UNIVERSITÀ DI GENOVA
FACOLTÀ DI SCIENZE M.F.N.
Dispense del corso di Laboratorio di Calcolo A
a.a. 2001-2002

Patrizia Boccacci, Paolo Morettini
Claudia Gemme, Maurizio Lo Vetere, Fabrizio Parodi

2nd October 2001

Copyright Notice

HIGZ – High level Interface to Graphics and Zebra

HPLOT – User's Guide

HBOOK – Statistical Analysis and Histogramming

MINUIT – Function Minimization and Error Analysis

CERN Program Library entry **Y250 Y251 D506**

Copyright CERN, Geneva 1991

Copyright and any other appropriate legal protection of these computer programs and associated documentation reserved in all countries of the world.

These programs or documentation may not be reproduced by any method without prior written consent of the Director-General of CERN or his delegate.

Permission for the usage of any programs described herein is granted apriori to those scientific institutes associated with the CERN experimental program or with whom CERN has concluded a scientific collaboration agreement.

Requests for information should be addressed to:

CERN Program Library Office
CERN-CN Division
CH-1211 Geneva 23
Switzerland
Tel. +41 22 767 4951
Fax. +41 22 767 8630
Email: cernlib@cern.ch

Trademark notice: All trademarks appearing in this guide are acknowledged as such.

Table of Contents

1 Introduzione **6**

1.1 Che cos' è un programma 6
1.2 Che cos' è un algoritmo 6

2 Configurazione base di un calcolatore **9**

2.1 Come è fatto un PC 9
2.2 Scheda Madre 11
2.2.1 Il clock di sistema 13
2.2.2 Unitá a dischetti floppy 13
2.2.3 Spie luminose e dispositivi assortiti 14
2.2.4 Batteria tampone 14
2.2.5 Porte seriali 14
2.2.6 Porta parallela 14
2.2.7 Universal Serial Bus (USB) 14
2.2.8 Connettori al bus IDE 15
2.3 CPU 15
2.4 La memoria centrale 16
2.5 Hard disk 17

3 Il sistema operativo **19**

3.1 Livelli di funzionalitá di un calcolatore 19
3.2 Linux 19
3.2.1 Un po' di storia 19
3.2.2 La struttura di Linux: il kernel e la shell 21
3.3 Struttura dei files e delle directories 21
3.4 Controllo dei processi 23
3.5 Nomi dei files e delle directories 23
3.6 Comandi per la gestione di files e directories 23
3.6.1 Utilizzo di wildcard 25
3.6.2 Pipe e redirezione 25
3.6.3 Protezioni 25
3.7 Il text editor 26
3.8 Compilazione ed esecuzione di un programma 26
3.9 Il debugger simbolico 27
3.10 Procedure di comandi 28
3.10.1 Uso delle variabili 29
3.10.2 Realizzazione ed esecuzione di una procedura di comandi 29
3.11 Archiviazione di files 30
3.11.1 Compressione di files 30
3.11.2 Accesso a dischi rimuovibili 31
3.12 Come ottenere ulteriori informazioni 31
3.13 Il sistema grafico X11 31

4 I linguaggi evoluti **33**
4.1 Introduzione . . . 33
4.2 Compilazione e Link . . . 33
4.3 Elementi di un linguaggio evoluto . . . 33
4.4 Le variabili . . . 34
4.4.1 Tipo e dimensione di una variabile . . . 34
4.4.2 Posizione in memoria degli elementi di una matrice . . . 34
4.4.3 Tipi composti . . . 35
4.5 Allocazione statica ed automatica della memoria . . . 35
4.6 I puntatori . . . 35
4.6.1 Allocazione dinamica della memoria . . . 36
4.7 Programmazione strutturata . . . 37
4.8 Subroutines . . . 37
4.8.1 Passaggio dei parametri alle subroutines . . . 38
4.8.2 Functions . . . 38
4.8.3 Ricursività . . . 38
4.8.4 Moduli e variabili globali . . . 38
4.8.5 La programmazione object–oriented . . . 39
4.8.6 Interfacce con librerie di routines . . . 39

5 Sintassi del linguaggio C **40**
5.1 Introduzione . . . 40
5.2 Il comando per la compilazione . . . 41
5.3 Formato del file sorgente . . . 42
5.4 Precompilazione . . . 42
5.5 Struttura del programma . . . 44
5.6 Functions . . . 45
5.7 Interfacce esplicite . . . 47
5.8 Operatori . . . 47
5.9 Costanti . . . 48
5.10 Dichiarazione delle variabili . . . 49
5.11 Manipolazione di stringhe . . . 50
5.12 Tipi composti . . . 51
5.13 Puntatori ed allocazione dinamica della memoria . . . 52
5.14 Istruzioni condizionate e cicli . . . 55
5.15 Input/Output su terminale . . . 55
5.16 Input/Output su file . . . 57

6 Tecniche algoritmiche e numeriche di base **59**
6.1 Introduzione 59
6.2 I vettori 59
6.2.1 Manipolazione dei vettori 59
6.2.2 Ordinamento degli elementi di un vettore 61
6.2.3 Aggiunta di elementi a un vettore 62
6.3 Tecniche Numeriche 63
6.3.1 Calcolo di integrali definiti 63
6.3.2 Ricerca del minimo di una funzione 64
6.3.3 Il metodo dei minimi quadrati 64

7 Uso delle librerie CERN **66**
7.1 Introduzione alle routines di Best-Fit 66
7.1.1 Basic concepts of MINUIT 68

Capitolo 1: Introduzione

1.1 Che cos' è un programma

Il termine programma puó essere utilizzato per riferirsi a sequenze di azioni che si prefiggono il raggiungimento di certi obiettivi precisi: in un corso di studi il termine programma indica una sequenza temporale di argomenti da svolgere e di obiettivi didattici da raggiungere o in politica con termine programma di un partito si indicano una serie di azioni e di impegni da mantenere nell'immediato futuro.

Nonostante i numerosissimi casi in cui si utilizza il termine programma, uno degli usi piú frequenti è diventato quello riferito al mondo dei calcolatori.

DEF. Per **programma** si intende *una sequenza di istruzioni scritte in un linguaggio comprensibile al calcolatore* che le esegue per ottenere i risultati richiesti.

L'esecutore del programma è quindi il calcolatore. Nel 1642 il francese Blaise Pascal inventa una macchina da calcolo (chiamata pascalina) per aiutare il padre nel suo lavoro di funzionario delle imposte; il principio di funzionamento si basava sul movimento di ruote dentate.

All'inizio del Novecento le macchine calcolatrici divennero da meccaniche ad elettriche. Nel 1952 l'ungherese John von Neumann realizza il primo vero prototipo dei moderni elaboratori elettronici che poteva essere programmato per risolvere problemi diversi.

L'elaboratore entra nel mondo commerciale e da allora con una crescita esponenziale aumentano le prestazioni e si riducono le dimensioni e i costi di produzione.

Nel 1965 Gordon Moore (uno dei fondatori della Intel) formuló una legge che sosteneva che ogni 18 mesi sarebbe raddoppiato il numero dei transistor contenuti nei circuiti integrati. Questa legge, che è rimasta valida anche nel nuovo millennio con una piccola correzizone: da 18 mesi si e' passatiad un anno. (informazioni ulteriori http://www.intel.com/research/silicon/mooreslaw.htm)

Potete notare che il grafico inizia nel 1970, due anni dopo lo sbarco (presunto) dell'uomo sulla Luna.

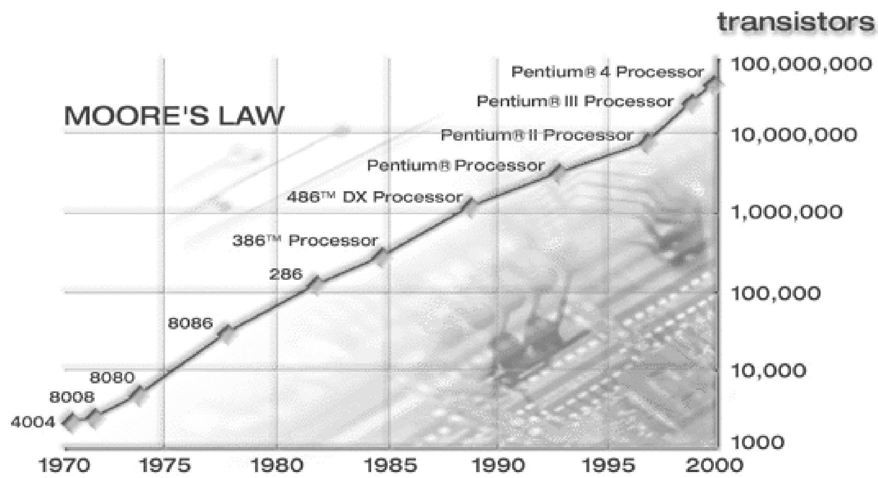
1.2 Che cos' è un algoritmo

Abbiamo appena accennato al concetto di programma nel campo degli elaboratori elettronici. Abbiamo detto che l'esecutore del programma è l'elaboratore elettronico e che il programma è una sequenza di istruzioni scritte in un linguaggio comprensibile al calcolatore. La *sequenza di istruzioni*, prima di essere tradotta in programma, rappresenta la *strategia* per raggiungere il risultato richiesto.

DEF. Si definisce **algoritmo** (dal nome del matematico arabo Al-Khuwarizmi) una serie finita di operazioni univocamente interpretabili, eseguite in sequenza, che trasforma i dati iniziali nel risultato richiesto.

Se il programma è la traduzione dell'algoritmo per poter essere eseguito senza errori dall'elaboratore questo deve essere univocamente interpretabile e deve essere rappresentato in una forma che ne consenta una facile conversione in programma.

L'insieme dei valori permessi dai dati in ingresso si definisce **dominio** dell'algoritmo (analogamente alle funzioni matematiche) mentre l'insieme dei valori che possono assumere i dati in uscita rappresenta il **codominio**.



In generale, ai fini di una migliore utilizzazione delle risorse disponibili (in termini di spazio e tempo) e per poter trattare problemi, anche di grandi dimensioni, con costi contenuti, è comunque fondamentale una scelta attenta nella formulazione dell'algoritmo risolutivo di un dato problema.

Una volta formulato l'algoritmo occorre verificare che questo fornisca:

- una soluzione per qualunque valore dei dati iniziali purchè appartenenti al dominio
- e che questa venga raggiunta in un numero finito di passi ovvero in un tempo finito.

E' chiaro che per descrivere un algoritmo funzionante devono essere considerati tutti i casi particolari anche con l'analisi dei risultati intermedi.

Una forma molto usata per la descrizione degli algoritmi è quella del *flow-chart*.

Esempio di algoritmo

Supponiamo di voler risolvere una equazione di secondo grado espressa in forma normale:

$$ax^2 + bx + c = 0 \tag{1.1}$$

Descriviamo questo semplice algoritmo prima a parole poi con un flow-chart.

1. Acquisisco a, b, c (supponiamo per semplicitá che a sia diversa da 0)
2. calcolo il $Delta = b^2 - 4ac$
3. Verifico se $Delta < 0$?
4. Se la risposta alla domanda precedente è *vero*, stampo che non ci sono soluzioni reali e vado al passo 10; se la risposta è *falso* passo al punto successivo.
5. Verifico se $Delta = 0$?
6. se la risposta alla domanda precedente è *vero*, pongo $x_1 = x_2 = -b/2a$, vado al passo 9; se la risposta è *falso* passo al punto successivo.
7. $x_1 = \frac{-b-\sqrt{\Delta}}{2a}$

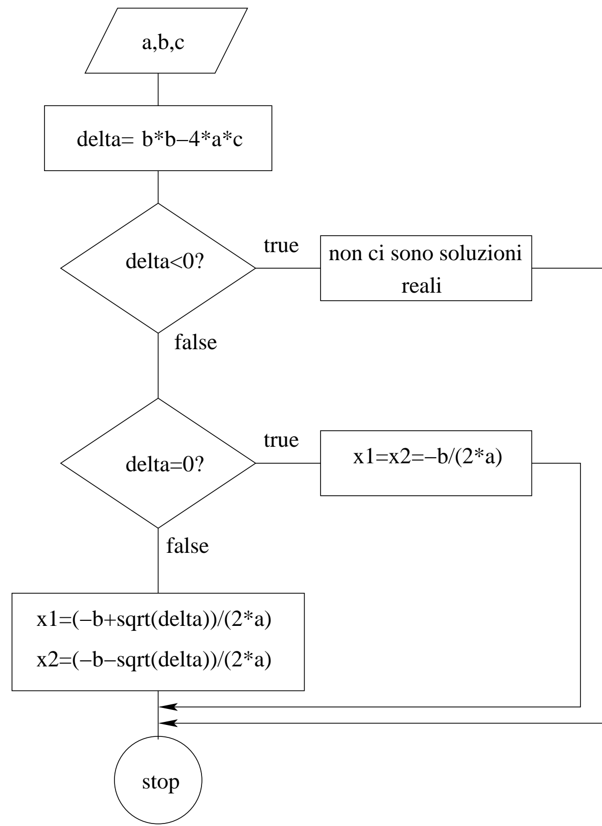


Figura 1.1: Grafico di flusso che descrive l'algoritmo per la risoluzione delle equazioni di secondo grado nel campo reale.

8. $x_2 = \frac{-b+\sqrt{\Delta}}{2a}$
9. stampo il risultato
10. termine della procedura

Capitolo 2: Configurazione base di un calcolatore

2.1 Come è fatto un PC

Abbiamo visto che l'esecutore dell'algoritmo è l'elaboratore elettronico (PC Personal Computer), ne discuteremo qui le componenti base.

Per descrivere i componenti interni di un calcolatore si usano spesso sigle che, a volte, non sono molto note.

Inseriamo un breve glossario delle sigle e del loro significato:

ALU	Arithmetic Logic Unit - Unitá logico-aritmetica. Parte integrante della CPU
ANSI	American National Standards Institute - Ente governativo statunitense che si occupa di definire lo standard dei linguaggi di programmazione
ASCII	American Standard Code for Information Interchange - Codice standard americano a 7 bit per l'interscambio di informazioni fra sistemi di elaborazione e di comunicazione definito dall'ANSI
BIOS	Basic Input-Output System Programma diagnostico e di inizializzazione del sistema attivato automaticamente all'accensione del computer
BIT	BInary digiT - Cifra binaria. Unitá elementare di informazione del sistema binario assume i valori 0 e 1
BUS	Circuito seriale o parallelo, interno al computer che consente lo scambio di informazioni tra i diversi dispositivi elettronici che compongono il sistema
BUFFER	Tampone - Area riservata della memoria nella quale sono temporaneamente immessi i dati che si trasferiscono da una unitá ad un'altra.
BYTE	Byte - Unitá di informazione, costituita da una sequenza di 8 bit
CAD	Computer Aided Design - Progettazione assistita dal computer
CLOCK	Orologio - Dispositivo elettronico, basato su di un cristallo di quarzo che genera impulsi ad intervalli regolari allo scopo di sincronizzare le operazioni del processore di un computer
CMOS	Complementary Metal-Oxide Semiconductor - Dispositivo semiconduttore integrato impiegato nelle memorie RAM
CPU	Central Process Unit (Unitá di elaborazione centrale)
CRT	Cathode Ray Tube - Tubo a raggi catodici. Dispositivo di output di un computer che produce immagini mediante l'illuminazione dei fosfori

Db	Debugger - Software progettato per la ricerca e la correzione degli errori di un programma
DRAM	Dynamic RAM - Ram Dinamica Tempo di accesso 70 80 nanosec
EIDE (IDE)	Interfaccia di comunicazione tra PC e dispositivi
FPU	Floating Point Unit - Processore interno alla CPU dedicato al calcolo in virgola mobile
OS	Operating System - Sistema Operativo
Gb	GigaByte - Unitá di misura della memoria equivale a $1.073.741.824 = 2^{30}$ bytes
HD	Hard Disk - Disco rigido
I/O	Input/Output - Immissione/emissione di dati.
Kb	Kilobyte - unitá di misura della memoria equivale $1024 = 2^{10}$ bytes
LAN	Local Area Network - Rete locale di comunicazione
Mb	Megabyte - Unitá di misura della memoria equivale a $1.048.576 = 2^{20}$ bytes
RAM	Random Access Memory - Memoria primaria del computer
ROM	Read Only Memory - Memoria a sola lettura
R/W	Read-Write - Lettura-scrittura
SCSI	Small Computer System Interface - Interfaccia parallela standard ad elevata velocitá con protocollo definito dall'ANSI
SDRAM	Versione sincrona della DRAM, piú veloce
SLOT	Alloggiamento - Presa che consente di installare vari dispositivi: espansioni di memoria, interfacce etc.
USB	Universal Serial Bus - permette la connessione di molti dispositivi esterni

Per descrivere un PC, è di aiuto pensarlo in termini di sottosistemi che lo compongono:

- Aspetto esteriore: si vedono lo schermo, la tastiera, il mouse e l'involucro del PC. L'involucro contiene tutti i componenti vitali del PC e fornisce anche l'accesso alle unitá CD-ROM e floppy sul davanti e a varie porte sul retro. L'involucro inoltre protegge l'utente dalle radiazioni elettromagnetiche prodotte dai componenti interni e consente al flusso d'aria canalizzato di ridurre i problemi di surriscaldamento.

- Componenti interni

  CPU: L'unitá di elaborazione centrale (Central Processing Unit), o processore, gestisce tutte le operazioni aritmetiche che permettono il funzionamento del PC. La velocitá della CPU, combinata con il suo design interno, determina la velocitá e l'efficienza con cui puó elaborare i dati e i codici dei programmi.

  - Memoria di sistema: Chiamata memoria ad accesso casuale (Random Acces Memory), o RAM, la memoria di sistema immagazzina i codici e i dati che il PC sta utilizzando. Piú memoria si ha, migliori sono le prestazioni.

  - Disco rigido: Il dispositivo per l'immagazzinamento permanente dei dati sul PC. I programmi e i dati contenuti nel disco rigido vengono trasferiti alla RAM di sistema per essere usati. I dischi rigidi odierni vantano capacitá di 10 GB e oltre.
  - Unitá CD-ROM: E' solitamente presente anche un supporto rimovibile di grande capacitá. Le unitá CD-ROM permettono agli utenti del PC di accedere ai contenuti multimediali, di installare applicazioni di grande dimensioni e di lavorare con enormi data base e con altre risorse. Questa tecnologia sta lasciando il posto alle unitá DVD-ROM, che offrono capienze ancora piú grandi.
  - Scheda grafica: Tuttoció che si vede sullo schermo passa attraverso la scheda grafica. Questo componente hardware accelera la visualizzazione delle immagini, sia in due sia in tre dimensioni, e delle riproduzioni dei video. Una buona scheda grafica fondamentale per avere una buona visione e per prestazioni veloci.
  - Scheda audio: Tuttoció che va oltre gli elementari bip e che è emesso dall'altoparlante del PC passa attraverso il sottosistema audio. Le schede audio permettono di riprodurre i suoni digitali attraverso gli altoparlanti del PC, nonchè di registrare suoni analogici in formato digitale.
  - Modem: Consente al PC di parlare con Internet, con i fax e con altri PC, usando una normale linea telefonica. Puó essere sia una scheda aggiunta internamente sia un modem esterno.
  - Scheda di interfaccia con la rete: Questa scheda supplementare serve a connettere il PC a una rete locale.

- Dispositivi di input/output
  - Schermo: Lo schermo CRT (cathode ray tube, tubo a raggi catodici) simile all'apparecchio TV, che si trova praticamente su tutti i PC da tavolo o desktop. Oggi, si usano soprattutto gli schermi da 15 e da 19 pollici. I monitor piú grandi consentono una visione di gran lunga migliore.
  - Tastiera e mouse: I principali dispositivi di input del PC. Praticamente tutti i PC richiedono, e possiedono, una tastiera e un mouse.

2.2 Scheda Madre

Se si guarda all'interno di un PC, si potrá constatare che quasi ogni cosa è montata sopra un solo, grande circuito stampato, che viene detto scheda madre. Questo circuito è l'ossatura di base del PC. Ogni elemento, dalle schede video ai dischi fissi alla CPU e alle memorie RAM è collegato o inserito sulla scheda madre e comunica con gli altri componenti tramite le connessioni sulla scheda madre. In pratica, la scheda madre determina completamente ció che il PC é in grado di fare. La funzione della scheda madre non è soltanto quella di fungere da contenitore passivo dei vari dispositivi che compongono il PC, bensí ha il compito di determinare la velocitá e l'efficienza con cui possono comunicare e lavorare gli elementi in essa contenuti (CPU, RAM, Chipset, slot vari, ecc.). E' da tenere in considerazione che la tecnologia applicata alle schede madri si sta velocemente evolvendo con continue innovazioni aumentando sempre piú la frequenza di clock, adottando vari tipi di memoria, aggiungendo nuove funzioni e per ultimo integrando sempre piú all'interno delle schede madri dispositivi che tempo addietro erano al suo esterno (schede audio e video, controller ecc.).

Le schede madri sono prodotti complessi, ma facili da comprendere, se le si suddivide nei loro singoli componenti.

