LINUX

Generalità

Perché Linux? 1
Linux & C 4
Unix su Macintosh 4

Comandi della shell

ls, man e type 1
Semplici comandi Unix 2
Gli script di shella 2
Altri comandi della shell 3
gzip e tar 3
split 3
find e slocate 3
mtools 3
Tasti speciali della shell 4
La scelta della password 4
Come spegnere 7
Redirezione dell'output 8
Redirezione dell'input 8
Redirezione degli errori 8
Pipelines della shell 8
Link e link simbolici 9
shutdown e poweroff 9
Diritti d'accesso 10
chown e chgrp 10
chmod 10
tee 11
sort 11
Un errore pericoloso 11
Espressioni regolari 11
grep 11
I files .profile e .bash_rc 12
Il file .bash_logout 12
Il file .plan 12

Emacs

Emacs 5
Comandi fondamentali di Emacs 5
Richard Stallman 5
ange ftp 5
I tasti di Emacs 6
L'aiuto di Emacs 6
Comandi Emacs (schema) 6
Il file .emacs 7
Le funzioni di Elisp 7
Scrivere programmi con Emacs 7
Le directory di Emacs 7

Internet

telnet 5
ftp 7
Salvare il lavoro 12

PROGRAMMAZIONE IN PERL

Generalità

Programmare in Perl 13
Variabili nel Perl 13
Input dalla tastiera 13
Files e operatore <> 15
Funzioni del Perl 15
Moduli 15
Il modulo files 15
Vero e falso 16
Operatori logici del Perl 16
Operatori di confronto 16
Istruzioni di controllo 16
Nascondere le variabili con my 17
I puntatori del Perl 25
I moduli CPAN 25
Puntatori a variabili locali 26
Passaggio di parametri 26
Numeri casuali e rand 30
Typeglobs 30
eval 31
Problemi di sicurezza con eval 31
Attributi di files e cartelle 31
Cercare gli errori 32
Cartelle e diritti d'accesso 32

Stringhe, liste e vettori associativi

Liste 14
La funzione grep del Perl 14
Alcuni operatori per le liste 14
Contesto scalare e contesto listale 14
map 17
Vettori associativi 17
index e rindex 24
Liste di liste e matrici 25
Concatenazione di stringhe 25
substr 26
Strumenti per le stringhe 26
lc e uc 26
Invertiparola e eliminacaratteri 26
Ordinare una lista con sort 30
printf e sprintf 30
Puntatori a vettori associativi 32
Vettori associativi anonimi 32

Espressioni regolari

Espressioni regolari nel Perl 21
Gli operatori *m* ed *s* 21
I modificatori */m* ed */s* 21
I metacaratteri 22
I metasimboli 22
Il modificatore */g* 22
I modificatori */i* ed */o* 22
La funzione pos 22
Il modificatore */e* 23
Riassunto dei modificatori 23
$_ sottinteso nelle espressioni regolari 23
Alternative a /.../ 23
Parentesi tonde 23
split e join 23
Uso di | nelle espressioni regolari 24
Ricerca massimale e minimale 24

Programmi elementari

I numeri di Fibonacci 18
Il sistema di primo ordine 18
La moltiplicazione russa 19
Trovare la rappresentazione binaria 19
La potenza russa 19
Lo schema di Horner ricorsivo 19
Lo schema di Horner classico 20
Zeri di una funzione continua 20

Programmazione funzionale

Funzioni anonime 33
Funzioni come argomenti di funzioni 33
Il λ-calcolo 33
Funzioni come valori di funzioni 33
Programmazione funzionale 33

Programmazione logica

Uso procedurale degli operatori logici 32
Programmazione logica 34
Un problema di colorazione 34
Prolog e Perl 34
Il prodotto cartesiano di liste 35
Esercizi sull'uso di and/or 35
Liste anonime e basi di dati 35
Lettura dei dati da un file 35

Intelligenza artificiale

ELIZA 27
Struttura dei files tematici 27
Risposte casuali 27
Una conversazione con ELIZA 28
Il file Eliza/alfa 28
Il file Eliza/saluti.pm 28
Il file Eliza/dialogo.pm 29
Il file Eliza/aus.pm 29

BIOINFORMATICA

Fattori di una parola 24
Lettura a triple del DNA 24

GRAFICA

Programmi di grafica

gqview 12

Programmazione in PostScript

Forth e PostScript 36
Lo stack 36
Programmare in Forth 36
Usare ghostscript 37
Il comando run di PostScript 37
Usare def 37
Diagrammi di flusso per lo stack 37
Lo stack dei dizionari 37
Argomenti di una macro 38
show e selectfont 38
if e ifelse 38
Cerchi con PostScript 38
Disegnare con PostScript 40

Curve di Bézier

L'algoritmo di Casteljau 39
Curve di Bézier cubiche 40
Invarianza affine 40
I punti di controllo 40
La parabola 41
Il cerchio 41
L'ellisse 41
L'iperbole 42
Alcuni esempi di curve di Bézier 42
Il file bezier.pm 42

LIBRI

Perl 30

VARIA

Obiettivi del corso 1
Tirocini all'ARDSU 16
Numeri esadecimali 18
Linux Day al dipartimento di matematica 29

PROGRAMMAZIONE IN C

Generalità

Programmare in C 43
I header generali 43
Il preprocessore 44
I commenti 44
Passaggio di parametri 48
Parametri di main 50
Le funzioni matematiche del C 78
atan 2 78
Funzioni per determinare il tipo di un carattere 78

Il makefile

Comandi di compilazione 44
Il makefile 47
Il comando make 47
I blocchi elementari 47

Programmi elementari

Il programma minimo 43
Calcoliamo il fattoriale 44
Operatori abbreviati 45
Confronto di stringhe 46
if ... else 46
Operazioni aritmetiche 48
Input da tastiera 48
scanf 48
Altre funzioni per le stringhe 49
for 49
I numeri binomiali 49
Operatori logici del C 50
Un semplice menu 50
Funzioni per i files 51
fseek e ftell 51
system 51
I numeri di Fibonacci in C 52
Il metodo del sistema di primo ordine 52
sprintf 53
Copiare una stringa 53
Una funzione di cronometraggio 58
switch 60
Punto interrogativo e virgola 61
L'algoritmo binario per il m.c.d. 61
Funzioni con un numero variabike di argomenti 61
A→b come abbreviazione di (*A).b 70
%* in printf 70
Un piccolo filtro 81

Tipi di dati

Vettori e puntatori 45
Stringhe e caratteri 45
Aritmetica dei puntatori 45
Puntatori generici 46
Conversione di tipo 46
Tipi di interi 49
static e extern 50
struct e typedef 52
Variabili di tipo static 52
Allocazione di memoria in C 53
Il tipo enum 60

Algoritmi di ordinamento

Il quicksort 56
La mediana 56
Versione generale di quicksort 57
Il counting sort 57

Numeri casuali

Numeri casuali 58
Numeri casuali in crittografia 58
La discrepanza 59
Integrali multidimensionali 59
Il generatore lineare 59
La struttura reticolare 60
Numeri casuali in C 60

Funzioni per le stringhe

Quindici comuni: la trasformazione dei dati 64
Le funzioni del C per le stringhe 64
strlen, strcat e strncat 64
strcpy e strncpy 65
strcmp e strncmp 65
strstr 65
strchr e strrchr 65
strspn, strcspn e strpbrk 65
strtok 70
Operazioni sui bytes in memoria 81

Programmazione di sistema

Processi 75
Il fork 75
Il PID 76
exit e wait 76
Esecuzione in background 76
I comandi exec 76
Esempi di comandi exec 77
fork e exec 77
environ e getenv 77
Terminare un processo 77
Pipelines 79
read e write 79
pipe, close e dup2 79
pipemail 80
pipemaills 80
Sull'uso delle pipelines 80
Trasferimento in entrambe le direzioni 81

PROGRAMMAZIONE IN C++

Programmazione in C++ 71
Classi 71
Costruttori e distruttori 71
this 72
Overloading di funzioni 72
Classi derivate 72
Overloading di operatori 72
Unioni 72
Numeri complessi 73
Riferimenti 74
new e delete 74

SICUREZZA DEI DATI

La funzione crypt 74
Sniffing e spoofing 74
Sicurezza dei dati 82
Locali e apparecchiature 82
La shell protetta SSH 82
Sicurezza in rete 83
Autenticazione 83
Sicurezza dei dati in medicina 84
I principi di Anderson 84
Il teorema di Fermat-Euler 85
Crittografia a chiave pubblica 85
La steganografia 85

OTTIMIZZAZIONE GENETICA

Generalità

Ottimizzazione genetica 54
Problemi di ottimizzazione 54
L'algoritmo di base 55
Il metodo spartano 55
Un'osservazione importante 55
Sul significato degli incroci 55
Confronto con i metodi classici 55
Incroci tra più di due individui 68

Cluster analysis

Cluster analysis 62
Il criterio della varianza 62
Il numero delle partizioni 63
Quindici comuni: i dati grezzi 63
Quindici comuni: l'algoritmo genetico 66
Quindici comuni: l'interfaccia utente 66
Il calcolo di $\sum \Delta\alpha$ 66
Partizioni proposte dall'utente 67
Dichiarazioni in cluster.c 67
Organizzazione dei dati sul file e lettura 67
Visualizzazione della partizione migliore 67
Elementi nuovi, mutazioni e incroci 68
La costruzione della graduatoria 68
cluster.c 70

Esempi vari

Il problema degli orari 54
Un problema di colorazione 69

LIBRI

Algoritmi genetici 65
C++ 74

VARIA

Il codice ASCII 57

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica Anno accademico 2001/2002 Numero 1 ◇ 25 Settembre 2001

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 origini 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

Comandi Unix. Uso dell'editor programmabile Emacs.

Programmazione in Perl (il miglior linguaggio di programmazione attualmente disponibile), C (gestione di progetti con make, programmazione classica, librerie standard) e C++ (programmazione orientata agli oggetti). Interazione tra C e Unix (processi, programmazione di sistema, pipelines).

Algoritmi, strutture dei dati e un po' di bioinformatica.

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
gzip e tar
split
find e slocate
mtools

4 Tasti speciali della shell
Linux & C
La scelta della password
Unix su Macintosh

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**. Eseguendo 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 2001
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 molto attenti perché 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?

Simile a **split** è **csplit**.

---

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 di 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.

Tasti speciali della shell

Alcuni tasti speciali della shell: **^c** per interrompere un programma (talvolta funziona anche **^d** oppure ^\), **^u** per cancellare la riga di comando, **^k** per cancellare la parte della riga di comando alla destra del cursore, **^a** per tornare all'inizio e **^e** per andare alla fine della riga di comando. Il capuccio ^ indica qui il tasto *Ctrl*, quindi **^a** significa che bisogna battere *a* tenendo premuto il tasto *Ctrl*. I tasti **^a**, **^e** e **^k** 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 **^b** e **^f**, permettono il movimento indietro e avanti nella riga di comando, per il tasto **^d** che cancella il carattere su cui si trova il cursore, e il tasto **^h** 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 **^p** e **^n**.

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.

Battendo due volte il tasto tabulatore, si può visualizzare un elenco di tutti i comandi disponibili. Sono migliaia! In questa come in altre occasioni, quando lo schermo è troppo piccolo per contenere tutto l'output, si può usare la combinazione di tasti *Shift Pag*↑ per ripercorrerlo all'indietro.

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 e in Perl) 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**).

Linux & C

Nell'armadietto dei libri in aula 9 trovate *Linux & C*, un'ottima rivista tutta dedicata a Linux. Ne fanno parte anche i *quaderni di informatica*, piccole guide tematiche (installazione di Linux, configurazioni, compilazione del kernel, StarOffice, PHP). Spesso alla rivista sono allegati dei CD che contengono raccolte di programmi o le ultime distribuzioni di Linux (RedHat, Slackware, Mandrake). Il numero 18 contiene una storia di Linux.

Particolare attenzione è 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), alle interfacce grafiche, 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 sfogliarli per avere un'idea sui molti aspetti e sulle novità del nostro sistema operativo.

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.

Per cambiare la propria password si usa il comando **passwd**; l'amministratore di sistema può usare **password rossi** per cambiare la password dell'utente *rossi*.

Unix su Macintosh

Recentemente la Apple ha introdotto il nuovo sistema operativo *MacOS X* per Macintosh, un ibrido tra Unix e il classico sistema operativo grafico dei Macintosh. Dovrebbe essere possibile utilizzare molti programmi scritti per Linux anche con questo sistema operativo.

L'installazione e l'uso di MacOS X sono molto semplificati dalle capacità di autoconfigurazione del sistema e finalmente esiste la possibilità di utilizzare Emacs anche su Macintosh!

Una presentazione del MacOS X si trova nel numero 18 (ottobre 2001) di *Linux & C*.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica Anno accademico 2001/2002 Numero 2 ◇ 2 Ottobre 2001

Emacs

Emacs è un programma di scrittura 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.

Comandi fondamentali di Emacs

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.

Nell'impostazione di default invece di **f4** si usa **^xc**, e **^xs** invece di **f8**.

Comandi di movimento: **^a** per andare all'inizio della riga, **^e** per andare alla fine della riga, **^f** per andare avanti di un carattere, **^b** per andare indietro di un carattere, **^p** per andare una riga in su, **^n** per andare una riga in giù, **^v** per andare in giù di circa 3/4 di pagina, **^z** per andare in su di circa 3/4 di pagina, **^cv** per salvare il buffer e passare a un eventuale secondo buffer nella stessa finestra.

Funzioni di ricerca: **^s** ricerca in avanti, **^r** ricerca all'indietro. Molto comode e potenti.

Altri comandi: **^o** per aprire una riga (si usa spesso), **^k** per cancellare il resto della riga, **^y** per incollare, **^tu** per annullare l'effetto di un comando precedente, *TAB* o **^xf** per aprire un file.

I tre comandi più importanti: **^g** per uscire da un comando e, come abbiamo già visto, **^xc** o **f4** per uscire, **^xs** o **f8** per salvare il file.

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

L'inventore di Emacs è Richard Stallman, iniziatore della *GNU* e della *Free Software Foundation*.

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

Questa settimana

5 Emacs
Comandi fondamentali di Emacs
Richard Stallman
ange ftp
telnet

6 I tasti di Emacs
L'aiuto di Emacs
Comandi Emacs

7 Il file .emacs
Le funzioni di Elisp
Scrivere programmi con Emacs
Le directory di Emacs
ftp
Come spegnere

8 Redirezione dell'output
Redirezione dell'input
Redirezione dei messaggi d'errore
Pipelines della shell

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 remoto) 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.

***

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.

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 una 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.

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* (↖) sono spiegati nella pagina successiva.

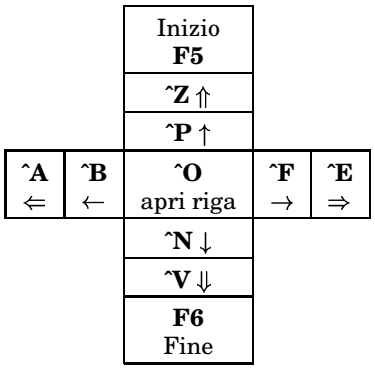
***

L'aiuto di Emacs

Emacs fornisce moltissime funzioni di aiuto, tra cui quelle utilizzate più frequentemente sono *describe-function* (**^tw**), *apropos-variable* (**^tx**), *describe-key* (**^th k**), *apropos-command* (**^th a**) e *describe-variable* (**^tv**).

Una documentazione online si trova nel World Wide Web sotto *http://www.geek-girl.com/emacs/emacs.html*. Per approfondire di più si può consultare il libro di Cameron/Rosenblatt nell'armadietto.

