Premessa per gli studenti.

Le lezioni di laboratorio vertono sulla *programmazione di applicazioni che comunicano su una interrete* ed, in particolare, progettate in accordo al *paradigma client-server*.

Questi appunti raccolgono gli argomenti presentati. In forma sintetica descrivono, principalmente relativamente all'uso dei protocolli dello stato di trasporto TCP e UDP, *tecniche* e *strumenti* utilizzabili per scrivere *agevolmente* applicazioni *efficienti* e *robuste*.

Il materiale è rinvenibile quasi totalmente nel testo di riferimento: *Unix: Network Programming. The Socket Networking API*, terza edizione, di W. R. Stevens, B. Fenner, e A. Rudolf, edito da Addison-Wesley. Alcuni richiami teorici sono tratti da *Reti di Calcolatori* di B. Forouzan, edito da McGraw-Hill, testo adottato per la parte teorica del corso.

Essendo le lezioni rivolte a studenti del secondo anno, che hanno mediamente poca familiarità con l'Inglese e non posseggono una piena maturità nell'ambito della programmazione di sistema, mi è sembrato utile raccogliere in forma scritta un sottoinsieme di concetti ed esempi del testo, che fosse facilmente accessibile a tutti gli studenti volenterosi. A tale scopo, ho seguito la trattazione che negli anni passati ha tenuto in laboratorio il prof. R. De Prisco, utilizzando le sue slide come riferimento per la presentazione, la suddivisione del materiale e per molte delle figure riportate. Naturalmente errori, imprecisioni e punti poco chiari sono soltanto opera mia :)

Invito ad usare gli appunti per *rivedere* gli argomenti trattati a lezione, e ad utilizzare il libro di testo per *approfondire e cogliere dettagli e sfumature* non discussi a lezione e non presenti in questa esposizione semplificata. La lettura anche parziale del libro di testo, che è *il riferimento* per eccellenza in materia, è estremamente piacevole, e può essere una buona occasione per migliorare la padronanza dell'Inglese tecnico.

E, naturalmente, invito a fare molta pratica, unico modo per comprendere a fondo le dinamiche delle comunicazioni su una interrete.

Draft 1.0

Laboratorio di Reti di Calcolatori

Obiettivi, ambiente di sviluppo e tool di base.

Paolo D'Arco

Abstract

Scopo della lezione è definire gli obiettivi del laboratorio, illustrare brevemente l'ambiente di sviluppo e presentare gli strumenti di base che verranno usati.

1 Per cominciare ...

L'obiettivo di queste lezioni, che completano il corso di Reti di Calcolatori, è di fornirvi i rudimenti per scrivere programmi che *comunicano* su una rete di calcolatori.

1.1 Ambiente

Sistema operativo, Linguaggio C e Socket API. Si assume che abbiate già seguito un corso introduttivo alla programmazione e ai sistemi operativi, e che abbiate, quindi, familiarità con un sistema operativo Unix-like e con il linguaggio C. Infatti, il nostro ambiente di sviluppo è costituito essenzialmente da tre elementi:

- Sistema operativo Linux
- Compilatore per il linguaggio C
- Socket API

I primi due elementi, dicevamo, vi sono più o meno noti. Il terzo, la Socket API, cioè l' *interfaccia alla programmazione dei socket*, consiste in una *collezione di funzioni e strutture di dati* che permette al programmatore di scrivere programmi che comunicano su una rete di calcolatori. Tramite le funzioni di questa interfaccia due (o più) programmi possono inviare e ricevere messaggi, e continuare la propria esecuzione in una direzione piuttosto che in un'altra a seconda dei messaggi ricevuti, di eventi che si sono verificati nel frattempo nel sistema e di eventuali condizioni di errore. Buona parte del nostro tempo sarà dedicata allo studio di questa interfaccia.

Nello sviluppo dei nostri programmi non utilizzeremo il desktop grafico del sistema operativo. Useremo, invece, *terminali testuali, shell* ed *editor di testo*.