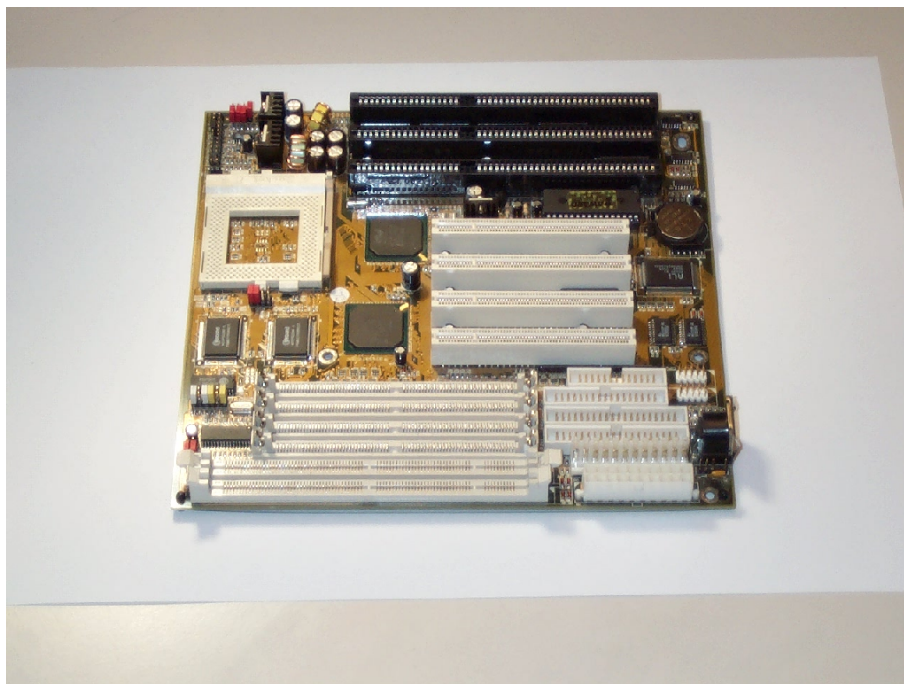


Figura 2.1: La scheda madre

- Zoccoli o slot per la CPU
- Slot ad innesto per le schede di espansione con bus ISA, PCI, AGP.
- Connettori per le memorie e (a volte) la cache secondaria con relativi chip.
- Chipset di sistema e CMOS del BIOS
- Cristallo di quarzo dell'oscillatore e una batteria tampone
- Connettori di I/O sul retro per le porte seriale, parallela, mouse, tastiera e USB
- Connettori per le unitá a disco, l'alimentazione e (a volte) SCSI

Un altro punto importante da considerare è il supporto alle schede di espansione. Si possono osservare degli slot (connettori a pettine) allineati perpendicolarmente al lato posteriore. In genere sono di due tipi, corrispondenti al bus utilizzato dal sistema. Questi slot sono importanti perchè determinano il tipo di schede che si possono aggiungere facilmente al sistema. Schede video, schede audio, schede di rete, e altre periferiche interne vengono tutte innestate sui bus delle schede di espansione. Punti chiave sono il numero e il tipo di slot presenti.
Ci sono tre tipi principali di bus nei PC:

- ISA:il sistema base di bus a bassa velocitá presente praticamente in ogni PC.
- PCI dominante in tutti i nuovi sistemi venduti negli ultimi anni
- AGP Accelerated Graphics Port, il nuovo bus solo per la grafica di recente costituzione, che lavora a una velocit da doppia a quadrupla del PCI, e assicura anche buone funzionalit

3D. Il bus è abbastanza veloce da permettere alle schede video di utilizzare la memoria di sistema come buffer per accelerare la velocità di creazione delle schermate nei giochi e con i video-clip.

I connettori ISA di solito sono di colore nero o marrone, mentre gli slot PCI si riconoscono dal loro colore chiaro, dall'essere piú corti e dai piedini molto piú ravvicinati. Lo slot AGP di solito è di colore marrone e un po' piú corto del PCI.

Il bus PCI si caratterizza soprattutto per la sua velocitá. Con una velocitá di trasferimento di 132 Mb/s il PCI puó trasportare i bit a una velocitá circa 25 volte quella dell'ISA. L'Accelerated Graphics Port è dedicato esclusivamente alla grafica, che permette alle schede video 3D di usare la grande disponibilitá di memoria di sistema per creare le scene tridimensionali sullo schermo.

Sulla scheda madre sono collocati vari componenti assai importanti, ma i piú critici sono senz'altro i chipset di sistema. Viene detto chipset un gruppo di circuiti integrati che sono direttamente saldati sulla scheda madre e si incaricano di tutte le operazioni di normale gestione, quali il risparmio energetico, attivitá sul bus, transazioni con memorie e temporizzazione della CPU. Il chipset fa parte integrante della scheda madre, e non puó essere aggiornato. Quindi quando si acquista una scheda madre va tenuto bene in considerazione il relativo chipset.

La sigla BIOS sta per Basic Input/Output Sistem, che costituisce il livello piú basso di software funzionante su un PC. Il BIOS ha la responsabilitá di tutto il lavoro che si svolge dietro le quinte, la configurazione dei dischi fissi e delle porte parallele, la gestione dei trasferimenti sul bus e del supporto a vari schemi di memoria e altre tecniche.

Come si fa ad entrare nel BIOS del sistema ? Quando la macchina si avvia subito dopo il test sulla memoria viena data un'indicazione sulla sequenza che permette di accedervi (si deve premere il tasto Canc, Del, ecc.). Una volta entrati nel programma di configurazione (che in genere è in modalit caratteri), verranno indicate le opzioni per gestire impostazioni e funzioni. Ci si puó spostare all'interno in genere usando i tasti cursore ed Esc e Invio.

ATTENZIONE: non salvate le modifiche apportate al BIOS se non siete sicuri di quello che avete fatto, altrimenti il PC potrebbe non ripartire.

2.2.1 Il clock di sistema

La frequenza di lavoro dei PC deriva da quella a cui oscilla l'orologio di sistema (clock) che stabilisce la cadenza per le attivitá della CPU e sul bus di sistema. Il clock in pratica è un cristallo di quarzo che oscilla in risonanza con una carica elettrica. Regolando l'input elettrico si può controllare la risonanza. Il clock della scheda madre di solito è individuabile sotto forma di un blocchetto rettangolare di colore nero montato presso il bordo.

Attualmente molte schede madri lavorano a una frequenza di 66 o 100 Mhz, mentre la CPU lavora internamente a una frequenza multipla di questa.

Di fatto non è possibile far funzionare le schede madri a velocitá prossime delle CPU attualmente in commercio per problemi fisici. Le piste interne della CPU, lunghe meno di un micron, possono operare a una velocità molto maggiore dei lunghi conduttori e dei diversi componenti e connettori alla scheda madre.

2.2.2 Unitá a dischetti floppy

Vicino al connettore IDE si trova sempre un connettore singolo per l'unitá a dischetti. Una piattina standard viene innestata da un lato su questo connettore e dall'altro sul retro dell'unitá a dischetti: di solito la piattina prevede l'attacco per due unitá.

2.2.3 Spie luminose e dispositivi assortiti

Lungo la parte anteriore della scheda madre si possono vedere di solito una serie di piccoli connettori. Questi servono per collegare l'altoparlantino interno del PC, le spie LED indicatrici del funzionamento del disco fisso e dell'alimentazione, e i pulsanti di accensione e di reset del computer. In questi connettori si innestano le prese che recano i conduttori che vanno direttamente ai vari dispositivi.

2.2.4 Batteria tampone

Questo componente piuttosto trascurato svolge invece un ruolo importante per il funzionamento del PC. La batteria fornisce l'alimentazione continuativa necessaria a conservare le impostazioni memorizzate nei CMOS del BIOS e nell'orologio in tempo reale del sistema. Ci sono vari tipi di batterie, fra cui accumulatori al nichel e idruro metallico, a ioni di litio e alcalini. I PC recenti utilizzano le batterie al litio, che in genere durano da 5 a 10 anni prima di richiedere una sostituzione.

2.2.5 Porte seriali

Molti PC dispongono di due connettori per porte seriali: un connettore con 9 piedini e uno grande con 25 piedini. Queste porte sono spesso usate da periferiche esterne quali modem, scanner, unitá di backup a nastro di fascia bassa. Le porte seriali moderne trasferiscono dati alla velocitá di 1,5 KB/s.

2.2.6 Porta parallela

Su molti PC la porta parallela serve per collegare una stampante. Capace di trasferire fino a 4 MB/s di dati, la moderna porta parallela è sufficientemente veloce per la maggior parte dei trasferimenti dati di media grandezza, fra cui i lavori di stampa e le telecomunicazione a connessione diretta via cavo. Scanner, unitá a nastro lettori di CD-ROM portatili e dispositivi di memoria di massa con dischi rimovibili quali le unitá Zip e jaz possono pure utilizzare la porta parallela. Non tutte le porte parallele sono create uguali. I sistemi recenti dispongono della cosiddetta ECP (Enhanced Capabilility Port), capace di operare in duplex (nelle due direzioni) con velocità di trasferimento maggiori. I PC dotati di ECP possono ridurre il tempo di stampa, ma non tutte le periferiche riconoscono i segnali di ottimizzazione cosí trasmessi.

2.2.7 Universal Serial Bus (USB)

Le porte USB hanno fatto la loro comparsa sul retro dei PC da alcuni anni, ma solo da poco cominciano a comparire sul mercato periferiche con esse compatibili. Nei prossimi anni si prevede che l'USB assumerá molti dei compiti oggi svolti dai bus delle porte seriali e parallela. Molti PC desktop attuali includono due porte USB montate sulla scheda madre. Questi connettori a innesto rapido servono come punto fisico di partenza per una catena di periferiche esterne. Si possono concatenare fino a 127 periferiche USB. Cosa puó essere collegato alla porta USB ?: Mouse, tastiere, modem, scanner, joystick, fotocamere digitali, e perfino altoparlanti audio possono lavorare collegati a questo bus Plug & Play. L'USB assicura una velocitá di trasferimento dati fino 12 MB/s, mediamente circa tre volte maggiore della porta parallela, sufficiente per la massima parte delle periferiche.

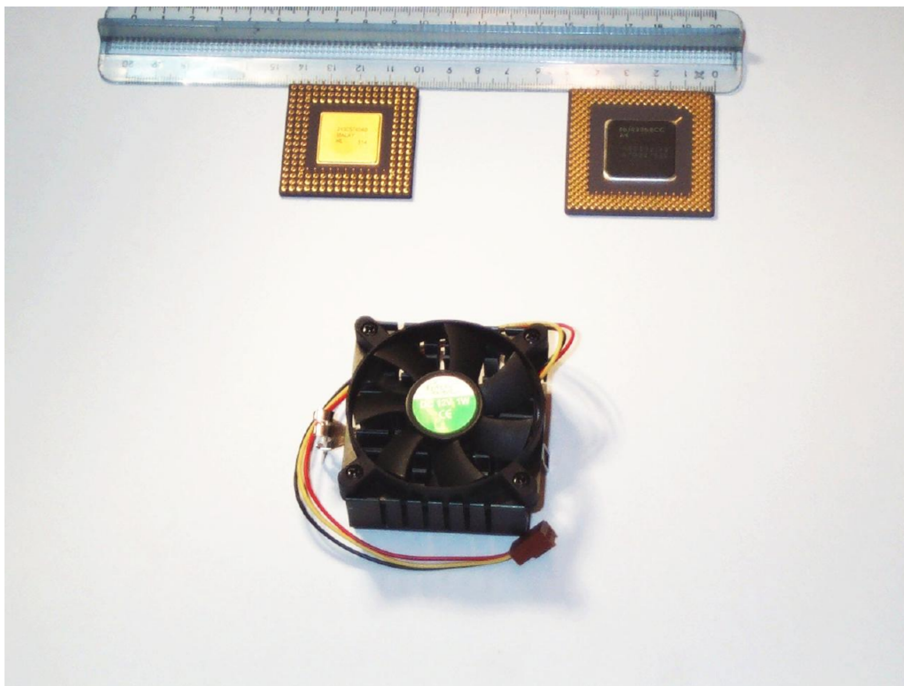


Figura 2.2: La CPU con lo zoccolo e il sistema di raffreddamento

2.2.8 Connettori al bus IDE

Anche i connettori IDE sono piuttosto evidenti sulla scheda madre. Su molte schede ci sono due connettori: uno per il bus IDE primario, l'altro per il bus IDE secondario. I connettori IDE permettono il collegamento con i dischi e le unità CD-ROM, DVD e/o di masterizzazione. Ogni connettore IDE puó controllare due dispositivi (device) uno master e uno slave. Il disco di avviamento deve essere collegato sul controller primario e impostato come dispositivo master. Per questo motivo molti utenti collegano il loro disco fisso al connettore IDE primario, mentre collegano lettore CD-ROM e altre periferiche IDE interne al connettore secondario.

2.3 CPU

CPU (Central Process Unit) è il microprocessore o unitá centrale che elabora le istruzioni prelevandole dalla memoria.

Uno degli elementi peculiari e conosciuti del funzionamento della CPU è la frequenza di clock (espressa in MHz o GHz). Una tipica CPU consiste di milioni di transistor concentrati su una fettina quadrata o rettangolare di silicio di meno di cinque centimetri per lato.

I principali componenti componenti della CPU sono:

- il bus dati
- il bus indirizzi
- la cache primaria o L1
- i registri

- i percorsi d'incanalamento (pipeline) per le istruzioni
- l'unitá a virgola mobile FPU (Floating Point Unit)

Il bus dati: è l'insieme di conduttori e di circuiti dedicati al trasferimento di informazioni dentro e fuori la CPU.

Il bus di indirizzi: come dice lo stesso nome, il gruppo di conduttori su cui viaggiano i bit che descrivono la posizione delle informazioni nella memoria di sistema.

La cache è una memoria aggiuntiva. Ci sono due tipi di cache: la cache interna di Livello 1 (L1), integrata nel processore, e la cache di secondo livello (L2) di dimensioni maggiori, solitamente esterna. La cache L1 occupa parecchio spazio prezioso sul chip e richiede complicati algoritmi per individuare ci di cui avrá bisogno la CPU in seguito: ma il guadagno di prestazioni è innegabile.

Il concetto che sta alla base della cache piuttosto semplice. Un processore puó accedere ai bit internamente molto pi rapidamente di quanto possa prelevandoli dalla memoria esterna. Non sorprende, quindi, che quanto maggiore è la cache tanto migliori i guadagni di prestazioni. La contropartita è che se il codice o i dati necessari non si trovano nella cache, il processore deve perdere un po' di tempo cercando inutilmente nella cache. Ed è qui che deve intervenire l'algoritmo complesso di cui si è parlato: deve prevedere ció che la CPU richiederá in seguito, cosí da rendere disponibili i dati corrispondenti.

L'unitá a virgola mobile: (FPU Floating Point Unit) è un processore dedicato al calcolo con i numeri non interi. Molte applicazioni per PC non ricorrono ai calcoli con numeri reali (in virgola mobile) e cosí spesso la FPU rimane inattiva. Tuttavia, le applicazioni che richiedono tali operazioni quali l'elaborazione delle immagini fotografiche, la progettazione in 3D e i software per il CAD usano intensivamente la FPU.

Il collegamento tra la memoria e la CPU è costituito da due *canali* di scambio chiamati *bus*: il bus dei dati e il bus degli indirizzi. Attraverso il primo si scambiano i dati prima e dopo l'esecuzione delle istruzioni, attraverso il secondo la CPU sa a quale indirizzo di memoria trova i dati e le istruzioni scritte in linguaggio macchina ovvero un linguaggio molto piú semplice di quello nel quale è scritto solitamente il programma. L'istruzione prima di essere eseguita viene copiata in una memoria interna della CPU che viene chiamata *registro*.

2.4 La memoria centrale

In termini fisici la memoria è facile da individuare. Rimosso il telaio del PC, esaminiamo la scheda madre e cerchiamo una o piú schede sottili disposte in fila (si vedranno probabilmente anche alcuni innesti vuoti). Delle dimensioni di poco piú di 10 cm di lunghezza per 2 di altezza.

Diversamente dal disco rigido o dal CD-ROM, la memoria di sistema è volatile: i contenuti della RAM vanno persi quando si toglie corrente al sistema. Questo è dovuto al fatto che la RAM di sistema deve essere costantemente rigenerata con segnali elettrici.

Due sono i tipi di moduli di memoria che prendiamo in considerazione:

- i moduli di memoria single inline (Single Inline Memory Module: SIMM);
- i moduli di memoria dual inline (Dual Inline Memory Module: DIMM).

I moduli di memoria SIMM sono caratterizzati da connessioni a 72 pin e i singoli chip della RAM sono montati su uno dei lati della scheda. I SIMM a 72 pin consentono capacitá che arrivano fino a 32 MB per modulo. Quale che sia la dimensione della memoria, tutti i SIMM devono essere installati nei banchi a coppie.

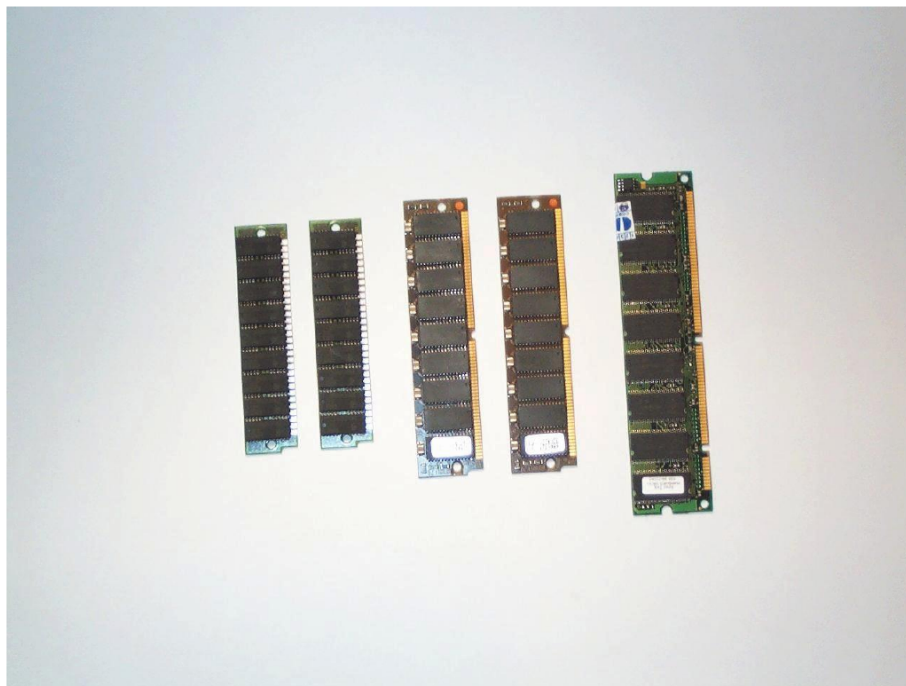


Figura 2.3: Alcuni tipi di RAM

I moduli di memoria DIMM sono relativamente nuovi sulla scena, presentano una connessione a 168 pin piú ampia per fornire prestazioni piú sofisticate e capacitá piú elevate. Diversamente dai SIMM possono essere installati singolarmente. Questo perché i DIMM forniscono un percorso per i dati a 64 bit, equivalente all'ampiezza in bit di un singolo banco di memoria. I DIMM cominciarono a comparire nei PC nel 1996, quando i produttori di PC introdussero una nuova variante di memoria chiamata DRAM sincrona (SDRAM) nei PC piú moderni. I DIMM si possono riconoscere per la loro maggiore lunghezza e per un maggior numero di collegamenti elettrici. I DIMM sono, inoltre piú lunghi dei SIMM e di solito montano RAM su entrambi i lati della scheda.

2.5 Hard disk

Un altro importante componente del PC l'hard disk (disco fisso). All'interno di ogni sistema troveremo sempre almeno un hard disk che ha il compito di mettere a disposizione sia alla memoria di sistema che alla CPU le informazioni, sia in lettura che in scrittura. Ad esempio quando si lancia un'applicazione per l'elaborazione testi, il programma eseguibile viene letto dal disco fisso e caricato nella RAM. Poi, quando si apre un documento salvato in precedenza, anche questo file viene letto dal disco e caricato nella RAM. Tutte le modifiche che vengono apportare a quel documento vengono poi riscritte sul disco fisso quando si salva il file.

Tutti sistemi operativi fanno fare inoltre al disco fisso anche gli straordinari, impegnandolo come serbatoio nel caso di superamento della capacitá di memoria RAM di sistema. Quando viene caricato un numero eccessivo di applicazioni e di file, quelli usati meno di recente vengono nuovamente riscritti sul disco. Ai programmi si fa intendere che i vari bit stiano ancora sulla RAM, ma quando i dati vengono richiamati il sistema operativo ricava in realtá le informazioni dal disco fisso. Questo modo di operare, chiamato memoria virtuale, amplia le funzionalitá di

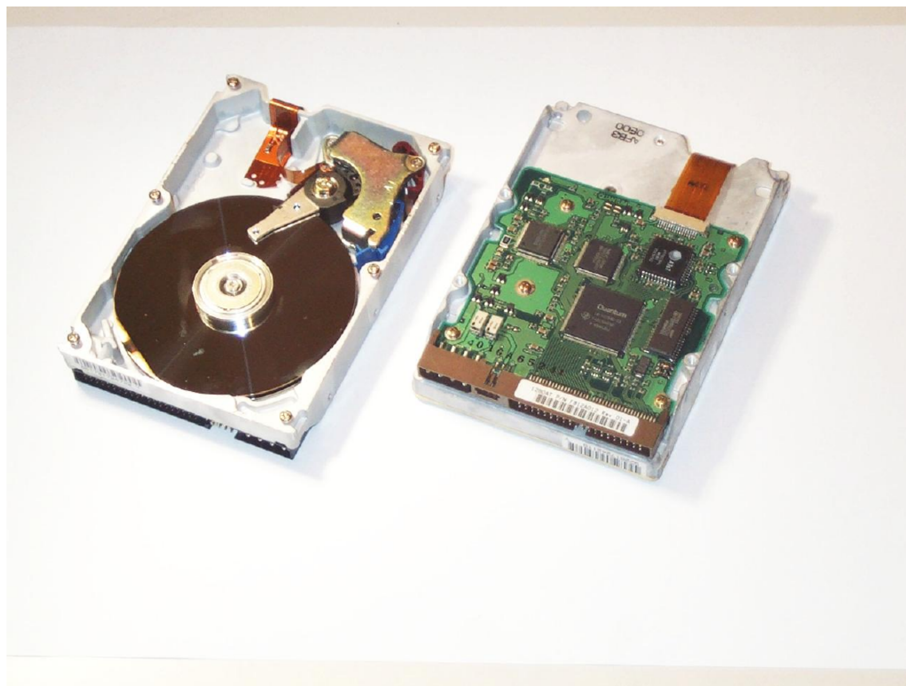


Figura 2.4: L'interno di un disco rigido

tutti i PC, ma esige un caro prezzo in termini di prestazioni: infatti nei trasferimenti di dati i dischi fissi sono 100 volte meno veloci della RAM.