Comandi Emacs



elenco buffer aperti	**F1**	incolla	**^Y**
uscire	**F4** (**^XC**)	cancella carattere	**^D**
salvare	**F8** (**^XS**)	cancella ←	**^H**
comando	**F9**	cancella resto riga	**^K**
sostituzione	**F10**	cancella riga	**^AK**
apri file	**TAB** (**^XF**)	cancella da qui	**^TR**
inserisci file	**^TI**	cerca →	**^S**
togli buffer	**^TK**	cerca ←	**^R**
tutta la finestra	**F7**	termina comando	**^G**
cambia mezzafinestra	**^CV**	undo	**^TU**
apropos espressione	**^TH A**	maiuscole	**^TM**
apropos tasto	**^TH K**	minuscole	**^TN**
apropos variabile	**^TX**	compila	**Canc**
descrivi comando	**^TW**	esegui alfa	**Fine**
descrivi variabile	**^TV**	goto	↖

Per battere un carattere tabulatore bisogna usare **^Q TAB**.
Per scambiare due righe si utilizza **^XT**.

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*.

Scrivere programmi con Emacs

Emacs è un editor ideale per scrivere programmi in qualsiasi linguaggio di programmazione. La possibilità di combinare la scrittura dei files sorgenti con l'esecuzione del programma o di operazioni aggiuntive permette un lavoro efficiente e creativo.

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 di un programma in C, se il file su cui si sta lavorando sotto Emacs fa parte di un progetto. I messaggi di compilazione appaiono 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 nostri 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)
```

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.

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 e **put** o **mput** per deporre dei files. 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://...*).

Come spegnere

Dalla consolle a solo testo si entra nell'interfaccia grafica X con il comando **startx** oppure con il nostro alias **win**.

Per spegnere il PC prima uscire da X con la combinazione *Ctrl-Alt-Del*, poi battere il comando **poweroff**.

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 al 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 **^c**). 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 possono usare questi numeri nella redirezione dei messaggi d'errore: con **comando 2**$>$ **alfa** si fa in modo che i messaggi d'errore vengano scritti nel file **alfa**. In questo caso in **2**$>$ non ci 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 della shell

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 **ls -l** $|$ **mail rossi**.

Le pipelines possono essere combinate, **ls** $|$ **grep [Rr]ob** $|$ **lpr –s** ad esempio fa in modo che venga stampato l'elenco di tutti i files (nella directory **.**) il cui nome contiene *rob* o *Rob*.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 3 ◇ 9 Ottobre 2001

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 comando* ***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., provare ***stat alfa****.*

Questa settimana

9 Link e link simbolici
shutdown e poweroff

10 Diritti d'accesso
chown e chgrp
chmod

11 tee
sort
Un errore pericoloso
Espressioni regolari
grep

12 I files .profile e .bash_rc
Il file .bash_logout
Salvare il lavoro
Il file .plan
gqview

shutdown e poweroff

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.

Come vedremo a pagina 12 ad ogni uscita dal sistema vengono inoltre eseguite le istruzioni contenute nel file **.bash_logout**.

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** (cfr. pagina 7). Talvolta 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** di **shutdown** effettua lo shutdown subito, **19:10** alle 19.10, **+17** fra 17 minuti.

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. La seconda colonna contiene nel caso di un file il numero dei suoi link, nel caso di una cartella il numero delle sottocartelle (comprese **.** e **..**).

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` finale 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--
```

Attenzione: 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 non `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**.

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.

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**.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 4 ◇ 16 Ottobre 2001

Programmare in Perl

Il Perl è il linguaggio preferito dagli amministratori di sistema e una felice combinazione di concezioni della linguistica e di abili tecniche di programmazione; è usato nello sviluppo di software per l'Internet e in molte applicazioni scientifiche semplici o avanzate.

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 alla programmazione funzionale e all'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 (pagina 22) 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.

A differenza dal C 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 nel Perl, 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*. Una sintassi simile vale per i componenti di una lista.

Questa settimana

13 Programmare in Perl
Variabili nel Perl
Input dalla tastiera

14 Liste
La funzione grep del Perl
Alcuni operatori per liste
Contesto scalare e contesto listale

15 Files e operatore <>
Funzioni del Perl
Moduli
Il modulo files

16 Vero e falso
Operatori logici del Perl
Operatori di confronto
Istruzioni di controllo
Tirocini all'ARDSU

Input dalla tastiera

Esaminiamo il programma in Perl visto a pag. 2; assumiamo che sia contenuto nel file **alfa**. Dopo il comando *alfa* dalla shell (dobbiamo ricordarci di rendere il file eseguibile con *chmod +x alfa*) ci viene chiesto il nome che possiamo inserire dalla tastiera; il programma ci saluta utilizzando il nome specificato.

```
#! /usr/bin/perl -w
# alfa
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.

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*), che tratteremo quando parleremo della programmazione funzionale.

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* (pag. 16) 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)*.

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
```

Il cappuccio ^ denota l'inizio della riga, e quindi */^F/* è vera se la riga inizia con *F*; un punto esclamativo anteposto significa negazione.

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.

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.

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 che tratteremo più avanti: 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
```

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 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*, come abbiamo visto a pagina 13; 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);
```

oppure

```
$/=undef;
open(File,"lettera");
print < File> ;
close(File);
```

Il separatore di fineriga (una stringa) è il valore della variabile speciale *$/* e può essere impostato dal programmatore; di default è uguale a "\n". Se lo rendiamo indefinito con *$/=undef* possiamo leggere tutto il file in un blocco solo come nel secondo esempio.

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 somma2 {my ($a,$b)=@_; $a+$b}
sub somma {my $s=0;
  for (@_) {$s+=$_} $s}
print raddoppia(4)," ";
print somma2(6,9)," ";
print somma(0,1,2,3,4),"\n";
```

con output *8 15 10*.

Alcuni operatori abbreviati che vengono usati in C e Perl:

```
$a+=$b ... $a=$a+$b
$a–=$b ... $a=$a–$b
$a*=$b ... $a=$a*$b
$a/=$b ... $a=$a/$b
$a++   ... $a=$a+1
$a–    ... $a=$a–1
```

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* (pagina 14) 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);
```

Abbiamo usato la funzione *join* per unire con caratteri di nuova riga gli elementi della lista ottenuta con *files::catalogo* in un'unica stringa.

Vero e falso

La verità di un'espressione in Perl viene sempra 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.

Operatori logici del Perl

Per la congiunzione logica (AND) 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)* $\{\alpha\}$ è equivalente a *if (A)* $\{$*if (B)* $\{\alpha;\}\}$ e *if (A||B)* $\{\alpha;\}$ è equivalente a *if (A)* $\{\alpha\}$ *else* $\{$*if (B)* $\{\alpha\}\}$.

Il punto esclamativo viene usato per la negazione logica; anche *not* può essere usato a questo scopo.

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”;
   # Nessun output!
```

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
```

Istruzioni di controllo

Nelle alternative di un *if* si possono usare sia *else* che *elsif* (come abbreviazione di *else* $\{$*if* ...$\}$):

```
sub sgn {my $a=shift; if ($a< 0) {-1}
   elsif ($a> 0) {1} else {0}}
```

if può anche seguire un'istruzione o un blocco *do*:

α *if* A oppure *do* $\{\alpha;\ \beta;\ \gamma\}$ *if* A.

Il *goto* è un'istruzione di salto come nel seguente esempio:

```
$k=0;
ciclo: $k=$k+1; if ($k==7) {goto fine}
   # altre istruzioni
   goto ciclo;
fine: print “Fine\n”;
```

Le etichette (in questo caso due) si riconoscono dal doppio punto finale.

In Perl esistono 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 *for* classico ha la seguente forma:

for $(\alpha;A;\beta)$ $\{\gamma\}$

equivalente a

```
α;
ciclo: if (A) {γ; β; goto ciclo}
```

oppure anche a

for $(\alpha;A;)$ $\{\gamma;\ \beta\}$

α, β e γ sono successioni di istruzioni separate da virgole; l'ordine in cui le istruzioni in ogni successione vengono eseguite non è prevedibile. Ciascuno dei tre campi può anche essere vuoto.

while (A) è equivalente a *for (;A;)* e *until (A)* equivalente a *for (;not A;)*.

Da un *for* (o *while* o *until*) si esce con *last* (o con *goto*), mentre *next* fa in modo che si torni ad eseguire il prossimo passaggio del ciclo, dopo esecuzione di β. Quindi

for (;A;β*)* $\{\gamma_1;$ *if (B) break;* $\gamma_2;\}$

è equivalente a

```
ciclo: if (not A) {goto fuori}
   γ1; if (B) {goto fuori}
   γ2; β;
   goto ciclo;
fuori:
```

mentre

for (;;β*)* $\{\gamma_1;$ *if (B)* $\{$*continue*$\}$ $\gamma_2\}$

è equivalente a

ciclo: γ_1*; if (!B)* $\{\gamma_2\}$ β*; goto ciclo;*

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 ...
```

Esistono anche le costruzioni *do* $\{\ldots\}$ *while* A e *do* $\{\ldots\}$ *until* A in cui le istruzioni nel blocco vengono eseguite sempre almeno una volta.

In *for $k (@a)* la variabile *$k* (locale per il *for*) percorre la lista *@a*. Se manca *$k* come in *for (@a)*, viene utilizzata la variabile speciale *@_*.

```
for (0..10) {last if $_> 5; print $_}
   # output: 012345

for ($k=0;$k< =10;$k++)
   {next if $k< 5; print $k}
   # output: 5678910

sub max {my $max=shift;
   for (@_)
   {$max=$_ if $_> $max} $max}
sub min {my $min=shift;
   for (@_)
   {$min=$_ if $_< $min} $min}
```

Tirocini all'ARDSU

L'ufficio *Orientamento al Lavoro* dell'ARDSU di Ferrara è un buon punto di partenza per chi cerca contatti con il mondo del lavoro. Proprio in questi mesi ha avuto, da buone aziende a Ferrara o nella provincia di Ferrara, molte offerte di tesi di laurea e tirocini (per informatici e matematici) con possibilità di assunzioni in seguito.

L'ufficio inoltre organizza da anni seminari (ad esempio su *tecniche e strumenti per la ricerca attiva del lavoro*), corsi di informatica di base, corsi di lingua inglese e diversi progetti formativi. L'ARDSU collabora strettamente con l'università e con gli enti pubblici e ha contatti con molte aziende.

L'ufficio si trova a Ferrara in via Cairoli 32 (cortile interno). Telefonare prima alla signora Ornella Gandini, tel. 299812.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 5 ◇ 23 Ottobre 2001

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. 14). 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

$pi=3.14159265358979323846; $pid180=$pi/180;
sub alfaecosalfa {my $alfa=shift; ($alfa,cos($alfa*$pid180))}
%tabellacoseni = map {alfaecosalfa($_)} (0..360);
# crea una tabella dei coseni
# gli angoli vengono indicati in gradi
print $tabellacoseni{30};
# output: 0.866025403784439
```

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
```

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*!

Questa settimana

17 map
Vettori associativi
Nascondere le variabili con my

18 I numeri di Fibonacci
Il sistema di primo ordine
Numeri esadecimali

19 La moltiplicazione russa
Trovare la rappresentazione binaria
La potenza russa
Lo schema di Horner ricorsivo

20 Lo schema di Horner classico
Zeri di una funzione continua

Nascondere le variabili con my

Consideriamo le seguenti istruzioni:

```
$a=7;
sub quadrato {$a=shift; $a*$a}
print quadrato(10),"\n";
# output: 100
print "$a\n";
# output: 10
```

Vediamo che il valore della variabile esterna *$a* è stato modificato dalla chiamata della funzione *quadrato* che utilizza anch'essa la variabile *$a*. Probabilmente non avevamo questa intenzione e si è avuto questo effetto solo perché accidentalmente le due variabili avevano lo stesso nome. Per evitare queste collisioni dei nomi delle variabili il Perl usa la specifica **my**:

```
$a=7;
sub quadrato {my $a=shift; $a*$a}
print quadrato(10),"\n";
# output: 100
print "$a\n";
# output: 7
```

In questo modo la variabile all'interno di *quadrato* è diventata una variabile privata o locale di quella funzione; quando la funzione viene chiamata, alla *$a* interna viene assegnato un nuovo indirizzo in memoria, diverso da quello corrispondente alla variabile esterna. Consideriamo un altro esempio:

```
sub quadrato {$a=shift; $a*$a}
sub xpiu1alcubo {$a=shift;
   quadrato($a+1)*($a+1)}
print xpiu1alcubo(2);
# output: 36 invece di 27
```

Viene prima calcolato *quadrato(2+1)*, ponendo la variabile globale *$a* uguale all'argomento, cioè a 3, per cui il risultato finale è $9(3 + 1) = 36$.

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 $n \geq 2$. Quindi si inizia con due volte 1, poi ogni termine è la somma dei due precedenti: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Un programma iterativo in Perl per calcolare l'n-esimo numero di Fibonacci:

```
sub fib1 {my $n=shift; my ($a,$b,$k);
  return 1 if $n<=1;
  for ($a=$b=1,$k=2;$k<=$n;$k++)
  {($a,$b)=($a+$b,$a)} $a}
```

Possiamo visualizzare i numeri di Fibonacci da F_0 a F_{20} e da F_{50} a F_{60} con la seguente funzione:

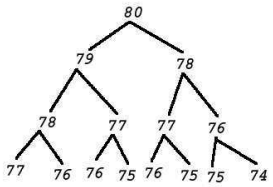
```
sub visfibonacci
  {for (0..20,50..60) {printf("%3d %-12.0f\n",$_,fib1($_))}}
```

La risposta è fulminea.

La definizione stessa dei numeri di Fibonacci è di natura ricorsiva, e quindi sembra naturale usare invece una funzione ricorsiva:

```
sub fib2 {my $n=shift;
  return 1 if $n<=1;
  fib2($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 i velocissimi Pentium non sembrano in grado di calcolare F_{50}. 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 ad esempio 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, 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

Più avanti in analisi si imparerà che 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 $n \geq 0$ 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 $n \geq 1$). Per applicare questo algoritmo dobbiamo però in ogni passo generare due valori, avremmo quindi bisogno di una funzione che restituisce due valori numerici. Ciò in Perl, dove una funzione può restituire come risultato una lista, è molto facile:

```
sub fib3 {my $n=shift; my ($x,$y);
  return (1,0) if $n==0;
  ($x,$y)=fib3($n-1); ($x+$y,$x)}
```

Per la visualizzazione dobbiamo ancora modificare la funzione **fibonacci** nel modo seguente:

```
sub visfibonacci {my ($x,$y);
  for (0..20,50..60)
  {($x,$y)=fib3($_);
  printf("%3d %-12.0f\n",$_,$x)}}
```

Numeri esadecimali

Nei linguaggi macchina e assembler molto spesso si usano i numeri esadecimali o, più correttamente, la rappresentazione esadecimale dei numeri naturali, cioè la loro rappresentazione in base 16.

Per $2789 = 10 \cdot 16^2 + 14 \cdot 16 + 5 \cdot 1$ potremmo ad esempio scrivere

`2789 = (10,14,5)`$_{16}$.

In questo senso `10,14` e `5` sono le cifre della rappresentazione esadecimale di `2789`. Per poter usare lettere singole per le cifre si indicano le cifre `10,...,15` mancanti nel sistema decimale nel modo seguente:

10	A
11	B
12	C
13	D
14	E
15	F

In questo modo adesso possiamo scrivere $2789 = (AE5)_{16}$. In genere si possono usare anche indifferentemente le corrispondenti lettere minuscole. Si noti che `(F)`$_{16}$ `= 15` assume nel sistema esadecimale lo stesso ruolo come il `9` nel sistema decimale. Quindi `(FF)`$_{16}$ `= 255` $= 16^2 - 1 = 2^8 - 1$. Un numero naturale n con $0 \leq n \leq 255$ si chiama un **byte**, un **bit** è invece uguale a 0 o a 1. Esempi:

	0	(0)$_{16}$
	14	(E)$_{16}$
	15	(F)$_{16}$
	16	(10)$_{16}$
	28	(1C)$_{16}$
2^5	32	(20)$_{16}$
2^6	64	(40)$_{16}$
	65	(41)$_{16}$
	97	(61)$_{16}$
	127	(7F)$_{16}$
2^7	128	(80)$_{16}$
	203	(CB)$_{16}$
	244	(F4)$_{16}$
	255	(FF)$_{16}$
2^8	256	(100)$_{16}$
2^{10}	1024	(400)$_{16}$
2^{12}	4096	(1000)$_{16}$
	65535	(FFFF)$_{16}$
2^{16}	65536	(10000)$_{16}$

Si vede da questa tabella che i byte sono esattamente quei numeri per i quali sono sufficienti al massimo due cifre esadecimali. Nell'immissione di una successione di numeri esadecimali come un'unica stringa spesso si pone uno zero all'inizio di quei numeri (da `0` a `9`) che richiedono una cifra sola, ad esempio la stringa `0532A2014E586A750EAA` può essere usata per rappresentare la successione (5,32,A2,1,4E,58,6A,75,E,AA) di numeri esadecimali.

