitted, the exit status is that of the last command executed. A trap on **EXIT** is executed before the shell terminates.
- **export** [-fn] [*name*[=*word*]] ...
- **export -p** The supplied *names* are marked for automatic export to the environment of subsequently executed commands. If the **-f** option is given, the *names* refer to functions. If no *names* are given, or if the **-p** option is supplied, a list of all names that are exported in this shell is printed. The **-n** option causes the export property to be removed from each *name*. If a variable name is followed by =*word*, the value of the variable is set to *word*. **export** returns an exit status of 0 unless an invalid option is encountered, one of the *names* is not a valid shell variable name, or **-f** is supplied with a *name* that is not a function.
- **fc** [-e *ename*] [-nlr] [*first*] [*last*]
- **fc -s** [*pat=rep*] [*cmd*] Fix Command. In the first form, a range of commands from *first* to *last* is selected from the history list. *First* and *last* may be specified as a string (to locate the last command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). If *last* is not specified it is set to the current command for listing (so that “fc -l -10” prints the last 10 commands) and to *first* otherwise. If *first* is not specified it is set to the previous command for editing and -16 for listing.

The **-n** option suppresses the command numbers when listing. The **-r** option reverses the order of the commands. If the **-l** option is given, the commands are listed on standard output. Otherwise, the editor given by *ename* is invoked on a file containing those commands. If *ename* is not given, the value of the **FCEDIT** variable is used, and the value of **EDITOR** if **FCEDIT** is not set. If neither variable is set, *vi* is used. When editing is complete, the edited commands are echoed and executed.

In the second form, *command* is re-executed after each instance of *pat* is replaced by *rep*. A useful alias to use with this is “r=fc -s”, so that typing “r cc” runs the last command beginning with “cc” and typing “r” re-executes the last command.

If the first form is used, the return value is 0 unless an invalid option is encountered or *first* or *last* specify history lines out of range. If the **-e** option is supplied, the return value is the value of the last command executed or failure if an error occurs with the temporary file of commands. If the second form is used, the return status is that of the command re-executed, unless *cmd* does not specify a valid history line, in which case **fc** returns failure.

- **fg** [*jobspec*] Resume *jobspec* in the foreground, and make it the current job. If *jobspec* is not present, the shell’s notion of the *current job* is used. The return value is that of the command placed into the foreground, or failure if run when job control is disabled or, when run with job control enabled, if *jobspec* does not specify a valid job or *jobspec* specifies a job that was started without job control.
- **getopts** *optstring name* [*args*] **getopts** is used by shell procedures to parse positional parameters. *optstring* contains the option characters to be recognized; if a character is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. The colon and question mark characters may not be used as option characters. Each time it is invoked, **getopts** places the next option in the shell variable *name*, initializing *name* if it does not exist, and the index of the next argument to be processed into the variable **OPTIND**. **OPTIND** is initialized to 1 each time the

shell or a shell script is invoked. When an option requires an argument, **getopts** places that argument into the variable **OPTARG**. The shell does not reset **OPTIND** automatically; it must be manually reset between multiple calls to **getopts** within the same shell invocation if a new set of parameters is to be used.

When the end of options is encountered, **getopts** exits with a return value greater than zero. **OPTIND** is set to the index of the first non-option argument, and **name** is set to ?.

getopts normally parses the positional parameters, but if more arguments are given in *args*, **getopts** parses those instead.

getopts can report errors in two ways. If the first character of *optstring* is a colon, *silent* error reporting is used. In normal operation diagnostic messages are printed when invalid options or missing option arguments are encountered. If the variable **OPTERR** is set to 0, no error messages will be displayed, even if the first character of *optstring* is not a colon.

If an invalid option is seen, **getopts** places ? into *name* and, if not silent, prints an error message and unsets **OPTARG**. If **getopts** is silent, the option character found is placed in **OPTARG** and no diagnostic message is printed.

If a required argument is not found, and **getopts** is not silent, a question mark (?) is placed in *name*, **OPTARG** is unset, and a diagnostic message is printed. If **getopts** is silent, then a colon (:) is placed in *name* and **OPTARG** is set to the option character found.

getopts returns true if an option, specified or unspecified, is found. It returns false if the end of options is encountered or an error occurs.

- **hash** [-lr] [-p *filename*] [-dt] [*name*] For each *name*, the full file name of the command is determined by searching the directories in **\$PATH** and remembered. If the **-p** option is supplied, no path search is performed, and *filename* is used as the full file name of the command. The **-r** option causes the shell to forget all remembered locations. The **-d** option causes the shell to forget the remembered location of each *name*. If the **-t** option is supplied, the full pathname to which each *name* corresponds is printed. If multiple *name* arguments are supplied with **-t**, the *name* is printed before the hashed full pathname. The **-l** option causes output to be displayed in a format that may be reused as input. If no arguments are given, or if only **-l** is supplied, information about remembered commands is printed. The return status is true unless a *name* is not found or an invalid option is supplied.
- **help** [-s] [*pattern*] Display helpful information about builtin commands. If *pattern* is specified, **help** gives detailed help on all commands matching *pattern*; otherwise help for all the builtins and shell control structures is printed. The **-s** option restricts the information displayed to a short usage synopsis. The return status is 0 unless no command matches *pattern*.
- **history** [*n*]
- **history -c**
- **history -d** *offset*
- **history -anrw** [*filename*]
- **history -p** *arg* [*arg* ...]

