Programmare Linux

Table of Contents

Table of Contents

Guida alla programmazione C in ambiente GNU/Linux

Giorgio Zoppi
deneb@linux.it
versione 0.0.1e, 11 Gennaio 1999

1. Introduzione

Programmare Linux ha come obiettivo primario insegnare ai principianti la programmazione C in ambiente GNU/Linux. Vuole essere in secondo luogo un riferimento per tutti coloro che, provenendo da altre esperienze di programmazione, desiderano programmare in Linux.

Il libro è suddiviso in due parti essenziali, la prima che riguarda la programmazione C vera e propria e la seconda che è inerente alla programmazione del sistema operativo.

1.1 Perche questa guida?

Quando iniziai a programmare su Unix, mi sarebbe servito proprio un libro come questo che parta della basi di programmmazione e arrivi fino al *kernel−hacking*. Come già sottolineato questa guida vuole essere sia l'incipit per imparare a programmare per gli utenti neofiti, che un riferimento completo per gli utenti esperti.

1.2 Come collaboro a questa guida?

Per contribuire allo sviluppo di questa guida, occorre iscriversi alla mailing list del progetto nel seguente modo.

Per iscrivervi, mandate una e−mail vuota al: proglinux−subscribe@topica.com

Per togliere il vostro nominativo, mandate una email vuota: proglinux−unsubscribe@topica.com

Per mandare i messaggi alla mailing list: proglinux@topica.com

Nel caso troviate scorrettezze, vi prego di contattarmi all'indirizzo e−mail specificato, o mandare un post alla mailing list.

Durante il libro verranno presentati esercizi di programmazione, per le soluzioni e/o correzioni fate riferimento alla mailing list sopra citata.

1.3 Uso e ridistribuzione

1.4 Requisiti iniziali

Questa guida presuppone, che voi vi sappiate muovere nell'ambiente Linux, una passata esperienza programmativa è utile ma non necessaria. Sapersi muovere in Linux significa che si presuppone che sappiate creare un file, cambiare directory, vedere quali file sono presenti, usare il comando grep, ecc. Se non siete a conoscenza di tali comandi, vi consiglio vivamente di consultare le man page o gli ottimi libri in circolazione tra cui Appunti Linux, disponibile on−line al http://www.pluto.linux.it/ildp/AppuntiLinux.

2. Strumenti di programmazione

È particolarmente importante per un programmatore avere strumenti flessibili di programmazione. In Linux sono stati sviluppati contemporaneamente al sistema operativo una miriade di linguaggi di programmazione ed alcuni strumenti utilissimi per testare e verificare i programmmi. In questo capitolo introdurremo gli strumenti di base, che useremo per tutto il resto del libro, tra cui il compilatore GCC *(Gnu C Compiler)*, il debugger GDB *(Gnu Debugger)* il GNU Make e l'editor Emacs. Iniziamo allora a capire come funziona un compilatore per poi passare a una breve descrizione del GCC.

2.1 Che cosa è un compilatore?

I compilatori sono fondamentali per la moderna informatica. Essi agiscono come traduttori, trasformando i linguaggi di programmazione orientati all'uomo *(human oriented)* nei linguaggi di programmazione orientati alla macchina *(machine oriented)* . Per la maggior parte degli utenti, un compilatore puó quindi essere visto come una scatola nera, che compie la trasformazione illustrata nella figura 1.1.

Figura 1.1

Un compilatore permette alla maggior parte degli utenti di computer di ignorare i dettagli del linguaggio macchina. I compilatori quindi permettono ai programmi di essere indipendenti dalla macchina. Questa è una risorsa particolarmente preziosa in un periodo dove il numero e la varietà di computer continua a crescere esplosivamente.

Che cosa fanno i compilatori ?

La figura 1.1 mostra un compilatore come un traduttore da un linguaggio di programmazione a linguaggio macchina. Questa descrizione ci suggerisce che tutti i compilatori fanno grossomodo la stessa cosa, la sola differenza sta nella loro scelta del linguaggio sorgente e del codice macchina destinazione. Tuttavia, la situazione è un pò più complicata di quello che appare,infatti prima di produrre il codice eseguibile si passa attraverso il linking.

Il linker

Il linker è un programma che attua il processo di linking, che consiste nell'unione di uno o più codici oggetto e (opzionalmente) i moduli di codice oggetto contenuti in una libreria, in modo da produrre un file eseguibile. Il GCC (a patto che non specifichiamo l'opzione −c) esegue automaticamente il processo di linking. Lo schema di figura 1.1 puo essere esteso nel seguente modo:

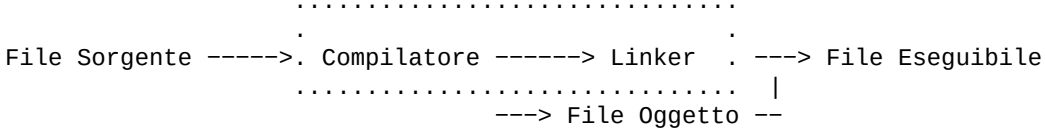


Figura 1.2

2.2 Il Gnu C compiler (GCC)

Il GCC è il compilatore C standard *de facto* per la piattaforma Linux, in quanto supporta tutti i moderni standard C, come l'ANSI C, ed ha molte estensioni proprie, che possono essere usate per l'ottimizzazione del codice. Essendo un compilatore multipiattaforma, consente anche di compilare codice per macchine non−Intel. Il GCC ha moltissime opzioni, ma noi durante tutto il libro ne useremo solo un sottoinsieme, che elenchiamo nella tabella sottostante fornendo esempi d'uso. Per coloro che fossere particolarmente curiosi, si consiglia la man page del GCC, che descrive in maniera dettagliata ogni opzione.

Opzione del GCC	**Descrizione**

−**o** nomefile	Specifica il nomefile di output eseguibile, utile quando si deve creare un esguibile. Se non viene specificata questa opzione, l'output di default è **a.out**.
	
	`Esempio di utilizzo:`
	`gcc −o targetfile sourcefile.c`

−**c** file_sorgente1, file_sorgente2, ecc..	Compila senza linkare i files sorgenti specificati, creando un file oggetto per ogni file sorgente specificato. Questa opzione è utile per la compilazione separata mediante l'utilizzo dell'utility make.
	
	`Esempio di utilizzo:`
	`gcc −c sourcefile.c`
	`Output:`
	`sourcefile.o`

−**D**pippo	Definisce la macro pippo per il preprocessore dalla linea di comando. Vedremo in seguito, che è una macro per preprocessore.
	
	`Esempio di utilizzo:`
	`gcc −Dpippo=\"pluto\" sourcefile.c`

−**I** directory	Aggiunge la directory specificata al percorso dove trovare i file di #include, usati dal programma sorgente.
	
	`Esempio di utilizzo:`
	`gcc −I/usr/local/include sourcefile.c`

−**L** directory	Aggiunge la directory specificata al percorso dove si trovano le librerie.
	
	`Esempio di utilizzo:`
	`gcc −L/usr/local/lib sourcefile.c`

−**g**	Produce informazioni aggiuntive utili ai debugger per poter lavorare.
	
	`Esempio di utilizzo:`
	`gcc −g −o target sourcefile.c`

−**ggdb**	Produce informazioni extra per il Gnu Debugger. Vengono con questa opzione aggiunte all'eseguibile molte più informazioni di quelle necessarie per il debug, per

	ottimizzare le operazioni di debug con il Gnu Debugger.
	
	`Esempio di utilizzo:`
	`gcc -ggdb -o target sourcefile.c`

-ansi	Supporta tutti i programmi scritti seguendo lo standard ANSI, ove vi siano confitti tra le estensioni GCC e lo standard ANSI, vengono disabilitate tali estensioni.
	
	`Esempio di utilizzo:`
	`gcc -ansi -o target sourcefile.c`

-Wall	Abilita tutti i messaggi di Warning di utilità generale, che il GCC fornisce. Molto utile quando il vostro programma non funziona come dovrebbe. Consiglio vivamente di usare quasi sempre questa opzione.
	
	`Esempio di utilizzo:`
	`gcc -Wall -o target sourcefile.c`

-On	Ottimizza il codice, dove n è il livello di ottimizzazione. Il massimo livello di ottimizzazione allo stato attuale è il 3, quello generalmente più usato è 2. Quando non si deve eseguire il debug è consigliato ottimizzare il codice.
	
	`Esempio di utilizzo:`
	`gcc -O2 -o target sourcefile.c`

-pedantic	Da durante la compilazione tutti i warning e gli errori fatti in un programma che rispetti l'ANSI C.
	
	`Esempio di utilizzo:`
	`gcc -pedantic -o target sourcefile.c`

-mcpu=tipo_cpu	Permette di specificare il tipo di cpu per il quale si vuole ottenere l'eseguibile. Per i comuni personal computer i valori ammessi sono:
	
	• Processore 80386. Valore ammesso:'i386'.
	• Processore 80486. Valore ammesso:'i486'.
	• Processore 80586. Valore ammesso:'i586'.
	• Processore 80686. Valore ammesso:'i686'.
	• Processore Pentium. Valore ammesso:'pentium'.
	• Processore PentiumPro. Valore ammesso:'pentiumpro'
	• Processore AMD K6. Valore ammesso:'k6'.
	`Esempio di utilizzo:`
	`gcc -mcpu='k6' -o target sourcefile.c`

-lnome_libreria	Usa la libreria nome_libreria durante il linking. Cerca la libreria nelle directory standard, dove sono poste le librerie.
	