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 (nello stesso modo abbiamo calcolato $10x$ a pag. 14):

$$86 \cdot x \longrightarrow 43 \cdot 2x \xrightarrow{\nearrow 2x} 42 \cdot 2x \longrightarrow 21 \cdot 4x \xrightarrow{\nearrow 4x} 20 \cdot 4x \longrightarrow$$
$$10 \cdot 8x \longrightarrow 5 \cdot 16x \xrightarrow{\nearrow 16x} 4 \cdot 16x \longrightarrow 2 \cdot 32x \longrightarrow$$
$$1 \cdot 64x \xrightarrow{\nearrow 64x} \longrightarrow 0 \cdot 128x \longrightarrow \bullet$$

Lo schema va interpretato così: p sarà il risultato della moltiplicazione; all'inizio poniamo $p = 0$. $86x = 43 \cdot 2x$, quindi possiamo sostituire x con $2x$ e dimezzare il primo fattore che così diventa dispari. Con il nuovo x abbiamo $43x = x + 42x$. Aggiungiamo x a p e procediamo con $42x = 21 \cdot 2x$ come nel primo passo: sostituiamo quindi x con $2x$ e dimezziamo il primo fattore. E così via: Quando arriviamo a un primo fattore dispari, sommiamo l'ultimo valore di x a p e diminuiamo il primo fattore di uno che così diventa pari. Quando il primo fattore è pari, sostituiamo x con $2x$ e dimezziamo il primo fattore. Ci fermiamo quando il primo fattore è uguale a 0.

Altrettanto semplice e importante è la formulazione puramente matematico-ricorsiva dell'algoritmo - scriviamo f per la funzione definita da $f(n, x) = nx$:

$$f(n,x) = \begin{cases} 0 & \text{se} \quad n = 0 \\ f(\frac{n}{2}, 2x) & \text{se} \quad n \text{ è pari} \\ x + f(n-1, x) & \text{se} \quad n \text{ è dispari} \end{cases}$$

Diamo una versione ricorsiva e una versione iterativa in Perl per questo algoritmo:

```
sub mrussa # $p=mrussa($n,$x)
  {my ($n,$x)=@_;
  return $x+mrussa($n-1,$x) if $n%2;
  return mrussa($n/2,$x+$x) if $n>0; 0}
sub mrussa # $p=mrussa($n,$x)
  {my ($n,$x)=@_; my $p;
  for ($p=0;$n;) {if ($n%2) {$p+=$x; $n--}
  else {$x+=$x; $n/=2}} $p}
```

Come trovare la rappresentazione binaria

Non è difficile convincersi che ogni numero naturale $n > 0$ possiede una rappresentazione binaria, cioè della forma

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_2 2^2 + a_1 2 + a_0 \qquad (*)$$

con coefficienti (o cifre) $a_i \in \{0, 1\}$ e $a_k = 1$ univocamente determinati. Sia $\text{rapp2}(n) = (a_k, \ldots, a_0)$ la lista i cui elementi sono queste cifre. Dalla rappresentazione (*) si deduce la seguente relazione ricorsiva, in cui utilizziamo il meccanismo della fusione di liste del Perl (cfr. pag. 19):

$$\text{rapp2}(n) = \begin{cases} (1) & \text{se} \quad n = 1 \\ (\text{rapp2}(\frac{n}{2}), 0) & \text{se} \quad n \text{ è pari} \\ (\text{rapp2}(\frac{n-1}{2}), 1) & \text{se} \quad n \text{ è dispari} \end{cases}$$

Questa relazione può essere tradotta immediatamente in un programma in Perl:

```
sub rapp2 # @cifre=rapp2($n)
  {my $n=shift; return (1) if $n==1;
  return (rapp2($n/2),0) if $n%2==0;
  (rapp2(($n-1)/2),1)}
```

La potenza russa

Per il calcolo dipotenze con esponenti reali arbitrari si può usare la funzione **pow** del Perl: la terza radice si ottiene ad esempio con *pow(x,1/3)*. Come molte altre funzioni matematiche (ad esempio *sin* e *cos*) anche *pow* richiede l'inclusione *use POSIX;* all'inizio del file.

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.

Anche in questo caso invece del ragionamente iterativo si può riformulare il problema in modo matematico-ricorsivo - stavolta $f(x, n) = x^n$:

$$f(x,n) = \begin{cases} 1 & \text{se} \quad n = 0 \\ f(x^2, \frac{n}{2}) & \text{se} \quad n \text{ è pari} \\ x f(x, n-1) & \text{se} \quad n \text{ è dispari} \end{cases}$$

È facile tradurre queste idee in un programma ricorsivo o iterativo in Perl:

```
sub potenza # $p=potenza($x,$n)
  {my ($x,$n)=@_;
  return $x*potenza($x,$n-1) if $n%2;
  return potenza($x*$x,$n/2) if $n>0; 1}
sub potenza # $p=potenza($x,$n)
  {my ($x,$n)=@_; my $p;
  for ($p=1;$n;) {if ($n%2) {$p*=$x; $n--}
  else {$x*=$x; $n/=2}} $p}
```

Lo schema di Horner ricorsivo

Lo schema di Horner per il calcolo del valore $f(\alpha)$ di un polinomio $f = a_0 x^n + a_1 x^{n-1} + \cdots + a_n$ permette una elegante versione ricorsiva. Infatti, se denotiamo quel valore con $\text{val}(\alpha, a_0, a_1, \ldots, a_n)$, allora abbiamo la relazione

$$\text{val}(\alpha, a_0, a_1, \ldots, a_n) = \alpha \cdot \text{val}(\alpha, a_0, a_1, \ldots, a_{n-1}) + a_n$$

come si vede da

$$a_0\alpha^n + a_1\alpha^{n-1} + \cdots + a_{n-1}\alpha + a_n =$$
$$= \alpha \cdot (a_0\alpha^{n-1} + a_1\alpha^{n-2} + \cdots + a_{n-1}) + a_n$$

con la condizione iniziale $\text{val}(\alpha) = 0$. Ciò può essere tradotto in un programma in Perl:

```
sub val # y=val($x,a0,...,a1)
  {my $x=shift; return 0 if not @_;
  my $a=pop; $a+$x*val($x,@_)}
```

Lo schema di Horner classico

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 $f = 3x^4 + 5x^3 + 6x^2 + 8x + 17$. Poniamo

$$\begin{array}{lcll} b_0 & = & 3 & \\ b_1 & = & b_0\alpha + 5 & = 3\alpha + 5 \\ b_2 & = & b_1\alpha + 6 & = 3\alpha^2 + 5\alpha + 6 \\ b_3 & = & b_2\alpha + 8 & = 3\alpha^3 + 5\alpha^2 + 6\alpha + 8 \\ b_4 & = & b_3\alpha + 17 & = 3\alpha^4 + 5\alpha^3 + 6\alpha^2 + 8\alpha + 17 \end{array}$$

e vediamo che $b_4 = f(\alpha)$. Lo stesso si può fare nel caso generale:

$$\begin{array}{lcl} b_0 & = & a_0 \\ b_1 & = & b_0\alpha + a_1 \\ \cdots & & \\ b_k & = & b_{k-1}\alpha + a_k \\ \cdots & & \\ b_n & = & b_{n-1}\alpha + a_n \end{array}$$

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

$$\begin{array}{rl} \alpha g = & \alpha b_0x^{n-1} + \alpha b_1x^{n-2} + \cdots + \alpha b_{n-1} = \\ = & (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \cdots + \\ & +(b_{n-1} - a_{n-1})x + b_n - a_n = \\ = & (b_1x^{n-1} + b_2x^{n-2} + \cdots + b_{n-1}x + b_n) - \\ & -(a_1x^{n-1} + a_2x^{n-2} + \ldots a_{n-1}x + a_n) = \\ = & x(g - b_0x^{n-1}) + b_n - (f - a_0x^n) = \\ = & xg - b_0x^n + b_n - f + a_0x^n = xg + b_n - f \end{array}$$

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 Perl si può usare la stessa variabile per tutti i b_k:

```
sub horner # y=horner($x,@a)
  {my ($x,@a)=@_; my $b=shift @a;
  for (@a) {$b=$b*$x+$_} $b}
```

Una frequente applicazione dello schemo di Horner è il calcolo del valore corrispondente a una rappresentazione binaria o esadecimale. Infatti

$$(1,0,0,1,1,0,1,1,1)_2 = \text{horner}(2,1,0,0,1,1,0,1,1,1)$$

$$(A,F,7,3,0,5,E)_{16} = \text{horner}(16,10,15,7,3,0,5,14).$$

Zeri di una funzione continua

Siano $a < b$ e $f : [a,b] \longrightarrow \mathbb{R}$ una funzione continua tale che $f(a) < 0$ e $f(b) > 0$. In analisi si impara che allora la funzione f deve contenere uno zero nell'intervallo (a,b). Da questo fatto deriva un buon metodo elementare e facile da ricordare per la ricerca delle radici di una funzione continua, che in un pseudolinguaggio che prevede procedure ricorsive può essere formulato nel modo seguente:

```
if b − a < ε then return (a, b)
x = (a + b)/2
if f(x) == 0 then return (x, x)
if f(x) > 0 then cerca in (a, x) # ricorsione
else cerca in (x, b) # ricorsione
```

$\varepsilon > 0$ 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 teoricamente deve terminare. In pratica però potrebbe non essere così. Infatti, se ε è minore della precisione della macchina, a un certo punto si avrà che il valore effettivamente calcolato come approssimazione di $x = \frac{a+b}{2}$ è uguale a b, e quindi, se al passo (4) dobbiamo sostituire b con x, rimaniamo sempre nella situazione (a,b) e avremo un ciclo infinito.

Assumiamo ad esempio che $a = 3.18$ e $b = 3.19$ e che la macchina arrotondi a due cifre decimali. Allora $a + b = 6.37$ e $x = \frac{a+b}{2} = 3.185$ che viene arrotondato a $3.19 = b$. Se noi avessimo impostato $\varepsilon = 0.001$, il programma possibilmente non termina.

Si può però sfruttare questo fenomeno a nostro favore: l'imprecisione della macchina fa in modo che prima o poi arriviamo a $x = a$ o $x = b$, e in quel momento ci fermiamo, potendo così applicare un criterio di interruzione indipendente dalla macchina.

In Perl possiamo formulare l'algoritmo nel modo seguente - bisogna prima verificare che veramente $f(a) < 0$ e $f(b) > 0$ e sostituire f con $-f$ quando accade il contrario:

```
sub zero # ($a,$b)=zero($f,$a,$b)
  {my ($f,$a,$b)=@_;
  my $x=($a+$b)/2;
  return ($a,$b) if $x==$a or $x==$b;
  my $fx=&$f($x); return ($x,$x) if $fx==0;
  return zero($f,$a,$x) if $fx>0;
  zero($f,$x,$b)}
```

$f(x)$ deve essere calcolato due volte, per questa ragione nella terzultima riga del programma abbiamo introdotto la variabile ausiliaria *$fx* a cui viene assegnato il valore $f(x)$.

Dobbiamo spiegare qui la sintassi usata nel Perl per argomenti che sono funzioni. L'argomento *$f* che rappresenta la funzione è uno scalare (cfr. pag. 18) di un tipo particolare: un *puntatore* (o *riferimento*) alla funzione. Per chiamare la funzione a cui punta questo riferimento bisogna utilizzare *&$f*. Se abbiamo definito una funzione *f* mediante *sub f*, il puntatore ad essa è *\&f*.

Vediamo queste nozioni nel seguente esempio in cui calcoliamo una (delle al massimo quattro) radici del polinomio $f = x^4 + x - 7$. Verifichiamo prima che $f(0) = -7 < 0$ e $f(2) = 16 + 2 - 7 = 11 > 0$. Adesso possiamo scrivere

```
($a,$b)=zero(\&f,0,2);
print "$a <= x <= $b\n";
# output: 1.52935936477247 <= x <= 1.52935936477247
```

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 6 ◇ 30 Ottobre 2001

Espressioni regolari nel Perl

Come abbiamo osservato a pag. 11, 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. 11. 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. 23. 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

21 Espressioni regolari nel Perl
Gli operatori *m* ed *s*
I modificatori */m* ed */s*

22 I metacaratteri
I metasimboli
Il modificatore */g*
I modificatori */i* ed */o*
La funzione pos

23 Il modificatore */e*
Riassunto dei modificatori
$_ sottinteso nelle espressioni regolari
Alternative a /.../
Parentesi tonde
split e join

24 Fattori di una parola
Lettura a triple del DNA
Uso di | nelle espressioni regolari
Ricerca massimale e minimale
index e rindex

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. 11, ^ e *$* vengono utilizzati per indicare l'inizio e la fine della stringa. Questo significato cambia se all'istruzione di matching aggiungiamo *m*: in questo caso ^ indica anche l'inizio di una riga (cioè l'inizio della stringa oppure una posizione preceduta da un carattere di nuova riga), e similmente *$* 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 ^); 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: \, | , *(,), [,]*, {, }, ^, *$*, ***, +, *?*, . 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 ^ 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*.
- **^** Questo carattere indica l'inizio di parola (oppure, quando è presente il modificatore */m*, anche l'inizio di una riga, cfr. pag. 21), 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. 21).
- **-** 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	[^0-9]
\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** ^ 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
```

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. 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*.

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 nondeterminanti. Un fattore r di una parola x si dice un *fattore nondeterminante a sinistra*, 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 nds {my ($r,$x)=@_;
   return 1 if $x=~/(.)$r/ and $x=~/[^$1]$r/; 0}
```

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. 14).

```
$a=$b="TATATCTGCAATTTGATAGATCGA";
```

Triple: TAT ATC TGC AAT TTG ATA GAT CGA.

```
@a=$a=~/AT[ACT]/g;
for (@a) {print "$_"} print "\n";
# output errato: ATA ATT ATA ATC
```

Lettura: T**AT A**TC TGC A**AT T**TG **ATA** G**AT C**GA.

```
$b=~s/(...)/($1)/g;
# $b="(TAT)(ATC)(TGC)(AAT)(TTG)(ATA)(GAT)(CGA)"
@a=$b=~/\((AT[ACT])\)/g;
for (@a) {print "$_"} print "\n";
# output corretto: ATC ATA
```

L'uso delle parentesi tonde nell'istruzione di matching fa in modo che @*a* contenga solo le parti catturate dalle parentesi.

```
@a=grep {/AT[ACT]/} $a=~/.../g;
# $a=~/.../g è la lista delle triple!
for (@a) {print "$_"} print "\n";
# output corretto: ATC ATA
```

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
```

Attenzione: 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 *?

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
```

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 7 ◇ 13 Novembre 2001

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; in Perl 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)
```

Questa settimana

25 I puntatori del Perl
Liste di liste e matrici
Concatenazione di stringhe
I moduli CPAN

26 substr
Strumenti per le stringhe
lc e uc
Puntatori a variabili locali
Passaggio di parametri in Perl
Invertiparola e eliminacaratteri

Concatenazione di stringhe

Il simbolo di concatenazione per stringhe è il punto: *$a."Roma".$b.$b* è la concatenazione delle stringhe *$a*, "*Roma*", *$b* e *$c*.

In questo caso avremmo anche potuto scrivere *"${a}Roma$b$c"*; il punto si usa ad esempio in *"(".f($a).")".riga(3)*, dove *f* e *riga* sono funzioni che restituiscono stringhe. Talvolta si può anche utilizzare la funzione *sprintf* (fra poco).

***

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. 15), 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*.

substr

Sia $\$a = a_0a_1 \ldots a_n$ una stringa. **substr($a,i)** fornisce allora la stringa $a_ia_{i+1} \ldots a_n$, mentre **substr($a,i,k)** è uguale a $a_ia_{i+1} \ldots a_{i+k-1}$ (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 $a_ia_{i+1} \ldots a_{i+k-1}$ 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. 22) 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
```

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.
```

Puntatori a variabili locali

A differenza dal C, nel Perl 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
```

Passaggio di parametri in Perl

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, prendendo come argomento l'indirizzo della variabile da modificare; 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. 14):

```
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.