1.2 Comandi ed Editing

Terminale Testuale. Un terminale testuale è una semplice finestra che permette all'utente di comunicare tramite *linee di comando*. Il terminale presenta all'utente nell'angolo in alto a sinistra un

simbolo (per esempio >, o % oppure $) detto *prompt*, preceduto da una stringa (personalizzabile) che spesso indica la directory di lavoro corrente. Comandi tipici che un terminale testuale Linux accetta sono

- *ls, cd, pwd, cp, rm, mv, cat, mkdir ...*

Attraverso il comando *man* seguito da un nome-comando specifico (e.g., *man ls*) è possibile accedere alla pagina del manuale del sistema operativo ed avere dettagli sull'uso e le opzioni con cui nome-comando può essere invocato.

Shell. In realtà il programma che *riceve, interpreta ed esegue* i comandi che l'utente digita di fronte al terminale testuale prende il nome di *shell*. I sistemi operativi Unix-like offrono all'utente diverse shell, che hanno opzioni più o meno simili. La Bash Shell è la shell che Linux associa all'utente per default. Potete controllare quale shell il sistema operativo vi mette a disposizione automaticamente, digitando il comando *echo* $SHELL. Per esempio:

```
macbook-di-paolo-darco:~ pd> echo $SHELL
/bin/bash
macbook-di-paolo-darco:~ pd>
```

In questo caso il prompt della shell è >, la stringa che lo precede è "macbook-di-paolo-darco:~ pd", nome della macchina e directory di lavoro, in questo caso la home directory ~ dell'utente pd. Il comando *echo* con parametro $SHELL, stampa a video il contenuto della variabile SHELL, contenuto che è la path che individua la shell utilizzata. Tale variabile è una variabile *d'ambiente*, cioè una variabile che contiene informazioni circa gli strumenti e le opzioni che il sistema operativo ha predisposto per l'utente pd.

Editor di testo. Un secondo strumento che occorre è un editor di testo, che permetta di editare file testuali, che nel nostro caso sono programmi in C. Ne esistono molti. Alcuni tra i più diffusi sono *vi* ed *emacs*. Non occorre diventare dei guru dell'editor, basta acquisire padronanza con i comandi fondamentali. In rete abbondano guide sintetiche alle principali shell ed agli editor di testo.

1.3 Compilazione

Compilatore. Una volta che un programma C è stato editato e salvato in un file .c, il programma va compilato. Compilare un programma significa generare un codice eseguibile dalla macchina. Tale generazione richiede fondamentalmente due[1] azioni:

1. trasformare il codice sorgente C in codice oggetto.
2. collegare tale codice oggetto con moduli di codice oggetto delle librerie (linkaggio).

Il compilatore che useremo è *gcc* (GNU C Compiler). Nel caso di un semplice programma C, memorizzato in *prog.c*, all'invocazione del comando *gcc prog.c*, il compilatore crea un file eseguibile, chiamato *a.out*. Usando invece l'opzione *-o*, cioè *gcc -o prog prog.c*, il compilatore memorizza l'output della compilazione nel file eseguibile *prog* invece che in *a.out*. In particolare, al termine della compilazione, per produrre l'eseguibile, *gcc* collega il codice oggetto prodotto dalla compilazione

[1] In realtà, come avete già avuto modo di vedere in altri corsi, il processo è piú complesso. Ai fini della discussione, la semplificazione non comporta perdita di generalitá.

di *prog.c* con le funzioni di libreria (e.g., *printf*), cioè con i moduli oggetto che corrispondono alle funzioni di libreria invocate in *prog.c*. Approfondiamo questo aspetto.

File multipli e librerie di funzioni. In genere un'applicazione scritta in C può avere una struttura ben più complessa. Per esempio, potremmo avere un file principale *main.c* che utilizza funzioni delle librerie fornite con il sistema operativo e funzioni definite dall'utente stesso, scritte e raccolte, in base agli obiettivi che realizzano, in diversi file di funzioni, tipo *func1.c, func2.c, func3.c* e *func4.c*. Il linguaggio C richiede però che all'interno del file *main.c* si riportino le dichiarazioni delle funzioni invocate. Inoltre, può accadere che alcune delle funzioni, per esempio in *func1.c*, siano richiamate da funzioni in *func4.c*. Pertanto, per l'utente risulta conveniente *raccogliere* i prototipi (valore di ritorno, nome della funzione, argomenti e tipo) di tutte le funzioni definite in *func1.c, func2.c, func3.c* e *func4.c* in un ulteriore file *myfunc.h*. Successivamente, ogni volta che sarà necessario, introducendo le due direttive

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

il compilatore saprà che il programma invoca sia funzioni della libreria standard del linguaggio C sia funzioni definite dall'utente, i cui prototipi si trovano, rispettivamente, nei file di intestazione *stdio.h* e *myfunc.h*. L'uso differente dell' instruzione #include dice al compilatore che il file *stdio.h* si trova in una directory prefissata dal sistema operativo per gli header file delle funzioni di libreria (/usr/include) mentre *mylib.h* è un header file definito dall'utente che si trova nella directory locale.