	`Esempio di utilizzo:`
	`gcc -lnome_libreria -o target sourcefile.c`

-static	Nei sistemi che supportano le librerie dinamiche, fa in

	modo che i files vengano linkati con le librerie statiche.
	
	`Esempio di utilizzo:`
	`gcc -static -o target sourcefile.c`
	Tabella 1.0

Non preoccupatevi se non vi è chiaro qualcosa delle opzioni presenti nella tabella 1.0, in seguito ci ritorneremo sopra. Ho solo voluto dare una discreta panoramica delle opzioni che useremo per compilare gli esempi di questo testo.

2.3 Introduzione agli editor di testo

Questa sezione è stata curata da Marco Abis abis@programmazione.it. Ho solo ampliato la parte relativa ad Emacs, in quanto sarà l' editor di riferimento per tutto *Programmare Linux.*

Gli editor di testo

Molti sono gli editor di testo disponibili per l'ambiente Linux, alcuni forniti di serie con tutte le distribuzioni altri reperibili in rete. Tengo a sottolineare che con editors di testo non si intendono programmi con avanzate funzionalità di formattazione (quelli sono infatti i word processor, di cui ApplixWare e StarOffice sono degni rappresentanti per il mondo Linux), bensí programmi che ci danno la possibilit' di scrivere e modificare del testo nudo e crudo.

L'utilità di questo mio intervento è chiara se si pensa che per programmare è indispensabile un editor in cui scrivere il nostro codice, da dare poi in pasto al nostro compilatore preferito.

Joe

Inizialmente volevo partire presentando l'editor di testi *VI* che è presente nella quasi totalità dei sistemi UNIX−like. In seguito però ho deciso di fare una breve introduzione ai comandi di Joe (Joe's Own Editor) in quanto questo è sicuramente più semplice da usare (in particolare per chi è alle prime armi), potente e con una guida in linea.

Una particolarità di questo programma è che riesce bene ad emulare altri editor. Questo è possibile grazie al fatto che in Linux si possono creare dei link che rimandano sempre allo stesso file ma con nomi diversi. Joe controlla con che nome è stato invocato e si adatta cambiando il proprio modo di operare di conseguenza.

Per eseguire joe nella sua vera natura scrivete semplicemente *joe nomefile*, per eseguirlo in modalità emulazione usate i seguenti nomi:

- jstar. Emulazione Wordstar.
- jmacs. Emulazione Emacs.
- jPico. Emulazione Pico.
- rjoe . Versione limitata di joe : si possono modificare solo i file specificati a riga di comando

I comandi che seguono sono quelli di base, tutti i comandi disponibili sono descritti nell'help in linea.

Comando	**Funzione**
Ctrl+K+H	**Help in linea (apri/chiudi) da scorrete premendo Esc+. O Esc+,**
`@da`	Sposta in cursore una linea su.
`@ua`	Sposta il cursore una linea giù.
`@ra`	Sposta il cursore sul carattere a destra.
`@la`	Sposta il cursore sul carattere a sinistra.
Ctrl+A	Inizio riga.
Ctrl+E	Fine riga.
Ctrl+B	Indietro di un carattere
Ctrl+C	**Esce senza salvare**
Ctrl+D	Cancella il carattere sotto al cursore
Ctrl+K+A	Centra la riga tra i margini
Ctrl+K+D	**Salva il file**
Ctrl+K+E	**Apre un altro file**
Ctrl+K+U	Inizio file
Ctrl+K+V	Fine file
Ctrl+N	Riga seguente
Ctrl+P	Riga precedente
Ctrl+R	Ridisegna lo schermo
Ctrl+U	Scroll di mezza schermata verso l'alto
Ctrl+V	Scroll di mezza schermata verso il basso
Ctrl+Y	Cancella la riga corrente
Ctrl+_	Undo
Ctrl+^	Redo
Ctrl+K+X	**Chiede se salvare le modifiche (y/n) ed esce**

Il modo ricerca :

Premendo Ctrl+K+F si entra nel modo ricerca di joe che ci permette di cercare e sostituire stringhe di caratteri. Ctrl+C abbandona la ricerca.

Molte sono le cose che joe può ancora fare per noi come ad esempio auto indentare blocchi di testo (molto utile per la programmazione), definire macro o usare finestre multiple, ma per questo vi rimandiamo all'help in linea.

Emacs

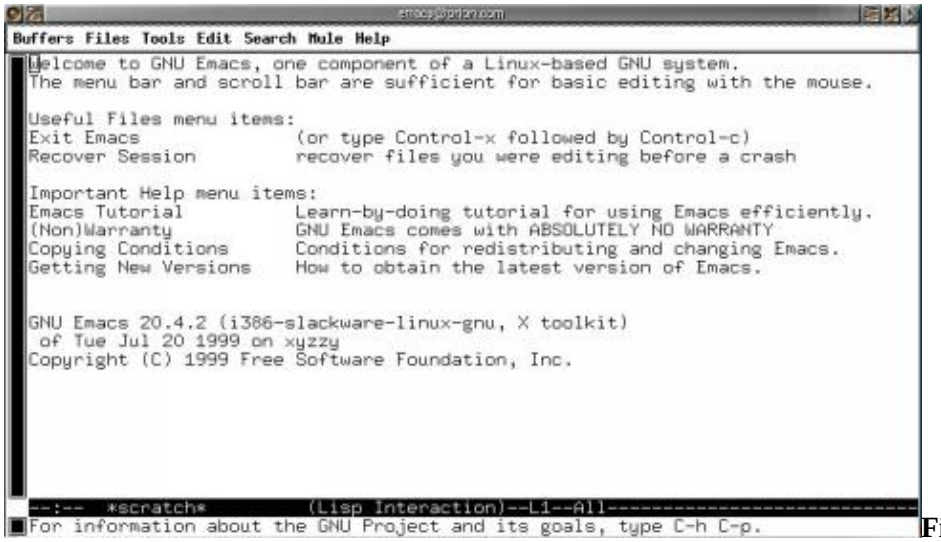


Figura 1.3 Emacs

Più produttivo per un programmatore, anche grazie a numerose estensioni che consentono di aggiungere funzionalità pensate espressamente per questo o quel linguaggio di programmazione.

Per far partire Emacs (naturalmente dovete averlo installato) digitate *emacs nomefile.* Vi troverete cosi davanti al vostro file aperto.

Nonostante sia più user−friendly di VI Emacs usa delle combinazioni di tasti che spesso sono tutto tranne che intuitive. Diciamo subito che per uscire da Emacs è necessario premere *Ctrl+X,Ctrl+S* per salvare ciò che si è scritto, quindi premere *Ctrl+X,Ctrl+C* per tornare al prompt della shell di Linux.

NOTA: per le combinazioni di tasti Emacs usa due tasti *Crtl* e *Meta*, però il tasto meta non è presente nella maggior parte delle tastiere e in sostituzione si usa *Alt*, se non avete neanche Alt (ma dove avete comprato il computer???) potete usare *Esc*

Detto questo passiamo all'elencazione dei comandi :

Comando	**Funzione**
Alt+<	Ci porta all'inizio del file
Alt+>	Ci porta alla fine del file
Alt+%	Permette di sostituire del testo con dell'altro testo.
Ctrl+A	Ci porta ad inizio riga
Crtl+E	Ci porta a fine riga
Ctrl+F	Avanti di un carattere
Ctrl+B	Indietro di un carattere
Alt+F	Avanti di una parola
Alt+B	Indietro di una parola

Ctrl+D	Cancella il carattere sotto il cursore
Alt+D	Cancella la parola corrente
Ctrl+G	Annulla il comando corrente
Ctrl+H	**Help in linea**
Ctrl+H+C	**Ci dice il nome del comando relativo ad una particolare combinazione di tasti**
Ctrl+H+T	**Introduzione ad Emacs**
Ctrl+K	Cancella tutta la riga dal cursore alla fine e la mette in un buffer (Taglia)
Ctrl+Y	Copia dopo il cursore il testo nel buffer (Incolla)
Ctrl+N	Linea seguente
Ctrl+P	Linea precedente
Ctrl+S	Cerca del testo, una volta trovato il testo premere *Esc*
Ctrl+T	Scambia il carattere sotto al cursore con il precedente
Alt+T	Scambia la parola sotto al cursore con la precedente
Alt+U	Rende maiuscole le lettere della parola sotto al cursore
Ctrl+V	Ci porta nella schermata seguente
Alt+V	Ci porta nella schermata precedente
Ctrl+X,U	Undo
Del	Cancella il carattere sotto al cursore

Emacs e la programmazione

da fare

Una sessione d'esempio

da fare

2.4 Il Gnu Make

Durante lo sviluppo di un programma, generalmente si applica il principio di programmazione top−down, suddividendolo in sottoprogrammi, che possono essere posti in file differenti per migliorare la manutenzione del codice stesso. L'utility `make` presente nei diversi tipi di Unix, viene incontro a questa necessità offrendo uno strumento efficente per la complilazione separata.