```
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*.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 8 ◇ 20 Novembre 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

27 ELIZA
Struttura dei files tematici
Risposte casuali

28 Una conversazione con ELIZA
Il file Eliza/alfa
Il file Eliza/saluti.pm

29 Il file Eliza/dialogo.pm
Il file Eliza/aus.pm
Linux Day al dipartimento

30 Numeri casuali e rand
Typeglobs
Ordinare una lista con sort
printf e sprintf
Libri sul Perl

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 *$femmina* del modulo *aus* se necessario, carica poi prima il file *trasformazioni* e, se non è applicabile, utilizza gli altri file tematici. Se non individua una risposta adatta, trova una risposta casuale. La terzultima riga serve per impedire che il programma ripeta quello che ha detto l'utente (eliminando così il problema visto nel dialogo a pag. 28) o la propria risposta precedente. La variabile *$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. 28, 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.
```

Linux Day al dipartimento di Matematica

Sabato 1 dicembre 2001 nell'aula magna del dipartimento di Matematica con inizio alle ore 9.30.

La giornata sarà organizzata in due momenti. La mattinata sarà dedicata alla presentazione del sistema GNU/Linux e del software libero con interventi di imprenditori locali, rappresentanti della amministrazione locale e della scuola/università. Seguirà un dibattito sui temi della manifestazione.

La seconda parte della giornata, nel pomeriggio, sarà dedicata a tutti coloro che vogliono avvicinarsi al mondo GNU/Linux. Sarà possibile recarsi alla manifestazione con il proprio PC ed essere seguiti nell'installazione del sistema operativo da volontari del flug. Alcuni soci saranno a disposizione per rispondere alle domande dei partecipanti e per aiutarli nella risoluzione dei problemi di installazione/configurazione del sistema, oppure semplicemente per fare due chiacchiere e scambiarsi opinioni.

Il programma dettagliato delle attività e degli interventi sarà disponibile sul sito Internet (*www.ferrara.linux.it/*) del *Ferrara Linux Users Group* nei prossimi giorni.

Numeri casuali e rand

rand(n) restituisce un numero (pseudo-)casuale 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)*, come abbiamo fatto a pagina 29 per le risposte casuali. Possiamo quindi definire una funzione *dado*:

```
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.

Parleremo nel secondo semestre un po' di più sui numeri pseudocasuali.

Typeglobs

Abbiamo visto a pagina 15 che per utilizzare una funzione *f* di un package *alfa* al di fuori di quest'ultimo la funzione deve essere invocata con *alfa::f*. Così abbiamo fatto in alcune istruzioni nelle due pagine precedenti:

```
saluti::presentazione();
dialogo::generico();
files::scrivi($file,$eliza);
my $utente=<main::stdin>;
saluti::fine();
files::aggiungi($saluti::file,shift)
```

Torneremo su questo meccanismo e sui pacchetti più avanti quando parleremo della programmazione orientata agli oggetti in Perl.

Esistono vari meccanismi in Perl per poter usare nomi di funzioni o variabili senza il prefisso che individua il modulo, uno dei quali è l'utilizzo dei *typeglobs* che, ai nostri scopi, avviene nel modo seguente (cfr. pag. 28-29):

```
*salva=\&aus::salva;
*carica=\&aus::carica;
*mf=\&aus::mf;
*rispostacasuale=\&aus::rispostacasuale;
*salva=\&aus::salva;
```

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
```

La funzione *sort* prende come argomento un criterio di ordinamento e una lista e restituisce la lista ordinata come risultato. La lista originale rimane invariata. Nel criterio di ordinamento le variabili *$a* e *$b* hanno un significato speciale, simile a quello di *$_* in altre occasioni.

printf e sprintf

Abbiamo incontrato la funzione **printf** a pagina 18 nell'istruzione

```
sub visfibonacci
  {for (0..20,50..60) {printf("%3d %-12.0f\n",$_,fib1($_))}}
```

per la stampa formattata dei numeri di Fibonacci.

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 `%3d` tiene il posto per il valore della variabile `n` che verrà visualizzata come intero di tre cifre, mentre `-12.0f` indica una variabile di tipo **double** di al massimo 12 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 %		

$a=sprintf(...) fa in modo che *$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;
```

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.

D. Conway: Object oriented Perl. Manning 2000.

J. Friedl: Mastering regular expressions. O'Reilly 2000.

S. Gundavaram: CGI programming on the World Wide Web. O'Reilly 2001.

S. Srinivasan: Advanced Perl programming. O'Reilly 1999.

J. Tisdall: Beginning Perl for bioinformatics. O'Reilly 2001.

Allo stesso tempo un'introduzione alla programmazione in Perl e al suo uso in bioinformatica, ad esempio nel trattamento di dati genetici o biochimici prelevati dalle grandi banche dati.

L. Wall/T. Christiansen/J. Orwant: Programming Perl. O'Reilly 2000.

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 2001/2002 | Numero 9 ◇ 27 Novembre 2001

eval

La funzione eval *permette di eseguire codice in Perl che è stato generato duranto 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
```

Nel prossimo esempio la parte variabile contiene 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
```

L'istruzione 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: Non viene più stampato* Adesso continuo.*, mentre appare il messaggio* 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 avere una sessione come la seguente - provare!

```
:) alfa
print 8;
8
$a=3; $b=7; print $a+$b;
10
exit;
:)
```

Problemi di sicurezza con eval

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 o di cui non ricordiamo con precisione il contenuto.

Questa settimana

31 eval
Problemi di sicurezza con eval
Attributi di files e cartelle

32 Cercare gli errori
Puntatori a vettori associativi
Vettori associativi anonimi
Cartelle e diritti d'accesso
Uso procedurale degli operatori logici

33 Funzioni anonime
Funzioni come argomenti di funzioni
Il λ-calcolo
Funzioni come valori di funzioni
Programmazione funzionale in Perl

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
```

Esercizio: Usare le funzioni per files e cartelle viste a pagina 15 e la funzione *cwd* (pagina 32) per esaminare il contenuto della cartella di lavoro e delle sue sottocartelle con un programma ricorsivo.

Cercare gli errori

Esistono vari strumenti per la ricerca degli errori sotto Perl. Il più semplice è l'inclusione del modulo **diagnostics** con l'istruzione *use diagnostics;* all'inizio del file. In questo modo si otterrà una dettagliata descrizione degli errori, soprattutto quelli di sintassi. Unico inconveniente è la lunghezza dei messaggi.

Il Perl contiene anche un proprio **debugger** che viene attivato con l'opzione *-d*, ad esempio **perl -d alfa**. Una descrizione dei comandi possibili sotto il debugger si ottiene con *h h* (corta) oppure con *|h* (lunga).

Spesso però la ricerca degli errori può essere anche fatta con semplici comandi di interruzione combinati con output di risultati o valori intermedi. Qui è utile anche il comando **exit** che ferma un programma nel punto in cui si trova.

Puntatori a vettori associativi

Creiamo una sorta di inversa della funzione *fundahash* definita a pagina 33:

```
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
```

In questo modo da una funzione otteniamo in pratica una tabella, la cui prima colonna contiene gli elementi del dominio di definizione della funzione (rappresentati dagli elementi della lista a cui punta *$x*), mentre la seconda contiene i rispettivi valori della funzione.

Studiare attentamente questo esempio; sembra semplice, ma è molto istruttivo perché contiene una buona selezione dei concetti trattati in questo numero.

Vettori associativi anonimi

A pagina 25 abbiamo introdotto puntatori a liste anonime; puntatori a 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 di 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.

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. 15.

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
```

Uso procedurale degli operatori logici

Abbiamo già osservato a pagina 16 che gli operatori logici *and* e *or* nel Perl non sono simmetrici. Più precisamente

$$A_1 \text{ and } A_2 \text{ and } \ldots \text{ and } A_n$$

è uguale ad A_n se tutti gli A_i sono veri, altrimenti il valore dell'espressione è il primo A_i a cui corrisponde il valore di verità *falso*. Similmente

$$A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_n$$

è uguale ad A_n se nessuno degli A_i è vero, altrimenti è uguale al primo A_i che è vero.

Inoltre l'operatore *and* lega più strettamente dell'*or*, quindi invece di *(A and B) or (C and D)* possiamo semplicemente scrivere *A and B or C and D*.

Possiamo utilizzare questo comportamento degli operatori logici per eliminare molti *if* (o anche tutti) nei programmi, avvicinandoci a uno stile di programmazione logica spesso più trasparente e più efficiente. Talvolta si vorrebbe che il valore di un'espressione booleana sia 1 se è vera e 0 se è falsa; il primo esempio definisce una funzione *boole* che si comporta in questo modo. Seguono altri esempi per illustrare l'utilizzo procedurale degli operatori logici.

```
sub boole {shift and 1 or 0}
# invece di: {return 1 if shift; 0}
sub positivo {shift>0 and 1 or 0}
# invece di: {return 1 if shift>0; 0}
sub sgn {my $a=shift;
   $a>0 and 1 or $a<0 and -1 or 0}
sub fatt {my $n=shift;
   $n==0 and 1 or $n*fatt($n-1)}
sub fattiterativo {my $n=shift;
   my $k=0; my $p=1;
   loop: $k==$n and $p or
    do {$k++; $p*=$k; goto loop}}
sub potenza {my ($x,$n)=@_;
   $n==0 and 1 or
   $n%2 and $x*potenza($x,$n-1)
   or potenza($x*$x,$n/2)}
```

La versione iterativa dell'ultima funzione è leggermente più complicata, rimane però sempre chiara e logica:

```
sub potenzaiterativa {my ($x,$n)=@_;
   my $p=1; loop: $n==0 and $p or
   do {$n%2==0 and do {$x*=$x; $n/=2}
   or do {$p*=$x; $n--}; goto loop}}
sub c01 {shift==0 and 1 or 0}
sub scambia01 {my $a=shift;
   $a=~s/([01])/c01($1)/ge; $a}
```

Un errore in cui si incorre facilmente è di usare invece, per ottenere una stringa in cui tutti gli 1 sono sostituiti da 0 e viceversa,

```
$a=~s/0/1/g; $a=~s/1/0/g;
```

Perché non funziona?

Nel prossimo numero elaboreremo queste idee per giungere a affascinanti tecniche di *programmazione logica* in Perl.

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);
```

Il λ-calcolo

Siccome bisogna distinguere tra la funzione f e i suoi valori $f(x)$, introduciamo la notazione $\bigcirc_x f(x)$ per la funzione che manda x in $f(x)$. Ad esempio $\bigcirc_x \sin(x^2+1)$ è la funzione che manda x in $\sin(x^2+1)$. È chiaro che $\bigcirc_x x^2 = \bigcirc_y y^2$ (per la stessa ragione per cui $\sum_{i=0}^{n} a_i = \sum_{j=0}^{n} a_j$), mentre $\bigcirc_x xy \neq \bigcirc_y yy$ (così come $\sum_i a_{ij} \neq \sum_j a_{jj}$) e, come non ha senso l'espressione $\sum_i \sum_i a_{ii}$, così non ha senso $\bigcirc_x \bigcirc_x x$. Siccome in logica si scrive $\lambda x.f(x)$ invece di $\bigcirc_x f(x)$, la teoria molto complicata di questo operatore si chiama λ-calcolo. Su esso si basano il Lisp e molti concetti dell'intelligenza artificiale.

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 di cui diamo adesso alcuni esempi. È un campo molto attuale dell'informatica con evidenti legami con molti concetti familiari al matematico che forse avremo tempo di approfondire nel secondo modulo del corso.

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 {shift}
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
# oppure
print comp(\&quad,\&piu1)→(2),“\n”;
$f=comp(\&quad,\&diff2);
print &$f(6,1),“\n”; # output: 25
# oppure
print comp(\&quad,\&diff2)→(6,1),“\n”;
$f=valut(3);
print &$f(\&cubo),“\n”; # output: 27
# oppure
print valut(3)→(\&cubo),“\n”;
```

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.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2001/2002 — Numero 10 ◇ 4 Dicembre 2001

Programmazione logica

Continuiamo in questo numero il progetto di esplorare le possibilità di realizzare concetti logici e matematici in Perl. Vedremo come definire il prodotto cartesiano di liste, una costruzione fondamentale trascurata in genere nei linguaggi di programmazione e nei corsi di informatica. Anche qui il Perl si rivela molto versatile e permette di definire il prodotto cartesiano di un numero finito arbitrario di liste in poche righe. Come applicazione risolveremo un problema di colorazione, in verità già un esempio tipico di programmazione logica elementare (statica).

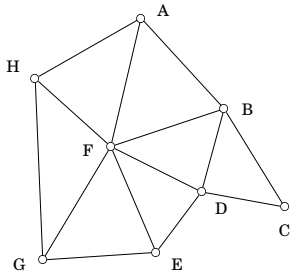
L'interpretazione procedurale della logica risale a Robert Kowalski ed Alain Colmerauer, l'acronimo Prolog (abbreviazione di programming in logic) fu inventato da P. Roussel (o forse da sua moglie Jacqueline). Durante gli anni '70 e i primi anni '80 il Prolog divenne popolare in Europa e in Giappone per applicazioni nell'ambito dell'intelligenza artificiale. È un linguaggio dichiarativo, negli intenti della programmazione logica il programmatore deve solo descrivere il problema e indicare le regole di base della soluzione, deve cioè specificare solo la componente logica di un algoritmo; ci pensa il sistema Prolog a trovare la soluzione. Purtroppo la cosa non è così facile e non è semplice programmare in Prolog.

La logica fornisce anche molti concetti fondamentali della teoria delle basi di dati. Infatti la logica generalizza il concetto di base di dati: un programma logico può essere considerato come una base di dati di regole che permettono di costruire una base di dati reali che può essere ampliata, ridotta, modificata. La logica aggiunge quindi un aspetto dinamico a una base di dati. Questo aspetto dinamico genera anche la possibilità di creare strutture complesse, profonde, tramite costruzioni ricorsive (mentre le basi di dati classiche non permettono la ricorsività).

Per questa volta vedremo soltanto, a pagina 35, come definire una tabella con più colonne in Perl e estrarne il contenuto. Impareremo anche come rappresentare una tabella su un file e come trasformare il contenuto del file in una tabella.

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 il metodo indicato a pagina 35, in cui utilizziamo il prodotto cartesiano di liste, troviamo tutte le soluzioni:



A	B	C	D	E	F	G	H
1	2	3	1	2	3	1	2
1	3	2	1	3	2	1	3
2	1	3	2	1	3	2	1
2	3	1	2	3	1	2	3
3	1	2	3	1	2	3	1
3	2	1	3	2	1	3	2

Questa settimana

34 Programmazione logica
Un problema di colorazione
Prolog e Perl

35 Il prodotto cartesiano di liste
Esercizi sull'uso di and/or
Liste anonime e basi di dati
Lettura dei dati da un file

Prolog e Perl

Un esempio in GNU-Prolog:

```
% alfa.pl

:-initialization(q).

padre(giovanni,maria).
padre(federico,alfonso).
padre(alfonso,elena).
padre(alfonso,carlo).

madre(maria,elena).
madre(patrizia,maria).
madre(roberta,alfonso).
madre(maria,carlo).

genitore(A,B):- padre(A,B); madre(A,B).
nonno(A,B):- padre(A,X), genitore(X,B).

trovanonni:- setof(X,nonno(X,elena),A),
   write(A), nl.

q:- trovanonni.
% output: [federico,giovanni]
```

Il GNU-Prolog si trova sul sito *pauillac.inria.fr/ diaz/gnu-prolog/*.