- **history -s** *arg* [*arg* ...] With no options, display the command history list with line numbers. Lines listed with a * have been modified. An argument of *n* lists only the last *n* lines. If the shell variable **HISTTIMEFORMAT** is set and not null, it is used as a format string for *strftime*(3) to display the time stamp associated with each displayed history entry. No intervening blank is printed between the formatted time stamp and the history line. If *filename* is supplied, it is used as the name of the history file; if not, the value of **HISTFILE** is used. Options, if supplied, have the following meanings:
 - **-c** Clear the history list by deleting all the entries.
 - **-d** *offset* Delete the history entry at position *offset*.
 - **-a** Append the “new” history lines (history lines entered since the beginning of the current **bash** session) to the history file.
 - **-n** Read the history lines not already read from the history file into the current history list. These are lines appended to the history file since the beginning of the current **bash** session.
 - **-r** Read the contents of the history file and use them as the current history.
 - **-w** Write the current history to the history file, overwriting the history file’s contents.
 - **-p** Perform history substitution on the following *args* and display the result on the standard output. Does not store the results in the history list. Each *arg* must be quoted to disable normal history expansion.
 - **-s** Store the *args* in the history list as a single entry. The last command in the history list is removed before the *args* are added.

If the **HISTTIMEFORMAT** is set, the time stamp information associated with each history entry is written to the history file. The return value is 0 unless an invalid option is encountered, an error occurs while reading or writing the history file, an invalid *offset* is supplied as an argument to **-d**, or the history expansion supplied as an argument to **-p** fails.

- **jobs [-lnprs]** [*jobspec* ...]
- **jobs -x** *command* [*args* ...] The first form lists the active jobs. The options have the following meanings:
 - **-l** List process IDs in addition to the normal information.
 - **-p** List only the process ID of the job’s process group leader.
 - **-n** Display information only about jobs that have changed status since the user was last notified of their status.
 - **-r** Restrict output to running jobs.
 - **-s** Restrict output to stopped jobs.

If *jobspec* is given, output is restricted to information about that job. The return status is 0 unless an invalid option is encountered or an invalid *jobspec* is supplied.

If the **-x** option is supplied, **jobs** replaces any *jobspec* found in *command* or *args* with the corresponding process group ID, and executes *command* passing it *args*, returning its exit status.

- **kill [-s sigspec | -n signum | -sigspec]** [*pid* | *jobspec*] ...

- **kill -l** [*sigspec* | *exit_status*] Send the signal named by *sigspec* or *signal* to the processes named by *pid* or *jobspec*. *sigspec* is either a case-insensitive signal name such as **SIGKILL** (with or without the **SIG** prefix) or a signal number; *signal* is a signal number. If *sigspec* is not present, then **SIGTERM** is assumed. An argument of **-l** lists the signal names. If any arguments are supplied when **-l** is given, the names of the signals corresponding to the arguments are listed, and the return status is 0. The *exit_status* argument to **-l** is a number specifying either a signal number or the exit status of a process terminated by a signal. **kill** returns true if at least one signal was successfully sent, or false if an error occurs or an invalid option is encountered.
- **let** *arg* [*arg* ...] Each *arg* is an arithmetic expression to be evaluated (see **ARITHMETIC EVALUATION**). If the last *arg* evaluates to 0, **let** returns 1; 0 is returned otherwise.
- **local** [*option*] [*name*[=*value*] ...] For each argument, a local variable named *name* is created, and assigned *value*. The *option* can be any of the options accepted by **declare**. When **local** is used within a function, it causes the variable *name* to have a visible scope restricted to that function and its children. With no operands, **local** writes a list of local variables to the standard output. It is an error to use **local** when not within a function. The return status is 0 unless **local** is used outside a function, an invalid *name* is supplied, or *name* is a readonly variable.
- **logout** Exit a login shell.
- **popd** [**-n**] [**+n**] [**-n**] Removes entries from the directory stack. With no arguments, removes the top directory from the stack, and performs a **cd** to the new top directory. Arguments, if supplied, have the following meanings:
 - **+n** Removes the *n*th entry counting from the left of the list shown by **dirs**, starting with zero. For example: “**popd +0**” removes the first directory, “**popd +1**” the second.
 - **-n** Removes the *n*th entry counting from the right of the list shown by **dirs**, starting with zero. For example: “**popd -0**” removes the last directory, “**popd -1**” the next to last.
 - **-n** Suppresses the normal change of directory when removing directories from the stack, so that only the stack is manipulated.