Che cosa fa make?

L'utility make determina automaticamente quali parti di codice di un programma di varie dimensioni devono essere ricompilati, ed esegue i comandi necessari alla ricompilazione. Per poter usare make, dovete scrivere un file, chiamato `makefile` che descriva le relazioni tra i file nel vostro programma e fornisca i comandi necessari per aggiornare ciascun file. In un programma, tipicamente, il file eseguibile è aggiornato dai file oggetto, che sono creati mediante la compilazione dei file sorgente. Generalmente parlando un programma può essere realizzato da un team di programmatori, che poi successivamente hanno il compito di riassemblare il tutto, ed è proprio questo che fa il make. Avendo creato un opportuno file di nome `Makefile`, per ricompilare il tutto, basta digitare dalla shell:

```
$ make
```

Oppure se avete creato un file make, con un vostro nome a piacimento, per ricompilare il tutto dovete, fare:

```
make −f nomefile
```

Una cosa importante da ricordare è la seguente: **In ogni progetto, se un file header (*.h) è cambiato, ciascun sorgente C, che include l'header deve essere ricompilato. Stessa cosa, per ogni sorgente modificato. Ogni compilazione produce un file oggetto corrispondente al file sorgente. Infine, tutti i file oggetto devono essere di nuovo linkati assieme per produrre l'eseguibile, il make fa questo per voi.**

Un semplice makefile.

Un semplice makefile è composto da un insieme di regole, con la seguente forma:

```
target: dipendenze
<TAB> comando
      .......
      .......
```

Un **target** è di solito il nome di un file, che è generato da un programma; esempi di file target sono i file eseguibili e i file oggetto.

Una **dipendenza** è un file che viene usato come input per creare il target. Un target spesso dipende da diversi file.

Un **comando** è un azione da compiere. Una regola può essere composta da più di un comando, su una linea. **Notate Attentamente: Si deve mettere un carattere TAB all'inzio di ogni linea di comando, per far sí che il make legga la linea in maniera corretta.**

Una **regola** spiega come e quando fare il make di dati file, che sono i target della data linea.

Ora che abbiamo chiarito questi concetti vediamo un semplice Makefile:

```
# Semplice Makefile
# Sysa Test
#
@echo Compilazione in corso
sysa : main.o token.o mostra.o \
        ins.o matrix.o
        gcc −o sysa main.o token.o mostra.o \
                     ins.o matrix.o
@echo Compilazione dei sottoprogrammi
main.o   : main.c header1.h
         gcc −c main.c
token.o  : token.c tokentype.h
         gcc −c token.c
mostra.o : mostra.c header1.h
         gcc −c mostra.c
ins.o    : ins.c tokentype.h
         gcc −c ins.c
matrix.o : matrix.c header1.h
         gcc −c matrix.c

clean : rm sysa main.o token.o mostra.o ins.o \
           matrix.o
install: cp sysa /usr/bin/sysa
```

A questo punto che avete editato il vostro Makefile, con il vostro editor preferito per fare in modo che il make vi compili i differenti file, in un unico, fate:

```
make
```

Se poi volete cancellare il tutto, fate `make clean`. Vediamo ora di spiegare a modo, come funziona il makefile. La prima cosa che si nota sono i commenti, che vengono posti dopo la #, il make ignora tutto ciò che appare dopo la # fino alla fine della linea. La seconda cosa sono i messaggi, che volete far apparire durante la compilazione, per far ciò si usa il comando :

```
@echo "Stringa che voglio far apparire"
```

Ora vediamo di distinguere tra target, dipendenze, e spiegare le diverse regole. Il file target principale, quello eseguibile dopo il make sarà `sysa`, che a sua volta dipende da n file oggetto, che vediamo subito dopo i due punti. Ciascuno di questi file oggetto, vengono compilati precedentemente e sono quindi essi stessi target e rispetto all'eseguibile sysa, dipendenze. Notate che la sequenza backslash−ritorno a capo (\), che suddivide alcune linee lunghe, è necessaria al fine della leggibilità del Makefile stesso. Il comando, che compare sotto i due punti, `gcc −c`, è necessario al make al fine di eseguire la chiamata al compilatore per ottenere i relativi target−dipendenze. Quando si vogliono ottenere i target ènecessario specificare nel make tutte le possibili dipendenze, compresi gli header, per esempio:

```
matrix.o : matrix.c header1.h
```

Questo significa, che il file oggetto matrix.o, dipende dal file matrix.c e dal file header1.h. Il file matrix.c andrà compilato, quindi si necessita di un comando che esegue tale operazione ed questo è:

```
gcc −c matrix.c
```

L'opzione −c serve, per evitare il linking automatico. Sono rimaste da spiegare le ultime righe. Il clean fa sí, che dopo che voi avete digitato make clean vengano cancellati i file binari li specificati ovvero : sysa, main.o, token.o, mostra.o, ins.o, matrix.o. La regola dopo il target install, copia il file appena compilato nella directory /usr/bin, dove un qualsiasi utente lo possa richiamare.

Macro e Makefile.

Molto spesso soprattuto per aumentare la flessibilità di progetti di grandi dimensioni vengono usate all'interno di un Makefile le macro. Una macro è un nome, che ha associato un valore. Tutte le volte che il make incontra quel nome, rimpiazza il nome con il valore ad esso associato. Facciamo un esempio più chiaro, supponiamo di aver bisogno di una data flessibilita quando compiliamo un file, e creiamo una macro `CFLAGS= −g −Wall`, faremo riferimento a tale macro in tutto il Makefile con $(CFLAGS). Tutte le volte che il make incontra l'espressione $(CFLAGS), la sostituisce con −g −Wall. Vediamo un esempio di un Makefile con macro:

```
# Tipo di compilatore
CC= gcc
# Opzioni per il compilatore
CFLAGS= −g −Wall

@echo Compilazione in corso
all : main.o token.o mostra.o \
        ins.o matrix.o
        $(CC) $(CFLAGS) −o sysa main.o token.o mostra.o \
        ins.o matrix.o
@echo Compilazione dei sottoprogrammi
main.o   : main.c header1.h
         $(CC) $(CFLAGS) −c main.c
token.o  : token.c tokentype.h
         $(CC) $(CFLAGS) −c token.c
mostra.o : mostra.c header1.h
         $(CC) $(CFLAGS) −c mostra.c
ins.o    : ins.c tokentype.h
         $(CC) $(CFLAGS) −c ins.c
matrix.o : matrix.c header1.h
         $(CC) $(CFLAGS) −c matrix.c
main.o   : main.c header1.h
         gcc −c main.c

clean : rm sysa main.o token.o mostra.o ins.o \
           matrix.o
install: cp sysa /usr/bin/sysa
```

Nell'esempio soprastante, vedete le macro CC, e CFLAGS. Tutte le volte che compaiono il make le rimpiazza con la stringa associata ad esse, quello che voglio farvi notare è quanto questo sia utile. Supponete che invece di avere un Makefile di dieci righe come questo, abbiate bisogno di un Makefile ben più grande. In questo caso se volete cambiare le opzioni del compilatore nel Makefile, dovrete sostituire tutte le opzioni, per esempio usando Emacs, mediante un Query and Replace. Questa è operazione un po' noiosa se il Makefile è lunghissimo, ma per fortuna l'esistenza delle macro ci dispensa da ció, basta infatti solo cambiare una linea e il lavoro è finito. La linea CFLAGS, è appunto la macro che rappresenta le opzioni da passare al compilatore che è rappresentato dalla macro CC. Ho scelto di definire una macro per il compilatore perchè in Linux

esitono molti compilatori C.

Ancora un altro Makefile.

Vediamo di spiegare questo nei dettagli questo Makefile, che è un pò piú complesso:

```
#  Directory dove andra installato il programma
INSTDIR= /usr/local/bin
# Tipo di compilatore
CC= gcc
# Opzioni per il compilatore
CFLAGS= -g -Wall

@echo Compilazione in corso
all : main.o token.o mostra.o \
        ins.o matrix.o
        $(CC) $(CFLAGS) -o sysa main.o token.o mostra.o \
        ins.o matrix.o
@echo Compilazione dei sottoprogrammi

main.o   : main.c header1.h
         $(CC) $(CFLAGS) -c main.c
token.o  : token.c tokentype.h
         $(CC) $(CFLAGS) -c token.c
mostra.o : mostra.c header1.h
         $(CC) $(CFLAGS) -c mostra.c
ins.o    : ins.c tokentype.h
         $(CC) $(CFLAGS) -c ins.c
matrix.o : matrix.c header1.h
         $(CC) $(CFLAGS) -c matrix.c

clean : -rm sysa main.o token.o mostra.o ins.o \
           matrix.o
install:
        @if [ -d $(INSTDIR) ] ;\
          then \
          install sysa $(INSTDIR)/sysa --mode 711 &&\
          echo "Installazione eseguita con successo!";\
        else \
          echo "Impossibile installare nella directory $(INSTDIR)" ;\
        fi
```

descrizione da fare.