In questo caso semplice possiamo direttamente tradurre il programma in Prolog in un programma in Perl:

```
$padri=[["giovanni","maria"],
   ["federico","alfonso"],["alfonso","elena"],
   ["alfonso","carlo"]];

$madri=[["maria","elena"],
   ["patrizia","maria"],["roberta","alfonso"],
   ["maria","carlo"]];

$persone=["giovanni","maria",
   "federico","alfonso","elena","carlo",
   "patrizia","roberta"];

sub madre {my ($a,$b)=@_; for (@$madri)
   {$_->[0] eq $a and $_->[1] eq $b
   and return 1} 0}

sub padre {my ($a,$b)=@_; for (@$padri)
   {$_->[0] eq $a and $_->[1] eq $b
   and return 1} 0}

sub genitore {my ($a,$b)=@_;
   (padre($a,$b) or madre($a,$b))
   and 1 or 0}

sub nonno {my ($a,$b)=@_;
   for (@$persone) {padre($a,$_) and
   genitore($_,$b) and return 1} 0}

@nonni=grep {nonno($_,"elena")}
   @$persone;

for (@nonni) {print "$_\n"}
```

Il prodotto cartesiano di liste

Nella funzione *tabelladafile* definita su questa pagina abbiamo usato che dopo ad esempio *@a=(0,1,2); @b=(7,8,9);* non solo la lista *(@a,3,4,5,6,@b,10,11)* è uguale a *(0,1,2,3,4,5,6,7,8,9,10,11)*, ma anche la corrispondente lista anonima *[@a,3,4,5,6,@b,10,11]* diventa uguale a *[0,1,2,3,4,5,6,7,8,9,10,11]*.

Possiamo usare questa proprietà per definire il prodotto cartesiano di liste, una di quelle costruzioni matematiche estremamente potenti e utili che purtroppo mancano in molti linguaggi di programmazione.

La funzione *cartl* prende come argomenti una lista di puntatori a liste e restituisce una lista anonima che corrisponde al prodotto cartesiano degli argomenti.

```
sub cartl {my ($A,@R)=@_; my @P=(); my @a;
  not defined $A and return [] or
  not @R and @P=map {[$_]} @$A and return \@P or
  do {my $B=cartl(@R);
  for $x (@$A) {for $L (@$B) {push(@P,[$x,@$L])}} return \@P}}
```

Si noti che gli elementi del prodotto cartesiano di n fattori sono sempre n-ple, in particolare *cartl([0,1,2,3])* è uguale a *[[0],[1],[2],[3]]*. Possiamo adesso il risolvere il nostro problema di colorazione; con le seguenti funzioni troviamo il risultato indicato a pagina 34.

```
$A=[1,2,3]; $P=cartl($A,$A,$A,$A,$A,$A,$A,$A,$A);

sub colori {my $u=shift; my ($a,$b,$c,$d,$e,$f,$g,$h)=@$u;
  $a!=$b and $a!=$f and $a!=$h and $b!=$c and $b!=$d and
  $b!=$f and $c!=$d and $d!=$e and $d!=$f and $e!=$f and
  $e!=$g and $f!=$g and $f!=$h and $g!=$h or 0}

@a = grep {colori($_)} @$P;

for (@a) {($a,$b,$c,$d,$e,$f,$g,$h)=@$_;
  print "$a $b $c $d $e $f $g $h\n"}
```

Esercizi sull'uso di and/or

Studiare le seguenti funzioni. I primi due esempi sono versioni leggermente migliorate delle funzioni omonime a pagina 32. Stavolta non usiamo il *do*.

```
sub fatt {my ($n,$k,$p)=(shift,0,1);
  loop: $k==$n and $p or
  ++$k and $p*=$k and goto loop}

sub potenza {my ($x,$n,$p)=(shift,shift,1);
  loop: $n==0 and $p or $x==0 and 0 or
  $n%2 and $p*=$x and $n-- and goto loop or
  $x*=$x and $n/=2 and goto loop}
```

Naturalmente per il fattoriale si userà piuttosto un *for* classico come

```
sub fatt {my ($n,$p,$k)=(shift,1,1);
  for (;$k<=$n;$k++) {$p*=$k} $p}
```

oppure la versione ricorsiva a pagina 32.

```
sub mcd {my ($a,$b)=@_;
  $a<0 and $a=-$a; $b<0 and $b=-$b;
  $b==0 and $a or mcd($b,$a%$b)}
```

Questa funzione calcola il massimo comune divisore di numeri interi; siccome l'espressione $a\&b$ calcola correttamente il resto sole per argomenti ≥ 0, dobbiamo prima sostituire eventuali argomenti negativi con il loro valore assoluto; il massimo comune divisore rimane invariato.

```
sub rip {my ($f,$w)=@_; my $i=index($w,$f);
  $i<0 and return 0 or
  index($w,$f,$i+1)<0 and 0 or 1}
```

Questa funzione prende come argomenti due stringhe e restituisce 1 se la prima compare più di una volta nella seconda, indica cioè, se la prima stringa è un fattore ripetuto della seconda.

Liste anonime e basi di dati

Una *base di dati relazionale* consiste di tabelle che vengono gestite in maniera coerente in modo che modifiche in una tabella (ad esempio dello stipendio di un impiegato) vengono riportate anche nelle altre tabelle interessate. Le liste anonime del Perl si prestano molto bene per rappresentare dati in questa forma. Consideriamo ad esempio una tabella che contiene alcuni paesi del Medio Oriente:

nome	*1000 km²*	*abitanti*	*lingua*
Iraq	*434*	*21800*	*arabo*
Israele	*21*	*6000*	*ebraico, arabo*
Giordania	*89*	*6300*	*arabo*
Iran	*1645*	*65800*	*persiano*
Arabia Saudita	*2248*	*20180*	*arabo*
Afghanistan	*652*	*21300*	*dari, pashto*
Emirati	*84*	*2353*	*arabo*
Egitto	*1001*	*66000*	*arabo*
Sudan	*2504*	*28300*	*arabo*

Possiamo rappresentare questa tabella mediante liste anonime ed estrarre i paesi di lingua araba o i paesi con più di un milione di km^2 di superficie nel modo seguente:

```
$paesi=[["Iraq",434,21800,"arabo"],
   ["Israele",21,6000,"ebraico,arabo"],
   ["Giordania",89,6300,"arabo"],
   ["Iran",1645,65800,"persiano"],
   ["Arabia Saudita",2248,20180,"arabo"],
   ["Afghanistan",652,21300,"dari,pashto"],
   ["Emirati",84,2353,"arabo"],
   ["Egitto",1001,66000,"arabo"],
   ["Sudan",2504,28300,"arabo"]];

sub arabi {my $p=shift; grep {$_->[3]=~/arabo/} @$p}

sub grandi {my $p=shift; grep {$_->[1]>=1000} @$p}

sub prova {for (arabi($paesi)) {print "$_->[0]\n"}
   print "\n"; for (grandi($paesi)) {print "$_->[0]\n"}}
```

Lettura dei dati da un file

Assumiamo che i dati siano contenuti in un file in semplice formato testo:

```
Iraq, 434, 21800, arabo
Israele, 21, 6000, ebraico:arabo
Giordania, 89, 6300, arabo
Iran, 1645, 65800, persiano
Arabia Saudita, 2248, 20180, arabo
Afghanistan, 652, 21300, dari:pashto
Emirati, 84, 2353, arabo
Egitto, 1001, 66000, arabo
Sudan, 2504, 28300, arabo
```

Ogni riga della tabella deve quindi stare su una riga del file, le colonne della tabella sono separate da virgole, una virgola nella tabella corrisponde a un doppio punto sul file. Usiamo la funzione *files::leggi* di pagina 15 per ottenere il contenuto del file come stringa che poi viene trasformata nella lista *$paesi* dell'esempio precedente.

```
sub tabelladafile {my $grezzi=files::leggi("Files/paesi");
  my @righe=split(/\n+/,$grezzi); my @paesi=();
  for (@righe) {@riga=split(/\s*,\s*/,$_);
  $riga[3]=~s/:/,/g; push(@paesi,[@riga])}
  \@paesi}

sub provafile {my $paesi=tabelladafile;
  for (arabi($paesi)) {print "$_->[0]\n"}
  print "\n"; for (grandi($paesi)) {print "$_->[0]\n"}}
```

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2001/2002 — Numero 11 ◇ 6 Dicembre 2001

Forth e PostScript

*Il **Forth** venne inventato all'inizio degli anni '60 da Charles Moore per piccoli compiti industriali, ad esempio il pilotaggio di un osservatorio astronomico. È allo stesso tempo un linguaggio semplicissimo e estremamente estendibile - dipende solo dalla pazienza del programmatore quanto voglia accrescere la biblioteca delle sue funzioni (o meglio macroistruzioni). Viene usato nel controllo automatico, nella programmazione di sistemi, nell'intelligenza artificiale. Un piccolo e ben funzionante interprete è **pfe**.*

*Uno stretto parente e discendente del Forth è il **PostScript**, un sofisticato linguaggio per stampanti che mediante un interprete (il più diffuso è **ghostscript**) può essere utilizzato anche come linguaggio di programmazione per altri scopi.*

Forth e PostScript presentano alcune caratteristiche che li distinguono da altri linguaggi di programmazione:

*(1) Utilizzano la notazione polacca inversa (RPN, reverse Polish notation) come alcune calcolatrici tascabili (della Hewlett Packard per esempio); ciò significa che gli argomenti precedono gli operatori. Invece di **a+b** si scrive ad esempio **a b +** (in PostScript **a b add**) e quindi **(a+3)*5+1** diventa **a 3 add 5 mul 1 add**. Ciò comporta una notevole velocità di esecuzione perché i valori vengono semplicemente prelevati da uno **stack** e quindi, benché interpretati, Forth e PostScript sono linguaggi veloci con codice sorgente molto breve.*

*(2) Entrambi i linguaggi permettono e favoriscono un uso estensivo di macroistruzioni (abbreviazioni) che nel PostScript possono essere addirittura organizzate su più dizionari (fornendo così una via alla programmazione orientata agli oggetti in questi linguaggi apparentemente quasi primitivi). Tranne pochi simboli speciali quasi tutte le lettere possono far parte dei nomi degli identificatori, quindi se ad esempio anche in PostScript volessimo usare **+** e * al posto di **add** e **mul** basta definire*

/+** {**add**} **def
/ {**mul**} **def***

(3) In pratica non esiste distinzione tra procedure e dati; tutti gli oggetti sono definiti mediante abbreviazioni e la programmazione acquisisce un carattere fortemente logico-semantico.

Sul sito di Adobe (www.adobe.com/) si trovano manuali e guide alla programmazione in PostScript.

Lo stack

Una **pila** (in inglese *stack*) è una delle più elementari e più importanti strutture di dati. Uno stack è una successione di dati in cui tutte le inserzioni, cancellazioni ed accessi avvengono a una sola estremità. Gli interpreti e compilatori di tutti i linguaggi di programmazione utilizzano uno o più stack per organizzare le chiamate annidate di funzioni; in questi casi lo stack contiene soprattutto gli indirizzi di ritorno, i valori di parametri che dopo un ritorno devono essere ripristinati, le variabili locali di una funzione.

Descriviamo brevemente l'esecuzione di un programma in PostScript (o Forth). Consideriamo ancora la sequenza

40 3 add 5 mul

Assumiamo che l'ultimo elemento dello stack degli operandi sia **x**. L'interprete incontra prima il numero 40 e lo mette sullo stack degli operandi, poi legge 3 e pone anche questo numero sullo stack (degli operandi, quando non specificato altrimenti). In questo momento il contenuto dello stack è **... x 40 3**. Successivamente l'interprete incontra l'operatore **add** che richiede due argomenti che l'interprete preleva dallo stack; adesso viene calcolata la somma 40+3=43 e posta sullo stack i cui ultimi elementi sono così **x 43**. L'interprete va avanti e trova **5** e lo pone sullo stack che contiene così **... x 43 5**. Poi trova di nuovo un operatore (**mul**), preleva i due argomenti necessari dallo stack su cui ripone il prodotto (43·5=215). Il contenuto dello stack degli operandi adesso è **... x 215**.

Questa settimana

36 Forth e PostScript
Lo stack
Programmare in Forth

37 Usare ghostscript
Il comando run di PostScript
Usare def
Diagrammi di flusso per lo stack
Lo stack dei dizionari

38 Argomenti di una macro
show e selectfont
if e ifelse
Cerchi con PostScript

Programmare in Forth

Qualche passo dal libro *Stack computers* di Philip Koopman:

"One of the characteristics of Forth is its very high use of subroutine calls. This promotes an unprecedented level of modularity, with approximately 10 instructions per procedure being the norm. Tied with this high degree of modularity is the interactive development environment used by Forth compilers ...

This interactive development of modular programs is widely claimed by experienced Forth programmers to result in a factor of 10 improvement in programmer productivity, with improved software quality and reduced maintenance costs ... Forth programs are usually quite small ...

Good programmers become exceptional when programming in Forth. Excellent programmers can become phenomenal. Mediocre programmers generate code that works, and bad programmers go back to programming in other languages. Forth ... is different enough from other programming languages that bad habits must be unlearned ... Once these new skills are acquired, though, it is a common experience to have Forth-based problem solving skills involving modularization and partitioning of programs actually improve a programmers effectiveness in other languages as well."

In ambiente Unix l'interattività tradizionale in Forth può essere sostituita con un'ancora più comoda organizzazione del programma su più files che come programma *script* (cfr. pag. 22), soprattutto se combinato con comandi di Emacs, diventa a sua volta praticamente interattivo.

Usare ghostscript

Sotto Unix si può battere **ghostscript** oppure semplicemente **gs** oppure meglio ad esempio **gs -g400x300** per avere una finestra di 400 x 300 pixel; sotto Windows cliccare sull'icona. Sotto Unix dovrebbe apparire una finestra per la grafica mentre sulla shell è attivo l'interprete che aspetta comandi. Proviamo prima a impostare alcuni comandi a mano (battere invio alla fine di ogni riga) facendo in modo che la finestra grafica rimanga visibile mentre battiamo i comandi:

0.05 ***setlinewidth***
33.3 33.3 ***scale***
0 0 ***moveto*** *5 4* ***lineto stroke***
/rosso {1 0 0 ***setrgbcolor****} def*
rosso 5 4 moveto 8 2 lineto stroke
/nero {0 0 0 setrgbcolor} def
nero 8 2 moveto 0 0 lineto stroke
/giallo {1 1 0 setrgbcolor} def
giallo 7 5 moveto 4 5 3 0 360 ***arc fill***
nero 7 5 moveto 4 5 3 0 360 arc stroke
rosso 6.5 5 moveto 4 5 2.5 0 360 arc fill
nero 6.5 5 moveto 4 5 2.5 0 360 arc stroke
/blu {0 0 1 setrgbcolor} def
blu 6 5 moveto 4 5 2 0 360 fill
blu 6 5 moveto 4 5 2 0 360 arc fill
nero 6 5 moveto 4 5 2 0 360 arc stroke
giallo 5.5 5 moveto 4 5 1.5 0 360 arc fill
nero 5.5 5 moveto 4 5 1.5 0 360 arc stroke
/verde {0 1 0 setrgbcolor} def
verde 4 5 moveto 8 6 lineto 7 7 lineto 4 5 lineto fill
nero 4 5 moveto 8 6 lineto 7 7 lineto 4 5 lineto stroke
quit

Per vedere una nuova pagina si usa **showpage**; questo comando è anche necessario se il file è destinato alla stampa.

Il comando run di PostScript

Invece di battere i comandi dalla tastiera li possiamo anche inserire in un file. Creiamo ad esempio una cartella **/home/sis/ps** e in essa un file **alfa** in cui trascriviamo i comandi dell'articolo precedente, tralasciando però il **quit** finale che serve solo per uscire dall'interprete e che non ci permetterebbe di osservare l'immagine.

A questo punto dopo aver aperto l'interprete possiamo battere **(alfa) run** se ci troviamo nella stessa cartella del file, altrimenti dobbiamo indicare il nome completo del file, quindi **(/home/sis/alfa) run**. Le parentesi tonde in PostScript servono per racchiudere una stringa e prendono quindi il posto delle virgolette in molti altri linguaggi. Sotto Windows si può usare ad esempio **(c:/sis/alfa) run** (con barre semplici) oppure **(c:\\sis\\alfa) run** (con backslash raddoppiati).