Il compilatore, leggendo gli header file, trova le definizione delle funzioni (dell'utente o di sistema) invocate nel main e riesce a produrre delle costanti simboliche che in fase di linkaggio servono per collegare il codice oggetto ottenuto compilando *main.c* con i codici oggetto delle funzioni dell'utente o della libreria. In particolare, il linker recupera automaticamente il codice oggetto delle funzioni di libreria del sistema nelle directory */usr/lib* e */usr/local/lib.* D'altra parte, l'utente può

- compilare separatamente ognuno dei quattro file contenenti le funzioni, invocando *gcc -c func1.o func1.c* per il primo file, *gcc -c func2.o func2.c* e così via, producendo i moduli oggetto. L'opzione -c dice al compilatore di arrestarsi non appena è stato prodotto il modulo oggetto.

- compilando il programma principale *main.c*, può chiedere di linkare anche le funzioni che si trovano nei file oggetto *func1.o, func2.o, func3.o* e *func4.o* con istruzioni tipo

  gcc main.c -o main func1.o func2.o func3.o func4.o

In realtà, l'utente potrebbe addirittura costruire una *propria libreria*, (esistono comandi appositi per far ciò, e.g., *ar*, per mettere assieme diversi file oggetto e costruire una libreria, ma la trattazione esula dagli obiettivi di questo corso) porla in una directory, diciamo *Mylibdir*, e dire al compilatore che le funzioni delle propria libreria si trovano nella directory *Mylibdir*. Questa ultima operazione può esser fatta attraverso l'istruzione *gcc -o main main.c -L Mylibdir*. L'opzione -L fornisce al compilatore il percorso di una directory che contiene la libreria costruita dall'utente.

In conclusione, il fine della discussione precedente è evidenziare che il processo di compilazione può diventare difficile ed oneroso. Ogni volta che si modifica uno dei file occorre ricompilare i file modificati e linkare nuovamente i moduli aggiornati. Manualmente questa operazione rischia di

diventare tediosa e soggetta ad errori. Sarebbe conveniente automatizzarla. Nota che nel prosieguo useremo funzioni di libreria del sistema operativo ma anche funzioni definite per il corso, i.e., il file *fun-corso-reti.c* contiene alcune funzioni utili a semplificare la gestione delle comunicazioni sulla rete.

1.4 Semplificare la compilazione

Comando make. Il processo di compilazione può essere semplificato *raccogliendo le direttive di compilazione* in un unico file e invocare un nuovo comando che legge il file e, a sua volta, *invoca* il compilatore *gcc* esattamente quanto serve (e.g., se un file dell'applicazione che stiamo scrivendo non è stato modificato non serve ricompilarlo) e con le opzioni giuste. Questo comando è *make* (gmake su alcune macchine) e il file di testo che raccoglie le direttive può essere chiamato *Makefile* o *makefile*. In soldoni, il comando *make* tiene traccia delle *modifiche* apportate e delle *dipendenze* fra i file. Quando invocato, ricompila solo ciò che è necessario. Un esempio ci aiuta a capire.

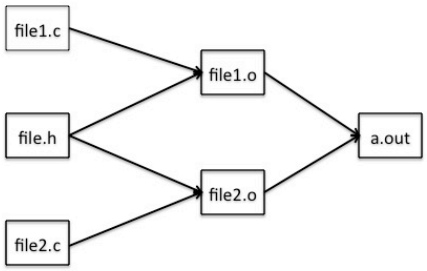


Figure 1: Grafo delle dipendenze

Il grafo mette in luce le dipendenze, e.g., *file2.o* dipende sia da *file.h* che da *file2.c*. Quindi in caso di modifiche a *file2.c* devono essere aggiornati *file2.o* e *a.out*, mentre gli altri file restano invariati. Il file *Makefile* codifica il *grafo delle dipendenze*. Relativamente al grafo precedente, avremo

```
a.out :   file1.o file2.o
    gcc file1.o file2.o

file1.o : file.h file1.c
    gcc -c file1.c

file2.o : file.h file2.c
```

```
    gcc -c file2.c

clean:
             rm -f *~
             rm -f *.o
```