L'uso dei suffissi in un makefile.

da fare

Un makefile completo.

da fare

```
# Ultimo esempio di makefile
```

```
prefix = /usr/local
CC = cc
CFLAGS = -O -I.

exec_prefix = ${prefix}
bindir = ${exec_prefix}/bin
mandir = ${prefix}/man

INSTALL = install -c
INSTALL_PROGRAM = ${INSTALL}
INSTALL_DATA = ${INSTALL} -m 644

.c.o:
        $(CC) -c $(CFLAGS) $*.c

MATRIX = main.o token.o mostra.o ins.o\
         matrix.o

all: sysa

sysa: $(MATRIX)
        $(CC) -o sysa $(MATRIX)
main.o   : main.c header1.h
token.o  : token.c tokentype.h
mostra.o : mostra.c header1.h
ins.o    : ins.c tokentype.h
matrix.o : matrix.c header1.h

install: sysa
        $(INSTALL_PROGRAM) sysa $(bindir)/sysa
        $(INSTALL_DATA) sysa.1 $(mandir)/man1/sysa.1
clean:
        -rm -f *.o sysa
```

Spiegazione da fare.

2.5 AutoMake e AutoConf

Si consiglia vivamente ai neofiti per una maggiore comprensione di saltare i paragrafi inerenti all'autoconf e all'automake e di leggerli una volta letto il capitolo 3. Tali paragrafi sono stati posti in questa parte della guida, al fine di rendere tematicamente organico il libro. Se invece conoscete bene un pò di C, iniziate a leggere.

Avete mai compilato un'applicazione scaricata da Internet? Se sì una volta scaricato il tarball avrete scritto i seguenti comandi:

```
cyberspace.org:~$ tar xfvz pacchetto.tar.gz
cyberspace.org:~$ ./configure
cyberspace.org:~$ make
```

E poi da root:

```
cyberspace.org:/home/gzoppi# make install
```

In questo paragrafo vediamo come creare un pacchetto e realizzare automaticamente lo script *configure* tramite le utility `automake` e `autoconf`. Per far ciò abbiamo essenzialmente la necessità di editare due file (`configure.in` e `Makefile.am`). L'insieme delle operazioni, che andremo a fare può essere suddiviso in diverse fasi che vediamo.

Prima fase: Dai sorgenti al configure.in

In questa fase si deve:

- Porsi nella directory dei file sorgenti.
- Eseguire l'utility `autoscan`, che analizzando i sorgenti crea un file `configure.in` di base, chiamandolo `configure.scan`.
- Editate nel modo, che vedremo il file configure.scan rinominandolo `configure.in`

Seconda fase: Documentazione.

Creiamo i file di documentazione : README, NEWS, AUTHORS, ChangeLog. Questa seconda fase è richiesta dall'utility automake e risulta essere particolarmente utile nello sviluppo del software e nel suo aggiornamento fornendo agli sviluppatori lo stato del nostro pacchetto e le eventuali modifiche. Dal punto di vista degli utenti, la documentazione è utile al fine della comprensione e utilizzo del pacchetto.
L'organizzazione di questi file richiesti dall'automake è cosi fatta:

- Changelog. Si usa questo file per memorizzare lo stato del vostro progetto e tutti i cambiamenti apportati al codice. Di solito i ChangeLog sono fatti nel seguente modo:

```
            DATA         AUTORE DEL CAMBIAMENTO       E-MAIL

            * Tipo cambiamento
            * Tipo cambiamento

Esempio:

1999-09-19  Arthur Fonzarelli  <arthur.fonzie@bayreuth.de>
        * Changed version to 0.12.4.
        * Added link breaking code (aka breaklinks) and additional
          \special operators to support them.
        * Fixed a bug in slanted font code
        * Fixed a bug in TFM code that won't actually execute
          unless you have a defective TFM file.
```

- NEWS. Questo file contiene e spiega le nuove caratteristiche aggiunte al pacchetto. Esempio:

```
Noteworthy changes in version 1.0.4 (1999-06-07)
------------------------------------

    * Add a very preliminary version of NOME_PACCHETTO the Handbook to
      the distribution (lynx doc/package/index.html).
```

```
* Changed the version number to NOME_PACCHETTO ;-)
```

- README. File generalmente richesto per spiegare il funzionamento del programma. In parole povere: *usate questo file quando avete qualcosa di fondamentale da dire sul programma*.
- AUTHORS. Questo file deve elencare una traccia di tutti coloro che hanno contribuito al programma. Un chiaro esempio è il seguente:

```
Authors of NOME_PACCHETTO
=================================

NOME_PACCHETTO Arthur Fonzarelli        1998-12-21
Assigns NOME_PACCHETTO and future changes.
arthur.fonzie@bayreuth.de
Designed and implemented NOME_PACCHETTO.


NOME_PACCHETTO  Matthew Zimmer         1999-01-10
Disclaims changes.
mat.zimmer@berknet.ca
Wrote io/basic.c.
```

Altri file che `automake` aggiunge e/o che dovrebbero essere presenti in ogni pacchetto sono:

- Il file THANKS. Scritto dall'autore originario dell'applicazione cita e ringrazia chiunque ha contribuito allo sviluppo, e a differenza del file AUTHORS non viene manenuto per ragioni legali. Esempio:

```
NOME_PACCHETTO was originally written by Arthur Fonzarelli.  Other people contribute
code.  Here is a list of those people.  Help me keep it complete and free of
errors.

Simon Clark             simon@seclink.com
Luigi Rossi             wildfire@securitynet.it
John  Rigel             rigel@univerisity.edu
Franz Muller            franz@friburg.de
Bryan Smith             bryan@usa.net
Luis  Deloire           deloire@chamberg.fr
```

- Il file INSTALL, creato automaticamente da `automake`, dove andranno specificate:
  - dipendenze da altri pacchetti
  - modalita di installazione
- Il file COPYING, creato da `automake`, che non è altro che la licenza GPL.

Terza fase: Creazione del file Makefile.am

Il file `Makefile.am` deve contenere una lista di assegnamenti per poi essere elaborato nel modo opportuno da `automake`. Vediamo in questa tabella le variabili più usate, che possono apparire in `Makefile.am`:

Variabile	Descrizione

LDADD= fileoggetto1.o, fileoggetto2.o, ecc	Elenca un insieme di file oggetto, che desiderate passare al compilatore per creare file esguibili.
SUBDIRS=directory 1, directory2, ecc	Elenca le sottodirectory conteneti i sorgenti, che volete compilare, prima compilare la directory corrente.
include_HEADERS = header1.h, header2.h, header3.h, ecc..	Include tutti i file header, che volete installare nel sistema al'PREFIX/include', dove PREFIX è di solito '/usr/local'
include_LIBRARIES = lib1.a, lib2.a, lib3.a, ecc..	Include tutti i file di libreria, che volete installare nel sistema al 'PREFIX/lib', dove PREFIX è di solito '/usr/local'
bin_PROGRAMS = eseguibile1, eseguibile2, eseguibile3, ecc..	Elenca tutti i file eseguibili, che volete compilare e installare nel sistema al 'PREFIX/bin', dove PREFIX è di solito '/usr/local'
programma_SOURCES = prog1.c, progr2.c, prog1.h, ecc..	Include tutti i file sorgenti, che volete compilare per creare il file eseguibile programma. **Tabella 1.1**

Quarta fase:Autoheader e acconfig.h

Il programma autoheader crea un file di direttive `#define` e `#undef` per lo script configure esaminando due file: configure.in e acconfig.h. Per quanto riguarda la struttura del primo file ne parleremo più avanti nel corso di questo paragrafo. Descriviamo ora brevemente il file acconfig.h, che può anche non essere presente in progetti di piccole dimensioni.

Il file acconfig.h, utile se usate un file header (*.h) per contenere direttive #define, viene elaborato (se presente) dall'autoheader e può avere la seguente struttura:

```
/* acconfig.h - used by autoheader to make config.h.in
 *      Copyright (C) 1998 Free Software Foundation, Inc.
 *
 *
  */
#ifndef PACK_CONFIG_H
#define PACK_CONFIG_H

/* need this, because some autoconf tests rely on this (e.g. stpcpy)
 * and it should be used for new programs
 */
#define _GNU_SOURCE  1

@TOP@

#undef M_DEBUG
#undef M_GUARD
#undef VERSION
#undef PACKAGE
#undef PRINTABLE_OS_NAME
#undef IS_DEVELOPMENT_VERSION

/* Define if your locale.h file contains LC_MESSAGES.  */
#undef HAVE_LC_MESSAGES

/* Define to 1 if NLS is requested.  */
#undef ENABLE_NLS
```

```
@BOTTOM@

#include "mydefine.h"

#endif
```

I `#define` e gli `#undef` sono direttive al compilatore di cui ora non ci occuperemo. Accentriamo invece la nostra attenzione sulla stringa **@TOP@.**L'autoheader nel momento in cui trova questa stringa, copia tutte le linee che la precedono nella parte iniziale del file configure.in.h, che è stato elaborato durante la scansione da parte di autoheader del file configure.in. Il programma autoheader svolge un simile procedura, quando incontra la stringa **@BOTTOM@**, solo che in questo caso l'autoheader copia le linee dopo quella stringa e le pone alla fine del file configure.in.h.