Il comando **run** può essere utilizzato anche all'interno di un file per chiedere l'esecuzione di altri files, tipicamente di qualche nostra raccolta di abbreviazioni.

Usare def

Le abbreviazioni vengono definite secondo la sintassi

/abbreviazione significato ***def***

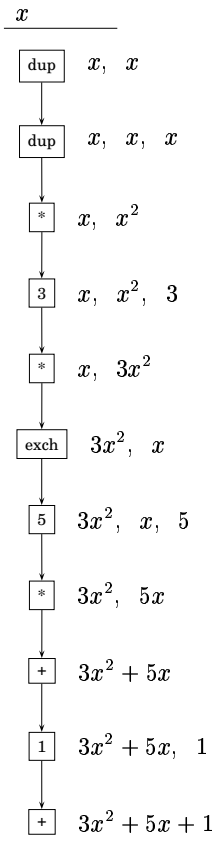
Se il significato è un operatore eseguibile bisogna racchiudere le operazioni tra parentesi graffe come abbiamo fatto sopra per i colori o a pagina 34 per *add* e *mul*, per impedire che vengano eseguite già la prima volta che l'interprete le incontra, cioè nel momento in cui legge l'abbreviazione.

Quando il significato invece non è eseguibile, non bisogna mettere parentesi graffe, ad esempio

/e 2.71828182845904523536 def
/pi 3.14159265358979323846 def

Diagrammi di flusso per lo stack

Vogliamo scrivere una macroistruzione per la funzione f definita da $f(x) = 3x^2 + 5x + 1$. Per tale compito in PostScript (e in Forth) si possono usare diagrammi di flusso per lo stack, simili ai diagrammi di flusso per altri linguaggi visti a pagina 14, dove però adesso indichiamo ogni volta gli elementi più a destra dello stack vicino ad ogni istruzione. Utilizziamo due nuove istruzioni: **dup**, che duplica l'ultimo elemento dello stack (vedremo subito perché), ed **exch** che scambia gli elementi più a destra dello stack. Assumiamo inoltre di avere definito gli operatori **+** e * invece di **mul** e **add** come a pagina 34. All'inizio l'ultimo elemento dello stack è x.



La macroistruzione che corrisponde a questo diagramma di flusso è

*/f {dup dup * 3 * exch 5 * + 1 +} def*

Lo stack dei dizionari

Il PostScript permette una gestione a mano di variabili locali mediante dizionari (*dictionaries*) che possono essere annidati perché organizzati tramite un apposito stack. Con

4 ***dict begin*** */x 1 def /y 2 def /z 3 def /w (Rossi) def*
...
end

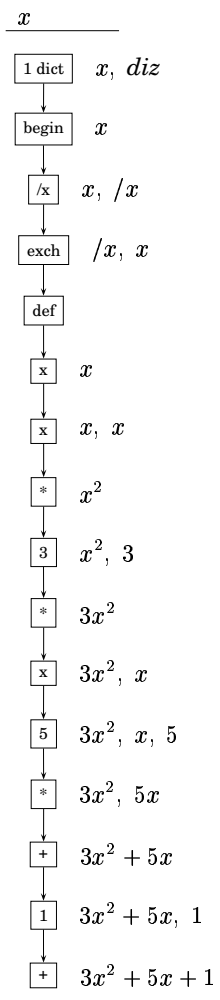
viene creato un dizionario di almeno 4 voci (il cui numero viene comunque automaticamente aumentato se vengono definite più voci). Tra *begin* e *end* tutte le abbreviazioni si riferiscono a questo dizionario se in esso si trova una tale voce (altrimenti il significato viene cercato nel prossimo dizionario sullo stack dei dizionari); con *end* perdono la loro validità.

Argomenti di una macro in PostScript

Consideriamo ancora la funzione $f(x) = 3x^2 + 5x + 1$. La macroistruzione che abbiamo trovato a pagina 35, benché breve, non è facilmente leggibile senza l'uso di un diagramma di flusso. L'uso di variabili locali mediante lo stack dei dizionari ci permette di ridefinire la funzione in un formato più familiare:

*/f {1 dict begin /x exch def x x * 3 * x 5 * + 1 + end} def*

Esaminiamo anche questa espressione mediante un diagramma di flusso.



L'istruzione *1 dict* crea un dizionario che prima viene posto sullo stack degli operandi; solo con il successivo *begin* il dizionario viene tolto dallo stack degli operandi e posto sullo stack dei dizionari. L'interprete adesso trova */x* e quindi in questo momento l'ultimo elemento dello stack è */x*, preceduto da *x*. Questi due elementi devono essere messi nell'ordine giusto mediante un *exch* per poter applicare il *def* che inserisce la nuova abbreviazione per *x* nell'ultimo dizionario dello stack degli operandi (cioè nel dizionario appena da noi creato) e toglie gli operandi dallo stack degli operandi. Ci si ricordi che *end* non termina l'espressione tra parentesi graffe ma chiude il *begin*, toglie cioè il dizionario dallo stack dei dizionari.

show e selectfont

Abbiamo già osservato a pagina 35 che il PostScript usa le parentesi tonde per raccogliere le stringhe. La visualizzazione avviene mediante il comando **show** che però deve essere preceduto dall'indicazione del punto dove la stringa deve apparire e del font. Il font viene definito come nel seguente esempio:

*/times-20 {/**Times-Roman** 20 scala **div selectfont**} def*

dopo aver definito la scala ad esempio con **/scala 33.3 def**. Adesso possiamo visualizzare una stringa:

times-20 1 8 moveto (Il cerchio si chiude) **show**

gsave 8 8 moveto (e) (d) (u) (i) (h) (c) ( ) (i) (s) ( )
(o) (i) (h) (c) (r) (e) (c) ( ) (l) (I) ( ) (*)
22 {show 360 23 **div neg rotate**} **repeat grestore**

Il comando **gsave** viene utilizzato per salvare le impostazioni grafiche prima di effettuare cambiamenti, ad esempio prima di una rotazione, mentre **grestore** ripristina il vecchio stato grafico – anche qui si usa uno stack!. **a b div** è i quoziente $\frac{a}{b}$, **a neg** è $-a$, mentre **t rotate** ruota il sistema di coordinate per t gradi in senso antiorario. **10** {**operazione**} **repeat** ripete un'operazione 10 volte. Provare con ghostscript!

---

if e ifelse

Illustriamo l'uso di *if* e *ifelse* con due esempi.

a m resto calcola il resto di a modulo m anche per $m < 0$ utilizzando la funzione **mod** del PostScript che dà il resto corretto invece solo per $m > 0$, **n fatt** è il fattoriale di n.

```
/resto {2 dict begin /m exch def /a exch def
  m 0 lt {/a a neg def} if a m mod end} def

/fatt {1 dict begin /n exch def
  n 0 eq {1} {n 1 - fatt n *} ifelse end} def
```

Per trasformare un numero in una stringa si può usare

*/stringa-numerica {20 **string cvs**} def*

Esempi da provare:

2 6 moveto 117 40 resto stringa-numerica show
3 6 moveto 8 fatt stringa-numerica show

Cerchi con PostScript

/cerchio {3 dict begin /r exch def /y exch def /x exch def
*gsave **newpath** x y r 0 360 **arc** stroke grestore end} def*

/cerchiopieno {3 dict begin /r exch def /y exch def /x exch def
*gsave newpath x y r 0 360 arc **fill** grestore end} def*

/rosetta {5 dict begin /f exch def /n exch def /y exch def
*/x exch def /alfa 360 n div def gsave x y **translate***
n {f alfa rotate} repeat grestore end} def

Provare adesso (dopo le solite impostazioni):

verde 3 3 2.3 cerchiopieno nero 3 3 2.3 cerchio
3 3 7 {giallo 1.5 0 0.5 cerchiopieno
nero 1.5 0 0.5 cerchio} rosetta
rosso 3 3 0.8 cerchiopieno nero 3 3 0.8 cerchio

Per cancellare una pagina si usa **erasepage**.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

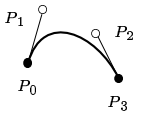
Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 12 ◇ 18 Dicember 2001

L'algoritmo di Casteljau

Come impareremo in questo numero, un arco di curva piana con rappresentazione parametrica polinomiale di terzo grado

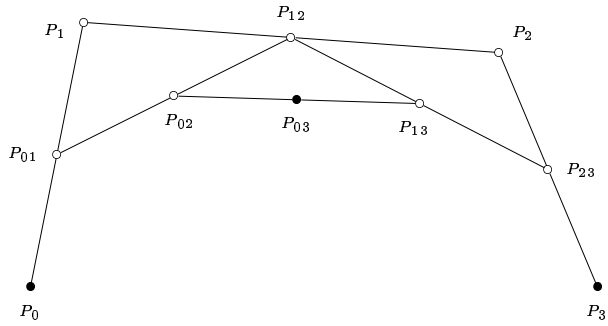
$$x = x(t) = a + bt + ct^2 + dt^3$$
$$y = y(t) = a' + b't + c't^2 + d't^3$$

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:

$$\begin{array}{lll} P_0 & & \\ & P_{01} = \frac{P_0+P_1}{2} & \\ P_1 & & P_{02} = \frac{P_{01}+P_{12}}{2} \\ & P_{12} = \frac{P_1+P_2}{2} & \qquad\qquad P_{03} = \frac{P_{02}+P_{13}}{2} \\ P_2 & & P_{13} = \frac{P_{12}+P_{23}}{2} \\ & P_{23} = \frac{P_2+P_3}{2} & \\ P_3 & & \end{array}$$

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

39 L'algoritmo di Casteljau
Disegnare con PostScript
40 Curve di Bézier cubiche
Invarianza affine
I punti di controllo
41 La parabola
Il cerchio
L'ellisse
42 L'iperbole
Alcuni sempi di curve di Bézier cubiche
Il file bezier.pm

Disegnare con PostScript

Abbiamo visto nel numero precedente 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:

$$\begin{array}{rcrcrcrcl} a & & & & & & & = & x_0 \\ & & b & & & & & = & \dot{x}_0 \\ a & + & b & + & c & + & d & = & x_3 \\ & & b & + & 2c & + & 3d & = & \dot{x}_3 \end{array}$$

da cui si trova facilmente

$$\begin{array}{rcl} a & = & x_0 \\ b & = & \dot{x}_0 \\ c & = & 3(x_3 - x_0) - 2\dot{x}_0 - \dot{x}_3 \\ d & = & 2(x_0 - x_3) + \dot{x}_0 + \dot{x}_3 \end{array}$$

e similmente

$$\begin{array}{rcl} a' & = & y_0 \\ b' & = & \dot{y}_0 \\ c' & = & 3(y_3 - y_0) - 2\dot{y}_0 - \dot{y}_3 \\ d' & = & 2(y_0 - y_3) + \dot{y}_0 + \dot{y}_3 \end{array}$$

In forma vettoriale la soluzione può essere scritta così:

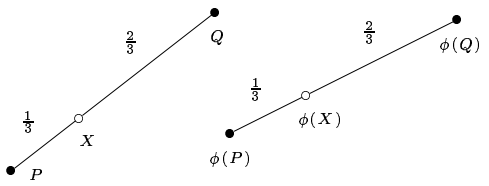
$$\begin{array}{rcl} \begin{pmatrix} a \\ a' \end{pmatrix} & = & P_0 \\ \begin{pmatrix} b \\ b' \end{pmatrix} & = & V_0 \\ \begin{pmatrix} c \\ c' \end{pmatrix} & = & 3(P_3 - P_0) - 2V_0 - V_3 \\ \begin{pmatrix} d \\ d' \end{pmatrix} & = & 2(P_0 - P_3) + V_0 + V_3 \end{array}$$

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 $t_0 = 0, t_1 = 1$):

$$\begin{array}{rcl} \begin{pmatrix} a \\ a' \end{pmatrix} & = & P_0 \\ \begin{pmatrix} b \\ b' \end{pmatrix} & = & 3(P_1 - P_0) \\ \begin{pmatrix} c \\ c' \end{pmatrix} & = & 3(P_0 - 2P_1 + P_2) \\ \begin{pmatrix} d \\ d' \end{pmatrix} & = & P_3 - P_0 + 3(P_1 - P_2) \end{array}$$

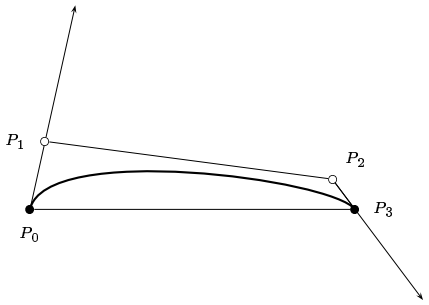
Invarianza affine

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

$$\begin{array}{rcl} P(t) & = & P_0 + 3(P_1 - P_0)t + 3(P_0 - 2P_1 + P_2)t^2 + \\ & & (P_3 - P_0 + 3(P_1 - P_2))t^3 \end{array}$$

che segue dall'osservazione in basso a sinistra. Perciò

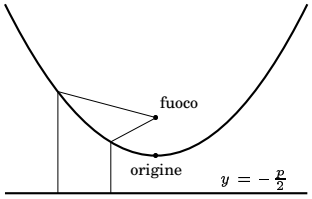
$$\begin{array}{rcl} P(\frac{1}{2}) & = & P_0 + \frac{3(P_1 - P_0)}{2} + \frac{3(P_0 - 2P_1 + P_2)}{4} + \frac{P_3 - P_0 + 3(P_1 - P_2)}{8} \\ & = & \frac{P_0 + 3P_1 + 3P_2 + P_3}{8} = P_{03} \end{array}$$

Esercizio 2: Dimostrare da soli che

$$\begin{array}{rcrcl} P_{01} & = & P_0 & + & \frac{1}{3}V_0 \\ P_{02} & = & P_{03} & - & \frac{1}{3}\dot{P}(\frac{1}{2}) \\ P_{13} & = & P_{03} & + & \frac{1}{3}\dot{P}(\frac{1}{2}) \\ P_{23} & = & P_3 & - & \frac{1}{3}V_3 \end{array}$$

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:

$$\begin{aligned} x_1 &= \tfrac{2t_0+t_1}{3} \\ x_2 &= \tfrac{t_0+2t_1}{3} \\ y_1 &= \tfrac{t_0}{2p}x_2 \\ y_2 &= \tfrac{t_1}{2p}x_1 \end{aligned}$$

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:

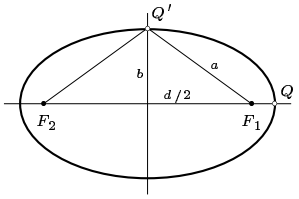
```
300 200 translate 20 20 scale 0.1 setlinewidth
2.25 5.75 translate 45 rotate
-1 0.707 moveto 0 -0.707 1 0 2 2.828 curveto stroke
-45 rotate -2.25 -5.75 translate
```

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 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 $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$.

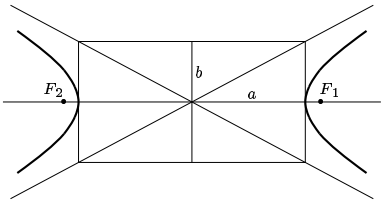
Ci sono due modi per disegnare un'ellisse con PostScript: la si può ottenere come trasformazione affine di un cerchio oppure approssimarla con curve di Bézier cubiche come nell'esercizio che segue.