Il file *Makefile* contiene regole specificate dalla seguente sintassi

```
target: source file(s)
  command               (deve essere preceduto da un TAB)
```

Il comando *make* legge il Makefile e controlla le date di modifica dei file coinvolti. Ogni volta che un source file ha una data di modifica *più recente* di quella del target file, il target file viene aggiornato eseguendo il comando specificato dalle regole nel Makefile. E' possibile invocare

```
make nome_target
```

per aggiornare solo il target specificato. Invece la semplice invocazione

```
make
```

esegue la prima regola del Makefile. Nel nostro caso, invocando *make* si ottiene l'esecuzione di

```
gcc -c file1.c
gcc -c file2.c
gcc file1.o file2.o
```

Si noti che alla fine del Makefile è specificato un *target senza dipendenze*. L'invocazione

```
make clean
```

provoca la cancellazione nella directory corrente di tutti i file temporanei e dei moduli oggetto. Tale opzione è utile soprattutto in fase di preparazione di una applicazione. Ogni volta che le compilazioni producono errori, possiamo ripulire e ricominciare daccapo invocando *make* clean.

Il comando *make* è potente e complesso. Una trattazione approfondita esula dagli scopi di questo corso. Vorrei solo far notare che *make* consente di definire delle macro per esprimere valori o sequenze di caratteri lunghe. Per esempio

```
OBJECTS = file1.o file2.o
```

Il valore di OBJECTS può essere utilizzato usando l'operatore $ e le parentesi tonde, cioè $(OBJECTS). In questo caso, nel Makefile precedente

```
a.out :   file1.o file2.o
```

equivale a

```
a.out :   $(OBJECTS)
```

Il Makefile che useremo nel nostro corso è

```
#Makefile

CFLAGS = -g -O0 -Werror -c
OFLAGS = -g -O0 -Werror -o

fun-corso-reti.o: fun-corso-reti.c
                gcc $(CFLAGS) fun-corso-reti.c

.c:   fun-corso-reti.o
        @echo compiling $< with rule 1
        gcc $< $(OFLAGS) $@ fun-corso-reti.o

clean:
        rm -f *~
        rm -f *.o
```

Il carattere # indica un commento. Le costanti CFLAGS ed OFLAGS rappresentano opzioni per il compilatore *gcc*, relative all'ottimizzazione del codice da produrre e alla gestione dei warning. Il target .c indica tutti i file con estensione .c. Il primo comando è una semplice stampa video che avverte il programmatore che si sta per compilare il file, mentre il secondo comando invoca realmente *gcc* con le dovute opzioni. Ogni volta che invocheremo *make* seguito da un nome-file contenente un sorgente .c (nell'invocazione l'estensione non va usata), il comando make ricompilerà nome-file. La semplice invocazione di *make* compilerà il file contenente le funzioni del nostro corso.

Infine, nella scrittura dei nostri programmi C utilizzeremo un file header *basic.h* che raccoglie i riferimenti a tutti i file header che definiscono i tipi di dati e le funzioni delle librerie di sistema che ci servono.

A questo punto abbiamo tutti gli strumenti di base per poter scrivere programmi C che comunicano su una rete di calcolatori. Prima però di addentrarci nello studio dell'interfaccia alla programmazione dei socket, occorre risolvere un problema preliminare: la *rappresentazione* dei dati all'interno delle architetture.

1.5 Rapprentazione dei dati

Architetture little-endian e big-endian. Le architetture dei calcolatori memorizzano i dati in modi diversi: le architetture little-endian memorizzano il byte meno significativo nel primo byte della locazione di memoria. Per intenderci, il valore decimale 288, rappresentabile in esadecimale come $0x0102$, in una locazione di memoria di 16 bit, memorizza il valore $0x02$ nel primo byte (meno significativo) della locazione e $0x01$ nel secondo (più significativo). Viceversa, in una architettura big-endian, il valore $0x01$ viene memorizzato nel primo byte (meno significativo) e $0x02$ nel secondo (più significativo).