If the **popd** command is successful, a **dirs** is performed as well, and the return status is 0. **popd** returns false if an invalid option is encountered, the directory stack is empty, a non-existent directory stack entry is specified, or the directory change fails.

- **printf** *format* [*arguments*] Write the formatted *arguments* to the standard output under the control of the *format*. The *format* is a character string which contains three types of objects: plain characters, which are simply copied to standard output, character escape sequences, which are converted and copied to the standard output, and format specifications, each of which causes printing of the next successive *argument*. In addition to the standard *printf*(1) formats, **%b** causes **printf** to expand backslash escape sequences in the corresponding *argument* (except that **\c** terminates output, backslashes in **\'**, ****, and **\?** are not removed, and octal escapes beginning with **\0** may contain up to four digits), and **%q** causes **printf** to output the corresponding *argument* in a format that can be reused as shell input.

The *format* is reused as necessary to consume all of the *arguments*. If the *format* requires more *arguments* than are supplied, the extra format specifications behave as if a zero value or null string, as appropriate, had been supplied. The return value is zero on success, non-zero on failure.

- **pushd** [-n] [*dir*]
- **pushd** [-n] [+n] [-n] Adds a directory to the top of the directory stack, or rotates the stack, making the new top of the stack the current working directory. With no arguments, exchanges the top two directories and returns 0, unless the directory stack is empty. Arguments, if supplied, have the following meanings:
 - +n Rotates the stack so that the *n*th directory (counting from the left of the list shown by **dirs**, starting with zero) is at the top.
 - -n Rotates the stack so that the *n*th directory (counting from the right of the list shown by **dirs**, starting with zero) is at the top.
 - -n Suppresses the normal change of directory when adding directories to the stack, so that only the stack is manipulated.
 - *dir* Adds *dir* to the directory stack at the top, making it the new current working directory.

If the **pushd** command is successful, a **dirs** is performed as well. If the first form is used, **pushd** returns 0 unless the **cd** to *dir* fails. With the second form, **pushd** returns 0 unless the directory stack is empty, a non-existent directory stack element is specified, or the directory change to the specified new current directory fails.

- **pwd** [-LP] Print the absolute pathname of the current working directory. The pathname printed contains no symbolic links if the **-P** option is supplied or the **-o physical** option to the **set** builtin command is enabled. If the **-L** option is used, the pathname printed may contain symbolic links. The return status is 0 unless an error occurs while reading the name of the current directory or an invalid option is supplied.
- **read** [-ers] [-u *fd*] [-t *timeout*] [-a *aname*] [-p *prompt*] [-n *nchars*] [-d *delim*] [*name ...*]
One line is read from the standard input, or from the file descriptor *fd* supplied as an argument to the **-u** option, and the first word is assigned to the first *name*, the second word to the second *name*, and so on, with leftover words and their intervening separators assigned to the last *name*. If there are fewer words read from the input stream than names, the remaining names are assigned empty values. The characters in **IFS** are used to split the line into words. The backslash character (\) may be used to remove any special meaning for the next character read and for line continuation. Options, if supplied, have the following meanings:
 - -a *aname* The words are assigned to sequential indices of the array variable *aname*, starting at 0. *aname* is unset before any new values are assigned. Other *name* arguments are ignored.
 - -d *delim* The first character of *delim* is used to terminate the input line, rather than newline.
 - -e If the standard input is coming from a terminal, **readline** (see **READLINE** above) is used to obtain the line.
 - -n *nchars* **read** returns after reading *nchars* characters rather than waiting for a complete line of input.
 - -p *prompt* Display *prompt* on standard error, without a trailing newline, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.

- **-r** Backslash does not act as an escape character. The backslash is considered to be part of the line. In particular, a backslash-newline pair may not be used as a line continuation.
- **-s** Silent mode. If input is coming from a terminal, characters are not echoed.
- **-t** *timeout* Cause **read** to time out and return failure if a complete line of input is not read within *timeout* seconds. This option has no effect if **read** is not reading input from the terminal or a pipe.
- **-u** *fdFP* Read input from file descriptor *fd*.

If no *names* are supplied, the line read is assigned to the variable **REPLY**. The return code is zero, unless end-of-file is encountered, **read** times out, or an invalid file descriptor is supplied as the argument to **-u**.