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

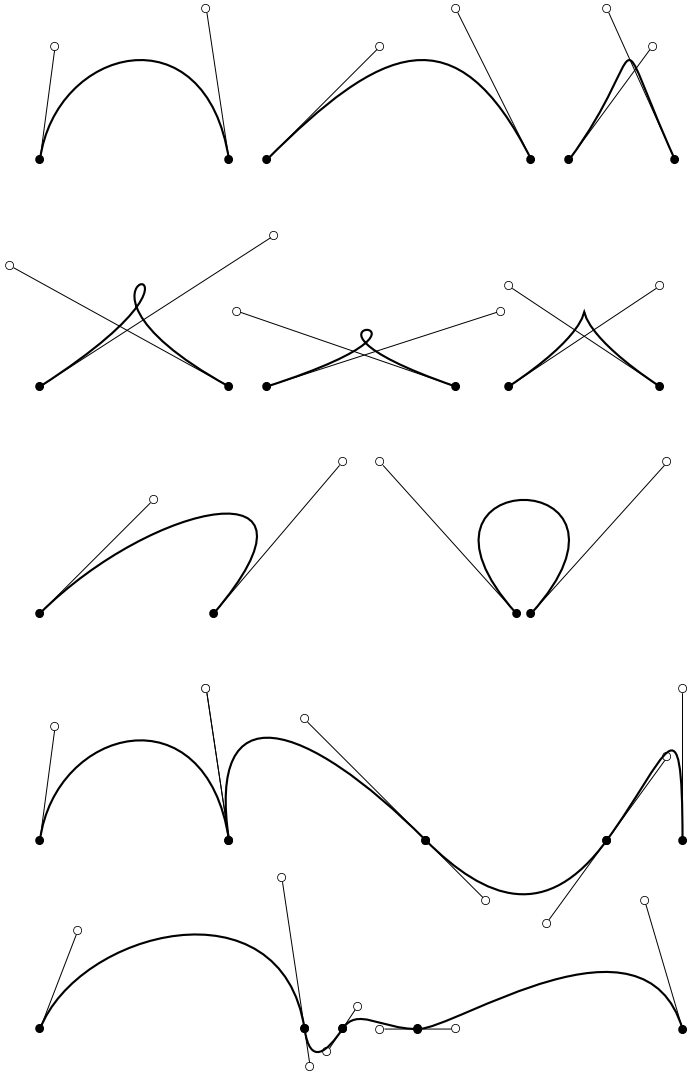
Un'iperbole è data da due punti distinti F_1 e F_2, detti *fuochi* e un valore s minore della distanza d tra i fuochi e maggiore di 0, e consiste di tutti i punti del piano per i quali, se con r_1 denotiamo la loro distanza da F_1 e con r_2 la distanza da F_2, si ha $r_2 - r_1 = s$ oppure $r_1 - r_2 = s$. 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



Il file bezier.pm

Questo file contiene un programma in Perl per la costruzione di curve di Bézier cubiche mediante l'algoritmo di Casteljau. I calcoli vengono effettuati in Perl e i risultati trascritti in un file PostScript (**casteljau.ps**) che risulta di circa 10 K. È solo un esperimento perché il PostScript contiene sue funzioni per le curve di Bézier (ad es. *curveto*, cfr. pag. 39).

```
1; # bezier.pm
use files;
sub provabezier {my ($P0,$P1,$P2,$P3)=
    ([-35,0],[-20,20],[24,36],[35,0]);
   $epsilon=0.5; # variabile globale
   my $a=sprintf("2.83 2.83 scale ".
    "100 142 translate 0.1 setlinewidth\n".
    "%.3f %.3f %.3f %.3f rectfill\n".
    "%.3f %.3f %.3f %.3f rectfill\n",
    $P0→[0]-$epsilon/2,$P0→[1]-$epsilon/2,
    $epsilon,$epsilon,
    $P3→[0],$P3→[1],$epsilon,$epsilon);
   files::scrivi("casteljau.ps",$a);
   casteljau($P0,$P1,$P2,$P3);
   files::aggiungi("casteljau.ps","showpage\n");}
####################################
sub casteljau {my ($P0,$P1,$P2,$P3)=@_;
   return if distl1($P0,$P3)<$epsilon;
   my $P01=mezzo($P0,$P1);
   my $P12=mezzo($P1,$P2);
   my $P23=mezzo($P2,$P3);
   my $P02=mezzo($P01,$P12);
   my $P13=mezzo($P12,$P23);
   my $P03=mezzo($P02,$P13);
   my $a=sprintf("%.3f %.3f %.3f %.3f rectfill\n",
    $P03→[0]-$epsilon/2,$P03→[1]-$epsilon/2,
    $epsilon,$epsilon);
   files::aggiungi("casteljau.ps",$a);
   casteljau($P0,$P01,$P02,$P03);
   casteljau($P03,$P13,$P23,$P3)}
sub distl1 {my ($P,$Q)=@_;
   abs($P→[0]-$Q→[0])+abs($P→[1]-$Q→[1])}
sub mezzo {my ($P,$Q)=@_;
   [($P→[0]+$Q→[0])/2,($P→[1]+$Q→[1])/2]}
```

Queste funzioni applicano l'algoritmo di Casteljau con una risoluzione impostata a ε = 0.5 mm (secondo la scala di misura che provabezier imposta all'inizio) e rappresenta i punti ottenuti con piccoli rettangoli di lato ε. Abbiamo usato le funzioni *files::scrivi* e *files::aggiungi* del modulo *files.pm* di pagina 15. La funzione *sprintf* è stata descritta a pagina 30.

Come distanza tra due punti del piano (a cui corrisponde la nostra funzione *distl1*) abbiamo usato la distanza L_1, definita da $\|x - y\|_1 = |x_1 - y_1| + |x_2 - y_2|$.

Il valore assoluto viene calcolato dalla funzione *abs* del Perl.

L'immagine che si ottiene è molto soddisfacente e potrebbe essere facilmente migliorata cambiando ε.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 13 ◇ 7 Marzo 2002

Programmare in C

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 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? 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 e i risultati 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, sotto Unix, automaticamente dal compilatore viene assegnato il diritto di esecuzione) dal linker (cfr. pag. 10).*

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, 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 file 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 dichiarazione 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. Stringhe nel C sono racchiuse tra virgolette, i caratteri tra apostrofi.

Questa settimana

43 Programmare in C
Il programma minimo
I header generali

44 Il preprocessore
Comandi di compilazione
I commenti
Calcoliamo il fattoriale

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**> (qui *io* sta per *input/output*, mentre *std* è sempre abbreviazione per *standard*).

I header necessari per le singole funzioni sono indicati in molti libri di testo sulla programmazione in C.

Le funzioni per l'allocazione di memoria (**malloc** ecc.) richiedono il header <**stdlib.h**> che riguarda anche alcune funzioni matematiche e per le stringhe non incluse in <**math.h**> e <**string.h**>.

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 del file 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 più semplici sono 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.

Comandi di compilazione

Se aggiungiamo la funzione per il fattoriale che è definita nella colonna a destra, il nostro progetto consiste di due files sorgente (**alfa.c** e **matematica.c**). Possiamo adesso effettuare la compilazione battendo dalla shell i seguenti comandi, dopo aver creato una cartella **Oggetti** che conterrà i files oggetto affinché non affollino la directory del progetto:

```
gcc -o Oggetti/alfa.o -c alfa.c
gcc -o Oggetti/matematica.o -c matematica.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 la riga successiva. 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.

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);
```

L'istruzione **printf** del C è praticamente identica a quella del Perl (pag. 30). Anche il C possiede un'istruzione **sprintf** su cui torneremo più avanti.

Potremmo a questo punto battere dalla tastiera i comandi di compilazione oppure raccoglierli in un **makefile** come impareremo nella prossima lezione.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2001/2002 — Numero 14 ◇ 12 Marzo 2002

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, *'\''* 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

45 Vettori e puntatori
Stringhe e caratteri
Aritmetica dei puntatori
Operatori abbreviati

46 Confronto di stringhe
if ... else
Puntatori generici
Conversioni di tipo

47 Il makefile
Il comando make
I blocchi elementari

Aritmetica dei puntatori

Puntatori vengono spesso usati come variabili 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 (o intero 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

Come in Perl (cfr. gli esempi alle pagine 14-15, 19, 32-33), 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 e comoda notazione, ad esempio *resistenza/=2;* e più breve e più leggibile di *resistenza=resistenza/2;*.

++ e -- si usano come in Perl (esempi alle pagine 16, 18-19, 26, 30-32). In *f(x++)* viene prima eseguita *f(x)* e successivamente aumentata la *x*, in *f(++x)* il valore di *x* aumentato di *1* prima dell'esecuzione di *f*.

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 (passaggio di parametri per valore, cfr. pag. 26 per il Perl) 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;}
```

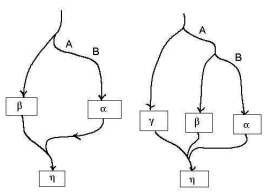
La differenza tra **uis** e **us** non è grande. In cosa consiste? Giustificare l'algoritmo. Anche qui potremmo usare le funzioni della libreria standard:

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

if ... else

L'*if* viene usato come nel Perl (pag. 16); nel C però non esiste *elsif* e non sono necessarie le parentesi graffe per blocchi che consistono di una singola istruzione.

```
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.

Il makefile

Per non dover battere ogni volta i comandi di compilazione dalla tastiera, li scriviamo in un file apposito, un cosiddetto *makefile* che verrà poi utilizzato come descritto in seguito su questa pagina.

```
# Makefile
librerie = -L/usr/X11R6/lib -lX11 -lm -lc
VPATH=Oggetti
progetto: alfa.o matematica.o
TAB gcc -o alfa Oggetti/*.o $(librerie)
%.o: %.c alfa.h
TAB gcc -o Oggetti/$*.o -c $*.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 il tasto tabulatore; non può qui essere sostituito da spazi. 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.

Non dimenticare di creare la directory **Oggetti**.

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.

Esaminiamo il nostro makefile. 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.

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*.

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.

I blocchi elementari

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 se 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 comprende adesso ricorsivamente le seguenti operazioni:

(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 un file **Makefile** così composto:

```
# Prove per capire il makefile
.SILENT:
primobersaglio : a b c
TAB echo io primobersaglio
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** 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.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 15 ◇ 19 Marzo 2002

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. 46. 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. 45), senza nessun 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++. Per il Perl cfr. pag. 26.

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 che in C (a differenza dal Perl) 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

48 Passaggio di parametri
Operazioni aritmetiche
Input da tastiera
scanf

49 Altre funzioni per le stringhe
for
Tipi di interi
I numeri binomiali

50 Parametri di main
Operatori logici del C
Un semplice menu
static e extern

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 non è 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;}
```

***

scanf

La funzione **scanf** 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.

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. 46.

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. 48. Con *break* si esce dal *for* come spiegato a lato. 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;}
```

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. il penultimo *for* in *invertiparola*); 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) {γ1; if (B) break; γ2;}
```

è equivalente a

```
ciclo: if (A) {γ1; if (B) goto fuori; γ2; β; goto ciclo;}
fuori:
```

mentre

```
for (;;β) {γ1; if (B) continue; γ2;}
```

è equivalente a

```
ciclo: γ1; if (!B) γ2; β; goto ciclo;
```

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

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 = $2^{32} - 1$.

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.

Parametri di main

Ogni progetto deve contenere esattamente una funzione **main**, che sarà la prima funzione ad essere eseguita. Essa viene 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 *na* 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**);*.

Come in Perl 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 usato 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 46, 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*!.

Operatori logici del C

Le espressioni booleane del C devono avere un valore numerico; ogni numero diverso da 0 è vero, il numero 0 è falso.

Per la congiunzione logica (AND) in C viene usato l'operatore *&&*, per la disgiunzione (OR) l'operatore | | . 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*.

Le differenze con il Perl (cfr. pag. 16) sono minime, soltanto che il Perl prevede anche gli operatori *and* e *or* che non esistono nel C.

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**?

---

static e extern

Variabili e funzioni dello stesso nome in files diversi devono essere dichiarate **static**, 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.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 16 ◇ 9 Aprile 2002

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);
```

Questa settimana

51 Funzioni per i files
fseek e tell
system

52 I numeri di Fibonacci in C
Il metodo del sistema di primo ordine
struct e typedef
Variabili di classe static

53 sprintf
Copiare una stringa
Allocazione di memoria in C

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, size_t 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 *α)*

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; size_t n;
File=fopen(A,"r");
if (File==0) return 0;
fseek(File,0,SEEK_END);
n=ftell(File); fclose(File); return n;}
```

***

system

Come in Perl (pagg. 9 e 31) 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("rm -i lettera");
system("ls -l > alfa");
```

I numeri di Fibonacci in C

Un programma iterativo in C per calcolare l'n-esimo numero di Fibonacci (pag. 18):

```
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;}}
```

Come abbiamo già visto, risulterebbe molto inefficiente la seguente funzione ricorsiva di secondo ordine:

```
double fib2 (int n)
{if (n<=1) return 1;
return fib2(n-1)+fib2(n-2);}
```

Il metodo del sistema di primo ordine

Come a pagina 18 poniamo
$x_n := F_n, y_n := F_{n-1}$,
ottenendo il sistema

$$x_{n+1} = x_n + y_n$$
$$y_{n+1} = x_n$$

per $n >= 0$ con le condizioni iniziali $x_0 = 1, y_0 = 0$. 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 (sotto) oppure più semplicemente definire una funzione con due argomenti in più che vengono modificati dalla funzione come spiegato a pag. 48:

```
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;}}
```

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.

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.

sprintf

Come in Perl (pag. 30), la funzione **sprintf** è molto simile nella sintassi alla funzione **printf**, da cui in Csi 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.

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:

L'istruzione *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*).

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 17 ◇ 18 Aprile 2002

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

54 Ottimizzazione genetica
Problemi di ottimizzazione
Il problema degli orari

55 L'algoritmo di base
Il metodo spartano
Un'osservazione importante
Sul significato degli incroci
Confronto con i metodi classici

56 Il quicksort
La mediana

57 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. 56).

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 aprossimano 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. 57) 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);
```

con