E' come se nelle architetture little-endian i dati venissero interpretati leggendo da *destra a sinistra*, mentre in quelle big-endian da *sinistra a destra*. I termini little-endian e big-endian derivano appunto dal fatto che il byte meno significativo è vicino a bit meno significativo (LSB) oppure al bit più significativo (MSB). Ovviamente se due programmi, il primo in esecuzione su un calcolatore A con architettura little-endian e il secondo in esecuzione su un calcolatore B con architettura big-endian, interpretano i dati che si scambiano in modo diverso, le computazioni risultanti sono inconsistenti. Se il programma in esecuzione su A invia il valore decimale 258 in

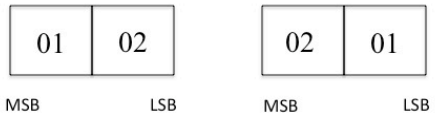


Figure 2: Architetture little endian e big endian

esadecimale $0x0102$, e il programma in esecuzione su B memorizza e interpreta il valore come $0x0201$, cioè 513, è facile intuire che ... i conti non tornano.

Pertanto, il programma in esecuzione su A e il programma in esecuzione su B debbono utilizzare la *stessa rappresentazione* dei dati nella comunicazione. Per convenzione, la rappresentazione prescelta nelle comunicazioni su reti è la rappresentazione big-endian. Quindi, il programma in esecuzione su un' architettura little-endian *deve* trasformare i dati che invia in formato big-endian e, viceversa, trasformare i dati che riceve in formato little-endian.

Curiosità. Si noti che problemi simili esistono anche nelle scritture che utilizzano gli esseri umani. I popoli occidentali scrivono orizzontalmente, da sinistra a destra. In passato, e ancora oggi, molti popoli orientali scrivono da destra a sinistra e/o dall'alto verso il basso. Pare che addirittura le prime forme di scrittura fossero *bustrofediche*, cioè si procedeva da un verso all'alto e si tornava indietro, ruotando al ritorno anche gli elementi dell'alfabeto.

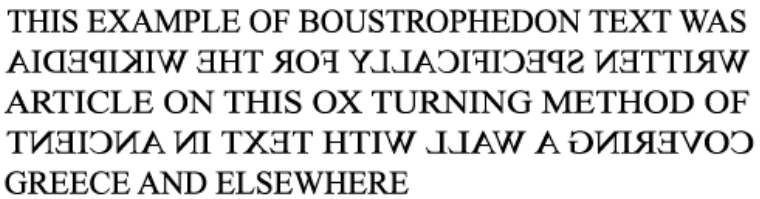


Figure 3: Esempio di scrittura bustrofedica

Seppur conoscessimo gli alfabeti di queste lingue, diversi dai nostri, leggendo nel nostro verso non riusciremmo ad interpretare correttamente ciò che c' è scritto. Tutto sommato, il mondo dei computer digitali è molto più semplice: l'alfabeto è unico (binario) e le direzioni sono soltanto due.

La nostra architettura. Come facciamo a sapere se la nostra architettura è little-endian o big-endian? Il programma in Tabella 1 permette di capirlo.

```
/* Verifica il tipo di architettura sottostante */
/* little-endian o big-endian */

#include <stdio.h>
#include <stdlib.h>

int main(int  argc, char **argv) {
        union {
          short s;
          char  c[sizeof(short)];
        } un;

        un.s = 0x0102;
        printf("CPU = %s - byte ordering: ",getenv("CPU"));

        if (sizeof(short) == 2) {
           if ( un.c[0] == 1 && un.c[1] == 2 )
                printf ("big-endian\n");
           else if ( un.c[0] == 2 && un.c[1] == 1 )
                printf ("little-endian\n");
           else
                printf("unknown\n");
        }
        else
           printf("size of short: %d.\n",sizeof(short));

        exit(0);
}
```

Table 1: Architettura: little-endian o big-endian?

Analizziamolo insieme e cerchiamo di capire cosa fa. Le istruzioni #include iniziali dicono al compilatore che nel seguito verranno utilizzate funzioni e tipi di dati i cui prototipi sono definiti nei file di intestazione *stdio.h* e *stdlib.h* (i.e., *printf* e *getenv*). Il programma vero e proprio (funzione main) alloca una *union* che può contenere sia uno *short* che un vettore di caratteri avente tanti elementi quanti sono i byte che si usano per rappresentare il tipo short. Solitamente uno short è 16 bit, ovvero 2 byte, per cui il vettore dovrebbe avere taglia 2.