Quinta Fase: Realizzazione dello script configure

Ora abbiamo creato in maniera appropriata i file: configure.in e Makefile.am. Ci resta solo da modificare i sorgenti e includere in essi il seguente spezzone di codice (utile per l'autoconf) all'inizio del primo file, che deve essere compilato:

```
#if HAVE_CONFIG
#include <config.h>
#endif
```

Salvate il tutto e ora siamo pronti per generare il nostro file configure, tramite i seguenti comandi:

```
cyberspace.org:~$ aclocal
cyberspace.org:~$ autoconf
cyberspace.org:~$ autoheader
cyberspace.org:~$ automake −a
```

2.6 Creiamo un pacchetto con Automake e Autoconf.

Passiamo ora dalla teoria alla pratica eseguendo le diverse fasi sopra indicate.

Prima fase: Dai sorgenti al file configure.in

Supponiamo di aver gia eseguito l'autoscan, ora ci accingiamo a editare il file configure.in nel seguente modo:

```
dnl Process this file with autoconf to produce a configure script.
AC_INIT(main.c)
dnl Checks for programs.
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(matrix,0.1)
AC_PROG_CC
AC_PROG_INSTALL
```

```
dnl Checks for libraries.

dnl Checks for header files.
AC_HEADER_STDC
if test $ac_cv_header_stdc = no; then
AC_MSG_ERROR([Matrix richiede ANSI C])
fi

dnl Checks for typedefs, structures, and compiler characteristics.

dnl Checks for library functions.
AC_CHECK_FUNCS(getline)
AC_OUTPUT(Makefile)
```

Il file configure.in mostrato sopra è suddiviso in diverse sezioni, suddivisione che viene rispettata da tutti i file configure.in. Queste sezioni sono:

- Inizializzazione
- Controllo programmi presenti nel sistema, da cui il vostro programma dipende
- Controllo librerie necessarie e presenti nel sistema
- Controllo degli header per la compilazione
- Controllo delle funzioni di libreria

Per ogni sezione si specificano delle macro (leggi regole di controllo), che servono a svolgere ogni singolo controllo sulla configurazione richiesta dal programma per funzionare. Penso che abbiate già capito, che i commenti nel file configure.in sono differenti rispetto ai `Makefile` tradizionali, dove erano rappresentati da #. Qui invece i commenti si esprimono mediante la parola dnl, ovvero tutto ciò, che è specificato dopo la parola dnl fino a fine linea viene ignorato dall'autoconf.

Iniziamo la descrizione del configure.in presentato, cercando di chiarire le cose passo dopo passo:

```
riga 1: dnl Process this file with autoconf to produce a configure script.
riga 2: AC_INIT(main.c)
riga 3: dnl Checks for programs.
riga 4: AM_CONFIG_HEADER(config.h)
riga 5: AM_INIT_AUTOMAKE(matrix,0.1)
riga 6: AC_PROG_CC
riga 7: AC_PROG_INSTALL
riga 8: dnl Checks for libraries.
riga 9: dnl Checks for header files.
riga 10:AC_HEADER_STDC
riga 11:if test $ac_cv_header_stdc = no; then
riga 12:AC_MSG_ERROR([Matrix richiede ANSI C])
riga 13:fi
riga 14:dnl Checks for typedefs, structures, and compiler characteristics.
riga 15:dnl Checks for library functions.
riga 16:AC_CHECK_FUNC(getline)
riga 17:AC_OUTPUT(Makefile)
```

- la riga 2 (AC_INIT(main.c)) imposta lo script configure in modo che controlli il file specificato tra parentesi per assicurarsi che la directory corrente sia quella contente i file sorgente del pacchetto.Questa è la fase di inizializzazione.
- la riga 4 serve all'autoconf per generare lo script configure

- la riga 5 (AM_INIT_AUTOMAKE(matrix,0.1)), serve a svolgere le inizializzazioni per l'uso con automake, dove l'espressione posta tra parentesi rappresenta rispettivamente il nome e la versione del pacchetto. In generale ogni macro, che ha come prime due lettere 'AM' è una macro per l'uso di automake. Se volete scrivervi manualmente i Makefile non avete bisogno di questa macro.
- AC_PROG_CC. Questa macro controlla il tipo di compilatore che avete.
- AC_PROG_INSTALL. Questa riga serve a controllare se avete un'utility di installazione compatibile BSD.
- AC_HEADER_STDC. Questa macro controlla se i file header dello standard ANSI C sono presenti.

Ora abbiamo 3 righe interessanti (11,12,13):

```
riga 11:if test $ac_cv_header_stdc = no; then
riga 12:AC_MSG_ERROR([L'applicazione richiede ANSI C])
riga 13:fi
```

In parole queste tre righe si traducono in :

```
se il test precedente è negativo
         ==> allora scrivi su schermo il messaggio:
                      "L'applicazione richiede ANSI C"
```

Facciamone una breve descrizione. La riga 11 usa una variabile (simile a quelle dei makefile) `$ac_cv_header_stdc`, che viene impostata durante l'esecuzione dello script *configure*. Questa variabile corrisponde alla macro $AC_HEADER_STDC. Da notare che ogni macro ha una variabile simile, per esempio la macro AC_PROG_FUNCS ha una variabile $ac_cv_prog_funcs associata. Durante l'esecuzione, *configure* testa se la richiesta di una data macro è soddifatta o meno nel sistema e imposta la variabile associata alla macro. Nelle righe sopra illustrate (11,12,13) si va a verificare la compatibilità con l'ANSI C, se tale compatibilità è presente allora la $ac_cv_header_stdc sarà uguale a 'yes', altresí sarà uguale a 'no'. In quest'ultimo caso l'esecuzione di *configure* sarà interrotta e apparirà sullo schermo il messaggio tra parentesi quadre presente nella macro AC_MSG_ERROR.

È obbligo ora elencare un sottoinsieme delle macro più utilizzate e darne una breve descrizione nella seguente tabella.

Variabile	Descrizione
AC_CHECK_HEADERS(header1.h, header2.h, header3.h, ecc)	Controlla nella directory standard di include, se gli header specificati esistono
AC_CHECK_FILE(nomefile)	Controlla se nel percorso corrente esiste il file nomefile
AC_CHECK_PROG(nome_eseguibile)	Controlla se nel percorso corrente esiste il programma nome_eseguibile
AC_CHECK_LIB(nome_libreria)	Controlla se esiste la libreria nome_libreria nella directory standard delle librerie.
AC_CHECK_FUNC(nome_func)	Controlla se nelle librerie presenti esiste la funzione nome_func

Tabella 1.2 (n.b. la tabella è da ampliare)

Ritorniamo al nostro esempio in quanto mancano da descrivere le linee 16 e 17, vediamole:

- La riga 16, AC_CHECK_FUNC(getline), controlla se nel sistema è presente la funzione getline, usata dal nostro programma. Anche in questo caso potevamo controllare il risultato del test, mediante la variabile $ac_cv_check_func, nello stesso modo fatto precendemente con $ac_cv_header_stdc.
- La riga 17 invece dice di generare al configure il Makefile dal Makefile.in, che **deve essere presente prima di essere lanciato il configure**. Il Makefile.in si può ottenere mediante automake, come vedremo.

Saltiamo la seconda fase, che consiste nel creare dei semplici file di testo secondo la struttura già descritta e passiamo alla terza fase.

Terza fase: Creazione del Makefile.am.

Nella creazione del Makefile.am, utile all'automake per generare il Makefile.in richiesto da configure, vale quanto detto finora sui Makefile. Possiamo inoltre aggiugere le opzioni già descritte in precedenza. Per i nostri scopi il makefile sarà fatto in questo modo:

```
bin_PROGRAMS= matrix
matrix_SOURCES= main.c token.c
```

Quarta fase: Autoheader e acconfig.h

Vediamo come ho edito il acconfig.h:

```
/* acconfig.h − used by autoheader to make config.h.in
 *      Copyright (C) 1998 Free Software Foundation, Inc.
 *
 *
  */

/* need this, because some autoconf tests rely on this (e.g. stpcpy)
 * and it should be used for new programs
 */
#define _GNU_SOURCE  1

@TOP@


@BOTTOM@

#include "filedef.h"
```

La mia unica necessità era di specificare il _GNU_SOURCE al compilatore, per avere la massima compatibilità con tutti gli standard esistenti e tutte le librerie. Poi ho incluso il mio file header, dove sono contenute le #define.