```
int minoreperpuntatori (double *A, double *B)
{return (*A<*B);}
```

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	^
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 2001/2002 | Numero 18 ◇ 30 Aprile 2002

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 $(x_1, \ldots, x_N)$ 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

58 Numeri casuali
Numeri casuali in crittografia
Una funzione di cronometraggio

59 La discrepanza
Integrali multidimensionali
Il generatore lineare

60 La struttura reticolare
Numeri casuali in C
switch
Il tipo enum

61 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,"si")) 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 $(x_1, \dots, x_N)$ di numeri reali. Denotiamo con $\{a\}$ la parte frazionaria di un numero reale a e definiamo

$$D_N(x_1, \dots, x_N) := \sup_{0 \le u < v \le 1} |\tfrac{1}{N} \sum_{n=1}^{N} (u \le \{x_n\} < v) - (v - u)|$$

$$D_N^*(x_1, \dots, x_N) := \sup_{0 < v \le 1} |\tfrac{1}{N} \sum_{n=1}^{N} (0 \le \{x_n\} < v) - v|$$

$D_N(x_1, \dots, x_N)$ si chiama la **discrepanza** della successione data, $D_N^*(x_1, \dots, x_N)$ la sua ***-discrepanza**.

Una successione infinita $(x_1, x_2, \dots)$ di numeri reali si chiama **uniformemente distribuita** se è verificata una delle seguenti due condizioni (di cui si dimostra facilmente l'equivalenza):

(1) $\lim_{N \to \infty} D_N(x_1, \dots, x_N) = 0.$
(2) $\lim_{N \to \infty} D_N^*(x_1, \dots, x_N) = 0.$

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 $x_1, \dots, x_N$ in $[0, 1)$ si ha

$$|\tfrac{1}{N} \sum_{n=1}^{N} f(x_n) - \int_0^1 f(u)du| \le V(f) D_N^*(x_1, \dots, x_N)$$

(*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 $D^*(x_1, \dots, x_N)$ 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 $x_1, x_2, \dots$ 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 $x_1, x_2, \dots$ 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 $\mathrm{mcd}(a, m) = 1$.

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 $\{x_0, x_1, \dots\}$ 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 $\mathrm{mcd}(a, m) = 1$, ciò implica $x_{r-1} = x_{s-1}$.

Corollario: La successione $x_0, x_1, \dots$ 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 $x_0, \dots, x_{m-1}$ 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.

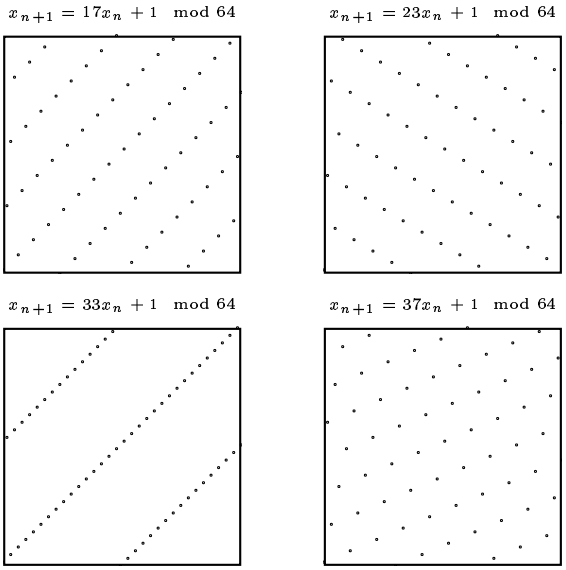
Teorema: Sia $m = 2^k \ge 4$. Allora la successione ha periodo massimale se e solo se $a \in 4\mathbb{Z} + 1$ e b è dispari.

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 $\mathrm{mcd}(m, b) = 1$.

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 $\mathrm{mcd}(m, b) = 1$.

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.



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. 59 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. Vedere pag. 30 per il Perl.

La funzione **rand** restituisce un numero intero pseudocasuale x con $0 \le x \le$ 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*:

(α, β, γ, u) è un valore, che è uguale al valore di u dopo esecuzione, nell'ordine indicato, di α, β e γ. Quindi *x=(α, β, γ, u)* è equivalente con *α; β; γ; 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 $\Rightarrow \mathrm{mcd}(a,b) = 2\,\mathrm{mcd}(\frac{a}{2}, \frac{b}{2})$.
a pari e b dispari $\Rightarrow \mathrm{mcd}(a,b) = \mathrm{mcd}(\frac{a}{2}, b)$.
a dispari e b pari $\Rightarrow \mathrm{mcd}(a,b) = \mathrm{mcd}(a, \frac{b}{2})$.
a, b dispari $\Rightarrow \mathrm{mcd}(a,b) = \mathrm{mcd}(\frac{a-b}{2}, b)$.
$a \geq 0 \Rightarrow \mathrm{mcd}(a,0) = a$.
$b \geq 0 \Rightarrow \mathrm{mcd}(0,b) = b$.
$\mathrm{mcd}(-a,b) = \mathrm{mcd}(a,b)$.
$\mathrm{mcd}(a,-b) = \mathrm{mcd}(a,b)$.

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 2001/2002 | Numero 19 ◇ 30 Aprile 2002

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

62 Cluster analysis
Il criterio della varianza

63 Il numero delle partizioni
Quindici comuni: i dati grezzi

64 Quindici comuni:
la trasformazione dei dati
Le funzioni del C per le stringhe
strlen, strcat e strncat

65 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. 63).

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 $\{1,\ldots,n\}$ può contenere $\{n\}$ come elemento (in tal caso n è equivalente solo a se stesso) oppure no.

Il numero delle partizioni di $\{1,\ldots,n\}$ in k classi di cui una coincide con $\{n\}$ è evidentemente uguale al numero delle partizioni di $\{1,\ldots,n-1\}$ in $k-1$ classi, cioè uguale a $S(n-1,k-1)$.

Se una partizione di $\{1,\ldots,n\}$ con k classi non contiene $\{n\}$ come elemento, essa si ottiene da una partizione di $\{1,\ldots,n-1\}$ 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
$S(20,5) = 749206090500$,
$S(50,4) = 52818655359845226611906445312$.

Tabella degli $S(n,k)$ per $n,k \leq 10$ (con - al posto di 0):

n

$\downarrow k$	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:

$$S(n,k) = \frac{k^n - \binom{k}{1}(k-1)^n + \binom{k}{2}(k-2)^n - + \ldots}{k!}$$

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.

Il comune di 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. Oltre a Ravenna e Venezia abbiamo trovato solo questi comuni italiani più grandi di Ferrara (sup. in km^2): Roma 1508, Foggia 506, Grosseto 475, L'Aquila 467, Perugia 450, Altamura 428, Caltanissetta 416, Viterbo 406.

	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

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. 63) 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. 46). 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. 46. 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. Attenzione all'uso: 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 errore 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*.

Attenzione: 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. 46.

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. 46 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. 64; **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
```

Esercizio: 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. 64. **strcspn** funziona in modo complementare a **strspn**, più precisamente **strcspn(A,L)** è la lunghezza del segmento iniziale di A che non 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 e.a.: Genetic programming. 1998.

D. Corne: Applied evolutionary computation. Springer 2000.

D. Goldberg: Genetic algorithms in search, optimization, and machine learning. Addison-Wesley 1989.

J. Holland: Adaptation in natural and artificial systems. MIT Press 1992.

C. Jacob: Principia evolvica. dpunkt 1997.

J. Koza: Genetic programming. MIT Press 1992.

M. Mitchell: An introduction to genetic algorithms. MIT Press 1999.

V. Nissen: Einführung in evolutionäre Algorithmen. Vieweg 1997.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica — Anno accademico 2001/2002 — Numero 20 ◇ 7 Maggio 2002

Quindici comuni: l'algoritmo genetico

Utilizziamo l'algoritmo di base presentato a pag. 55 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. 57.*

Per i quindici comuni a pag. 63 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. 64) 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

66 15 comuni: l'algoritmo genetico
15 comuni: l'interfaccia utente
Il calcolo di $\sum_{\alpha\in P}\Delta\alpha$
67 Partizioni proposte dall'utente
Dichiarazioni in cluster.c
Organizzazione dei dati sul file
Visualizzazione della partizione migliore
68 Elementi nuovi, mutazioni e incroci
La costruzione della graduatoria
Incroci tra più di due individui
69 Un problema di colorazione
strtok
70 cluster.c
A→b abbreviazione di (*A).b
%* in printf

Il calcolo di $\sum_{\alpha\in P}\Delta\alpha$

Ricordiamo che $\Delta\alpha := \sum_{x\in\alpha}|x-\bar\alpha|^2$ (pag. 62). 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 $\rightarrow$ per i puntatori è spiegato a pag. 70.

```
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. 63 e confrontandola con il risultato che abbiamo ottenuto a pag. 66, 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. 63-64, 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. 57) 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. 55). **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. 55: 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. 70.

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:

$$
\begin{aligned}
a_1 &= (a_{10}, a_{11}, a_{12}, \ldots)\\
a_2 &= (a_{20}, a_{21}, a_{22}, \ldots)\\
a_3 &= (a_{30}, a_{31}, a_{32}, \ldots)\\
a_4 &= (a_{40}, a_{41}, a_{42}, \ldots)
\end{aligned}
$$

Gli oggetti ottenuti mediante gli incroci saranno codificati nel modo seguente:

$$
\begin{aligned}
b_1 &= (b_{10}, b_{11}, b_{12}, \ldots)\\
b_2 &= (b_{20}, b_{21}, b_{22}, \ldots)\\
b_3 &= (b_{30}, b_{31}, b_{32}, \ldots)\\
b_4 &= (b_{40}, b_{41}, b_{42}, \ldots)
\end{aligned}
$$

Adesso scegliamo a caso una delle 24 permutazioni, ad esempio $\tau_0 = \sigma_{\mathrm{dado}(24)}$ e definiamo

$$
\begin{aligned}
b_{10} &= a_{\tau_0(1)0}\\
b_{20} &= a_{\tau_0(2)0}\\
b_{30} &= a_{\tau_0(3)0}\\
b_{40} &= a_{\tau_0(4)0}
\end{aligned}
$$

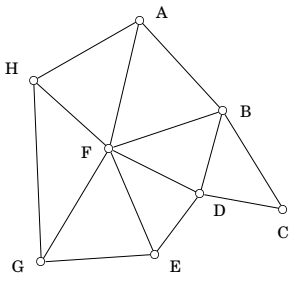
Mediante un'altra permutazione casuale τ_1 troviamo le componenti $b_{11}, b_{21}, b_{31}, b_{41}$ ecc. Abbiamo usato lo stesso metodo, ma per incroci tra due soli individui, nella funzione **incroci** per i comuni.

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. 64).

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 modificata 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. 52). 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.

%* in printf

A pag. 66 abbiamo usato un'istruzione della forma

```
printf("%*.0f ",n,x);
```

in cui il valore di *n* viene inserito al posto dell'asterisco all'interno del termine di formato definito da %.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 21 ◇ 14 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. 47, 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);}
```

Classi

Una **classe** nel C++ è simile a una struttura del C (pag. 52). 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) {...}
```

Questa settimana

71 Programmazione in C++
Classi
Costruttori e distruttori

72 this
Overloading di funzioni
Classi derivate
Overloading di operatori
Unioni

73 Numeri complessi
Il valore assoluto
Il quoziente di numeri complessi
La radice complessa
Funzioni trigonometriche

74 Riferimenti
new e delete
Libri sul C++
La funzione crypt
Sniffing e spoofing

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. 73).

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. 73.

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. 71 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:

```
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:

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	–	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	*=
<<=	>>=	[]	()	->	->*	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 private.

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

$$\begin{aligned}
e^{z} &= e^{x+iy} &&= e^{x}(\cos y + i \sin y)\\
e^{iz} &= e^{-y+ix} &&= e^{-y}(\cos x + i \sin x)\\
e^{-iz} &= e^{y-ix} &&= e^{y}(\cos x - i \sin x)\\
\cos z &= \tfrac{e^{iz}+e^{-iz}}{2} &&= \tfrac{e^{y}+e^{-y}}{2}\cos x - i\tfrac{e^{y}-e^{-y}}{2}\sin x\\
\sin z &= \tfrac{e^{iz}-e^{-iz}}{2i} &&= \tfrac{e^{y}+e^{-y}}{2}\sin x + i\tfrac{e^{y}-e^{-y}}{2}\cos x
\end{aligned}$$

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. 48). 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++.

La funzione crypt

Abbiamo già osservato a pag. 4 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 in Perl:

```
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 possiamo fare una prova con

```
static void provacrypt()
{printf("%s\n",crypt("toroseduto","Bk"));}
```

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.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 22 ◇ 21 Maggio 2002

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

75 Processi
Il fork
Il PID

76 exit e wait
Esecuzione in background
I comandi exec

77 Esempi di comandi exec
fork e exec
environ e getenv
Terminare un processo

78 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. 4. 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. 51 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

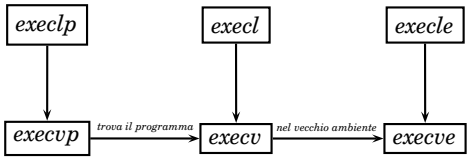
```
int execl (const char*, const char*, ..., 0);
int execlp (const char*, const char*, ..., 0);
int execv (const char*, char **);
int execvp (const char*, char**);
int execle (const char*, const char*, ..., 0);
int execve (const char*, char**, char**);
```

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. 50). Quindi la chiamata *execlp(α,α,β,γ,δ,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 atan2(y, x) è uguale al valore principale di 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 $\alpha =$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

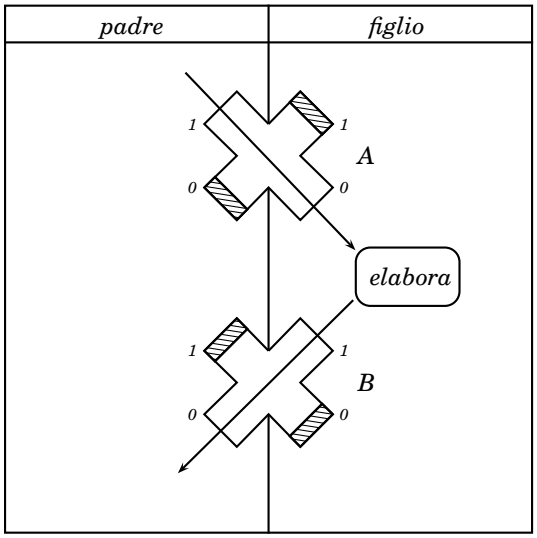
int tolower (int x);
int toupper (int x);
come abbiamo già fatto nelle funzioni *invertiparola* (pag. 49) e *modificafile* (pag. 51).

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

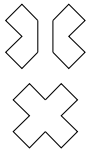
Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 23 ◇ 28 Maggio 2002

Pipelines

Abbiamo visto a pag. 8 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. 8), mentre le funzioni del ISO C, ad esempio *fopen*, *fclose*, *getc*, *putc*, *fread* e *fwrite*, di cui abbiamo incontrato le prime quattro a pag. 51, 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

79 Pipelines
read e write
pipe, close e dup2

80 pipemail
pipemaills
Sull'uso delle pipelines

81 Un piccolo filtro
Trasferimento in entrambe le direzioni
Operazioni sui bytes in memoria

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.

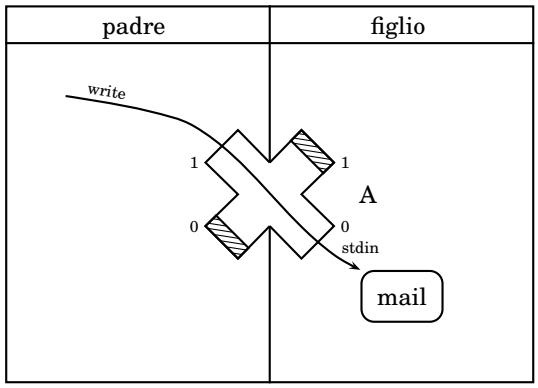
La funzione *close* è una funzione generale che viene usata per chiudere una via di comunicazione qualsiasi identificata da un *file descriptor*, che può essere anche utilizzata per chiudere, separatamente, le due vie di un incrocio, quindi ci saranno due istruzioni di chiusura, semplicemente *close(A[0]);* 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. 8). 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. 77.

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);}}
```

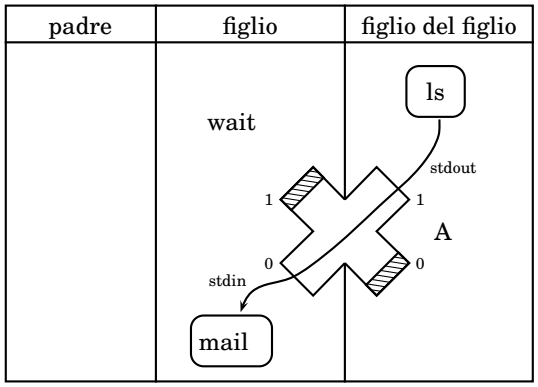
Il programma (che mettiamo in un nuovo file **pipe.c**) non è difficile e vale la pena provarlo, ad esempio con l'istruzione

```
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 sostituire 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);
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;}
```

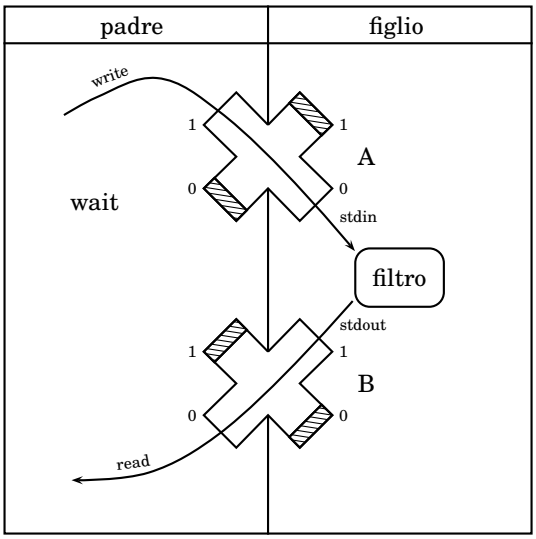
Per la compilazione usiamo il seguente semplice makefile:

```
# 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. 57.

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.

SISTEMI di ELABORAZIONE dell'INFORMAZIONE

Corso di laurea in matematica | Anno accademico 2001/2002 | 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

82 Sicurezza dei dati
Locali e apparecchiature
La shell protetta SSH

83 Sicurezza in rete
Autenticazione

84 Sicurezza dei dati in medicina
I principi di Anderson

85 Il teorema di Fermat-Euler
Crittografia a chiave pubblica
La steganografia

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 (*Linux - massima sicurezza, Apogeo 2000*) 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 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 (pag. 82) 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.

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.

Teorema 1: 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 $\phi(p) = p - 1$.

Corollario: p e q siano primi distinti e $n := pq$. Allora $\phi(n) = (p-1)(q-1)$.

Sempre dall'algebra sappiamo che in ogni gruppo G vale $g^{|G|} = 1$ 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.

Teorema 2: 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 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 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.

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 e 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).