Capitolo 3: Il sistema operativo

3.1 Livelli di funzionalitá di un calcolatore

La figura 3.1 rappresenta la gerarchia dei termini che formano i livelli di funzionalitá del PC. Se con hardware indichiamo tutte le parti fisiche dell'eleboratore notiamo che questo livello contiene una parte detta firmware che individua la parte, normalmente registrata in una memoria ROM, dove sono presenti microistruzioni necessarie per la fase di avvio (bootstrap) e per il caricamento del sistema operativo.

Il livello piú basso é il **linguaggio macchina** (sequenza di numeri binari) che é l'unico linguaggio che riesce la CPU riesce a comprendere.

Non é molto comodo scrivere un programma utilizzando questo linguaggio; il divario tra un linguaggio macchina e un linguaggio piú evoluto é colmato dal sistema operativo (OS Operating System).

Un sistema operativo si preoccupa di:

- gestire l'interfaccia utente cioé di interpretare ed eseguire una serie di comandi comunicati dall'utente tramite tastiera o mouse. Ció permette di manipolare insiemi di dati (files) e di accedere alle periferiche senza avere una conoscenza approfondita dell'hardware del PC.
- permettere l'utilizzo contemporaneo della CPU da parte di piú programmi.
- gestire l'utilizzo della memoria centrale.

Probabilmente molti di voi conoscono i sistemi operativi commerciali piú diffusi (DOS, Windows95, 98, NT, 2000, ME etc.., MAC), durante il corso utilizzeremo un sistema operativo completemente gratuito chiamato Linux, che deriva da un sistema operativo creato per workstation (Unix) e adattatato all'architettura dei PC.

3.2 Linux

3.2.1 Un po' di storia

Linux è una versione per PC del sistema operativo (molto potente) Unix studiato per la gestione delle workstation utilizzate come server (ovvero macchine potenti in grado di gestire un centro calcolo con molti utenti e reti locali).

Il sistema UNIX sviluppato dalla AT&T Bell Laboratories nacque essenzialmente come reazione ai sistemi operativi di grandi dimensioni, complessi e poco flessibili, disponibili alla fine degli anni sessanta. Il principale obiettivo di progettazione fu quello di ottenere un sistema operativo d'elevata potenzialità.

Originariamente (anni '70) il sistema operativo fu sviluppato da un gruppo di esperti di computer come uno strumento di sviluppo per uso personale, ignorando cosí le esigenze dei principianti a vantaggio della velocitá e della precisione. A causa delle leggi antitrust il software non poteva essere venduto e prese la strada delle Universitá Americane che a loro volta iniziarono a sviluppare varie versioni di Unix facendolo maturare velocemente ma creando anche un po' di disordine e confusione. In questo modo iniziarono a formarsi una casta di persone che una volta laureate e pronte per il mondo del lavoro iniziarono a divulgare e far conoscere Unix nelle aziende. Pian piano Unix potè diventare un prodotto commerciale e si iniziarono a vedere sul mercato varie versioni proprietarie e quindi a pagamento. Linux è nato come estensione del Minix (sistema operativo Unix per processori i86 sviluppato ad uso didattico dal professor Andrew

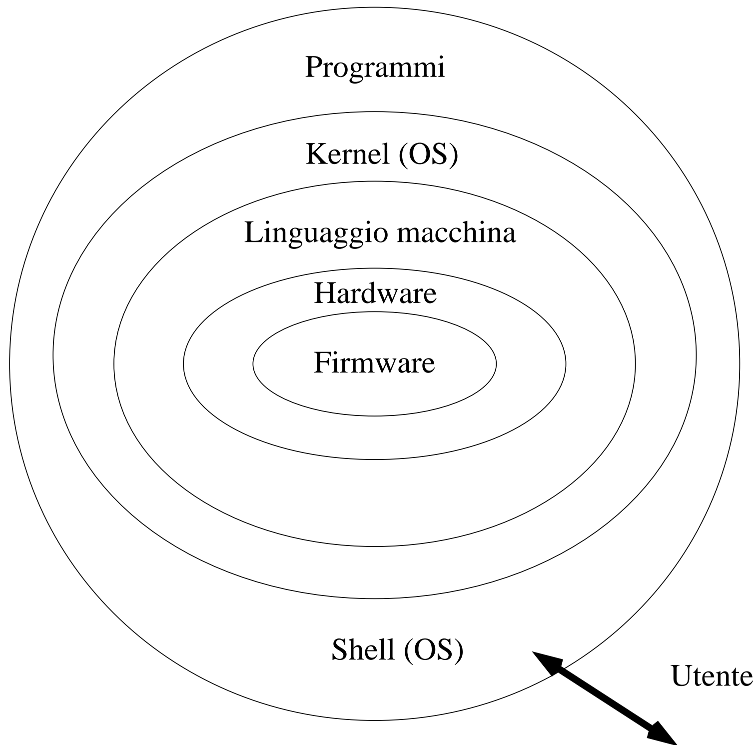


Figura 3.1: Il livelli di funzionalità di un calcolatore

S. Tanenbaum) e come clone di Unix, dallo studente finlandese dell' Universitá di Helsinki Linus Beneditc Torvalds, per un progetto universitario sullo studio delle funzionalitá multitasking dei microprocessori i386.

Linux viene mantenuto da un gruppo di programmatori sparsi per tutto il mondo e la sua flessibilitá e la sua potenza sono accompagnati da un vantaggio che non deve essere trascurato. Linux è completamente gratuito cioé non "si è obbligati" a pagare per poter utilizzare questo sistema operativo. In pratica esiste una licenza GNU (sigla ricorsiva GNU's not Unix) che assicura che Linux rimanga gratuito ma anche standard.

Uno dei vantaggi principali di Linux è che in esso sono forniti tutti gli strumenti per il calcolo scientifico, come per esempio: editori di testo molto complessi, tutti i linguaggi di programmazione, librerie grafiche etc..

Non è da trascurare il fatto che è un sistema multiutente e multitasking, ovvero piú utenti possono utilizzare lo stesso computer (da terminali differenti) e richiedere al sistema di eseguire piú operazioni (task) contemporaneamente.

Recentemente le interfacce grafiche per l'utilizzo del sistema operativo sono si fatte sempre piú elaborate e "simili" a quelle dei sistemi commerciali.

Per la cronaca, dopo sette anni dalla nascita di Linux, Linus Torvalds si é laureato e dopo aver collaborato per diversi anni come ricercatore all'interno dell'Universitá di Helsinki si é trasferito in America dove continua, anche se non come attivitá primaria, a dirigere lo sviluppo del sistema operativo e a lavorare su una versione real-time di Linux.

3.2.2 La struttura di Linux: il kernel e la shell

Il sistema operativo è costituito da un sistema di base e da un certo numero di estensioni. Il sistema di base comprende il kernel, le shell e le utility di base. Le estensioni sono ad esempio le funzionalitá grafiche, le librerie , i compilatori. Il kernel di Linux, che consiste nell'interfaccia tra l'hardware e i processi, gestisce l'accesso alla CPU e la memoria, e si occupa, fra le altre cose, del controllo e della gestione di ciascuna periferica (device). Rappresenta, quindi, il nocciolo dell'intero sistema operativo e idealmente puó assumersi come una sorta di astrazione della macchina fisica essendo posto al livello software piú basso.

Per quanto riguarda le versioni del kernel c'e' da dire che esse utilizzano uno specifico schema per la loro numerazione : le versioni V.x.y con x pari rappresentano le versioni stabili, mentre al contrario le versioni con x dispari sono le beta-release (ancora in prova) utilizzate normalmente dagli sviluppatori e possono essere instabili o generare effetti collaterali non conosciuti: mano a mano che si risolvono i bug (bachi!) il numero y viene incrementato.

Bisogna stare attenti a non confondere la versione del kernel con la distribuzione che contiene anch'essa un numero. Esistono infatti diversi distributori di Linux (cambia praticamente solo l'aspetto grafico e alcune utilities) RedHat, Mandrake, Slackware, Suse, Debian, quindi posso avere per esempio la Mandrake 8.2 che fornisce il kernel 2.4.3-20mdk (V=2,x=4,y=3).

La shell è il collegamento tra l'utente e il kernel. Mentre il kernel è definito, per una data installazione, una volta per tutte, esistono diverse possibili realizzazioni di SHELL per lo stesso computer, che possono essere presenti simultaneamente nello stesso sistema ed eventualmente essere richiamate o abbandonate dall'utente nel corso di una sessione interattiva.
Per ogni utente viene definita una cosidetta SHELL di login, ossia che è attiva all'atto del collegamento; nel nostro caso la SHELL di login è la "**tcsh**".
È importante osservare che le SHELL Linux distinguono le lettere maiuscole da quelle minuscole; in questo senso si dice che Linux è "case sensitive".

I comandi Linux sono terminati dal tasto **Enter** che serve a passare il comando stesso al sistema operativo per la sua esecuzione. Essi possono contenere delle opzioni e/o dei parametri, per i quali occorre rispettare la stessa distinzione tra lettere minuscole e maiuscole.
Per richiamare comandi eseguiti in precedenza si possono utilizzare i tasti **Up** e **Down** (↑ e ↓).
Il comando `history n` mostra su terminale gli ultimi n comandi eseguiti.

3.3 Struttura dei files e delle directories

Dal punto di vista logico un file è un contenitore di informazioni memorizzate in modo permanente (cioè che non vengono perse quando il computer viene spento). Dal punto di vista fisico un file è una porzione di un disco magnetico sulla quale si può scrivere, leggere e cancellare.

Ogni file possiede un nome, che ne consente l' identificazione. Inoltre, per consentire una migliore gestione dello spazio disponibile, i files sono, dal punto di vista logico, racchiusi in contenitori che si chiamano "directories". Ogni directory può contenere files o altre directories, di modo che l' intero sistema (detto "filesystem") risulta organizzato ad albero (Fig. 3.2). Il nome completo di un file è dato dalla serie completa dei nomi delle directories in cui è contenuto a partire dalla radice dell' albero oltre che dal suo nome proprio. La radice si chiama **/**, e lo stesso carattere **/** divide i nomi delle directories ed il nome dell' ultima directory dal nome proprio del file. Ad esempio il nome del file **prova.c** nella directory **labc** sarà

/usr/users/cognome1/labc/prova.c

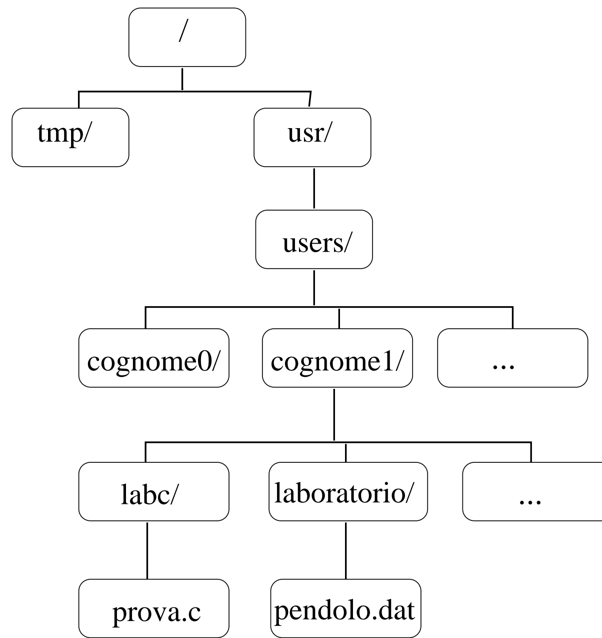


Figura 3.2: Esempio di struttura ad albero di directories

Ovviamente è piuttosto scomodo utilizzare i nomi completi dei files, visto che di solito sono lunghi; di conseguenza Linux fornisce alcuni strumenti per abbreviare i nomi. Il più importante si chiama "working directory" (WD): si tratta di una particolare directory che viene usata come punto di partenza per espandere i nomi di files che non cominciano per **/**. Ad esempio, se la WD è **/usr/users/cognome1** il nome del file precedente può essere abbreviato in

labc/prova.c

Per sapere quale è la WD si usa il comando **pwd**; per cambiare la WD si usa il comando **cd**, che accetta come parametro il nome di una directory (con o senza **/** iniziale a seconda che si voglia partire dalla radice o dalla WD).

Il nome della WD può essere abbreviato con il simbolo **.** (punto), mentre per la directory genitrice della WD si può usare il simbolo **..** (due punti). Ad esempio, se siamo nella directory **labc**, potremmo accedere al file **pendolo.dat** nella directory **laboratorio**,utilizzando il nome

../laboratorio/pendolo.dat

in quanto, se la WD è **labc**, **..** è un sinonimo di

/usr/users/cognome1/

All'inizio di una sessione la WD è una directory specifica per ciascun utente (il cui nome coincide con lo username); questa directory viene detta "home directory" (HD) e il suo nome può essere abbreviato con il simbolo ˜.

Ogni utente può creare nuove directories all' interno della HD utilizzando il comando **mkdir**.

La quantità di dati che ogni utente può scrivere nella propria HD e nelle directories che essa contiene è limitato. Il comando **quota** consente di verificare la situazione del proprio account in termini di spazio libero (i dati sono espressi in kbytes). Nel caso lo spazio disponibile nella HD non fosse sufficiente, è possibile utilizzare un' area temporanea per salvare files di grandi dimensioni; l' area temporanea si chiama **/tmp**. È evidente che devono essere messi in tale area solo files temporanei o facilmente ricreabili; ad esempio i files eseguibili ottenuti dalla compilazione di un sorgente C possono essere scritti in **/tmp**.

3.4 Controllo dei processi

Come già detto Linux è un sistema multi-task ed ogni SHELL può eseguire diversi processi contemporaneamente. Il comando `ps` mostra tutti i processi che sono in esecuzione nella SHELL. Questo comando è utile per trovare l' identificatore (PID) di un processo che non risponde e che si vuole fermare. L' arresto di processo si esegue con il comando `kill PID`.
Linux permette di sospendere l' esecuzione di un programma e riattivarla in seguito.
Quando un programma è "lanciato" dalla linea di comando digitando il suo nome, si dice che tale processo è eseguito in "foreground" (il terminale resta bloccato fino al termine del programma). Un programma in "foreground" che non risponde può essere fermato bruscamente con Ctrl-C, o sospeso con Ctrl-Z.
Se un stesso programma è "lanciato" aggiungendo dopo il nome il carattere &, si dice che il programma è avviato in "background" (il terminale è libero di ricevere comandi).
Il comando `jobs` mostra la lista dei programmi in "background". Ogni programma è preceduto da un numero (`n`). Per fermare un programma in "background" si esegue il comando `kill %n`. Il comando `fg %n` (`bg %n`) manda in "foreground" ("background") il programma corrispondente. Se `n` non è specificato `fg` e `bg` agiscono sul programma corrente (quello con il + nella lista stampata da `jobs`).

3.5 Nomi dei files e delle directories

Come abbiamo giá detto il SO Linux é "Case sensitive" quindi un file o una directory con il nome **Pippo** é distinta da quello/a con il nome **pippo**.

I nomi dei files e delle directories non possono contenere nessuno dei seguenti caratteri: spazio, ?, * ,! ,apici, accenti e virgolette varie, <, >, |, il carattere separatore '/', '%', '$', '; parentesi varie. Questi sono caratteri che hanno speciali significati per la shell. Anche far cominciare il nome di un file con il carattere - non é una buona idea, in quanto usato per indicare le opzioni di un comando.

3.6 Comandi per la gestione di files e directories

La maggior parte dei comandi del SO agisce su files: di conseguenza è molto importante imparare bene a scrivere i nomi dei files ed a manipolare le directories.
La Tab. 3.1 contiene è una lista non esaustiva dei comandi per la manipolazione di files in Linux, le opzioni riportate sono solo quelli più utilizzate: *nome* indica un generico nome di file o di directory, mentre *fil* e *dir* si riferiscono in modo specifico a un file e a una directory. Le opzioni si possono combinare; se, per esempio, si vuole eseguire il comando `ls` con l' opzione `-a` e `-l` si scriverà `ls -la`.
Per una descrizione più completa si rimanda al manuale online (vedere Par. 3.12).

comm	opt		
pwd			Stampa su schermo il nome della WD (Working Directory)
echo		*textline*	Mostra la linea di testo *textline*
cd		*dir*	Cambia la WD in dir
ls		*nome*	Lista i files contenuti nella dir specificata
	-l		stampa infomazioni aggiuntive sui files (data, protezioni)
	-a		lista tutti i files (compresi quelli del tipo .nome)
	-F		aggiunge un classificatore tra i simboli (*/=@—) a ciascun file
	-t		ordina i files per data dell' ultima modifica
mkdir		*dir*	Crea *dir*
rmdir		*dir*	Cancella *dir*
mv		*nome1 nome2*	Ridenomina *nome1* in *nome2*
cp		*nome1 nome2*	Copia *nome1* in *nome2*
	-r		copia il contenuto della dir *nome1* nella dir *nome2*
rm		*fil*	Cancella *fil*
	-i		chiede conferma prima di cancellare ciascun file
cat		*fil*	Stampa su schermo il contenuto di *fil* (accetta più files)
less		*fil*	Impagina il contenuto di *fil* su schermate
			(spazio → schermata successiva, b → schermata precedente)
tail	-n N	*fil*	Stampa su schermo le ultime N linee di *fil*
head	-n N	*fil*	Stampa su schermo le prime N linee di *fil*
grep		*string fil*	Cerca la stringa *string* nel file *fil*
	-i		non fa differenza tra lettere maiuscole e minuscole (in *string*)
find	-name	*fil*	Cerca un file in WD e nelle sue sotto-directories
diff		*fil1 fil2*	Stampa su schermo le linee di testo diverso tra due files
ln	-s	*fil link-name*	Crea un equivalenza simbolica tra *link-name* e *fil*
			(tutte le operazioni eseguite su *link-name* sono applicate a fil)
	-f		se *link-name* esista già lo cancella
lpr	-P *printer*	*fil*	Manda in stampa sulla stampante *printer* il file *fil*

Tabella 3.1: Breve elenco di alcuni comandi Linux utili per la gestione di files

3.6.1 Utilizzo di wildcard

Può essere utile, in alcuni casi, lasciare le lettere non definite nei nomi dei files (wildcards) (ad esempio se si vogliono eseguire comandi su più files aventi una stringa di caratteri comune nel nome).
Le wildcards consentite sono:

*	sostituisce qualsiasi stringa di caratteri (di qualsiasi lunghezza)
?	sostituisce un singolo carattere
[03]	sostituisce il carattere 0 o 3
[0-3]	sostituisce i caratteri corrispondenti ai numeri da 0 a 3 (per i caratteri vale l' ordine alfabetico)

Es.:
Se in una directory sono contenuti i files: test00.f test1.f test2.f test3.f, il comando `ls test*.f` li listerà tutti, `ls test?.f` listerà solo gli ultimi tre e il comando `ls test[12].f` listerà il secondo e il terzo.

3.6.2 Pipe e redirezione

In Linux i comandi possono essere eseguiti in sequenza. Per passare l' output di un comando come input al seguente si utilizza il carattere | (chiamato pipe). Vediamo un esempio:

```
ls | grep ps | less
```

questo comando lista il contenuto dei files nelle directory (`ls`), cerca quelli che contengono la stringa ps nel nome (`grep`) e li passa a `less` che li visualizza su schermo.
I simboli `<` e `>` permettono di ridirigere lo standard input (tastiera) e lo standard output (terminale).
Se scriviamo:

```
ls > listato
```

il listato della directory invece di comparire sul terminale viene scritto sul file *listato* (se non esiste il file viene creato, se esiste sovrascritto).
Il comando

```
ls >> listato
```

è simile al precedente salvo che, ora, il risultato di `ls` è scritto in coda al file *listato* (se non esiste il file viene creato).

3.6.3 Protezioni

Le protezioni definiscono come i diversi utenti possono accedere a files e directories.
Le protezione per un file/directory possono essere controllate con il comando `ls -l` che fornisce (per una directory) un risultato di questo tipo:

```
drwxr-xr-x   2 smith                1024 Oct  7 20:01 Work
```

La legenda della prima serie di caratteri (che contiene le protezioni) è data dalla seguente tabella

```
Type  User Group  Other
 *    ***    ***    ***
 d    rwx    rwx    rwx
 l    ---    ---    ---
 -
```

Il primo carattere definisce il tipo di file (d=directory,l=link simbolico,−= file), gli altri regolano i privilegi di accesso ai files (r=lettura,w=scrittura,x=esecuzione) da parte di diversi gruppi di persone.
Lo User è l' utente che si è collegato, Group il suo gruppo di appartenenza, Other tutte le altre persone che possono avere accesso a quella macchina.
Per modificare le protezioni si utilizza il comando `chmod`.
Es.:
`chmod g+x` *fil*: aggiunge a Group il diritto di eseguire *fil*
`chmod o-x` *fil*: toglie a Group il diritto di leggere *fil*

3.7 Il text editor

Il text editor è uno strumento che serve a creare files; ogni tipo di file "leggibile" (un testo, il sorgente di un programma, un file di dati) può essere creato con un text editor. Esistono molti tipi di editor, e la scelta dipende essenzialmente dai gusti personali. Quello che vi consigliamo si chiama **emacs**. Per attivarlo è sifficiente dare il comando

emacs nome-fil &

o semplicemente

emacs &

Se la variabile DISPLAY (Par. 3.13) è correttamente definita, verrà creata sul vostro X-terminal una nuova finestra per l' editor; **emacs** possiede un numero enorme di comandi che consentono funzioni di editing molto sofisticate, ma la descrizione di queste possibilità esula dagli scopi di queste note. Il menu "Help" contiene tutte le informazioni che servono per usare al meglio **emacs**. Basterà qui osservare che **emacs** è un editor full-screen, e che quindi è possibile muoversi all' interno del file usando i tasti con le frecce; inoltre la maggior parte dei comandi possono essere attivati usando il mouse e la "menu-bar" che appare nella parte alta della finestra. È possibile aprire più files contemporaneamente, utilizzando l' opzione "Open File" nel menu "Files" e il menu "Buffers" per scegliere tra i diversi files. Per salvare un file modificato si usa il comando "Save Buffer" nel menu "Files".
Siccome disponete, per le esercitazioni, di terminali che consentono di lavorare su più finestre, è conveniente evitare di aprire e chiudre l' editor in continuazione. All' inizio della seduta potrete aprire **emacs** utilizzando la "&" a fine riga in modo da creare una finestra indipendente per l' editor e lasciare il vostro terminale libero di eseguire i comandi di compilazione ed esecuzione. Dopo avere eseguito una modifica al vostro file, salvatelo senza uscire dall' editor con "Save Buffer" nel menu "Files", compilate ed eseguite utilizzando il terminale e tornate a modificare il file nella finestra dell' editor, che dovrà essere chiusa solo alla fine del lavoro.