- **readonly** [-**apf**] [*name*[=*word*] ...] The given *names* are marked readonly; the values of these *names* may not be changed by subsequent assignment. If the **-f** option is supplied, the functions corresponding to the *names* are so marked. The **-a** option restricts the variables to arrays. If no *name* arguments are given, or if the **-p** option is supplied, a list of all readonly names is printed. The **-p** option causes output to be displayed in a format that may be reused as input. If a variable name is followed by =*word*, the value of the variable is set to *word*. The return status is 0 unless an invalid option is encountered, one of the *names* is not a valid shell variable name, or **-f** is supplied with a *name* that is not a function.
- **return** [*n*] Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed in the function body. If used outside a function, but during execution of a script by the . (**source**) command, it causes the shell to stop executing that script and return either *n* or the exit status of the last command executed within the script as the exit status of the script. If used outside a function and not during execution of a script by ., the return status is false. Any command associated with the **RETURN** trap is executed before execution resumes after the function or script.
- **set** [-**abefhkmnptuvxBCHP**] [-**o** *option*] [*arg* ...] Without options, the name and value of each shell variable are displayed in a format that can be reused as input. The output is sorted according to the current locale. When options are specified, they set or unset shell attributes. Any arguments remaining after the options are processed are treated as values for the positional parameters and are assigned, in order, to **\$1**, **\$2**, ... **\$n**. Options, if specified, have the following meanings:
 - **-a** Automatically mark variables and functions which are modified or created for export to the environment of subsequent commands.
 - **-b** Report the status of terminated background jobs immediately, rather than before the next primary prompt. This is effective only when job control is enabled.
 - **-e** Exit immediately if a *simple command* (see **SHELL GRAMMAR** above) exits with a non-zero status. The shell does not exit if the command that fails is part of the command list immediately following a **while** or **until** keyword, part of the test in an *if* statement, part of a **&&** or **||** list, or if the command's return value is being inverted via **!**. A trap on **ERR**, if set, is executed before the shell exits.
 - **-f** Disable pathname expansion.

- **-h** Remember the location of commands as they are looked up for execution. This is enabled by default.
- **-k** All arguments in the form of assignment statements are placed in the environment for a command, not just those that precede the command name.
- **-m** Monitor mode. Job control is enabled. This option is on by default for interactive shells on systems that support it (see **JOB CONTROL** above). Background processes run in a separate process group and a line containing their exit status is printed upon their completion.
- **-n** Read commands but do not execute them. This may be used to check a shell script for syntax errors. This is ignored by interactive shells.
- **-o** *option-name* The *option-name* can be one of the following:
 - **allexport** Same as **-a**.
 - **braceexpand** Same as **-B**.
 - **emacs** Use an emacs-style command line editing interface. This is enabled by default when the shell is interactive, unless the shell is started with the **-noediting** option.
 - **errtrace** Same as **-E**.
 - **functrace** Same as **-T**.
 - **errexit** Same as **-e**.
 - **hashall** Same as **-h**.
 - **histexpand** Same as **-H**.
 - **history** Enable command history, as described above under **HISTORY**. This option is on by default in interactive shells.
 - **ignoreeof** The effect is as if the shell command “IGNOREEOF=10” had been executed (see **Shell Variables** above).
 - **keyword** Same as **-k**.
 - **monitor** Same as **-m**.
 - **noclobber** Same as **-C**.
 - **noexec** Same as **-n**.
 - **noglob** Same as **-f**. **nolog** Currently ignored.
 - **notify** Same as **-b**.
 - **nounset** Same as **-u**.
 - **onecmd** Same as **-t**.
 - **physical** Same as **-P**.
 - **pipefail** If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

- **posix** Change the behavior of **bash** where the default operation differs from the POSIX 1003.2 standard to match the standard (*'posix mode'*).
- **privileged** Same as **-p**.
- **verbose** Same as **-v**.
- **vi** Use a vi-style command line editing interface.
- **xtrace** Same as **-x**.

If **-o** is supplied with no *option-name*, the values of the current options are printed. If **+o** is supplied with no *option-name*, a series of **set** commands to recreate the current option settings is displayed on the standard output.

- **-p** Turn on *privileged* mode. In this mode, the **\$ENV** and **\$BASH_ENV** files are not processed, shell functions are not inherited from the environment, and the **SHELLOPTS** variable, if it appears in the environment, is ignored. If the shell is started with the effective user (group) id not equal to the real user (group) id, and the **-p** option is not supplied, these actions are taken and the effective user id is set to the real user id. If the **-p** option is supplied at startup, the effective user id is not reset. Turning this option off causes the effective user and group ids to be set to the real user and group ids.
- **-t** Exit after reading and executing one command.
- **-u** Treat unset variables as an error when performing parameter expansion. If expansion is attempted on an unset variable, the shell prints an error message, and, if not interactive, exits with a non-zero status.
- **-v** Print shell input lines as they are read.
- **-x** After expanding each *simple command*, **for** command, **case** command, **select** command, or arithmetic **for** command, display the expanded value of **PS4**, followed by the command and its expanded arguments or associated word list.
- **-B** The shell performs brace expansion (see **Brace Expansion** above). This is on by default.
- **-C** If set, **bash** does not overwrite an existing file with the **>**, **>&**, and **<>** redirection operators. This may be overridden when creating output files by using the redirection operator **>|** instead of **>**.
- **-E** If set, any trap on **ERR** is inherited by shell functions, command substitutions, and commands executed in a subshell environment. The **ERR** trap is normally not inherited in such cases.
- **-H** Enable **!** style history substitution. This option is on by default when the shell is interactive.
- **-P** If set, the shell does not follow symbolic links when executing commands such as **cd** that change the current working directory. It uses the physical directory structure instead. By default, **bash** follows the logical chain of directories when performing commands which change the current directory.
- **-T** If set, any trap on **DEBUG** is inherited by shell functions, command substitutions, and commands executed in a subshell environment. The **DEBUG** trap is normally not inherited in such cases.