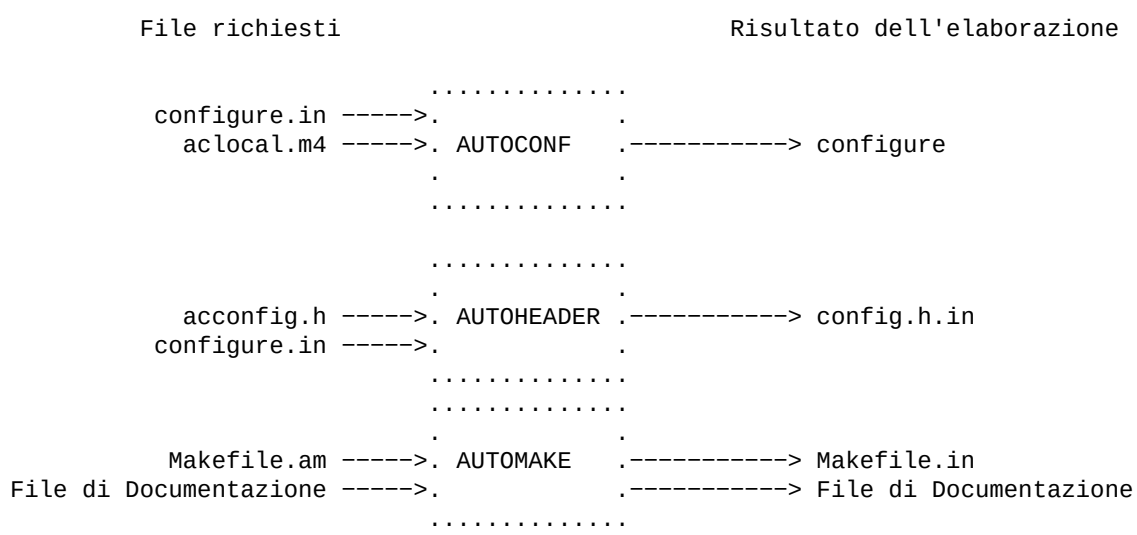
Quinta fase: Realizziamo lo script configure

Siamo arrivati in fondo, la parte più noiosa è stata svolta, ora basta digitare i seguenti comandi:

```
cyberspace.org:~$ aclocal
cyberspace.org:~$ autoconf
cyberspace.org:~$ autoheader
cyberspace.org:~$ automake −a
```

Cerchiamo di comprendere ogni singolo comando digitato:

- **aclocal** crea un file chiamato aclocal.m4. Questo file è utilizzato sia per scrivere le macro definite dall'utente che dall'automake. Infatti automake usa la macro AM_INIT_AUTOMAKE, che non è riconosciuta dall'autoconf, la cui definizione è posta in aclocal.m4.
- **autoconf, autoheader, automake** eseguono le seguenti elaborazioni:



C'è, solo da fare un piccolo appunto, prima di terminare questo paragrafo. Qualcuno di voi si chiederà il significato dell'opzione −a data ad automake. Precedentemente avevamo parlato di file, che automake aggiunge automanticamente che poi vanno editati, quali COPYING, INSTALL, ecc. Ebbene automake li aggiunge solo se gli viene passata l'opzione −a, altresi da un messaggio di errore simile:

```
automake: configure.in: required file `./install−sh' not found
automake: configure.in: required file `./mkinstalldirs' not found
automake: configure.in: required file `./missing' not found
automake: Makefile.am: required file `./INSTALL' not found
automake: Makefile.am: required file `./COPYING' not found
```

Note finali.

Ora abbiamo finito, il nostro pacchetto è pronto, creiamo il tarball, e nel caso in cui ritenessimo il nostro programma un lavoro ben fatto, potremmo segnalarlo al http://www.freshmeat.net, mettendolo così a disposizione di altri programmatori e/o utenti.

2.7 Documentazione sul Make

da elencare.

2.8 Il Gnu Debugger

Come funziona il GDB?

Il GDB (Gnu Source−Level Debugger) può fare essenzialmente quattro tipi di operazioni:

- Avviare il vostro programma, specificando tutto ciò che influisce sul suo comportamento.
- Fare in modo che il vostro programma si interrompa sotto date condizioni
- Esaminare quello che è accaduto, quando il vostro programma è stato interrotto.
- Cambiare le cose nel vostro programma, in modo che possiate sperimentare gli effetti delle correzioni di un dato bug

Per usare lavorare con il GDB occorre produrre le informazioni necessarie al debugging nel formato specifico del sistema. Nella maggior parte dei sistemi che usano il formato "stabs", l'opzione '−g' data al GCC, in fase di compilazione abilita l'uso delle informazioni extra necessarie al debugging con il GDB. Ecco un esempio di compilazione:

```
cyberspace.org:~$ gcc −g −o mio_programma mio_programma.c
```

Una volta compilato il programma, avviamo il GDB:

```
cyberspace.org:~$ gdb mio_programma

oppure se vogliamo analizzare anche il file core

cyberspace.org:~$ gdb mio_programma core
```

Ora per eseguire il programma fino al primo breakpoint o watchpoint è necessario digitare:

```
(gdb) run
```

Il lettore attento, ora si starà chiedendo che cosa sia un breakpoint o un watchpoint, vediamolo.

Breakpoint e Watchpoint.

Breakpoint

Un **breakpoint** fa in modo che il vostro programma si interrompa ogni volta che viene raggiunto un dato punto. L'impostazione di un breakpoint è abbastanza flessibile, in quanto si possono specificare diverse condizioni ad esso relative. Facciamo un piccolo schema relativo alle varie opzioni che possiamo dare al comando `break` al fine di impostare un breakpoint.

Comando	**Descrizione**
`break`	In questo modo si imposta il breakpoint alla istruzione successiva, che deve essere eseguita.
`break` nome_funzione	Imposta un breakpoint alla funzione nome_funzione, dove nome_funzione è una funzione del vostro programma.
`break` numero_linea	Imposta un breakpoint, alla linea numero_linea del file corrente
`break` nomefile:nome_funzione	Imposta un breakpoint, alla funzione nome_funzione del file sorgente nomefile
`break` nomefile:numero_linea	Imposta un breakpoint, alla linea numero_linea del file sorgente nomefile
`break`if cond	Imposta un breakpoint, con una condizione cond; valuta l'espressione ogni volta, che viene raggiunta e si ferma se e solo se la condizione è un valore diverso da zero.
Tabella 1.3	

Molto probabilmente ora vorrete visualizzare tutti i breakpoints oppure lo stato degli stessi (compreso il loro numero), basta fare allora:

```
(gdb) info breakpoints
```

Watchpoint

A differenza di un breakpoint, un watchpoint ferma il programma quando il valore di una data espressione cambia.

Potete gestire un watchpoint in maniera simile ad un breakpoint; potete quindi assegnarlo, abilitarlo, ottenere le informazioni relative, disabilitarlo e cancellarlo con comandi molto simili. Per impostare un watch point, e vedere le informazioni relative basta digitare:

Per settare un watchpoint:

```
(gdb) watch espressione
```

Per vedere i watchpoint correnti e il loro relativo numero:

```
(gdb) info watch
```

Dove *espressione*, è una variabile del vostro programma sorgente. Il GDB interromperà l'esecuzione quando l'espressione viene scritta dal programma e il suo valore cambia.

Abilitazione e Disabilitazione di watchpoints e breakpoints

Come detto precedentemente un breakpoint o un watchpoint può avere uno dei quattro stati di abilitazione:

- **Abilitato.** Il breakpoint (watchpoint) ferma l'esecuzione del vostro programma.
- **Disabilitato.** Il breakpoint (watchpoint) non ha alcun effetto sul vostro programma.
- **Abilitato una volta**. Il breakpoint (watchpoint) ferma il vostro programma e subito dopo viene disabilitato.
- **Abilitato per la cancellazione**. Il breakpoint (watchpoint) ferma il vostro programma e subito dopo, viene cancellato in maniera permanente.

I comandi relativi sono descritti nella seguente tabella:

Comando	Descrizione
`enable numero_breakpoint`	Breakpoint abilitato
`disable numero_breakpoint`	Breakpoint disabilitato
`enable once numero_breakpoint`	Breakpoint abilitato una volta
`enable delete numero_breakpoint`	Breakpoint abilitato per la cancellazione
Tabella 1.4	

Muoversi durante l'esecuzione.

Ora per poter illustrare il nostro futuro esempio riassuntivo, che dia una visione completa di una sessione gdb, ci servono ancora due cose essenziali. La prima sono i comandi per muoversi durante il debug e per impostare gli eventuali parametri, la seconda è il concetto di backtrace che vedremo nel prossimo paragrafo. Forniamo una tabella riassuntiva di tutti i comandi necessari per muoversi durante una sessione di debug.

Comando	Descrizione
`set args`	Se il nostro programma accetta degli argomenti, questo comando permette di specificarli in questo modo: `set args argomento1 argomento 2 ... ecc.`
`show args`	Mostra tutti gli argomenti specificati
`run`	Esegue il programma finchè non incontra un breakpoint o un watchpoint.
`next`	Avanza alla linea successiva del programma senza entrare in sottofunzioni
`list`	Visualizza il codice sorgente relativo al programma
`step`	Avanza alla linea successiva del programma entrando nelle sottofunzioni
`print nomevariabile`	Stampa il contenuto della variabile nome variabile
`step`	Avanza alla linea successiva del programma entrando nelle sottofunzioni
`help`	Richiama l'help in linea interno al GDB.
`clear tipo_argomento`	Cancella un breakpoint (watchpoint). Il tipo_argomento è simile a quello di break.
`finish`	Completa l'esecuzione del programma sino alla terminazione della funzione
`continue`	Continua l'esecuzione fino al breakpoint (watchpoint) successivo.
`delete numero_breakpoint`	Cancella il breakpoint (watchpoint) numero_breakpoint
`quit` oppure `CTRL+D`	Esce dal GDB
Tabella 1.5	

Il GDB e lo stack

Siamo ora in grado di utilizzare il GDB, ma ogni volta che il vostro programma è interrotto, si rende molto utile capire quali sottofunzioni ha chiamato, tutto ciò per avere una piena comprensione dello stato del vostro programma una volta interrotto.