3.8 Compilazione ed esecuzione di un programma

In questo paragrafo daremo una breve descrizione dei comandi necessari per compilare un programma allo scopo di realizzare il primo semplice programma in C. Un' introduzione più completa sarà fornita in Par. 5.2.

Il comando di compilazione permette di tradurre un programma scritto in linguaggio di "alto" livello (in linguaggio C nel nostro caso) in un codice eseguibile dalla macchina. La compilazione prevede in genere due fasi: la compilazione vera e propria che trasforma il file sorgente (.c) in file oggetto (.o) e la fase di "link" in cui si produce il file eseguibile includendo, se necessario, files di libreria o altri files oggetto.
Il comando:

```
cc -o file file.c
```

compila il sorgente `file.c` e crea l' eseguibile `file`.
Questo comando crea in un solo passo l' eseguibile ed è equivalente a

```
cc -c -o file.o file.c     (compilazione)
cc    -o file   file.o     (link)
```

Il file eseguibile può essere eseguito semplicemente digitando il suo nome completo:

```
./file
```

3.9 Il debugger simbolico

Il debugger simbolico è uno strumento software che consente di eseguire un programma una istruzione alla volta e che permette di esaminare il contenuto delle variabili (sia locali che globali) in una qualunque fase dell' esecuzione. È quindi uno strumento utilissimo per capire le cause di malfunzionamento di un programma. Esistono sul mercato vari tipi di debugger: quello che vi consigliamo si chiama **gdb**.

Esistono due modi per utilizzare un debugger; il primo modo è "post–mortem", nel senso che utilizza il file chiamato `core` che ogni programma genera in caso di errore. In questo caso i comandi da usare sono:

```
gdb <nome programma> core
where
```

Il comando `where` del debugger vi informarà di quale linea del programma ha causato l' errore. Per uscire dal debugger utilizzate il comando `quit`.

Il secondo utilizzo del debugger è quello di eseguire un programma passo–passo. In questo caso si comincia con

```
gdb <nome programma>
```

Si deve quindi predisporre almeno un "breakpoint", ovvero una posizione del programma alla quale l' esecuzione deve sospendersi; un breakpoint può essere specificato in due modi:

```
break <routine>   ! Sospende l' esecuzione all' ingresso nella routine
```

oppure

```
break <numero linea>  ! Sospende l' esecuzione alla linea specificata
```

Il numero di linea da specificare è quello del programma sorgente.

Dichiarati i breakpoints si può cominciare l' esecuzione con il comando `run`. Quando l' esecuzione si interrompe si può:

- riprendere fino al successivo break con il comando `cont`;
- eseguire solo la linea successiva senza entrare in eventuali routines chiamate con il comando `next`;
- eseguire solo la linea successiva entrando in eventuali routines chiamate con il comando `step`;
- proseguire fino all' uscita dalla routine corrente con il comando `return`;
- leggere il file sorgente nel punto corrispondente alla linea successiva a quella appena eseguita con il comando `list`;
- esaminare il contenuto di una variabile visibile nella routine corrente con il comando `print <nome_variabile>`;
- modificare il contenuto di una variabile con il comando `set <nome_variabile> = <valore>`;
- creare nuovi breakpoints.

Le componenti dei vettori possono essere esaminate individualmente, usando le parentesi quadre (per i programmi scritti in C).

L' utilizzo di questi comandi, e dei molti altri disponibili per i quali potete consultare il manuale (vedi Par. 3.12), consente di trovare le cause di malfunzionamenti e può aiutare a comprendere meglio il modo di operare di alcune istruzioni dei linguaggi di programmazione.

Il debugger **gdb** possiede una versione grafica cui potete accedere con il comando **xxgdb**. In questa versione il file sorgente è sempre visibile ed è possibile definire un breakpoint semplicemente "cliccando" a fianco di ogni linea.

3.10 Procedure di comandi

Accade sovente di dover usare ripetutamente sequenze di comandi complesse. Per consentire un risparmio di tempo ed un incremento di flessibilità gli interpretatori dei comandi (shell) sono dotati di un certo grado di programmabilità: è cioè possibile scrivere delle sequenze di comandi in un file (che viene detto "script") ed eseguirle scrivendo un solo comando. Inoltre all' interno di uno script si possono utilizzare variabili e strutture complesse tipo if-then-else, do-while e così via. Un qualsiasi manuale di Linux riporta in dettaglio le possibilità di programmazione incluse nei diversi tipi di shell (quella che noi utilizziamo si chiama `csh`,"C shell"[1]). La scelta della "C shell" è dettata dal fatto che, come dice il nome, la sua sintassi è molto simile a quella del C.
Nel seguito analizzeremo solo alcuni elementi sintattici di base che possono servire per scrivere scripts molto semplici.

[1] la shell di login `tcsh` è semplicemente la versione "interattiva" della `csh`

3.10.1 Uso delle variabili

Nella C shell possono essere definiti variabili e vettori di tipo stringa. Esistono variabili locali e variabili globali: le prime esistono solo all' interno della procedura che le crea, le seconde restano attive nell' arco di tutta la sessione. Le variabili locali vengono dichiarate con il comando **set**, mentre quelle globali con il comando **setenv**; ad esempio:

`set a1 = val_1`	Crea la variabile locale a1 e le assegna il valore val_1
`setenv g1 val_2`	Crea la variabile globale g1 e le assegna il valore val_2
`set vect = (v1 v2 v3)`	Crea il vettore a tre componenti vect ed assegna i valori v1, v2 e v3 alle tre componenti

Il contenuto della variabile `a1` è dato dal simbolo `$a1`, il valore dell' n-esima componente del vettore `vect` dal simbolo `$vect[n]`, mentre il numero di componenti del vettore `vect` è dato da `$#vect`.

È possibile assegnare ad una variabile l' output di un comando, includendo il comando medesimo tra apici rovesciati (‘); ad esempio

```
set files = (‘ls‘)
```

crea un vettore di nome `files` che contiene i nomi dei file presenti nella WD, dati come output dal comando **ls**.

Alcune variabili globali sono automaticamente definite dal sistema nella fase di login. Per vedere quali sono basta eseguire il comando `setenv` senza argomenti.

Una importante variabile globale è *PATH* che contiene la lista delle directories in cui il sistema cerca un file eseguibile se questo non è specificato con il nome completo.

3.10.2 Realizzazione ed esecuzione di una procedura di comandi

Una procedura di comandi può essere scritta con un text editor. La prima riga deve contenere la specificazione della shell che eseguirà i comandi. Nel caso della C shell la prima riga sarà

```
#!/usr/bin/csh
```

Le linee successive conterranno la sequenza di comandi che si desidera eseguire. Le righe che cominciano com `#` sono considerate commenti e non vengono eseguite. Una volta preparato il file è necessario renderlo eseguibile con il comando

```
chmod u+x nome-fil
```

Per eseguirlo basta quindi scriverne il nome completo, o solo il nome proprio nel caso lo script sia in una directory contenuta nel `PATH`. Il nome dello script può essere seguito da uno o più parametri. Questi verranno passati allo script come un vettore locale di nome `argv`. Di seguito trovate il listato di uno script che esegue compilazione e link del sorgente specificato come primo argomento ed esegue il programma così ottenuto. Se il secondo parametro è `-c` viene eseguita la sola compilazione, se è `-r` il programma viene solo eseguito.

```
#!/usr/bin/csh
if ($#argv == 1) then
  cc -o /tmp/$argv[1] argv[1] fun1.c ‘cernlib graflib‘
```

```
  chmod u+x /tmp/$argv[1]
  /tmp/$argv[1]
else
  switch ($argv[2])
    case -c:
    cc -o /tmp/$argv[1] argv[1] fun1.c `cernlib graflib`
    chmod u+x /tmp/$argv[1]
    breaksw

    case -e:
    /tmp/$argv[1]
    breaksw

    default:
    echo "Unknown option $argv[2]"
endif
```

Sulla base di questo esempio vi sarà possibile creare procedure di comandi che semplifichino le operazioni che eseguite frequentemente.

3.11 Archiviazione di files

È in generale buona norma archiviare e salvare il lavoro fatto su un apposito supporto per evitare che vada perso per erronee operazioni dell' utente o problemi di sistema. Nel corso non vi dovrete preoccupare di questo problema perchè l' archiviazione di files verrà eseguita automaticamente; tuttavia nel seguito descriveremo le operazioni di base necessarie.

3.11.1 Compressione di files

Per limitare l' occupazione dello spazio si disco può essere utile comprimere, con un processo riproducibile, i files in modo che occupino meno spazio. Per esemplificare il metodo di compressione consideriamo un file di testo: invece di scrivere tutti i caratteri si può scrivere il singolo carattere e quante volte questo è ripetuto (basti pensare a quanti spazi bianchi consecutivi sono contenuti in un file di testo per capire quanto si può guadagnare). Ovviamente gli attuali algoritmi di compressione utilizzano metodi più sofisticati di quello descritto.
Il comando per comprimere[2] è
`gzip` *fil*
che crea il file compresso *fil*.gz, che può essere decompresso con il comando:
`gunzip` *fil*.gz
Per archiviare un' intera struttura di directory in unico file si usa il comando:
`tar cvf` *tarfile.tar dir1 dir2 dir3*
che crea (`c`=create) il file archivio *tarfile*.
Per riottenere la struttura di directories dal file *tarfile.tar*:
`tar xvf` *tarfile.tar dir1 dir2 dir3*
(`x`=extract).
Il file di archivio può essere automaticamente compresso se si aggiunge l' opzione `z`.

[2] i comandi riportati sono modifiche elaborate dalla fondazione GNU ai comandi standard `zip` e `unzip`

3.11.2 Accesso a dischi rimuovibili

Uno dei supporti più semplici e più economici su cui salvare files sono i floppy disk.
Prima di utilizzare un floppy disk occore creare su di esso un "file-sistem" (formattazione) con il comando:
`mkfs -t` *os device blocks*
dove *os* è il tipo di "file-system" (*ext2* è il default in Linux, *msdos* è invece utile se volete rileggere il dischetto anche da Windows), *device* è l' indetificatore del device dell' unità floppy (in genere e nel nostro caso /dev/fd0) e *blocks* è il numero di blocchi (in unità di 1Kb) da allocare sul disco (per un floppy *blocks* = 1440).
Fatta questa operazione il dischetto è pronto per essere scritto ma non fa ancora parte del sistema su cui state lavorando; per includerlo nel sistema occorre montarlo con il comando:
`mount /mnt/floppy` (se il "file-system" è ext2),
`mount /mnt/winfloppy` (se il "file-system" è msdos).
A questo punto potete fare sul dischetto tutte le operazioni che usualmente fate sul disco rigido (quello che contiene la vostra HD).
Finite le operazioni dovete smontare il dischetto con il comando `umount` (analogo a `mount`).

3.12 Come ottenere ulteriori informazioni

Ulteriori informazioni sono reperibili, oltre che sui manuali cartacei, direttamente sul computer. La sintassi di ogni comando e funzione di libreria di Linux è documentata per mezzo di un help "on–line" cui potete accedere tramite il comando **man** (o **xman** se state utilizzando un X-terminal; la sintassi è

```
man <comando>
```

o semplicemente

```
xman &
```

Se non si sa il nome del comando che interessa, ma si conosce qualche parola chiave relativa all'argomento si può dare il comando

```
man -k parola_chiave
```

3.13 Il sistema grafico X11

Per utilizzare un elaboratore è necessario avere un terminale. Il terminale in senso stretto è oggetto "stupido", nel senso che si limita a mostrare sullo schermo i caratteri che riceve dal computer cui è collegato e a trasmettere al computer i caratteri digitati sulla tastiera.
Più spesso, però, quello che si indica con terminale è un piccolo computer capace di elaborazioni grafiche piuttosto complesse: in generale possiede un "sistema a finestre" dove ogni finestra può emulare un terminale tradizionale che può essere connesso ad uno o piu' elaboratori. Inoltre può ricevere comandi grafici da altri computers ed eseguirli mostrando il risultato su una finestra.
L' architettura "client-server" gestisce, nei sistemi LINUX, il funzionamento dei terminali.
Il "server" è un computer che invia dei comandi grafici, il "client" è un "oggetto" che riceve i comandi grafici e li esegue mostrando il risultato su terminale grafico. La comunicazione tra client e server avviene tramite una rete di trasmissione dati, nella quale ad ogni macchina è attribuito un "indirizzo" che ne consente l' identificazione (vedere Fig. 3.3).
Il protocollo di comunicazione si chiama X11.

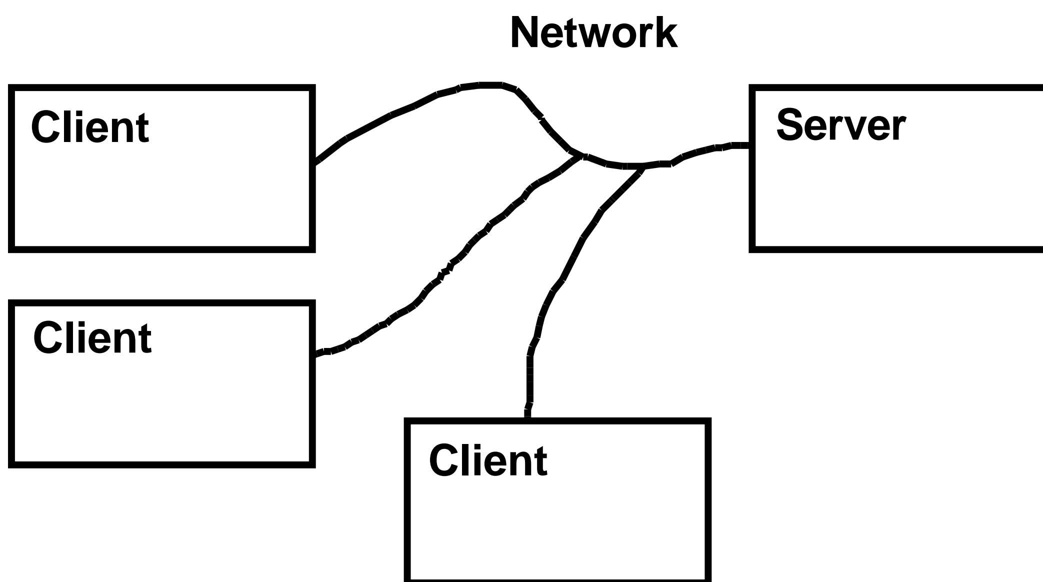


Figura 3.3: Esempio di sistema client-server

Per le esercitazioni utilizzerete dei PC con il sistema operativo LINUX.
Il loro indirizzo è **stationN.aula.fisica** con $N = 1..18$. In questo caso non si tratta di terminali ma di veri e propri computers che possono essere usati, oltre che per collegarsi ad un sistema remoto, anche per eseguire programmi localmente. In questo caso ciascuna macchina è un sistema "client-server".
Per utilizzare questi PC come semplici "client" nel collegamento remoto (as es. a server3) occorre seguire i seguenti passi:

- su una delle finestre inviare il comando

```
telnet server3.fisica.unige.it
```

- Seguire la normale procedura di login (vedere Par 3.2.2).
- Informare **server3** dell' indirizzo dell' X-terminal che state usando, in modo che sia in grado di aprire le varie finestre grafiche su quel terminale. In generale questa operazione viene eseguita automaticamente durante il login; in alcuni casi però è necessario eseguirla manualmente. Se avete dubbi potete controllare quale X-terminal **server3** pensa voi stiate usando con il comando **echo $DISPLAY**, e fornire l' indirizzo con il comando **setenv DISPLAY xxxxN.fisica.unige.it:0** (Per maggiori dettagli vedere Par. 3.10)

Tutte le esercitazioni del corso utilizzeranno come i PC macchine locali, in questo caso la variabile DISPLAY viene automaticamente attribuita.
Altre macchine a vostra disposizione sono i PC dell'aula di Laboratorio 1 esp1N.fisica.unige.it $N = 01..10$ e i PC dell' aula terminali al quarto piano linuxdN.fisica.unige.it $N = 1..8$. Tutti questi PC hanno il sistema operativo Linux e condividono la stessa directory di login (vedi Par. 3.3).

Capitolo 4: I linguaggi evoluti

4.1 Introduzione

I linguaggi evoluti sono insiemi di regole sintattiche che consentono di specificare operazioni eseguibili da una generica CPU. Rispetto ai linguaggi macchina, i linguaggi evoluti hanno il vantaggio di rendere possibile l' utilizzo di nomi simbolici per le variabili e per le subroutines; inoltre mettono a disposizione del programmatore un insieme di istruzioni elementari molto più ampio di quello disponibile in un normale linguaggio macchina, ed hanno l' ulteriore vantaggio di essere totalmente indipendenti nella sintassi e nella funzionalità dal tipo di CPU utilizzata. Quindi, grazie all' utilizzo di linguaggi evoluti e di sistemi operativi diventa possibile scrivere programmi in modo del tutto indipendente dal tipo di computer utilizzato.

Naturalmente, siccome alla fine ogni particolare CPU non puó che eseguire programmi scritti nel suo linguaggio macchina, dovrà essere svolta una operazione di traduzione che porti dal programma scritto in linguaggio evoluto al codice eseguibile scritto in linguaggio macchina.

4.2 Compilazione e Link

L' operazione di traduzione di un programma in linguaggio evoluto in uno in linguaggio macchina si chiama compilazione, e viene eseguita per mezzo di un programma specifico detto "compilatore". I compilatori sono specifici, oltre che del linguaggio evoluto utilizzato, anche del tipo di CPU utilizzata, della quale devono conoscere il linguaggio macchina; sono invece di solito indipendenti dai dettagli di configurazione del computer utilizzato.

Il codice in linguaggio macchina generato da un compilatore non è tuttavia eseguibile dalla CPU, e prende il nome di "codice oggetto". Il codice oggetto è in effetti solo la traduzione in linguaggio macchina del programma effettivamente scritto dal programmatore in linguaggio evoluto: mancano tutte quelle routines che permettono di realizzare in linguaggio macchina le operazioni elementari in linguaggio evoluto, come pure mancano tutte le routines specifiche del sistema operativo che rendono possibile il dialogo con le periferiche. Ad esempio, in qualunque linguaggio evoluto la stampa sul monitor del valore di una variabile reale chiamata `A` si può eseguire con una semplice istruzione del tipo `PRINT A`. Dal punto di vista della CPU questo significa leggere la memoria alla locazione corrispondente alla variabile `A`, decodificarla tenendo conto della codifica utilizzata, ricavare una rappresentazione corrispondente in base 10, scriverla su una stringa e passare questa stringa al driver del monitor perché la visualizzi. Queste operazioni vengono eseguite da routines specifice del linguaggio e del sistema operativo che devono essere prelevate da opportune librerie. L' operazione di completamento del codice oggetto con le opportune routines prelevate dalle librerie di linguaggio o di sistema operativo si chiama "link". Durante tale operazione possono essere aggiunte al codice oggetto anche routines prelevate da librerie utente.

Osserviamo infine che nel sistema operativo UNIX le operazioni di compilazione e link vengono di solito eseguite dallo stesso comando (**cc** per il C); è però possibile, usando opportune opzioni dei comandi (vedere Par.5.2), eseguirle separatamente.

4.3 Elementi di un linguaggio evoluto

In un linguaggio evoluto si possono distinguere diverse componenti:

- Sintassi: indica l' insieme delle regole generali che governano la scrittura del programma.

- Struttura: indica il modo in cui è possibile suddividere un programma complesso in sottoprogrammi e fornisce le regole per far comunicare tra loro i sottoprogrammi.
- Variabili: sono le regole per la definizione e l' utilizzo delle aree di memoria che contengono i dati da elaborare.
- Controllo di Flusso: sono le regole per eseguire test, cicli etc.
- Funzioni: sono i comandi per eseguire operazioni complesse, sia di tipo aritmetico che di input/output.

Nel seguito ci occuperemo degli aspetti generali (cioè sostanzialmente indipendenti dal linguaggio) di variabili e struttura. Sintassi, Controllo di Flusso e Funzioni sono invece specifiche dei diversi linguaggi; per il C, un breve riassunto di queste caratteristiche si trova al Capitolo 5

4.4 Le variabili

Una variabile è una porzione di memoria a cui il compilatore assegna un nome. È importante ricordare che il nome ha un senso solo nell' ambito del programma in linguaggio evoluto; dopo la compilazione resteranno solo gli indirizzi numerici di memoria, ed il programma eseguibile non saprà più nulla dei nomi. Soltanto nel caso la compilazione venga eseguita con l' opzione **-g** al codice eseguibile verrà aggiunta una tabella di corrispondenza nome-indirizzo che consentrà al "debugger simbolico" (Par 3.9) di accedere, nel corso dell' esecuzione del programma, alle variabili utilizzando il loro nome.

Nel seguito esamineremo le caratteristiche principali delle variabili.