Una *union* è una specie di struttura con più campi, con una differenza importante: per essa, il compilatore, invece di allocare spazio per *ogni* campo della struttura, alloca spazio in grado di contenere il più grande dei campi della union. In parole diverse, alla union si può accedere con la stessa sintassi con cui si accede ai campi delle strutture in C, ma essa contiene in ogni istante *un solo* dato, che può essere di uno dei tipi specificati nella definizione della union.

Nel nostro caso, nello spazio della union, visto come spazio che memorizza uno short, viene

memorizzato il valore esadecimale $ox0102$. Successivamente, il programma accede allo stesso spazio, visto come un vettore di due caratteri. Se il byte in $C[0]$, primo elemento del vettore, è 01, e quello in $C[1]$ è 02, allora l'architettura è big-endian. Viceversa, se il byte in $C[0]$ è 02 e quello in $C[1]$ è 01, allora l'architettura è little-endian. In casi diversi dai precedenti c' è qualche problema o il tipo short non viene memorizzato con 2 byte nella macchina in questione.

1.6 Conclusioni

Nella lezioni di oggi, abbiamo convenuto che:

- Alla fine del corso saremo in grado di scrivere programmi C che comunicano su una rete di calcolatori, usando le funzioni e i tipi di dati di una interfaccia alla programmazione di applicazioni, la Socket API, che è l'oggetto di studio principale di questo corso.
- Diversi strumenti riempiono il nostro ambiente di lavoro: terminali testuali, shell ed editor di testo. Dovremo prendere familiarità con essi quanto prima.
- Useremo nei nostri programmi C sia funzioni delle librerie fornite dal sistema operativo che una nostra libreria, *fun-corso-reti.c*. Poichè il processo di compilazione rischia di diventare inefficiente e facilmente soggetto ad errori, utilizzeremo il comando *make* che semplifica e rende efficiente il tutto. Il Makefile che dà indicazioni al comando *make*, il file *basic.h* e il file *fun-corso-reti.c*. sono scaricabili dalla pagina del corso.
- Nella scrittura dei nostri programmi, faremo sempre attenzione al problema delle diverse rappresentazioni dei dati, tra i vari calcolatori.

1.7 Esercizi: warm-up stringhe e file.

Nelle prossime lezioni useremo abbondantemente stringhe e file. I due esercizi che seguono aiutano a rivedere le funzioni principali.

Esercizio 1. Si scriva un programma C che:

- Chiede all'utente di digitare una riga contenente due numeri.
- Legge da tastiera la riga e la memorizza in una stringa.
- Estrae dalla stringa i due numeri, ne calcola la somma e la stampa a video.
- Chiede all'utente di digitare un nome.
- Chiede all'utente di digitare un cognome.
- Costruisce le concatenazioni nomecognome e cognomenome e le stampa a video.
- Se le due stringhe coincidono, stampa a video "Nome e Cognome coincidono"; altrimenti stampa "Nome e Cognome diversi".
- Estrae le iniziali da nome e cognome.
- Costruisce un codice utente concatenando alle iniziali la somma dei due numeri letti da tastiera.
- Stampa a video il codice utente.
- Conclude con un messaggio di saluto.

L'output atteso è qualcosa del tipo:

```
macbook-di-paolo-darco:ESERCIZI pd$ ./estring
Inserisci una riga contenente due numeri
5 7
La somma dei due numeri e': 12
Inserisci un nome
Paolo
Inserisci un cognome
D'Arco
La stringa NomeCognome risultante e' PaoloD'Arco
La stringa Cognomenome risultante e' D'ArcoPaolo

Come nella norma, Cognome e Nome sono diversi

Le iniziali di Paolo D'Arco sono: PD

Il tuo codice e': PD12.

Complimenti, maneggi le stringhe abbastanza bene!!
macbook-di-paolo-darco:ESERCIZI pd$
```

Funzioni utili allo svolgimento dell'esercizio possono essere: *fgets, sscanf, strcat, strncat, strcpy, strcmp, sprintf.*

Esercizio 2. Si scriva un programma C che:

- Chiede all'utente il *nome* di un file di testo.
- Apre il file e ne fa una copia, che chiama *backupnome.*
- Contemporanemente, conta il numero di vocali, il numero di cifre, il numero totale di caratteri, e il numero di righe del file.
- Stampa a video un resoconto relativo ad i quattro parametri considerati.