Ogni volta che il vostro programma svolge una chiamata ad una funzione vengono generate delle informazioni relative alla chiamata stessa. Tali informazioni includono la posizione della chiamata, gli argomenti della chiamata e le variabili locali della funzione che è stata chiamata. Le informazioni vengono salvate in un blocco di dati chiamato *stack frame*. Gli stack frame sono allocati in una regione di memoria chiamata *stack*. Il GDB ci fornisce diversi comandi per interagire con lo stack, quello che viene usato di più e il comando `backtrace`. Il backtrace mostra una linea per frame, per differenti stack frame, partendo dal frame correntemente in esecuzione susseguito dal suo chiamante, continuando per tutto lo stack. Vediamo come si applica:

Comando	Descrizione
`backtrace`	Stampa un backtrace intero dello stack, una linea per frame per ogni frame dello stack. Si interrompe premendo CTRL+C.
`backtrace n`	Stampa un backtrace per i primi n frame dello stack.
`backtrace −n`	Simile, ma stampa un backtrace per gli ultimi n frame dello stack. **Tabella 1.6**

Detto questo ora siamo in grado di vedere la nostra sessione di esempio, che riassuma tutta la teoria finora vista.

Sessione di esempio

Nella sessione di esempio, userò il carattere ## per indicare un commento che non fa parte della sessione stessa.

```
##compiliamo il nostro file a dovere:

cyberspace.org:~$ make
gcc −g −c −Wall main.c
gcc −g −c −Wall token.c
gcc −g −o matrix main.o token.o

## avviamo il gdb
cyberspace.org:~$ gdb matrix

GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
##
## impostiamo un breakpoint alla linea 23
(gdb) break 23
Breakpoint 1 at 0x8048478: file main.c, line 23.
##
## eseguiamo il programma fino alla linea 23.
(gdb) run
Starting program: /home/gzoppi/last/matrix

Breakpoint 1, main () at main.c:24
24      int main (void) {
## Andiamo avanti con l'esecuzione passo dopo passo
(gdb) step
main () at main.c:30
30      do {
## Andiamo avanti con l'esecuzione passo dopo passo
(gdb) step
31      dim=0;
## Andiamo avanti con l'esecuzione passo dopo passo
(gdb) step
32      input=NULL;
# Stampiamo il valore della variabile dim
(gdb) print dim
$1 = 0
## Impostiamo un watchpoint sulla variabile dim.
(gdb) watch dim
Hardware watchpoint 2: dim
## Continiamo l'esecuzione fino al prossimo watchpoint
(gdb) continue
Continuing.
#0  main () at main.c:32
32      input=NULL;
Hardware watchpoint 2: dim
## Qui viene modificato il valore della dim e si verifica un watchpoint
Old value = 0
New value = 120
0x4006969f in getdelim () from /lib/libc.so.6
## Guardiamo cosa c'è nello stack
(gdb) backtrace
#0  0x4006969f in getdelim () from /lib/libc.so.6
#1  0x40068a18 in getline () from /lib/libc.so.6
#2  0x80484a1 in main () at main.c:33
## Continuamo l'esecuzione fino alla fine della funzione getdelim, chiamata inte# rnamente
(gdb) finish
Run till exit from #0  0x4006969f in getdelim () from /lib/libc.so.6
#0  0x4006969f in getdelim () from /lib/libc.so.6
0x40068a18 in getline () from /lib/libc.so.6
## Visualizziamo tutte le informazioni relative ai nostri watchpoint
(gdb) info watch
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048478 in main at main.c:23
        breakpoint already hit 1 time
2   hw watchpoint  keep y   dim
## Visualizziamo il sorgente
(gdb) list
27      char *input;
28      int dim,size;
29
30      do {
31      dim=0;
```

```
32      input=NULL;
33      size=getline(&input,&dim,stdin);
34      stop=reserved_token(input,size);
35      } while (stop!=QUIT);
36      return EXIT_SUCCESS;
## disabilitiamo il breakpoint 2 (che è un watchpoint)
(gdb) disable 2
## ancora passo−passo
(gdb) step
Single stepping until exit from function getline,
which has no line number information.
main () at main.c:34
34      stop=reserved_token(input,size);
(gdb) step
reserved_token (stringa=0x80499c8 "LUIGI\n", dimensione=6) at token.c:24
24        char *tkn;
## Guardiamo i sorgenti della sottofunzione
(gdb) list
19                      else return ID;
20      }
21
22      keyword reserved_token (char *stringa, int dimensione) {
23
24        char *tkn;
25        char *running;
26
27        keyword token;
28
#finiamo la sottofunzione
(gdb) finish

Run till exit from #0  reserved_token (stringa=0x80499c8 "LUIGI\n",
    dimensione=6) at token.c:24
0x80484b6 in main () at main.c:34
34      stop=reserved_token(input,size);
Value returned is $2 = EMPTY
## Continuo dopo...e quindi faccio quit per uscire
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

2.9 Documentazione sul GDB.

Per ultertiori approfondimenti leggetevi: **Debugging with GBD**, il manuale on−line presso il sito http://www.gnu.org/manual/gdb−4.17/gdb.html.

2.10 Il Data Display Debugger (DDD)

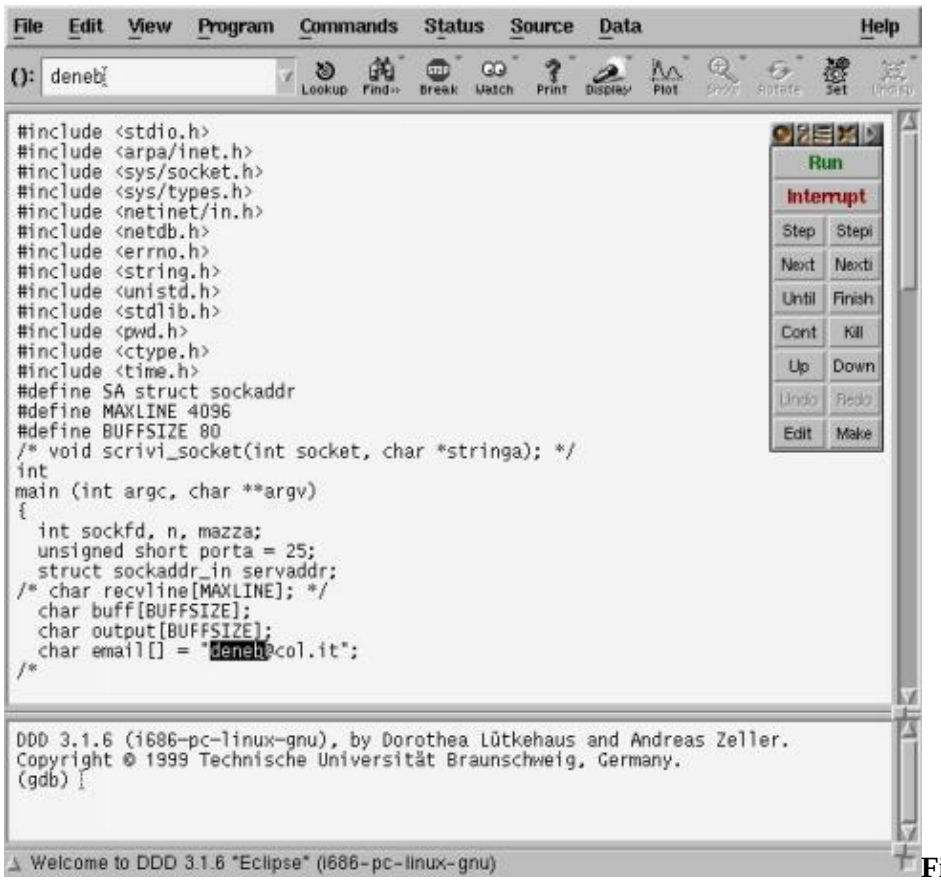


Figura 1.4 Il DDD

dafare

2.11 Scrivere la guida in linea (man page)

Scrivere la guida in linea, la cosidette man page, è un'operazione abbastanza elementare, che chiunque può fare. Per essere tutte standard, le man page devono seguire lo schema, che vediamo sotto. Si tratta semplicemente di scrivere un file con elementari regole sintattiche, che viene poi rielaborato dal programma groff, che restiuisce la man page. Vediamo come deve essere fatta una man page:

```
   .TH nome sezione data
          specifica l'intestazione della man page che ha la seguente struttura:

+-------------------------------------------+
|nome(sezione)                 nome(sezione)|
|                                           |
/                                           /
|                                           |
|               data           numero_pagina|
+-------------------------------------------+

   .SH "descrizione"
        Intestazione della sezione descrizione

   .SS "descrizione"
           Intestazione della sotto-sessione di "descrizione"
```

```
.B scritta
       scritta in grassetto

.I scritta
       scritta in corsivo

.BI a b a b a b a...
       .IB a b a b a b a...
       .RI a b a b a b a...
       .IR a b a b a b a...
       .RB a b a b a b a...
       .BR a b a b a b a...
       BI  pone le a in grassetto e le b in corsivo
       RI  pone le a in roman e la le b in corsivo
       IR  pone le a in corsivo e le b in roman
       RB  pone le a in roman e le b in grassetto
       BR  pone le a in grassetto e le b in roman
       etc.