4.4.1 Tipo e dimensione di una variabile

Si dice "tipo" di una variabile il metodo di codifica utilizzato per scrivere i dati in memoria. Come abbiamo visto esistono codifiche adatte ai numeri interi, ai numeri reali, alle variabili alfanumeriche e così via; i diversi linguaggi utilizzanno diverse parole chiave per indicare lo stesso tipo di codifica; si veda Tab. 5.2 per i nomi utilizzati dal C.

Il tipo di codifica utilizzato determina anche la lunghezza in bits di un elemeto di quel tipo: ad esempio un reale in singola precisione è scritto su 32 bits, ovvero su 4 bytes, mentre un reale in doppia precisione occupa 64 bits, ovvero 8 bytes. Oltre alla lunghezza di un singolo elemento i linguaggi evoluti consentono di creare vettori e matrici di singoli elementi di un dato tipo. La dimensione di una variabile sarà quindi data dal numero complessivo di elementi del vettore o della matrice; vale la pena di ricordare che la lunghezza totale occupata in memoria da una variabile sarà data dal prodotto della dimensione per la lunghezza in bytes del singolo elemento.

4.4.2 Posizione in memoria degli elementi di una matrice

Gli elementi di un vettore vengono sempre posti in memoria in locazioni consecutive. Per quel che concerne le matrici e i vettori a più indici invece la disposizione degli elementi in memoria è meno ovvia; in particolare in C la disposizione degli elementi avviene nell' ordine $i[0][0]$, $i[0][1]$, ..., $i[0][n-1]$, $i[1][0]$, ... L' ordine esatto della disposizione delle matrici in memoria è importante, in quanto molto spesso i computers accedono molto più velocemente a blocchi contigui di memoria che non a elementi sparsi; quindi quando si realiza un loop che legge o scrive gli elementi di un vettore è bene cercare di assecondare l' ordine naturale della posizione dei singoli elementi: basterà ricordare che in C devono scorrere più velocemente gli indici più a destra.

4.4.3 Tipi composti

I tipi composti sono agglomerati di variabili standard (sia scalari che vettori o matrici). Una volta definiti (vedere Par. 5.12), i tipi composti possono essere utilizzati, come quelli predefiniti, per definire variabili scalari, vettori o matrici. Pure una variabile appartenente a un tipo composto può essere passata come argomento ad una subroutine specificandone semplicemente il nome: per questo motivo i tipi composti consentono di semplificare molto la scrittura dei programmi ed il trasferimento di dati tra diverse routines e tra i programmi ed i files di dati. La sola limitazione, superata solo nei linguaggi "object oriented" (come il C++), è che non si possono definire operazioni sui tipi composti, ma si deve sempre ricorrere alle operazioni di base facendole agire sui singoli "campi" che definiscono il tipo composto.

4.5 Allocazione statica ed automatica della memoria

Con allocazione della memoria si indica l' operazione che il compilatore esegue quando riserva lo spazio di memoria necessario per ospitare una variabile.

L' allocazione statica viene eseguita al momento della compilazione, e di conseguenza non può essere alterata nel corso dell' esecuzione. Questo metodo di allocazione viene usato normalmente dai compilatori per le variabili globali di un programma: queste variabili vengono allocate una volta per tutte e rimangono nella medesima posizione di memoria durante tutta l' esecuzione del programma. In questo modo possono essere utilizzate con sicurezza da tutte le componenti del programma.

L' allocazione statica è svantaggiosa per variabili che non sono necessarie durante tutto lo svolgimento del programma, ma solo durante fasi particolari, in quanto mantiene lo spazio relativo occupato inutilmente. In questi casi viene usata l' allocazione automatica, che consiste nel riservare lo spazio in memoria al momento in cui la variabile deve essere utilizzata (durante l' esecuzione del programma quindi) e nel rilasciarlo quando la variabile non serve più. Questa tecnica è utilizzata normalmente per le variabili locali delle subroutine, che vangono allocate al momento della chiamata della routine e buttate via quando la routine termina l' esecuzione. Di conseguenza non si deve mai fare affidamento sul fatto che una variabile locale conservi il suo valore tra una chiamata e la successiva, a meno che non si utilizzino dichiarazioni particolari (Par. 5.10) che forzano l' allocazione statica anche per le variabili locali.

4.6 I puntatori

I metodi di allocazione che abbiamo visto fino ad ora sono eseguiti e comandati dal compilatore, vuoi in fase si compilazione (allocazione statica) vuoi in fase di esecuzione (allocazione automatica). In alcuni casi tuttavia è necessario poter intervenire direttamente sul meccanismo di allocazione delle variabili: il caso tipico è quello che si presenta quando si devono disporre n dati in un vettore, ma n non è noto al momento della compilazione ma viene deciso volta per volta al momento dell' esecuzione del programma. In questo caso si vorrebbe decidere la lunghezza del vettore, e quindi eseguire l' allocazione, al momento dell' esecuzione.

Lo strumento che consente di intervenire sul processo di allocazione è il "puntatore". Si tratta di una normale variabile, di solito allocata staticamente, che contiene, invece di un dato, l' indirizzo in memoria di un' altra variabile.

Quamdo si accede ad una variabile per mezzo di un puntatore il sistema prima legge il contenuto del puntatore, quindi lo utilizza per trovare la posizione in memoria sulla quale effettuare l' operazione richiesta. La variabile "puntata" dal puntatore viene detta "target".

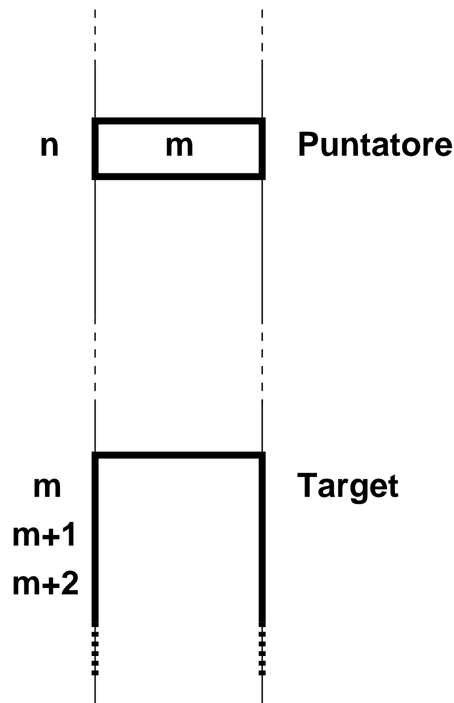


Figura 4.1:

4.6.1 Allocazione dinamica della memoria

Grazie all' utilizzo dei puntatori e di speciali routines del sistema operativo che consentono, nel corso dell' esecuzione di un programma, di cercare aree di memoria libere e di riservarle ad un particolare uso, è possibile realizzare la allocazione dinamica delle variabili in memoria.

La sola operazione necessaria in fase di compilazione consiste nel definire un puntatore. Durante l' esecuzione del programma, una volta deciso quanto spazio è necessario per la variabile che vogliamo allocare, dovremo solo chiedere al sistema operativo di trovare una zone di memoria sufficientemente grande e dovremo scrivere nel puntatore l' indirizzo iniziale di questa zona. Ciò fatto potremo accedere alla nostra variabile tramite il puntatore come se si trattasse di una variabile allocata in modo statico; la variabile sarà locale o globale a seconda che il puntatore, e quindi l' indirizzo in esso contenuto, siano leggibili o no da diverse routines. Quando la nostra variabile non ci servirà più potremo rilasciare lo spazio ed azzerare il puntatore, o eventualmente potremo riutilizzare il puntatore per puntare a un target di dimensione differente: in questo modo sarà possibile, ad esempio, incrementare progressivamente, secondo le esigenze che si presentano nel corso dell' esecuzione, la dimensione di un vettore allocato dinamicamente.

Malgrado gli enormi vantaggi di versatilità e flessibilità che presenta, l' allocazione dinamica scarica completamente sul programmatore la responsabilità di controllare la corretta gestione della memoria. Per questo motivo va utilizzata solo quando realmente necessario e facendo la dovuta attenzione.

Al Paragrafo 5.13 sono descritte le tecniche usate in C per realizzare praticamente l' allocazione dinamica. Riassumiamo qui solo i concetti fondamentali. Il C utilizza simboli specifici per il puntatore ed il target (se `p` è un puntatore `*p` è il suo target. I puntatori si comportano all' incirca come variabili intere, per cui le operazioni sui puntatori si possono eseguire (quasi) come su normali interi positivi. Ogni puntatore è specifico del tipo del target (i puntatori a

interi sono diversi dai puntatori a float ad esempio) e può essere usato indifferentemente per puntare a scalari o a vettori; nel secondo caso la notazione per l' accesso al vettore è identica a quella che si usa per i vettori allocati staticamente.

4.7 Programmazione strutturata

Con il termine "programmazione strutturata" si intende la suddivisione di un programma complicato in sottoprogrammi aventi ciascuno una funzionalità ben definita. Questo metodo di programmazione consente di semplificare notevolmente le fasi di messa a punto di un programma, dal momento che i vari pezzi possono essere provati e messi a punto indipendentemente, anche da persone differenti. Inoltre sottoprogrammi scritti in un particolare contesto possono poi essere riutilizzati, sostanzialmente senza modifiche in situazioni diverse.

Un altro vantaggio della programmazione strutturata è la possibilità di essere applicato in una progettazione del software di tipo top-down, nella quale cioè il progetto complessivo viene suddiviso in parti più semplici, per ciascuna delle quali è definita una "interfaccia" con le altre, e la procedura è ripetuta iterativamente per ciascun sottoprogetto fino ad arrivare a blocchi elementari. Il vantaggio della progettazione top-down è quello di fornire descrizioni schematiche a diversi livelli di progetti complessi, consentendo a chi lavora in un particolare settore e quindi conosce i dettagli di quella parte, di avere delle descrizioni funzionali di massima du tutte le rimanenti componenti senza per altro dover vemire a conoscenza di alcun dettagio realizzativo.

Gli elementi dei linguaggi di programmazione che consentono la realizzazione dei blocchi elementari di un programma strutturato sono le subroutines.

4.8 Subroutines

Una subroutine è un insieme di istruzioni che vengono eseguite in sequenza; quando la sequenza termina, o quando viene incontrata un' istruzione specifica detta "return" la subroutine termina e il programma riprende dall' istruzione successiva a quella che ne ha causato l' attivazione (istruzione che appartiene per definizione ad un' altra subroutine). Una subroutine particolare è il "main program". Il main program non viene chiamato da nessuna subroutine, ma viene attivato al momento in cui il programma parte; quando la sequenza di istruzioni del main program termina anche il programma termina.

Ogni subroutine può definire al suo interno variabili locali. Tali variabili sono accessibili solo alla subroutine che le dichiara e, in assenza di differenti direttive, sono allocate in modo automatico, per cui vanno perdute non appena l' esecuzione della subroutine termina.

Nella logica della programmazione strutturata, le routines devono comunicare tra loro, scambiandosi dati e risultati delle elaborazioni, e questo non può essere fatto usando variabili locali. Ogni subroutine possiede quindi un insieme di parametri, che vengono usualmente specificati tra parentesi dopo il nome; si tratta variabili fittizie o "dummy", per le quali cioè non esiste una allocazione di memoria. Ciò nonostante possono essere utilizzate nell' ambito della subroutine come normali variabili: infatti, al momento della chiamata, il programma chiamante sepcificherà quali delle sue variabili locali devono essere passate come parametri, e quando la subroutine utilizzerà i nomi delle variabili dummy lavorerà in realtà su queste variabili.

In questo modo si può fare sì che due subroutines comunichino tra loro; in più si può ottenere che una subroutine esegua le operazioni per cui è stata progettata su variabili diverse semplicemente passando parametri differenti in successive chiamate: la subroutine non dovrà sapere nulla dei veri nomi delle variabili su cu lavora, in quanto utilzzerà sempre i nomi delle variabili dummy.

Bisogna sempre fare molta attenzione a che il programma chiamante passi come parametri delle variabili identiche, per tipo e dimensione, a quelle specificate nella definizione della subroutine

come variabili fittizie; in caso contrario i dati scritti in memoria dal programma chiamante saranno decodificati in modo errato dalla subroutine ed i risultati saranno pure errati.

La sintassi per la definizione e la chiamata delle subroutines è descritta al Par. 5.6.

4.8.1 Passaggio dei parametri alle subroutines

In C il passaggio dei parametri alle subroutines avviene "by value": per ogni parametro passato alla subroutine viene eseguita una copia, e questa copia è resa accessibile come variabile dummy; il sottoprogramma può quindi leggere i parametri, ma eventuali modifiche verranno eseguite sulla copia, e non saranno quindi visibili al programma chiamante quando la subroutine sarà terminata. Un modo per ovviare a questa difficoltà consiste nel passare come parametro non una variabile ma un puntatore ad essa (passaggio "by reference"): in questo modo verrà eseguita una copia del puntatore, ma la subroutine, leggendo l' indirizzo della variabile, potrà comunque leggerla e modificarne il contenuto. Questa tecnica viene implicitamente usata per i vettori, in quanto in C il nome di un vettore non seguito da parentesi quadre indica il puntatore al vettore.

4.8.2 Functions

Una variante delle subroutine sono le functions, che si usano di solito per implementare funzioni matematiche o comunque operazioni assimilabili a funzioni matematiche. Le functions posseggono un parametro dummy aggiuntivo, che è associato al nome stesso della function e che viene sempre e solo passato (by value) dalla function al programma chiamante. In questa maniera il nome di una function seguito dalla lista dei parametri posta tra parentesi può essere utilizzato dal programma chiamante all' interno di espressioni aritmetiche o di istruzioni di assegnazione o di test, esattamente come se si trattasse di una normale variabile; tuttavia mentre una normale variabile viene semplicemente letta dalla memoria e usata nell' espressione, una function viene chiamata come una subroutine, le vengono passati i parametri e viene eseguito il codice eseguibile in essa contenuto che termina con la assegnazione della variabile dummy di ritorno; questo valore viene poi sostituito nell' espressione. Di conseguenza l' utilizzo di functions in espressioni, specie all' interno di cicli ripetuti molte volte, può risultare ben più "CPU time consuming" dell' uso di una normale variabile.

Va notato che in C di fatto esistono solo le functions. Una subroutine è vista come una function che ritorna una variabile di tipo `void`.

4.8.3 Ricursività

La ricursività è la capacità da parte di una function di richiamare se stassa. Da quello che abbiamo detto sull' allocazione automatica delle variabili locali si intuische che linguaggi come il C ammettono la ricursività. In effetti ad ogni chiamata (o auto–chiamata) viene creato un nuovo insieme di variabili locali, e di conseguenza si cancellano le possibili interferenze tra due istanze della stessa function contemporaneamente presenti nello stack delle subroutines. Va però osservato che le variabili globali o le variabili locali statiche sono potenzialmente fonti di interferenza tra chiamate ricursive, e vanno usate con molta attenzione nel caso si implementino algoritmi ricursivi.

4.8.4 Moduli e variabili globali

L' utilizzo dei parametri come metodo di comunicazione tra le routines può diventare scomodo se si ha un gruppo di routines che lavorano su un numero elevato di variabili; in questo caso si preferisce ricorrere alle variabili globali. Si tratta di variabili, allocate staticamente, che sono visibili non ad una sola routine ma a gruppi di routines.

Il meccanismo più sicuro per l' implementazione delle variabili globali consiste nella realizzazione di un modulo, ovvero di un gruppo di routines: ogni routine definirà al suo interno le proprie variabili locali, ma tutte le variabili definite entro il modulo ma fuori da ogni routine saranno automaticamente considerate comuni a tutto il modulo. Le regole per la realizzazione dei moduli si trovano al Par. 5.5.

Vale la pena di osservare come l' approccio delle variabili globali cancelli l' indipendenza delle subroutines dai nomi delle variabili su cui esse agiscono: i nomi delle variabili globali sono fissi, e quindi una subroutine che agisce su una variabile globale agisce sempre e solo su quella (a meno che la variabile globale non sia un puntatore che viene ridefinito dinamicamente nel corso dell' esecuzione...)

Se le routines singole sono quindi adatte ad implementare blocchi funzionali semplici, che, nell' ambito di un programma possono essere applicati a diversi insiemi di dati, i moduli vanno utilizzati per per implementare blocchi complessi, che agiscono su un' unica base di dati.

4.8.5 La programmazione object–oriented

I moduli sono collezioni di procedure che agiscono su una base di dati fissata ed univoca. In alcune circostanze questa limitazione (che pure può essere aggirata con un uso sapiente di puntator ed allocazione dinamica) risulta fastidiosa. È il caso delle interfacce grafiche a finestre: ogni finestra corrsiponde a dati diversi, ma i programmi di manipolazione sono similari; si deisdererebbe quindi un accoppiamento naturale tra dati e subroutines, del tipo che tutte le volta che viene creata la struttura dati corrispondente ad una finestra venisse pure creata una copia delle routines di manipolazione specifiche per quella particolare finestra.

Questa èla logica dei linguaggi di programmazione object-oriented (come ad esempio il C++). In questi linguaggi è possibile definire dei tipi composti (detti classi) che contengono, oltre che variabili, subroutines e functions. Ogni volta che si crea una variabile del tipo composto (nel linguaggio OO, ogni volta che si istanzia una classe) vengno anche create copie dedicate delle funzioni. Questo incapsulamento del codice risulta molto utile in tutte quelle circostanze in cui sia necessario, all' interno di un singolo programma, di realizzare molte ripetizioni di algoritmi identici o molto simili agenti su basi di dati differenti.

4.8.6 Interfacce con librerie di routines

Le routines che appartengono ad uno stesso modulo si conoscono tra loro, e di conseguenza si possono chiamare senza bisogno di particolari definizioni; inoltre il compilatore sarà in grado di controllare se le variabili dummy ed i parametri effettivamente passati sono definiti in modo coerente. Si dice in questo caso che l' interfaccia tra le varie routines di un modulo è definita implicitamente.

In alcuni casi tuttavia si devono chiamare routines per le quali non è disponibile una interfaccia implicita: in tal caso si deve creare una interfaccia esplicita, che, nella versione minimale, deve contenere la definizione del tipo dei parametri di ritorno di tutte le function che si desidera utilizzare. Tale definizione è identica a quella di una variabile, con in aggiunta l'opzione "extarnal" (vedere Par. 5.6).

L' interfaccia minimale tuttavia non consente al compilatore di controllare la coerenza dei parametri passati alle subroutines; se si desidera (ed è molto consigliabile) rendere possibile questo test si deve creare una interfaccia completa che specifichi, per ogni subroutine o function, tipo e dimensione di tutti i parametri. La sintassi è descritta al Paragrafo 5.7.

Capitolo 5: Sintassi del linguaggio C

5.1 Introduzione

L' operazione di apprendimento di un particolare linguaggio di programmazione deve in qualche misura essere smitizzata. La maggior parte dei linguaggi evoluti, al di là degli aspetti estetici o semantici, sono tra loro molto simili nella struttura logica: gli strumenti che forniscono al programmatore sono spesso del tutto analoghi. La realizzazione di un programma può essere divisa in due parti: la prima consiste nel passaggio da una descrizione formale di un problema ad una descrizione di tipo algoritmico della sua soluzione; tale descrizione algoritmica, dovendo essere eseguita da una macchina “stupida”, deve prevedere tutte le possibili variazioni del problema e tutti i possibili errori che si possono incontrare nel giungere alla soluzione, e trattarli adeguatamente. La descrizione algoritmica utilizzerà dei “building blocks” fondamentali (ad esempio il ciclo DO-WHILE) che sono propri del linguaggio di programmazione prescelto. Ora, proprio la similitudine nella struttura logica dei linguaggi permette di affrontare questa prima fase della programmazione in modo del tutto indipendente da quella che sarà la scelta finale del linguaggio. La seconda fase della programmazione, ovvero la realizzazione del codice sorgente scritto in un determinato linguaggio, sarà in qualche modo una pura opera di traduzione dal linguaggio astratto dei “building blocks” al linguaggio reale che si è scelto. Servirà quindi una sorta di vocabolario che, data una operazione fondamentale, ne illustri l' utilizzo e spieghi come realizzarla in un dato linguaggio: questo è lo scopo di queste pagine.

Ciò premesso sembrerebbe legittimo affermare che, noto il linguaggio astratto e dato un vocabolario, si possa programmare in qualunque linguaggio. In realtà non tutti i linguaggi di programmazione sono tra loro strutturalmente uguali: alcuni (ad esempio il BASIC) hanno funzionalità inferiori rispetto al C, altri (ad esempio il C++ e tutti i linguaggi “Object Oriented”) posseggono una struttura differente (più flessibile ma più difficile da usare). Vi è tuttavia una famiglia piuttosto ampia di linguaggi che sono del tutto similari al C (tra questi i più diffusi sono probabilmente il FORTRAN e il PASCAL), e che potrete legittimamente affermare di “conoscere” non appena abbiate compreso a fondo la struttura logica comune.

Da queste poche considerazioni si comprende comunque come tutte le disquisizioni sulla superiorità di questo o quel linguaggio siano, nella sostanza, del tutto inutili. La vera scelta sta nel modo di realizzare l' algoritmo che risolve un dato problema, e questa è la vera essenza (e la vera difficoltà) della programmazione. La scelta finale del linguaggio dipenderà da considerazioni di tipo estetico, affettivo, economico o pratico; comunque sarà piuttosto irrilevante. Saper programmare significa essere capaci a risolvere problemi in modo algoritmico (ovvero in modo razionale, controllato e riproducibile, esaminando tutte le possibilità e prevedendo tutte le eccezioni). È un esercizio di logica più che di informatica, e può essere molto utile anche quando non ci si trova davanti ad una tastiera.

Prima di proseguire con l' illustrazione del linguaggio è bene osservare che le note che seguono non pretendono di introdurre tutti gli aspetti e le potenzialità del C, ma solo un sottoinsieme sufficiente ad affrontare le problematiche tipiche delle esercitazioni di laboratorio o della maggior parte delle tesi di laurea. Molti strumenti avanzati non verranno neppure citati, ed in alcuni casi si riporteranno solo sintassi semplificate di alcuni comandi. Sebbene, come detto, ciò che segue dovrebbe risultare sufficiente per la maggior parte delle applicazioni, ulteriori dettagli si possono trovare sul seguente testo, disponibile in biblioteca studenti, come pure su qualunque manuale del linguaggio C.