- – If no arguments follow this option, then the positional parameters are unset. Otherwise, the positional parameters are set to the *args*, even if some of them begin with a *-*.
- - Signal the end of options, cause all remaining *args* to be assigned to the positional parameters. The **-x** and **-v** options are turned off. If there are no *args*, the positional parameters remain unchanged.

The options are off by default unless otherwise noted. Using *+* rather than *-* causes these options to be turned off. The options can also be specified as arguments to an invocation of the shell. The current set of options may be found in **\$-**. The return status is always true unless an invalid option is encountered.

- **shift** [*n*] The positional parameters from *n*+1 ... are renamed to **\$1** Parameters represented by the numbers **\$#** down to **\$#-n+1** are unset. *n* must be a non-negative number less than or equal to **\$#**. If *n* is 0, no parameters are changed. If *n* is not given, it is assumed to be 1. If *n* is greater than **\$#**, the positional parameters are not changed. The return status is greater than zero if *n* is greater than **\$#** or less than zero; otherwise 0.
- **shopt** [-**pqsu**] [-**o**] [*optname* ...] Toggle the values of variables controlling optional shell behavior. With no options, or with the **-p** option, a list of all settable options is displayed, with an indication of whether or not each is set. The **-p** option causes output to be displayed in a form that may be reused as input. Other options have the following meanings:
 - **-s** Enable (set) each *optname*.
 - **-u** Disable (unset) each *optname*.
 - **-q** Suppresses normal output (quiet mode); the return status indicates whether the *optname* is set or unset. If multiple *optname* arguments are given with **-q**, the return status is zero if all *optnames* are enabled; non-zero otherwise.
 - **-o** Restricts the values of *optname* to be those defined for the **-o** option to the **set** builtin.

If either **-s** or **-u** is used with no *optname* arguments, the display is limited to those options which are set or unset, respectively. Unless otherwise noted, the **shopt** options are disabled (unset) by default.

The return status when listing options is zero if all *optnames* are enabled, non-zero otherwise. When setting or unsetting options, the return status is zero unless an *optname* is not a valid shell option.

The list of **shopt** options is:

- **cdable_vars** If set, an argument to the **cd** builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to.
- **cdspell** If set, minor errors in the spelling of a directory component in a **cd** command will be corrected. The errors checked for are transposed characters, a missing character, and one character too many. If a correction is found, the corrected file name is printed, and the command proceeds. This option is only used by interactive shells.
- **checkhash** If set, **bash** checks that a command found in the hash table exists before trying to execute it. If a hashed command no longer exists, a normal path search is performed.
- **checkwinsize** If set, **bash** checks the window size after each command and, if necessary, updates the values of **LINES** and **COLUMNS**.

- **cmdhist** If set, **bash** attempts to save all lines of a multiple-line command in the same history entry. This allows easy re-editing of multi-line commands.
- **dotglob** If set, **bash** includes filenames beginning with a `'.'` in the results of pathname expansion.
- **execfail** If set, a non-interactive shell will not exit if it cannot execute the file specified as an argument to the **exec** builtin command. An interactive shell does not exit if **exec** fails.
- **expand_aliases** If set, aliases are expanded as described above under **ALIASES**. This option is enabled by default for interactive shells.
- **extdebug** If set, behavior intended for use by debuggers is enabled:
 - 1. The **-F** option to the **declare** builtin displays the source file name and line number corresponding to each function name supplied as an argument.
 - 2. If the command run by the **DEBUG** trap returns a non-zero value, the next command is skipped and not executed.
 - 3. If the command run by the **DEBUG** trap returns a value of 2, and the shell is executing in a subroutine (a shell function or a shell script executed by the `.` or **source** builtins), a call to **return** is simulated.
- **extglob** If set, the extended pattern matching features described above under **Pathname Expansion** are enabled.
- **extquote** If set, `'$string'` and `$$string` quoting is performed within `${parameter}` expansions enclosed in double quotes. This option is enabled by default.
- **failglob** If set, patterns which fail to match filenames during pathname expansion result in an expansion error.
- **force_ignore** If set, the suffixes specified by the **FIGNORE** shell variable cause words to be ignored when performing word completion even if the ignored words are the only possible completions. See **SHELL VARIABLES** above for a description of **FIGNORE**. This option is enabled by default.
- **gnu_errfmt** If set, shell error messages are written in the standard GNU error message format.
- **histappend** If set, the history list is appended to the file named by the value of the **HISTFILE** variable when the shell exits, rather than overwriting the file.
- **histredit** If set, and **readline** is being used, a user is given the opportunity to re-edit a failed history substitution.
- **histverify** If set, and **readline** is being used, the results of history substitution are not immediately passed to the shell parser. Instead, the resulting line is loaded into the **readline** editing buffer, allowing further modification.
- **hostcomplete** If set, and **readline** is being used, **bash** will attempt to perform hostname completion when a word containing a `@` is being completed (see **Completing** under **READLINE** above). This is enabled by default.

- **huponexit** If set, **bash** will send **SIGHUP** to all jobs when an interactive login shell exits.
- **interactive_comments** If set, allow a word beginning with **#** to cause that word and all remaining characters on that line to be ignored in an interactive shell (see **COMMENTS** above). This option is enabled by default.
- **lithist** If set, and the **cmdhist** option is enabled, multi-line commands are saved to the history with embedded newlines rather than using semicolon separators where possible.
- **login_shell** The shell sets this option if it is started as a login shell (see **INVOCATION** above). The value may not be changed.
- **mailwarn** If set, and a file that **bash** is checking for mail has been accessed since the last time it was checked, the message “The mail in *mailfile* has been read” is displayed.
- **no_empty_cmd_completion** If set, and **readline** is being used, **bash** will not attempt to search the **PATH** for possible completions when completion is attempted on an empty line.
- **nocaseglob** If set, **bash** matches filenames in a case-insensitive fashion when performing pathname expansion (see **Pathname Expansion** above).
- **nullglob** If set, **bash** allows patterns which match no files (see **Pathname Expansion** above) to expand to a null string, rather than themselves.
- **progcomp** If set, the programmable completion facilities (see **Programmable Completion** above) are enabled. This option is enabled by default.
- **promptvars** If set, prompt strings undergo parameter expansion, command substitution, arithmetic expansion, and quote removal after being expanded as described in **PROMPTING** above. This option is enabled by default.
- **restricted_shell** The shell sets this option if it is started in restricted mode (see **RESTRICTED SHELL** below). The value may not be changed. This is not reset when the startup files are executed, allowing the startup files to discover whether or not a shell is restricted.
- **shift_verbose** If set, the **shift** builtin prints an error message when the shift count exceeds the number of positional parameters.
- **sourcepath** If set, the **source** (.) builtin uses the value of **PATH** to find the directory containing the file supplied as an argument. This option is enabled by default.
- **xpg_echo** If set, the **echo** builtin expands backslash-escape sequences by default.
- **suspend** [-f] Suspend the execution of this shell until it receives a **SIGCONT** signal. The **-f** option says not to complain if this is a login shell; just suspend anyway. The return status is 0 unless the shell is a login shell and **-f** is not supplied, or if job control is not enabled.
- **test** *expr*
- [*expr*] Return a status of 0 or 1 depending on the evaluation of the conditional expression *expr*. Each operator and operand must be a separate argument. Expressions are composed of the primaries described above under **CONDITIONAL EXPRESSIONS**.

Expressions may be combined using the following operators, listed in decreasing order of precedence.

- **!** *expr* True if *expr* is false.
- **(expr)** Returns the value of *expr*. This may be used to override the normal precedence of operators.
- *expr1* **-a** *expr2* True if both *expr1* and *expr2* are true.
- *expr1* **-o** *expr2* True if either *expr1* or *expr2* is true.

test and **[** evaluate conditional expressions using a set of rules based on the number of arguments.

- 0 arguments The expression is false.
- 1 argument The expression is true if and only if the argument is not null.
- 2 arguments If the first argument is **!**, the expression is true if and only if the second argument is null. If the first argument is one of the unary conditional operators listed above under **CONDITIONAL EXPRESSIONS**, the expression is true if the unary test is true. If the first argument is not a valid unary conditional operator, the expression is false.
- 3 arguments If the second argument is one of the binary conditional operators listed above under **CONDITIONAL EXPRESSIONS**, the result of the expression is the result of the binary test using the first and third arguments as operands. If the first argument is **!**, the value is the negation of the two-argument test using the second and third arguments. If the first argument is exactly **(** and the third argument is exactly **)**, the result is the one-argument test of the second argument. Otherwise, the expression is false. The **-a** and **-o** operators are considered binary operators in this case.
- 4 arguments If the first argument is **!**, the result is the negation of the three-argument expression composed of the remaining arguments. Otherwise, the expression is parsed and evaluated according to precedence using the rules listed above.
- 5 or more arguments The expression is parsed and evaluated according to precedence using the rules listed above.
- **times** Print the accumulated user and system times for the shell and for processes run from the shell. The return status is 0.
- **trap [-lp] [[arg] sigspec ...]** The command *arg* is to be read and executed when the shell receives signal(s) *sigspec*. If *arg* is absent (and there is a single *sigspec*) or **-**, each specified signal is reset to its original disposition (the value it had upon entrance to the shell). If *arg* is the null string the signal specified by each *sigspec* is ignored by the shell and by the commands it invokes. If *arg* is not present and **-p** has been supplied, then the trap commands associated with each *sigspec* are displayed. If no arguments are supplied or if only **-p** is given, **trap** prints the list of commands associated with each signal. The **-l** option causes the shell to print a list of signal names and their corresponding numbers. Each *sigspec* is either a signal name defined in *<signal.h>*, or a signal number. Signal names are case insensitive and the SIG prefix is optional. If a *sigspec* is **EXIT** (0) the command *arg* is executed on exit from the shell. If a *sigspec* is **DEBUG**, the command *arg* is executed before every *simple command*, *for* command, *case* command, *select* command,