`A. Kelley, I. Pohl - A BOOK ON C - The Benjamin Cummings Publishing Co. Inc.`

5.2 Il comando per la compilazione

Il comando da usare per la compilazione è **cc**. Questo comando possiede un numero piuttosto grande di varianti che sono illustrate nei manuali o nelle “man pages” (vedi Par. 3.12); qui ci limiteremo alle opzioni fondamentali.

I comandi di compilazione usano l’ estensione dei files forniti come input per decidere che tipo di operazioni eseguire; in particolare:

Estensione	Contenuto del file	Operazione svolta dal compilatore
.c	Sorgente C	Pre-compilazione, compilazione e link
.o	Codice oggetto	Link

In generale quindi le operazioni di compilazione e link verranno svolte da un comando del tipo:

```
> cc -o <nome_eseguibile> <sorgente_1> ... <sorgente_n> <oggetto_1> ...
<oggetto_n>
```

L’ opzione `-o` specifica il nome del file di output, che nel caso precedente è un eseguibile. In alcuni casi è necessario aggiungere all’ eseguibile gruppi di routines prelevati da una libreria (le librerie sono di solito fornite con il sistema operativo o con prodotti software opzionali). In questo caso si utilizza l’ opzione `-l`, ed il comando generico diventa

```
> cc -o <nome_eseguibile> <sorgente_1> ... <oggetto_1> ... -l<nome_lib_1> ...
```

Per convenzione, i nomi delle librerie cominciano con `lib` e hanno estensione `.a` o `.so`. Il prefisso `lib` deve essere omesso nella linea di comandi. Le librerie CERN possono essere aggiunte semplicemente usando il comando `cernlib`

```
> cc -o nome_eseguibile sorgente_1 ... oggetto_1 ... `cernlib graflib`
```

È possibile eseguire solo la compilazione, senza effettuare il link, ottenendo così un codice oggetto che potrà essere utilizzato in un secondo tempo per fabbricare un eseguibile; per farlo si utilizza l’ opzione `-c`:

```
> cc -c -o <nome_oggetto> <sorgente_1> ... <sorgente_n>
```

Come detto in precedenza, i codici oggetto hanno usualmente estensione `.o`.

Un’ altra opzione importante è `-g` che deve essere specificata tutte le volte che si intende utilizzare un debugger simbolico (vedi Par. 3.9) per analizzare il comportamento del programma. Questa opzione dice al compilatore di aggiungere all’ eseguibile una tabella di corrispondenza tra i nomi delle variabili e le corrispondenti locazioni in memoria. In questo modo il debugger simbolico vi permetterà di ispezionare e modificare il contenuto di una variabile specificandone il nome. L’ opzione `-g` disabilita anche tutte le forme di ottimizzazione; questo fa sìche il programma si comporti esattamente come specificato nel sorgente: risulterà quindi più facile da debuggare, anche se sarà decisamente più lento in fase di esecuzione.

5.3 Formato del file sorgente

Il file sorgente è quello che contiene il codice che deve essere tradotto dal compilatore. Questo file deve essere scritto seguendo alcune regole che dipendono dal linguaggio.

Per il C le regole sono piuttosto blande: i comandi possono essere scritti in qualunque posizione, uno o più per riga; si deve porre un punto e virgola (;) alla fine di ciascun comando. I comandi sono usualmente composti da più elementi, ed uno o più di questi elementi possono essere a loro volta comandi completi; è il caso del comando `if`, che è fatto così :

```
if (espressione-logica) comando;
```

dove `comando` è un qualunque comando C (incluso un altro `if` che viene eseguito se l' espressione logica è vera. Gruppi di comandi possono essere realizzati utilizzando coppie di parentesi graffe ({}); sintatticamente un gruppo di comandi equivale ad un comando singolo; nel caso dell' `if` scriveremo:

```
if (espressione-logica) {
  comando-1;
  comando-2;
  ...
}
```

Da notare che se un blocco è posto alla fine di un comando si può omettere il punto e virgola finale.

Non esistono regole di ordinamento delle istruzioni, a parte il fatto che ogni simbolo deve essere definito prima di essere utilizzato. Le lettere maiuscole e quelle minuscole sono considerate diverse: questo è rilevante sia per i comandi, sia per i nomi delle variabili. È possibile inserire commenti nel programma racchiudendoli tra coppie /* - */. Ad esempio:

```
/* Questo e' un commento */
```

5.4 Precompilazione

L' operazione di precompilazione consiste in una trasformazione del codice sorgente che viene eseguita da un programma specializzato o dallo stesso compilatore prima che la fase di compilazione abbia inizio. Il principale scopo di questa operazione è quello di consentire al programmatore di risparmiare tempo e fatica evitando di scrivere per esteso blocchi identici che si ripetono più volte nel programma. Un modo di ottenere questo risultato consiste nell' utilizzo degli "include files". Si tratta di normali files che contengono i blocchi che devono essere ripetuti. Nel file sorgente vero e proprio si sostituirà a questi blocchi una riga che contiene il nome dell' include file:

```
#include "nome-file"
```

In generale gli include files si troveranno nella stessa directory in cui si trova il file sorgente. In caso contrario il nome del file dovrà contenere la specificazione della directory.

In alcuni casi si utilizzano degli include files che definiscono l' interfaccia tra il linguaggio ed il sistema operativo (per esempio le funzioni di input-output). Questi files hanno nomi standard e si trovano in particolari directories note ai compilatori (e di solito non note agli utenti). Per questi files esiste una sintassi speciale:

```
#include <nome-file>
```

Ad esempio:

```
#include <stdio.h>
```

Il precompilatore del C consente inoltre di definire simboli che possono essere usati all' interno del codice:

```
#define EPSILON 1.0e-5
...
if (x < EPSILON) {
...
```

Tali simboli possono anche servire per decidere se compilare o meno alcune parti del programma:

```
#define DEBUG /* Commentare questa linea se non si
                 vogliono stampe di debug */
...
#ifdef DEBUG
  printf("Errore nella routine xx\n");
#else
  /* Qui si deve fare qualcosa per gestire l' errore */
#endif
```

A seconda che il simbolo DEBUG sia definito o meno, il precompilatore deciderà quali linee inserire nel programma che verrà compilato. In questo modo, cambiando una sola linea del codice, si possono ottenere funzionalità diverse del programma.

Inoltre il comando **cc** consente di attivare un simbolo come DEBUG nell' esempio precedente (ovvero di fare l' operazione corrispondente a `#define DEBUG`) senza neppure modificare il sorgente, usando l' opzione **-D**:

```
> cc -DDEBUG -o test test.c
```

5.5 Struttura del programma

Ogni programma è formato da un insieme di routines (vedi paragrafo successivo); le routines possono a loro volta essere raggruppate in moduli. Un modulo contiene usualmente routines tra loro affini (per esempio che lavorano sugli stessi dati o che svolgono funzioni tra loro correlate); in generale le routines appartenenti ad un modulo condividono un set di variabili globali (vedi Par. 5.10).

In C le routines contenute in uno stesso file sorgente (ovvero in uno stesso file .c) formano un unico modulo. Tutte le variabili dichiarate in un file al di fuori di ogni routine sono considerate globali e possono essere usate da tutte le routines del modulo.

Gruppi di routines o moduli che compongono uno stesso programma non devono necessariamente essere compilati insieme: è possibile compilare ogni singolo file sorgente indipendentemente e poi unire il tutto in un unico eseguibile (vedi Par. 5.2).

Ogni programma completo deve contenere una ed una sola routine, detta "main program", dalla quale partirà l' esecuzione. In C il main program è una routine che si chiama `main`.

```
main() {
  ...
}
```

Il main program (al pari di qualunque altra function) può ricevere dei parametri; in questo caso i parametri vengono specificati nella riga di comandi che si usa per eseguire il programma (che, da un punto di vista logico, potrebbe essere pensata come la function che chiama `main`). Il main program può specificare come argomenti il numero dei parametri specificati nella linea di comando e un vettore di puntatori a `char` che contiene i vari parametri. Il seguente esempio dovrebbe chiarificare il meccanismo: consideriamo il programma `main_args.c`:

```
main(int argc, char **argv) {

  int i;

  for (i=0; i<argc; i++) {
    printf("argv[%d] = %s\n",i,argv[i]);
  }

}
```

lo compiliamo con il comando:

```
> cc -o main_args main_args.c
```

quindi lo eseguiamo con il comando

```
> main_args uno due tre quattro
```

ed otteniamo in output:

```
argv[0] = main_args
argv[1] = uno
argv[2] = due
argv[3] = tre
argv[4] = quattro
```

5.6 Functions

Una function è una porzione di codice che esegue una particolare funzione. Può essere chiamata più volte durante l' esecuzione del programma, dal main program o da altre functions; al termine della sequenza di istruzioni previste il programma continua dall' istruzione immediatamente successiva a quella che ha generato la chiamata. La struttura generale di una function è:

```
tipo nome(def_var1,...,def_varn) {

  /* Def variabili locali */
  ...
  /* Codice eseguibile */
  ...
  return;
  ...
}
```

L' istruzione **return** provoca il ritorno dell' esecuzione al programma chiamante.

Le variabili definite dentro una function sono locali, ovvero visibili solo all' interno della function medesima. Le variabili indicate tra parentesi dopo il nome sono dette parametri. Non sono vere variabili, ma solo nomi simbolici che possono essere usati all' interno della function (vengono anche dette variabili "dummy"); al momento della chiamata il chiamante specificherà quali variabili vere prenderanno il posto dei simboli specificati nella definizione della. A questo proposito bisogna notare che in C le functions ricevono una copia della variabile originale, per cui l' accesso ai parametri è solo in lettura: ogni modifica fatta a un parametro risulta visibile solo all' interno della function, ma non si riperquote sulla variabile originale del programma chiamante. Nel caso si desideri che una function modofichi il valore di una variabile del programma chiamante, si dovrà passare un puntatore a tale variabile (`&nome_var`): il puntatore verrà passato come copia, ma la copia punterà comunque direttamente alla variabile originale, consentendo quindi sia la lettura che la modifica.

Il C richiede una definizione completa dei parametri tra le parentesi che seguono il nome della routine; ad esempio:

```
void esempio(float x,int y) {
  ...
}
```

Per chiamare una subroutine si usa la sintassi

```
  float a;
  int b;
  ...
  esempio(a,b);
  ...
```

I parametri possono essere vettori; in questo caso non è necessario specificarne la lunghezza nella definizione data nella subroutine, in quanto la lunghezza sarà sempre quella del vettore specificato dalla routine chiamante. Per esempio:

```
void esempio(float a1[],int a2[][]) {
  ...
}
```

Si deve osservare che, visto che in C il nome di un vettore non seguito da parentesi indica il puntatore al vettore, si può passare un vettore ad una subroutine via puntatore semplicemente specificandone il nome.
Anche il nome della function deve essere preceduto da una definizione di tipo, in modo da indicare chiaramente il tipo del valore che viene ritornato al programma chiamate al termine dell' esecuzione. Nel caso non si voglia ritornare alcun parametro si specificherà il tipo `void`; altrimenti si potrà specificare un valore nell' istruzione `return`:

```
float cubo( float x ) {
  return x*x*x;
}
```

Il programma chiamante userà il valore ritornato dalla function in modo analogo al valore di una variabile:

```
extern float cubo();
float val,val_cube;
...
val = 10.0
val_cube = cubo(val);
...
```

La function deve essere definita nel programma chiamante, con l'opzione external, a meno che il programma chiamante non sia nello stesso modulo della function.

Una function emula il comportamento di una normale variabile: può essere usata all' interno di espressioni numeriche o logiche, e può anche essere passata come parametro ad un' altra routine; si deve però ricordare che ogni utilizzo di una function corrisponde alla attivazione di una porzione di codice che esegue un calcolo e, alla fine, restituisce un valore. Questa operazione è del tutto differente dall' utilizzo di una normale variabile, che richiede unicamente la lettura di un valore dalla memoria.

5.7 Interfacce esplicite

Nel caso in cui una routine ne chiami un altra appartenente allo stesso modulo, il compilatore conoscerà a priori tipo e dimensione di tutti i parametri della routine chiamata, e potrà quindi eseguire un test di correttezza dei parametri passati (in C questo è vero solo se la routine chiamata precede la chiamante all' interno del modulo).

Nel caso la routine chiamata non appartenga al modulo sarà necessario, al fine di rendere possibile un simile test, la definizione di una interfaccia esplicita secondo la sintassi seguente:

```
void  ext_sub(float x1, float x2[], int *i3);
float ext_fun(float x, float y);
```

È prassi comune tra i programmatori C di raccogliere le interfacce di tutte le functions di un modulo (ovvero di un file `module.c`) all' interno di un include file avente lo stesso nome (`module.h`). Tale include file è quindi inserito con un comando `#include` in tutti i moduli che devono accedere alle functions ivi definite.

5.8 Operatori

Gli operatori sono simboli che eseguono un operazione sulla o sulle variabili cui vengono applicati. Gli operatori che agiscono su una sola variabile si chiamano unari (u); un tipico operatore unario è il NOT. Gli operatori che agiscono su due variabili, come ad esempio l' operatore di somma, sono detti binari (b).

Gli operatori si possono poi dividere in relazionali (R), logici (L), numerici (N) e logici "bit a bit" o "bitwise" (B). Gli operatori relazionali, come ad esempio l' operatore di uguaglianza, sono usati nei tests e servono a formare espressioni che possono assumere il valore vero o falso; questo in C significa un valore intero che è zero per "falso" e diverso da zero per "vero". Gli operatori logici possono essere utilizzati insieme agli operatori relazionali per creare espressioni logiche complesse, ad esempio:

```
if ( a==b || c>=d ) {
  ...
}
```

Gli operatori numerici sono usati nel calcolo di espressioni aritmetiche, sia intere che reali.

Gli operatori bitwise agiscono sui singoli bit delle variabili intere; ad esempio l'operatore OR bitwise restituisce una variabile intera i cui bit sono ottenuti facendo l' or logico dei bit corrispondenti delle due variabili di ingresso.

La tabella 5.1 riassume i simboli usati per i principali operatori.

Nome	Tipo	Sintassi C
Assegnazione	b	=
Test uguaglianza	bR	==
Test non uguaglianza	bR	!=
Maggiore	bR	>
Minore	bR	<
Maggiore o uguale	bR	>=
Minore o uguale	bR	<=
AND logico	bL	&&
OR logico	bL	||
NOT logico	uL	!
Somma	bN	+
Sottrazione	bN	-
Moltiplicazione	bN	*
Divisione	bN	/
Cambio segno	uN	-
Elevamento a potenza	bN	pow(,)
Resto divisione intera	bN	%
Incremento	uN	++
Decremento	uN	--
Inclusive OR	bB	|
Exclusive OR	bB	^
AND	bB	&
NOT	uB	~
Shift destro	bB	>>
Shift sinistro	bB	<<

Tabella 5.1: I principali operatori in C

5.9 Costanti

Esistono fondamentalmente due tipi di costanti: le costanti numeriche e le costanti alfanumeriche. A loro volta le costanti numeriche possono essere intere o reali, e le costanti intere possono essere espresse in forma decimale, binaria o esadecimale. Di seguito vediamo alcuni esempi di costanti:

```
"Un Esempio" /* Alfanumerica */
123          /* Intera base 10 */
0xfa4        /* Intera base 16 */
1.4234       /* Reale          */
0.345e-4     /* Reale          */
```

5.10 Dichiarazione delle variabili

Una variabile è caratterizzata dal tipo, dalla dimensione, dal modo di allocazione e dalla visibilità o scope. Il tipo specifica la codifica con cui i dati vengono scritti in memoria; la dimensione indica di quanti elementi singoli di un dato tipo si compone una variabile: si possono fabbricare scalari, vettori o matrici a due o più dimensioni; il modo di allocazione indica come il compilatore riserva la memoria relativa alla variabile; lo scope indica quali parti di un programma possono accedere in lettura e/o scrittura alla variabile.
I tipi di variabili disponibili nel linguaggio C sono riportati in tabella 5.2.

```
char          /* intero con segno, 1 byte */
unsigned char /* intero positivo,  1 byte */
short int     /* intero con segno, 2 bytes */
int           /* intero con segno, 4 bytes */
float         /* variabile reale, 4 bytes */
double        /* Variabile reale, 8 bytes */
```

Tabella 5.2: Tipi predefiniti nel linguaggio C

Con i tipi predefiniti si possono costruire scalari, vettori oppure matrici; la forma generica dell' istruzione di definizione di una variabile è:

```
tipo nome[dim][...];
```

Ad esempio:

```
short int k,f[2][3];
float     r[4];
```

Si deve ricordare che in C la numerazione degli indici parte sempre da 0.
In generale, le variabili definite all' interno di una routine sono considerate locali e vengono allocate in modo automatico. Per ottenere una allocazione statica di una variabile locale si usa la sintassi

```
static int w;
```

Le variabili definite dentro a un modulo ma fuori da una routine sono considerate globali e sono visibili in tutte le routines del modulo che seguono la definizione. Le variabili globali sono allocate in modo statico. Possono essere inizializzate utilizzando la sintassi

```
int   npoints = 3;
float x[3] = { 1.0, 2.0, 3.0 };
```

Infine è possibile definire delle costanti usando la sintassi

```
const int      npoints = 4;
const float    pi = 3.14159;
```

È utile ricordare che in C le functions utilizzate e definite in un altro modulo devono necessariamente essere dichiarate. Esempio:

```
extern int fun();
```

5.11 Manipolazione di stringhe

Le stringhe sono variabili contenenti gruppi di caratteri alfanumerici. In in C esiste il tipo carattere, ed una stringa può essere costruita come un vettore di caratteri.

```
char stringa[21];
```

La stringa termina obbligatoriamente con un carattere NULL (il carattere con codice ASCII uguale a 0). La lunghezza specificata nella definizione è quindi la lunghezza massima (anzi, per essere precisi è $1 + L_{MAX}$).
La manipolazione delle stringhe avviene per mezzo di funzioni di libreria che richiedono, all' inizio del file sorgente, la definizione

```
#include <string.h>
```

Le stringhe possono essere inizializzate durante la definizione:

```
char stringa[21] = {"Valore Iniziale"};
```

Osservare come, essendo la costante usata nell' inizializzazione lunga 16 caratteri, soltanto 17 caratteri di `stringa` risulteranno effettivamente inizializzati (16 più il NULL finale) mentre i restanti 4 caratteri non verranno inizializzati.

L'assegnazione di una stringa viene eseguita con i seguenti comandi

```
#include <string.h>
...
char str1[26],str2[26];
...
strcpy(str1,'Nuovo Valore');
strcpy(str2,str1);
```

Sono inoltre possibili alcune operazioni sulle stringhe, che sono illustrate nell' esempio seguente:

```
#include <string.h>

char str1[26],str2[26];
char str3[50];
int ll;
...
/* Concatenazione */
strcpy(str3,strcat(str1,str2));
/* Estrazione caratteri */
strncpy(str2,str3+9,10);
/* Lunghezza */
ll = strlen(str2);
/* Test di uguaglianza */
if (strcmp(str1,str2)==0) {
  ...
}
/* Test di uguaglianza */
if (strncmp(str1+4,"Test",4)==0) {
  ...
}
```

5.12 Tipi composti

Un insieme di variabili di tipo e dimensione diversa può essere raggruppato in un tipo composto. Dopo la sua definizione il tipo composto potrà essere utilizzato, come un tipo predefinito, per definire variabili. La definizione di un tipo composto si esegue con le istruzioni

```
struct <nome> {
  <definizione elemento 1>
  ...
}; /* il ; e' obbligatorio */
```

dove la definizione degli elementi è una normale definizione di una variabile, che però non riserva

alcuno spazio in memoria, ma crea solo un paradigma che potrà essere usato successivamente per allocare effettivamente dello spazio. Ad esempio, possiamo definire un tipo `STUDENTE` e poi utilizzarlo per definire un vettore di 100 elementi:

```
struct studente {
  char nome[30];
  char cognome[30];
  int matricola;
  int data_di_nascita[3];
};  /* il ; e' obbligatorio */
...
struct studente st[100];
```

Avremo a questo punto a disposizione un vettore di 100 elementi, chiamato `st`, i cui elementi sono di tipo `studente`. Ogni singolo elemento del tipo composto può essere utilizzato come una variabile usuale, utilizzando la sintassi

```
st[3].matricola = 132456;
st[10].data_di_nascita[2] = 1972;
```

5.13 Puntatori ed allocazione dinamica della memoria

Un puntatore è una variabile il cui contenuto è l' indirizzo di un' altra variabile.

In C il puntatore è una variabile con un nome proprio, che viene usato solo per indicare il puntatore. La variabile puntata dal puntatore viene indicata con un nome differente, che è dato del nome del puntatore preceduto da un asterisco. Un puntatore C è specifico per un dato tipo (ad esempio `int` o `float`) ma può puntare sia scalari che vettori di quel tipo.

Ecco alcuni esempi di definizione di un puntatore, dove si deve osservare come la definizione di un puntatore ad uno scalare è identica alla definizione di un puntatore a vettore:

```
short int * s_point;
double    * v_point;
```

Vale la pena di osservare che non è possibile fabbricare puntatori a matrici; in realtà un oggetto del tipo

```
float  **matrix;
```

è un puntatore a oggetti di tipo "puntatore a float". `matrix` può quindi essere usato per fabbricare un vettore di puntatori, e ogni puntatore può a sua volta puntare a un vettore di float. Questo realizza una struttura che ha due indici, come una matrice, ma di fatto è più complessa di una matrice (ad esempio le righe, ovvero i vettori puntati dai vari puntatori, possono essere di lunghezza diversa).

L' assegnazione di un puntatore può avvenire tramite allocazione esplicita di una porzione di memoria; ad esempio le istruzioni

```
float *r_point;
...
r_point = malloc(100*sizeof(float));
```

chiedono al sistema di riservare (al momento dell' esecuzione) una porzione di memoria corrispondente a 100 reali (400 bytes) e di assegnare il puntatore `r_point` in modo che punti a quest' area. Si noti come è necessario passare alla funzione `malloc` il numero di bytes da allocare, il che può essere fatto sia direttamente sia usando la funzione `sizeof` che ritorna la lunghezza in byte di un dato tipo.

Un area allocata in questo modo (dinamicamente) deve essere liberata quando non serve più utilizzando le istruzioni

```
  free(r_point);
```

Un puntatore può poi essere assegnato eguagliandolo ad un altro puntatore; ad esempio:

```
  int *a, *b;
  ...
  a = b;
```

Dopo l' assegnazione, `a` e `b` puntano alla stessa locazione di memoria; di conseguenza possono essere usati in modo interscambiabile nel resto del programma.

Un terzo modo di assegnare un puntatore è quello di farlo puntare ad una variabile allocata staticamente. Per esempio:

```
  int *a;
  int b;
  ...
  a = &b;
```

In C il simbolo `&` seguito dal nome di una variabile indica l' indirizzo di quella variabile.

Un quarto modo di assegnare un puntatore è quello di fornire esplicitamente l' indirizzo in memoria della variabile puntata. Questa operazione è utile nel caso si debba manipolare un dispositivo mappato in memoria in una posizione fissa. Anche in questo caso il C esegue l' assegnazione del puntatore come quella di una normale variabile:

```
int *register_1;
...
register_1 = 0xff042000;
```

Come già detto, in C ogni riferimento alla variabile puntata viene fatto anteponendo un asterisco al nome del puntatore. Fanno eccezione vettori e matrici, nel senso che se `a` è un puntatore `a[3]` è la quarta componente del vettore puntato da `a`. Questo fatto è vero anche per i vettori allocati staticamente: se `b` è definito come in FORTRAN sia sufficiente specificare quante locazioni di un dato tipo (`REAL` nell' esempio) si desiderano, ed il computo dei bytes viene fatto automaticamente. In C invece

```
int b[100];
```

allora il simbolo `b` senza parentesi indica il puntatore al vettore, ovvero le due scritture

```
b e &(b[0])
```

sono equivalenti.

Vediamo ora alcuni esempi di uso dei puntatori:

```
int *a,*b;
int c;
...
b = malloc(10*sizeof(int));
a = &(b[4]);
...
b[0] = b[c] + c;
*a = b[2] + b[3];
...
fun(*a);
...
```

Si possono definire puntatori a tipi composti, come pure si possono fabbricare tipi composti che abbiano puntatori come elementi; rifacendoci all' esempio di Par. 5.12 avremo

```
struct studente *pt_s;
...
pt_s = malloc(sizeof(struct studente));
...
strcpy(pt_s->nome,"Luisa");
```

Si noti la sintassi particolare che, nel caso di un puntatore, sostituisce `nome.` con `nome->`.

5.14 Istruzioni condizionate e cicli

Le istruzioni condizionate sono blocchi di istruzioni che vengono eseguiti solo se si verificano particolari condizioni. Le condizioni che devono essere verificate sono date da espressioni logiche (vedi Par. 5.8). La sintassi è:

```
if (espressione_logica) {
  istruzioni
  ...
} else {
  istruzioni
  ...
}
```

I cicli sono gruppi di istruzioni che vengono ripetute più volte; esistono due tipi di cicli: quelli per cui si conosce a priori il numero di ripetizioni (a priori significa prima di cominciare la prima iterazione) e quelli per cui il numero di ripetizioni non è noto a priori. Vediamo le sintassi corrispondenti.

```
int i;
...
for(i=val_iniz; i<=valore_fin; i=i+passo) {
  istruzioni
  ...
}
```

```
do {
  istruzioni
  ...
} while (espressione_logica);

while (espressione_logica) {
  istruzioni
  ...
}
```

Il ciclo `do-while` si ripete fino a che l' espressione logica assume il valore "falso".

5.15 Input/Output su terminale

L' Input/Output (I/O) è l' operazione con la quale si trasferiscono i dati contenuti nella memoria su un terminale o un file e viceversa. Esistono due tipi di I/O: quello formattato e quello non formattato. Nell' I/O formattato un numero viene rappresentato in cifre; ad esempio un numero

reale può essere rappresentato come "0.00012945" (10 caratteri, cioè 10 bytes) o come ".13e-3" (6 bytes) malgrado il fatto che la rappresentazione di un reale in memoria sia sempre lunga 4 bytes. In questo caso naturalmente il programmatore deve decidere i dettagli del formato che vuole utilizzare (ad esempio quante cifre mostrare). Nell' I/O non formattato invece i dati vengono letti o scritti esattamente nello stesso formato usato in memoria (per esempio un reale viene sempre scritto in 4 bytes); ovviamente, nel caso non formattato i dati sono del tutto illeggibili, ma occupano meno spazio; inoltre ciò che viene scritto ha esattamente la stessa precisione di quanto contenuto in memoria, laddove l' output formattato ha usualmente una precisione inferiore. Nel seguito ci occuperemo esclusivamente di I/O formattato.

Il C fornisce istruzioni per leggere variabili dalla tastiera e scriverle sullo schermo del terminale (più precisamente queste istruzioni scrivono e leggono su due canali di comunicazione, detti STDIN ed STDOUT che sono usualmente assegnati alla tastiera e allo schermo). Il formato delle istruzioni è:

```
#include <stdio.h>
...
scanf("formato",&var_1,...,&var_n);

printf("formato",var_1,...,var_n);
```

Notare che è necessario includere il file `stdio.h` in ogni modulo che esegue operazioni di I/O; notare pure che la funzione `scanf` deve ricevere i puntatori alle variabili da leggere (`scanf` è una funzione che deve modificare il valore di alcuni ei suoi parametri).

La specificazione del formato è molto utile in fase di output, in quanto consente di presentare i dati in forma ordinata ed esteticamente piacevole. In input invece specificare un formato significa richiedere che i dati vengano immessi ESATTAMENTE nel modo specificato, e questo può risultare piuttosto scomodo. Per questo motivo in input si usano i cosiddetti "formati liberi", che consentono di scrivere i dati in un modo quasi arbitrario lasciando al computer il compito di fare del proprio meglio per interpretare correttamente quanto viene scritto in ingresso. In C è comunque necessario essere relativamente precisi, utilizzando `%d` per ogni numero intero, `%f` per ogni float, `%lf` per ogni double e `%s` per ogni parola (cioè sequenza di caratteri alfanumerici che non contiene spazi). Inoltre la sequenza di caratteri in input viene letta dopo che viene premuto il tasto "Enter"; è buona norma includere anche questo carattere alla fine di ogni formato di input (lo si fa aggiungendo un `%*c`) in modo che le operazioni successive non ricevano residui degli input precedenti.

Nel seguito vediamo alcuni esempi di utilizzo dei più comuni formati di output. Si deve notare che, oltre ai formati delle variabili da stampare, è possibile includere nella specifica del formato anche caratteri normali che vengono stampati senza alcuna interpretazione. Inoltre in C l' output di successive istruzioni `printf` avviene di seguito sulla stessa riga, a meno che il formato non termini con `\n`.

```
#include <stdio.h>
#include <string.h>
...
int    i1;
float  r1,r2;
char   c1[21];
...
i1 = 123;
r1 = 1.23456789;
r2 = 283.3344;
strcpy(c1,"Prova di I/O");
...
printf("i1 = %5d \n", i1);
printf("r1 = %5.3f \n", r1);
printf("r2 = %12.5e \n", r2);
printf("s1 = %s* \n", c1);
printf("%10.5f%10.5f \n", r1,r2);
```

Il progamma sopra scritto produce il seguente output:

```
i1 =   123
r1 = 1.235
r2 =  2.83330E+02
s1 = Prova di I/O*
  1.23457 283.33441
```

5.16 Input/Output su file

L' I/O su un file è del tutto analogo a quello su terminale. La sola differenza consiste nella necessità di definire il nome del file sul quale si desidera agire; a tale file verrà assegnato un identificatore di canale di comunicazione (di solito si tratta di un numero intero, che in C è considerato essere un puntatore ad un tipo di variabile detta `FILE`) che verrà utilizzato in tutte le operazione relative a quel file. Una volta terminate le operazioni di I/O si dovrà chiudere il canale (ed il file relativo).

Di seguito vediamo come si eseguono le operazioni di apertura e chiusura del file.

Le variabili indicate nell' esempio con i nomi `stat1` e `stat2` indicano codici di errore. Valori negativi indicano che qualcosa è andato storto.

```
#include <stdio.h>
...
FILE *f1,*f2;
int stat1,stat2;
...
f1 = fopen("nome_file_in","r");

f2 = fopen("nome_file_out","w");

...
stat1 = fscanf(f1,"formato", &var_1,...,&var_n);
stat2 = fprintf(f2,"formato", var_1,...,var_n);
...
fclose(f1);
fclose(f2);
```

Nel caso si debba leggere una linea generica di un file, che possa contenere spazi si utilizzeranno le istruzioni:

```
  char line[81];
  ...
  fgets(line,80,f1);
```

Osserviamo infine che è possibile, come variante della scrittura e lettura su file, scrivere su o leggere da una stringa di caratteri, come illustra l' esempio seguente:

```
#include <stdio.h>
...
char s1[21],s2[21];
...
sscanf(s1,"formato", &var_1,...,&var_n);
sprintf(s2,"formato", var_1,...,var_n);
```

Naturalmente è responsabilità del programmatore accertarsi che le stringhe siano lunghe abbastanza per contenere i dati prodotti dai formati di output.

Capitolo 6: Tecniche algoritmiche e numeriche di base

6.1 Introduzione

Illustreremo nel seguito alcune tecniche elementari di manipolazione dei dati e di calcolo numerico. Per ogni argomento tratteremo essenzialmente i soli concetti fondamentali e spesso ometteremo discussioni, anche rilevanti, su questioni di efficienza ed ottimizzazione degli algoritmi presentati. Lo scopo è, come al solito, quello di fornire le informazioni di base che possano consentirvi di risolvere semplici problemi e che possano servire da stimolo per uleriori approfondimenti.

6.2 I vettori

Il vettore è l' elemento chiave per la realizzazione di procedure indicizzate, nelle quali cioè ci si trovi a dover manipolare un insieme numerabile di oggetti simile che devono essere elaborati in sequenza o comunque organizzati in modo ordinato. Ad ogni oggetto si assegna un indice, ovvero un numero positivo che corrisponde alla posizione dell' oggetto all'interno del vettore.

6.2.1 Manipolazione dei vettori

La struttura sintattica di base per la manipolazione dei vettori è ovviamente il ciclo `for`; in effetti i vettori hanno lunghezza definita a priori (a livello di compilazione o, nel caso della allocazione dinamica, a livello di esecuzione del programma), e di conseguenza il ciclo `for` consente di manipolare sequenzialmente tutti gli elementi. Consideriamo l' esempio seguente, nel quale leggiamo una lista di nomi, la copiamo in un vettore e rileggiamo il conetnuto:

```
#include <stdio.h>
#include <string.h>

main() {

  int i,n_names;
  char **names;
  char buffer[100];

  /* Legge il numero di nomi e alloca il vettore */
  printf("Quanti sono i nomi ?  ");
  scanf("%d%*c",&n_names);
  names = (char **)malloc(n_names*sizeof(char*));

  /* Carica nel vettore i nomi */
  for (i=0; i<n_names; i++) {
    fgets(buffer, 100, stdin);
    /* Ogni elemento deve essere allocato con la lunghezza giusta */
    names[i] = (char *)malloc((strlen(buffer))*sizeof(char));
    strncpy(names[i], buffer, strlen(buffer)-1);
  }

  /* Stampa il contenuto del vettore */
  for (i=0; i<n_names; i++) {
```

```
    printf("%d = %s\n", i, names[i]);
  }

}
```

È chiaro da questo esempio come il vettore sia uno strumento molto andatto in tutte quelle cercostanze in cui il numero di oggetti da manipolare è noto all' inizio dell' elaborazione e resta costante.

Le matrici, avendo due indici, richiedono due cicli `for` uno dentro l' altro, come si vede nell' esempio seguente dove stampiamo il contenuto di un array a due indici di float:

```
float matrix[10][20];
int i,j;
...

for (i=0; i<10; i++) {
  for (j=0; j<20; j++) {
    printf("%f ", matrix[i][j]);
  }
  printf("\n");
}
```

È importante ricordare, specie nella manipolazione di array di grandi dimensioni (un megabyte o più), di rispettare con la sequenza degli indici la naturale disposizione in memoria degli elementi. In C questo significa che l' indice più a destra deve scorrere per primo, e quindi deve essere nel ciclo `for` più interno.

In alcuni casi è conveniente utilizzare un vettore per implementare un "buffer circolare"; si tratta di una struttura a n elementi nella quale vengono memorizzate informazioni. Nel momento in cui è necessario scrivere l' infromazione $n+1$ si cancella la prima informazione scritta e si mette la $n+1$-ima al suo posto.

Nell' esempio seguente vediamo un modulo che implementa un buffer circolare di 100 numeri interi. La routine `add` consente di aggiungere un numero al buffer, la routine `read` mostra gli ultimi numeri scritti (fino a un massimo di 100).

```
#define BUFLEN 100

int nel = 0; /* numero di elementi presenti nel buffer */
int pointer = 0; /* indice dell'elemento siccessivo */
int buffer[BUFLEN];

void add(int i) {
  buffer[pointer] = i;
  nel++; if (nel > BUFLEN) nel = BUFLEN;
  pointer++; if (pointer >= BUFLEN) pointer = 0;
}

void read() {
  int i;
  int ip = pointer;
```

```
  for (i=0; i<nel; i++) {
    ip--; if (ip <0) ip = BUFLEN - 1;
    printf("%d\n", buffer[ip]);
  }
}
```

I buffer circolari possono essere utilizzati per implementare strutture di tipo FIFO (First In First Out) o LIFO (Last In First Out) di dimensione fissa.
La ricerca in un vettore è spesso di tipo euristico: si leggono tutti gli elementi e si confronatano con il valore desiderato; quando si trova l' elemento giusto si può usare l' istruzione `break` per interrompre il ciclo senza scandire gli elementi rimanenti; nel modulo appena visto aggiungiamo ad esempio una function che ritorna la posizione nel buffer di un dato numero o -1 se il numero non è nel buffer:

```
int find(int val) {
  int i;
  int ip = pointer;
  int pos = -1;

  for (i=0; i<nel; i++) {
    ip--; if (ip <0) ip = BUFLEN - 1;
    if (buffer[ip] == val) {
      pos = ip;
      break;
    }
  }
  return pos;
}
```

Nel caso si abbia a che fare con vettori di grnde dimensione, la ricerca euristica risulta inefficiente. Si usa talvolta la ricerca "a chiave". Supponiamo ad esempio di aver caricato un elenco telefonico in un vettore (gli elementi del vettore saranno tipo composti da un nome e un numero di telefono), e di voler ricercare un dato nome. Potremmo fabbricare un vettore ausilirario che memorizzi, per ogni lettera dell' alfabeto, l' indice del primo nome nell' elenco che comincia per quella lettera. Invece di cercare euristicamente in tutto l' elenco potremmo con questa tecnica cercare soltanto in un sottoinsieme. Naturalmente il vettore ausilirario deve essere mantenuto aggiornato ogni volta che si aggiunge o si toglie un elemento dall' elenco.

6.2.2 Ordinamento degli elementi di un vettore

Ci sono molte tecnche per ordinare gli elementi di un vettore; la più intuitiva consiste nel cercare euristicamente nell' intero vettore l' elemento più piccolo, copiarlo nella prima locazione di un vettore ausiliario eliminandolo dal vettore originario e procedere coìfino ad avere esaurito gli elementi. Una tecnica un po' più raffinata, detta "bubble sort" consiste nell' analizzare tutti gli elementi, partendo dall' ultimo, scambiando ogni elemento $x(i)$ con il precedente $x(i-1)$ se $x(i) < x(i-1)$; in questo modo si porta l' elemento più piccolo nella posizione iniziale. La procedura va poi ripetuta nel sotto-vettore ottenuto ignorando la prima componente, ed iterata fino a quando non si esegue più alcuno scambio o si è rimasti cin un sotto-vettore di un elemento. Ecco un esempio di codifica in C del "bubble sort":

```
void sort(int *vect, int n) {
  int i,swap,top,temp;

  if (n < 2) return;

  top = 0;
  do {
    swap = 0;
    for (i=n-1; i>top; i--) {
      if (vect[i] < vect[i-1]) {
        swap = 1;
        temp = vect[i];
        vect[i] = vect[i-1];
        vect[i-1] = temp;
      }
    }
    top++;
  } while(swap != 0 && top < n-1);
}
```

6.2.3 Aggiunta di elementi a un vettore

Siccome un vettore consiste in una sequenza statica e di lunghezza definita di elementi, a rigore non è possibile aggiungere un elemento a un vettore. In pratica si usa spesso il trucco di definire in modo generoso le dimensioni dei vettori, usandone poi solo una parte; in questo caso è possibile aggiungere un elemento anche nel mezzo del vettore, a patto di spostare tutti gli elementi successivi di una posizione. Inoltre, se si usa l' alloazione dinamica, è possibile sostituire il vettore con uno più grande se manca lo spazio. Come esempio, vediamo una routine che aggiunge elementi ad un vettore allocato dinamicamente mantenendo costantemente l' ordinamento crescente degli elementi:

```
int *vect = NULL;
int n_el = 0, max_el = 0;

void increase_size(int siz) {
  int i,*temp;

  /* Crea un nuovo vettore */
  max_el = max_el + siz;
  temp = (int *)malloc(max_el*sizeof(int));

  /* Copia il vecchio vettore nel nuovo */
  for (i=0; i<n_el; i++) temp[i] = vect[i];

  /* Cancella il vecchio vettore e riassegna il puntatore */
  if (vect != NULL) free(vect);
  vect = temp;

}
```

```
int add_element(int val) {
   int i;

  /* Controlla se c'e' ancora spazio */
  if (n_el + 1 > max_el) increase_size(100);

  /* Aggiungi il nuovo elemento al suo posto */
  for (i=n_el++; i>0; i--) {
    if (vect[i-1]>val) {
      vect[i] = vect[i-1];
    } else {
      vect[i] = val;
      return i;
    }
  }
  vect[0] = val;
  return 0;
}
```

6.3 Tecniche Numeriche

Il calcolo numerico consiste nella soluzione di equazioni o nel calcolo di integrali ottenuta per sostituzione di particolari valori numerici. Si tratta ovviamente di un metodo molto meno elegante rispetto alla soluzione analitica del problema, ma che rischia di essere la sola possibile in molti casi in cui una soluzione analitica non è realizzabile. Malgrado l' apparente rozzezza, non si deve commettere l' errore di sottovalutare la complessità delle tecniche numeriche: un approccio troppo "spensierato" può condurre a risultati totalmente sbagliati che pure hanno una apparenza del tutto ragionevole.

Nel seguito illustreremo i rudimenti del calcolo numerico. I metodi che illustreremo hanno di fatto solo valenza didattica, e sono del tutto indadatti (per precisone, efficienza e dominio di applicabilità) alla soluzione di problemi "seri". Per maggiori approfondimenti si può ricorrere ad un qualunque testo di calcolo numerico; citiamo il seguente:

```
W. H. Press, S. A. Teukolsky, W. T. Wetterling, B. P. Flannering -
Numerical Recipes in C - Cambridge University Press
```

che ha il singolare vantaggio di essere disponibile on–line all' indirizzo `http:` `www.nr.com`.

6.3.1 Calcolo di integrali definiti

Il metodo intuitivo di calcolare un integrale definito di una funzione $f(x)$ non singolare, consiste nel dividere l' intervallo di integrazione in N sottointervalli uguali. Siano $x_0, x_1, ..., x_N$ i punti che delimitano i sottointervalli e h la lunghezza dei sottointervalli; potremo allora approssimare l' integrale nell' intervallo $[x_i, x_{i+1}]$ con l' area sotto la retta che passa per i punti $(x_i, f(x_i))$ e $(x_{i+1}, f(x_{i+1}))$. In formule scriveremo:

$$\int_{x_0}^{x_n} f(x)dx \simeq \sum_{i=0}^{N-1} \frac{h}{2}(f(x_i) + f(x_{i+1}))$$

È facile convincersi che questa formula è esatta per una retta; si può anche dimostrare che l' errore E, ovvero la differenza tra l' integrale e la somma, è proporzionale ad $h^3 f''(\bar{x})$ dove $\bar{x}$ è un punto non noto interno all' intervallo di integrazione (da cui si ritrova il risultato che per una retta, che ha derivata seconda sempre nulla, l' errore è zero).

Si può altresì dimostrare che, raggruppando due a due gli intervalli si ottiene una formula del tipo

$$\int_{x_i}^{x_{i+2}} f(x)dx = h\Big(\frac{f(x_i)}{3} + \frac{4f(x_{i+1})}{3} + \frac{f(x_{i+2})}{3}\Big) + O(h^5 f^{(4)}(\bar{x}))$$

dove l' errore è proporzionale ad h^5 moltiplicato per la derivata quarta di f in un punto interno all' intervallo (significa che è esatta per la polinomiali fino all' ordine x^3).

Le tecniche di integrazione presentate non possono ovviamente funzionare su intervalli illimitati e neppure nel caso che la unzione presenti singolarità all' interno dell' intervallo di integrazione.

6.3.2 Ricerca del minimo di una funzione

Per trovare il minimo (o il massimo) di una funzione continua e derivabile si sfrutta il fatto che in tale punto la derivata prima della funzione si annulla e cambia segno. partendo da un punto sufficientemente vicino al minimo ricercato (vuol dire che non ci devono essere altri minimi relativi nei paraggi) si calcola la derivata approssimandola con il rapporto incrementale su un intervallo finito di dimensione h:

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h}$$

Ci si sposta quindi di una quantità pari ad h e si continua fino a che il rapporto incrementale non cambia segno. A questo punto si dimezza il passo di campionamento e si ripete l' operazione nella direzione inversa. Ci si può fermare quando l' intervallo di campionamento scende al di sotto di una quantità minima definita come errore tollerato sulla posizione dell' estremo.