every arithmetic *for* command, and before the first command executes in a shell function (see **SHELL GRAMMAR** above). Refer to the description of the **extglob** option to the **shopt** builtin for details of its effect on the **DEBUG** trap. If a *sigspec* is **ERR**, the command *arg* is executed whenever a simple command has a non-zero exit status, subject to the following conditions. The **ERR** trap is not executed if the failed command is part of the command list immediately following a **while** or **until** keyword, part of the test in an *if* statement, part of a **&&** or **||** list, or if the command's return value is being inverted via **!**. These are the same conditions obeyed by the **errexit** option. If a *sigspec* is **RETURN**, the command *arg* is executed each time a shell function or a script executed with the **.** or **source** builtins finishes executing. Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals are reset to their original values in a child process when it is created. The return status is false if any *sigspec* is invalid; otherwise **trap** returns true.

- **type** [-**aftpP**] *name* [*name* ...] With no options, indicate how each *name* would be interpreted if used as a command name. If the **-t** option is used, **type** prints a string which is one of *alias*, *keyword*, *function*, *builtin*, or *file* if *name* is an alias, shell reserved word, function, builtin, or disk file, respectively. If the *name* is not found, then nothing is printed, and an exit status of false is returned. If the **-p** option is used, **type** either returns the name of the disk file that would be executed if *name* were specified as a command name, or nothing if “type -t name” would not return *file*. The **-P** option forces a **PATH** search for each *name*, even if “type -t name” would not return *file*. If a command is hashed, **-p** and **-P** print the hashed value, not necessarily the file that appears first in **PATH**. If the **-a** option is used, **type** prints all of the places that contain an executable named *name*. This includes aliases and functions, if and only if the **-p** option is not also used. The table of hashed commands is not consulted when using **-a**. The **-f** option suppresses shell function lookup, as with the **command** builtin. **type** returns true if any of the arguments are found, false if none are found.
- **ulimit** [-**SHacdflmnpstuv**] [*limit*] Provides control over the resources available to the shell and to processes started by it, on systems that allow such control. The **-H** and **-S** options specify that the hard or soft limit is set for the given resource. A hard limit cannot be increased once it is set; a soft limit may be increased up to the value of the hard limit. If neither **-H** nor **-S** is specified, both the soft and hard limits are set. The value of *limit* can be a number in the unit specified for the resource or one of the special values **hard**, **soft**, or **unlimited**, which stand for the current hard limit, the current soft limit, and no limit, respectively. If *limit* is omitted, the current value of the soft limit of the resource is printed, unless the **-H** option is given. When more than one resource is specified, the limit name and unit are printed before the value. Other options are interpreted as follows:
 - **-a** All current limits are reported
 - **-c** The maximum size of core files created
 - **-d** The maximum size of a process's data segment
 - **-f** The maximum size of files created by the shell
 - **-l** The maximum size that may be locked into memory
 - **-m** The maximum resident set size
 - **-n** The maximum number of open file descriptors (most systems do not allow this value to be set)

- **-p** The pipe size in 512-byte blocks (this may not be set)
- **-s** The maximum stack size
- **-t** The maximum amount of cpu time in seconds
- **-u** The maximum number of processes available to a single user
- **-v** The maximum amount of virtual memory available to the shell

If *limit* is given, it is the new value of the specified resource (the **-a** option is display only). If no option is given, then **-f** is assumed. Values are in 1024-byte increments, except for **-t**, which is in seconds, **-p**, which is in units of 512-byte blocks, and **-n** and **-u**, which are unscaled values. The return status is 0 unless an invalid option or argument is supplied, or an error occurs while setting a new limit.

- **umask** [-p] [-S] [*mode*] The user file-creation mask is set to *mode*. If *mode* begins with a digit, it is interpreted as an octal number; otherwise it is interpreted as a symbolic mode mask similar to that accepted by *chmod(1)*. If *mode* is omitted, the current value of the mask is printed. The **-S** option causes the mask to be printed in symbolic form; the default output is an octal number. If the **-p** option is supplied, and *mode* is omitted, the output is in a form that may be reused as input. The return status is 0 if the mode was successfully changed or if no *mode* argument was supplied, and false otherwise.
- **unalias** [-a] [*name* ...] Remove each *name* from the list of defined aliases. If **-a** is supplied, all alias definitions are removed. The return value is true unless a supplied *name* is not a defined alias.
- **unset** [-fv] [*name* ...] For each *name*, remove the corresponding variable or function. If no options are supplied, or the **-v** option is given, each *name* refers to a shell variable. Read-only variables may not be unset. If **-f** is specified, each *name* refers to a shell function, and the function definition is removed. Each unset variable or function is removed from the environment passed to subsequent commands. If any of **RANDOM**, **SECONDS**, **LINENO**, **HISTCMD**, **FUNCNAME**, **GROUPS**, or **DIRSTACK** are unset, they lose their special properties, even if they are subsequently reset. The exit status is true unless a *name* is readonly.
- **wait** [*n*] Wait for the specified process and return its termination status. *n* may be a process ID or a job specification; if a job spec is given, all processes in that job's pipeline are waited for. If *n* is not given, all currently active child processes are waited for, and the return status is zero. If *n* specifies a non-existent process or job, the return status is 127. Otherwise, the return status is the exit status of the last process or job waited for.