Questo metodo funziona in modo soddisfacente solo con funzioni regolari e prive di estremi relativi. Inoltre è di facile applicabilità solo con funzioni di una variabile. Nelle esercitazioni utilizzeremo delle routines delle librerie CERN per la determinazione dei minimi di funzione a molti parametri. Tali routines sono descritte al Par. 7.1.

6.3.3 Il metodo dei minimi quadrati

Il metodo dei minimi quadrati viene utilizzato per determinare quale tra le funzioni appartenenti ad una data famiglia descriva al meglio un insieme di punti sperimentali affetti (ovviamente) da errore. La situazione tipica è quella in cui si dispone di un set di misure y_i con $1 < i < N$ ciascuna affetta da un errore σ_i e ciascuna corrispondente ad una particolare scelta di variabili indipendenti $\vec{x}_i$ (assumeremo che non ci siano errori sugli $\vec{x}_i$, ipotesi di solito non verificata. Nel corso di ESP1 vedrete come trattare gli errori sulle variabili indipendenti). Si dispone poi di una teoria che prevede una relazione funzionale tra y e $\vec{x}$ espressa sotto forma di una famiglia di funzioni del tipo $y = f(\vec{x}, \vec{p})$, dove $\vec{p}$ rappresenta un insieme di parametri fisici che determinano

l' andamento delle funzioni f ma che non sono a priori note. Il problema è quello di determinare quale scelta di $\vec{p}$ fornisce il migliore accordo tra i dati sperimentali e la funzione teorica.

Il metodo dei minimi quadrati consiste nell' affermare che la migliore scelta di $\vec{p}$ è quella che minimizza la funzione $\chi^2(\vec{p})$ definita come

$$\chi^2(\vec{p}) = \sum_{i=0}^{N} \Big(\frac{(y_i - f(\vec{x}_i, \vec{p}))^2}{\sigma_i^2} \Big)$$

Nel caso la funzione f sia una retta, cioè

$$f(x, \vec{p}) = ax + b; \quad \vec{p} = (a, b)$$

il problema della minimizzazione del χ^2 ha una soluzione analitica che avete visto a ESP1. In tutti gli altri casi ci si deve accontentare di una soluzione numerica, che può essere ottenuta con i metodi illustrati nel paragrafo precedente.

Se la miglior funzione teorica si adatta in modo accettabile ai dati sperimentali, ovvero se il valore del minimo χ^2 è sufficientemente piccolo (imparerete nel corso del secondo semestre a dare un significato quantitativo, di tipo statistico, a queste affermazioni) il metodo dei minimi quadrati fornisce in sostanza una misura di $\vec{p}$ derivata dalla misura degli y_i. Questo significa che è necessario calcolare un errore sul valore di $\vec{p}$ ricavato: se ripetessimo la misura degli y_i otterremmo valori diversi (sebbene entro gli errori), e quindi una diversa funzione χ^2 da minimizzare ed un diverso valore di $\vec{p}$ che la minimizza. Quindi in qualche modo l' entità delle fluttuazioni degli y_i determinano una più o meno grande fluttuazione dei valori di $\vec{p}$. Si può dimostrare che l' incertezza su $\vec{p}$ è data dall' insieme dei punti tali che:

$$\chi^2(\vec{p}) - \chi^2(\vec{p}_{min}) < 1$$

dove $\vec{p}_{min}$ è il valore di $\vec{p}$ che minimizza la funzione χ^2. Pur senza dimostrare l' affermazione precedente, possiamo osservare che se abbiamo una sola misura y_0 ed un singolo parametro p potremo scegliere il valore della funzione teorica f in modo che sia identico a y_0, per cui il valore minimo del χ^2 sarà zero in corrispondenza di un dato p_0. Supponiamo ora di ripetere la misura e di trovare un valore $y_0 + \sigma_y$; il nuovo valore di p che annulla χ^2 sarà tale per cui

$$f(p_o + \Delta p) = y_0 + \sigma_i$$

Se calcoliamo la prima funzione χ^2 ottenuta in questo nuovo punto otteniamo:

$$\chi^2(p_0 + \Delta p) = \frac{(y_0 - f(p_0 + \Delta p))^2}{\sigma_y^2} =$$

$$\frac{(y_0 - y_0 - \sigma_y)^2}{\sigma_y^2} = 1$$

Cioè la variazione di p in corrispondenza della variazione di un sigma della misura y_0 corrisponde ad una variazione di χ^2 pari a 1.

Osserviamo che, nel caso di molti parametri, l' insieme dei $\vec{p}$ per cui χ^2 differisce dal minimo per meno di uno può avere una forma complicata, ed eventualmente potrebbe non trattarsi di una regione connessa. Nel caso la funzione χ^2 intorno al minimo sia assimilabile ad un paraboloide in M dimensioni (dove M è il numero di parametri ovvero la dimensione di $\vec{p}$) tale regione sarà un ellissoide.

Capitolo 7: Uso delle librerie CERN

È riportato nel seguito un estratto del manuale della libreria CERN chiamata MINUIT che serve alla realizzazione di analisi statistiche di dati sperimentali. Si tratta di un estratto molto parziale, limitato alle necessità di analisi relative al corso di Laboratorio di Calcolo; i manuali completi sono disponibili in laboratorio. Alcune routines sono state leggermente modificate con lo scopo di renderle più facili da usare.

7.1 Introduzione alle routines di Best-Fit

The fitting routine HMINUIT is based on the MINUIT package. MINUIT is conceived as a tool to find the minimum value of a multi-parameter function and analyze the shape of the function around the minimum. The principal application is foreseen for statistical analysis, working on chisquare functions, to compute the best-fit parameter values and uncertainties, including correlations between the parameters. It is especially suited to handle difficult problems, including those which may require guidance in order to find the correct solution.

The final fitted parameter values correspond to the minimum of the χ^2 function (defined by the user).

```
HMINUIT  ((*FCN)(int NP, double *XVAL), char *MODE, int NP,
         char *CPAR[], double PAR[], double STEP[], double PMIN[],
         double PMAX[], double OUTPAR[], double SIGPAR[], double *CHI2)
```

Action: Minimize the χ^2, defined by the user function FCN, and give the corresponding best estimate of the parameters and their erros.

Input parameters:

`FCN` (double) χ^2 function defined by the user:

```
double fcn(int np, double *xval){
  ...
  chi2 = ....
  return chi2;
}
```

`*MODE` Character specifying the desired options:

- `' '` Default minimization;
- `'M'` Invoke interactive MINUIT.

`NP` Number of parameters (dimension of `CPAR`, `PAR`, `STEP`, `PMIN`, `PMAX`, `OUTPAR` and `SIGPAR`).

`*CPAR[]` Array of characters specifying the name of the parameters.

`PAR[]` Array with initial values for the parameters.

`STEP[]` Array with initial step sizes for the parameters.

`PMIN[]` Array with the lower bounds for the parameters.

`PMAX[]` Array with the upper bounds for the parameters.

Output parameters:

`OUTPAR[]` Array with the final fitted values of the parameters.

SIGPAR[] Array with the standard deviations on the final fitted values of the parameters.

*CHI2 Chisquared of the fit.

Remarks:

- All the computations are performed in double precision to avoid rouding problems.
- If PMIN=PMAX=0.0 the parameters are left free to vary from $-\infty$ to $+\infty$.

7.1.1 Basic concepts of MINUIT

The MINUIT package acts on a multiparameter Fortran function to which one must give the generic name `FCN`. The value of `FCN` will in general depend on one or more variable parameters.

To take a simple example, suppose the problem is to fit a polynomial through a set of data points. Routine `FCN` called by `HMINUIT` calculates the chisquare between a polynomial and the data; the variable parameters of `FCN` would be the coefficients of the polynomials. Routine `HMINUIT` will request MINUIT to minimize `FCN` with respect to the parameters, that is, find those values of the coefficients which give the lowest value of chisquare.

Basic concepts - The transformation for parameters with limits.

For variable parameters with limits, MINUIT uses the following transformation:

$$P_{\mathrm{int}} = \arcsin\left(2\frac{P_{\mathrm{ext}} - a}{b - a} - 1\right) \qquad\qquad P_{\mathrm{ext}} = a + \frac{b-a}{2}\left(\sin P_{\mathrm{int}} + 1\right)$$

so that the internal value P_{int} can take on any value, while the external value P_{ext} can take on values only between the lower limit a and the upper limit b. Since the transformation is necessarily non-linear, it would transform a nice linear problem into a nasty non-linear one, which is the reason why limits should be avoided if not necessary. In addition, the transformation does require some computer time, so it slows down the computation a little bit, and more importantly, it introduces additional numerical inaccuracy into the problem in addition to what is introduced in the numerical calculation of the `FCN` value. The effects of non-linearity and numerical roundoff both become more important as the external value gets closer to one of the limits (expressed as the distance to nearest limit divided by distance between limits). The user must therefore be aware of the fact that, for example, if he puts limits of $(0, 10^{10})$ on a parameter, then the values 0.0 and 1.0 will be indistinguishable to the accuracy of most machines.

The transformation also affects the parameter error matrix, of course, so MINUIT does a transformation of the error matrix (and the "parabolic" parameter errors) when there are parameter limits. Users should however realize that the transformation is only a linear approximation, and that it cannot give a meaningful result if one or more parameters is very close to a limit, where $\partial P_{\mathrm{ext}}/\partial P_{\mathrm{int}} \approx 0$. Therefore, it is recommended that:

- Limits on variable parameters should be used only when needed in order to prevent the parameter from taking on unphysical values.
- When a satisfactory minimum has been found using limits, the limits should then be removed if possible, in order to perform or re-perform the error analysis without limits.

How to get the right answer from MINUIT.

MINUIT offers the user a choice of several minimization algorithms. The `MIGRAD` (Other algorithms are available with Interactive MINUIT, as described on Page 70) algorithm is in general the best minimizer for nearly all functions. It is a variable-metric method with inexact line search, a stable metric updating scheme, and checks for positive-definiteness. Its main weakness is that it depends heavily on knowledge of the first derivatives, and fails miserably if they are very inaccurate.

If parameter limits are needed, in spite of the side effects, then the user should be aware of the following techniques to alleviate problems caused by limits:

Getting the right minimum with limits.

If MIGRAD converges normally to a point where no parameter is near one of its limits, then the existence of limits has probably not prevented MINUIT from finding the right minimum. On the other hand, if one or more parameters is near its limit at the minimum, this may be because the true minimum is indeed at a limit, or it may be because the minimizer has become "blocked" at a limit. This may normally happen only if the parameter is so close to a limit (internal value at an odd multiple of $\pm\frac{\pi}{2}$ that MINUIT prints a warning to this effect when it prints the parameter values.

The minimizer can become blocked at a limit, because at a limit the derivative seen by the minimizer $\partial F/\partial P_{\mathrm{int}}$ is zero no matter what the real derivative $\partial F/\partial P_{\mathrm{ext}}$ is.

$$\frac{\partial F}{\partial P_{\mathrm{int}}} = \frac{\partial F}{\partial P_{\mathrm{ext}}}\frac{\partial P_{\mathrm{ext}}}{\partial P_{\mathrm{int}}} = \frac{\partial F}{\partial P_{\mathrm{ext}}} = 0$$

Getting the right parameter errors with limits.

In the best case, where the minimum is far from any limits, MINUIT will correctly transform the error matrix, and the parameter errors it reports should be accurate and very close to those you would have got without limits. In other cases (which should be more common, since otherwise you wouldn't need limits), the very meaning of parameter errors becomes problematic. Mathematically, since the limit is an absolute constraint on the parameter, a parameter at its limit has no error, at least in one direction. The error matrix, which can assign only symmetric errors, then becomes essentially meaningless.

Interpretation of Parameter Errors:

There are two kinds of problems that can arise: the **reliability** of MINUIT's error estimates, and their **statistical interpretation**, assuming they are accurate.

Statistical interpretation:

For discussuion of basic concepts, such as the meaning of the elements of the error matrix, or setting of exact confidence levels.

Reliability of MINUIT error estimates.

MINUIT always carries around its own current estimates of the parameter errors, which it will print out on request, no matter how accurate they are at any given point in the execution. For example, at initialization, these estimates are just the starting step sizes as specified by the user. After a `MIGRAD` or `HESSE` step, the errors are usually quite accurate, unless there has been a problem. MINUIT, when it prints out error values, also gives some indication of how reliable it thinks they are. For example, those marked `CURRENT GUESS ERROR` are only working values not to be believed, and `APPROXIMATE ERROR` means that they have been calculated but there is reason to believe that they may not be accurate.

If no mitigating adjective is given, then at least MINUIT believes the errors are accurate, although there is always a small chance that MINUIT has been fooled. Some visible signs that MINUIT may have been fooled are:

- Warning messages produced during the minimization or error analysis.
- Failure to find new minimum.

- Value of `EDM` too big (estimated Distance to Minimum).
- Correlation coefficients exactly equal to zero, unless some parameters are known to be uncorrelated with the others.
- Correlation coefficients very close to one (greater than 0.99). This indicates both an exceptionally difficult problem, and one which has been badly parameterized so that individual errors are not very meaningful because they are so highly correlated.
- Parameter at limit. This condition, signalled by a MINUIT warning message, may make both the function minimum and parameter errors unreliable. See the discussion above "*Getting the right parameter errors with limits*".

The best way to be absolutely sure of the errors, is to use "independent" calculations and compare them, or compare the calculated errors with a picture of the function. Theoretically, the covariance matrix for a "physical" function must be positive-definite at the minimum, although it may not be so for all points far away from the minimum, even for a well-determined physical problem. Therefore, if MIGRAD reports that it has found a non-positive-definite covariance matrix, this may be a sign of one or more of the following:

A non-physical region: On its way to the minimum, MIGRAD may have traversed a region which has unphysical behaviour, which is of course not a serious problem as long as it recovers and leaves such a region.

An underdetermined problem: If the matrix is not positive-definite even at the minimum, this may mean that the solution is not well-defined, for example that there are more unknowns than there are data points, or that the parameterization of the fit contains a linear dependence. If this is the case, then MINUIT (or any other program) cannot solve your problem uniquely, and the error matrix will necessarily be largely meaningless, so the user must remove the underdeterminedness by reformulating the parameterization. MINUIT cannot do this itself.

Numerical inaccuracies: It is possible that the apparent lack of positive-definiteness is in fact only due to excessive roundoff errors in numerical calculations in the user function or not enough precision. This is unlikely in general, but becomes more likely if the number of free parameters is very large, or if the parameters are badly scaled (not all of the same order of magnitude), and correlations are also large. In any case, whether the non-positive-definiteness is real or only numerical is largely irrelevant, since in both cases the error matrix will be unreliable and the minimum suspicious.

An ill-posed problem: For questions of parameter dependence, see the discussion above on positive-definiteness.

Possible other mathematical problems are the following:

Excessive numerical roundoff: Be especially careful of exponential and factorial functions which get big very quickly and lose accuracy.

Starting too far from the solution: The function may have unphysical local minima, especially at infinity in some variables.

MINUIT interactive mode

The routine `HMINUIT`, with the `M` option, moves to interactive mode and gives control to the MINUIT program. In this case, the user may enter MINUIT control statements directly.

Overview of available MINUIT commands

CLEar

Resets all parameter names and values to undefined. Must normally be followed by a PARAMETER command or equivalent, in order to define parameter values.

CONtour par1 par2 [devs] [ngrid]

Instructs MINUIT to trace contour lines of the user function with respect to the two parameters whose external numbers are **par1** and **par2**. Other variable parameters of the function, if any, will have their values fixed at the current values during the contour tracing. The optional parameter **[devs]** (default value 2.) gives the number of standard deviations in each parameter which should lie entirely within the plotting area. Optional parameter **[ngrid]** (default value 25 unless page size is too small) determines the resolution of the plot, i.e. the number of rows and columns of the grid at which the function will be evaluated.

EXIT

End of Interactive MINUIT. Control is returned to the calling routine.

FIX parno

Causes parameter **parno** to be removed from the list of variable parameters, and its value will remain constant (at the current value) during subsequent minimizations, etc., until another command changes its value or its status.

HELP [SET] [SHOw]

Causes MINUIT to list the available commands. The list of SET and SHOw commands must be requested separately.

HESse [maxcalls]

Instructs MINUIT to calculate, by finite differences, the Hessian or error matrix. That is, it calculates the full matrix of second derivatives of the function with respect to the currently variable parameters, and inverts it, printing out the resulting error matrix. The optional argument **[maxcalls]** specifies the (approximate) maximum number of function calls after which the calculation will be stopped.

IMProve [maxcalls]

If a previous minimization has converged, and the current values of the parameters therefore correspond to a local minimum of the function, this command requests a search for additional distinct local minima. The optional argument **[maxcalls]** specifies the (approximate) maximum number of function calls after which the calculation will be stopped.

MIGrad [maxcalls] [tolerance]

Causes minimization of the function by the method of Migrad, the most efficient and complete single method, recommended for general functions (see also MINImize). The minimization produces as a by-product the error matrix of the parameters, which is usually reliable unless warning messages are produced. The optional argument **[maxcalls]** specifies the (approximate) maximum number of function calls after which the calculation will be stopped even if it has not yet converged. The optional argument **[tolerance]** specifies required tolerance on the function value at the minimum. The default tolerance is `0.1`. Minimization will stop when the estimated vertical distance to the minimum (`EDM`) is less than `0.001*[tolerance]*UP` (see `SET ERR`).

MINImize [maxcalls] [tolerance]

Causes minimization of the function by the method of Migrad, as does the MIGrad command, but switches to the SIMplex method if Migrad fails to converge. Arguments are as for MIGrad.

MINOs [maxcalls] [parno] [parno]...

Causes a Minos error analysis to be performed on the parameters whose numbers **[parno]** are specified. If none are specified, Minos errors are calculated for all variable parameters. Minos errors may be expensive to calculate, but are very reliable since they take account of non-linearities in the problem as well as parameter correlations, and are in general asymmetric. The optional argument **[maxcalls]** specifies the (approximate) maximum number of function calls *per parameter requested,* after which the calculation will be stopped for that parameter.

RELease parno

If **parno** is the number of a previously variable parameter which has been fixed by a command: **FIX parno**, then that parameter will return to variable status. Otherwise a warning message is printed and the command is ignored. Note that this command operates only on parameters which were at one time variable and have been FIXed. It cannot make constant parameters variable; that must be done by redefining the parameter with a PARAMETER command.

REStore [code]

If no **[code]** is specified, this command restores all previously FIXed parameters to variable status. If **[code]=1**, then only the last parameter FIXed is restored to variable status.

SCAn [parno] [numpts] [from] [to]

Scans the value of the user function by varying parameter number **[parno]**, leaving all other parameters fixed at the current value. If **[parno]** is not specified, all variable parameters are scanned in sequence. The number of points **[numpts]** in the scan is 40 by default, and cannot exceed 100. The range of the scan is by default 2 standard deviations on each side of the current best value, but can be specified as from **[from]** to **[to]**. After each scan, if a new minimum is found, the best parameter values are retained as start values for future scans or minimizations. The curve resulting from each scan is plotted on the output unit in order to show the approximate behaviour of the function. This command is not intended for minimization, but is sometimes useful for debugging the user function or finding a reasonable starting point.

SEEk [maxcalls] [devs]

Causes a Monte Carlo minimization of the function, by choosing random values of the variable parameters, chosen uniformly over a hypercube centered at the current best value. The region size is by default 3 standard deviations on each side, but can be changed by specifying the value of **[devs]**.

SET ERRordef up

Sets the value of **up** (default value= 1.), defining parameter errors. MINUIT defines parameter errors as the change in parameter value required to change the function value by **up**. Normally, for chisquared fits **up=1**, and for negative log likelihood, **up=0.5**.

SET LIMits [parno] [lolim] [uplim]

Allows the user to change the limits on one or all parameters. If no arguments are specified, all limits are removed from all parameters. If **[parno]** alone is specified, limits are removed from parameter **[parno]**. If all arguments are specified, then parameter **[parno]** will be bounded between **[lolim]** and **[uplim]**. Limits can be specified in either order, MINUIT will take the smaller as **[lolim]** and the larger as **[uplim]**. However, if **[lolim]** is equal to **[uplim]**, an error condition results.

SET PARameter parno value

Sets the value of parameter **parno** to **value**. The parameter in question may be variable, fixed, or constant, but must be defined.

SET PRIntout level

Sets the print level, determining how much output MINUIT will produce. The allowed values and their meanings are displayed after a **SHOw PRInt** command. Possible values for **level** are:

-1 No output except from SHOW commands

0 Minimum output (no starting values or intermediate results)

1 Default value, normal output

2 Additional output giving intermediate results.

3 Maximum output, showing progress of minimizations.

SET STRategy level

Sets the strategy to be used in calculating first and second derivatives and in certain minimization methods. In general, low values of **level** mean fewer function calls and high values mean more reliable minimization. Currently allowed values are 0, 1 (default), and 2.

SHOw XXXX

All **SET XXXX** commands have a corresponding **SHOw XXXX** command. In addition, the SHOw commands listed starting here have no corresponding SET command for obvious reasons. The full list of SHOw commands is printed in response to the command **HELP SHOw**.

SHOw CORrelations

Calculates and prints the parameter correlations from the error matrix.

SHOw COVariance

Prints the (external) covariance (error) matrix.

SIMplex [maxcalls] [tolerance]

Performs a function minimization using the simplex method of Nelder and Mead. Minimization terminates either when the function has been called (approximately) **[maxcalls]** times, or when the estimated vertical distance to minimum (`EDM`) is less than **[tolerance]**. The default value of **[tolerance]** is `0.1*UP` (see `SET ERR`).