7.14.32 Restricted Shell

If **bash** is started with the name **rbash**, or the **-r** option is supplied at invocation, the shell becomes restricted. A restricted shell is used to set up an environment more controlled than the standard shell. It behaves identically to **bash** with the exception that the following are disallowed or not performed:

- • changing directories with **cd**
- • setting or unsetting the values of **SHELL**, **PATH**, **ENV**, or **BASH_ENV**
- • specifying command names containing /

- • specifying a file name containing a / as an argument to the `.` builtin command
- • Specifying a filename containing a slash as an argument to the `-p` option to the `hash` builtin command
- • importing function definitions from the shell environment at startup
- • parsing the value of `SHELLOPTS` from the shell environment at startup
- • redirecting output using the `>`, `>|`, `<>`, `>&`, `&>`, and `>>` redirection operators
- • using the `exec` builtin command to replace the shell with another command
- • adding or deleting builtin commands with the `-f` and `-d` options to the `enable` builtin command
- • Using the `enable` builtin command to enable disabled shell builtins
- • specifying the `-p` option to the `command` builtin command
- • turning off restricted mode with `set +r` or `set +o restricted`.

These restrictions are enforced after any startup files are read.

When a command that is found to be a shell script is executed (see **COMMAND EXECUTION** above), `} rbash` turns off any restrictions in the shell spawned to execute the script.

7.14.33 See Also

- *Bash Reference Manual*, Brian Fox and Chet Ramey
- *The Gnu Readline Library*, Brian Fox and Chet Ramey
- *The Gnu History Library*, Brian Fox and Chet Ramey
- *Portable Operating System Interface (POSIX) Part 2: Shell and Utilities*, IEEE
- *sh(1)*, *ksh(1)*, *cs(1)*
- *emacs(1)*, *vi(1)*
- *readline(3)*

7.14.34 Files

- */bin/bash* The `bash` executable
- */etc/profile* The systemwide initialization file, executed for login shells
- */etc/bash.bashrc* The systemwide per-interactive-shell startup file
- */etc/bash_logout* The systemwide login shell cleanup file, executed when a login shell exits
- */.bash_profile* The personal initialization file, executed for login shells
- */.bashrc* The individual per-interactive-shell startup file
- */.bash_logout* The individual login shell cleanup file, executed when a login shell exits
- */.inputrc* Individual *readline* initialization file

7.14.35 Authors

Brian Fox, Free Software Foundation

bfox@gnu.org

Chet Ramey, Case Western Reserve University

chet@po.CWRU.Edu

7.14.36 Bug Reports

If you find a bug in **bash**, you should report it. But first, you should make sure that it really is a bug, and that it appears in the latest version of **bash**. The latest version is always available from *ftp://ftp.gnu.org/pub/bash/*.

Once you have determined that a bug actually exists, use the *bashbug* command to submit a bug report. If you have a fix, you are encouraged to mail that as well! Suggestions and ‘philosophical’ bug reports may be mailed to *bug-bash@gnu.org* or posted to the Usenet newsgroup **gnu.bash.bug**.

ALL bug reports should include:

- The version number of **bash**
- The hardware and operating system
- The compiler used to compile
- A description of the bug behaviour
- A short script or ‘recipe’ which exercises the bug

bashbug inserts the first three items automatically into the template it provides for filing a bug report.

Comments and bug reports concerning this manual page should be directed to *chet@po.CWRU.Edu*.

7.14.37 Bugs

It’s too big and too slow.

There are some subtle differences between **bash** and traditional versions of **sh**, mostly because of the **POSIX** specification.

Aliases are confusing in some uses.

Shell builtin commands and functions are not stoppable/restartable.

Compound commands and command sequences of the form ‘a ; b ; c’ are not handled gracefully when process suspension is attempted. When a process is stopped, the shell immediately executes the next command in the sequence. It suffices to place the sequence of commands between parentheses to force it into a subshell, which may be stopped as a unit.

Commands inside of **\$(...)** command substitution are not parsed until substitution is attempted. This will delay error reporting until some time after the command is entered. For example, unmatched parentheses, even inside shell comments, will result in error messages while the construct is being read.

Array variables may not (yet) be exported.