.P
       .LP
       .PP
       Tre sinonimi per le interruzioni di paragrafo

.TP colonne
       termine
       paragrafo...

.IP inizio
       Inizia un paragrafo identato, l'argomento <it>inizio è               presente vi



.RS
       ...
       .RE
       Definisce un regione identata.



Commandi generici

 .\" linea di commento
```

Queste macro appena descritte compongono un elenco completo di come puo essere fatta una man page, ma effettivamente alcune di esse sono poco usate. Vediamo ora un esempio di man page.

```
.TH MIOPROGRAMMA 4 LOCAL
.SH NAME
mioprogramma − esegue qualcosa di utile
.SH SYNOPSIS
.B mioprogramma [opzioni]
.I option option
.B ["
.I f1 f2
```

```
.B  ..."]
.SH DESCRIPTION
.I mioprogramma
 Script bash che fa qualcosa di utile, e ha le seguenti opzioni:
.TP 5
f1
Descrivi la prima opzione
.TP
f2
Descrivi la seconda opzione
.P
Paragrafo con un esempio di utilizzo:
.br
.nf

mioprogramma "con opzioni"

.fi
.SH OPTIONS
.TP
.I −r
Fa qualcosa di utile.
.TP
.I −n
Opzione per non fare nulla
.SH AUTHOR
MioNome MioCognome
.SH SEE ALSO
utile(1), tools(2)
.SH DIAGNOSTICS
Programma utile.
.SH BUGS
```

Ora abbiamo bisogno di formattare il codice appena creato con

```
groff −man −Tascii manpage > mioprogramma.man
```

E cosí abbiamo creato la nostra man page. Questa operazione poteva essere fatta anche mediante un Makefile. Quasi tutti i programmi durante l'installazione creano la man page e poi la installano nella directory di sistema, di solito /usr/man/catN, dove N è il numero di man page nel nostro caso 4.

2.12 Creiamo un RPM.

Supponete ora di aver fatto la vostra applicazione, e volete che oltre al tarball, sia disponibile in formato RPM, ad uso e consumo degli utenti Linux, che usino il Red Hat Package Manager, cosa fareste? In questo paragrafo vediamo come creare un file RPM contentente i vostri eseguibili, e un RPM contenente i sorgenti.

Supponiamo di aver creato il nostro programma di elaborazioni di matrici, che chiameremo matrix, e questo programma abbia una man page e un file di configurazione, chiamato matrix.cfg, che va posto nella directory /etc. Per poter creare un rpm, contentente tali file è necessario :

- Assicurarsi che il file /etc/rpmrc sia impostato in maniera corretta
- Creare un file di specifiche sul pacchetto

- Creare il pacchetto mediante tale file di specifiche

Esempio di /etc/rpmrc.

Vediamo come puo essere settato il file di configurazione di RPM:

```
require_vendor: 1
  distribution: Red Hat
  require_distribution: 1
  topdir: /usr/src/packs
  vendor: NerSoft
  packager: NerSoft Gestione Software <nrsof@soft.it>

  optflags: i386 −O2 −m486
  optflags: alpha −O2
  optflags: sparc −O2

  signature: pgp
  pgp_name:
  pgp_path: /home/packages/.pgp

  tmppath: /tmp
```

La linea 1, obbliga RPM a trovare una linea vendor, dove si specifica di solito il produttore del pacchetto. Le linee:

```
 distribution: Red Hat
  require_distribution: 1
```

Sono direttamente collegate poichè se la seconda è settata a 0, nella prima non è necessario specificare per quale distribuzione debba essere creato il pacchetto. Tralasciamo le due linee successive, che sembrano abbastanza chiare e passiamo ad optflags, li vengono specificati i flag da passare al compilatore per rendere il programma ottimizzato per la data piattaforma, l'ottimizzazione viene fatta durante la compilazione del pacchetto stesso. Le restanti linee servono per l'autenticazione del pacchetto. Oltre a queste opzioni per il file /etc/rpmrc, ne esistono altre che potete scoprire con il comando : rpm −−showrc.

Creiamo il file di specifiche

Dopo aver settato a modo l'rpm, creiamo il nostro file di specifiche, matrix−1.0.SPEC:

```
Summary:Pacchetto di elaborazione matematica matrix
Name: matrix
Version: 1.0
Release: 1
Copyright: GPL
Group: Utility
Source: ftp://ftp.techno.it/matrix.tar.gz

%description
Progetto di Laboratorio. Parser per l'elaborazioni di matrici.
%files
%config /etc/matrix.cfg
```

```
/usr/bin/matrix
/usr/man/man4/matrix.4
```

Cerchiamo di descrivere a modo questo file, e i suoi campi:

- **Summary.** Contiene una descrizione riassuntiva del pacchetto
- **Name.** È la label che verrà assegnata al pacchetto all'interno del database dopo l'installazione.
- **Version.** Versione del software.
- **Release.** Numero di versione indicante il numero di volte che il pacchetto è stato impacchettato per questa versione.
- **Copyright.** Indica chi detiene il copyright su quel determinato pacchetto.
- **Group.** Indica a quale gruppo deve appartenere il pacchetto.
- **Source.** Serve per indicare l'URL in cui è possibile reperire il pacchetto.
- **Description** Da una generale panoramica sull'utilizzo del pacchetto.
- **Files** Indica quali siano i file che fanno parte del pacchetto. Ogni file di configurazione è necessario farlo precedere dal %config.

Creiamo il pacchetto RPM

Ora copiamo i file nelle posizioni specificate dal file .SPEC nel nostro sistema (tanto per intenderci l'eseguibile *matrix* va in /usr/bin,la man page in /usr/man/man4, il file di configurazione *matrix.cfg* in /etc). A questo punto possiamo fare la creazione del pacchetto con il comando:

```
cyberspace.org:~# rpm −bb matrix−0.1.SPEC
Processing files: matrix
Finding provides...
Finding requires...
Requires: ld−linux.so.2 libc.so.6 libc.so.6(GLIBC_2.0)
Wrote: /usr/src/rpm/RPMS/i386/matrix−0.1−1.i386.rpm
```

Abbiamo ora ottenuto il nostro pacchetto, installiamolo:

```
cyberspace.org:/usr/src/rpm/RPMS/i386 # rpm −i matrix−0.1−1.i386.rpm
E interroghiamolo:
cyberspace.org:~# rpm −qi matrix
Name        : matrix                       Relocations: (not relocateable)
Version     : 0.1                               Vendor: NerSoft
Release     : 1                             Build Date: Thu Feb 10 20:41:11 2000
Install date: Fri Feb 11 11:54:55 2000      Build Host: liz.cyberspace.org
Group       : Utility                       Source RPM: matrix−0.1−1.src.rpm
Size        : 25168                            License: GPL
Summary     : Pacchetto di elaborazione matrici matrix
Description :
Pacchetto di elaborazioni di semplici matrici
```

Un pacchetto contenente i sorgenti

da fare

2.13 Risorse e approfondimenti sugli strumenti di sviluppo

3. Iniziamo a programmare

3.1 Le parole chiave del C.

Per chi avesse già programmato in altri linguaggi, il C è case−sensitive, cioé il linguaggio distingue tra lettere minuscole e lettere maiuscole. Le parole chiave del C sono tutte miniscole e sono:

```
auto    break  case     char     const    continue goto   switch
default do     double   else     enum     extern   float  for    struct
if      int    long     register return   short    signed sizeof static
typedef union  unsigned void     volatile while
```

3.2 Come è composto un programma C?

La composizione di un programma C puo essere riassunta in quattro elementi di base:

- espressioni
- istruzioni
- blocchi di istruzioni
- blocchi di funzioni

Espressioni

Una espressione e composta da operandi e operatori e all'interno di un'espressione non possono essere usate parole chiavi. Non è lecito per esempio scrivere: `c=auto+2`

Diamo alcuni esempi di espressioni lecite:

```
ESEMPIO 1. 2+3
ESEMPIO 2. d+z
ESEMPIO 3. b>a
ESEMPIO 4. 300
ESEMPIO 5. b=c+a
```

Saltiamo i primi quattro esempi, che sono banali somme, confronti o valori costanti e passiamo all'esempio 5. Tale esempio significa:*"Prendi il contenuto della variabile c, prendi il contenuto della variabile a, sommali e ponili nella variabile b"*. Per capire quanto appena detto occorre la seguente definizione.

Definizione. *Una variabile è una locazione di memoria, ove si possono scrivere e/o leggere dati. In C viene rappresentata mediante un'identificatore alfanumerico.*

Esempio

```
 a=10;   /* a e' una variabile, a cui assegnamo il valore 10 */

Contenuto della variabile    | 10 | <- VARIABILE A
                             |____|
```

Le istruzioni

Un'istruzione è un comando C completo. Tutte le istruzioni terminano con il carattere ; . Esempio:

- Istruzione 1: a=1000;
- Istruzione 2: c=1;
- Istruzione 3: b=a+c;

Il contenuto delle variabili e il seguente (n.d, sta per non disponibile):

```
Istruzione 1 :  |1000|    |n.d |    | n.d|
                |____|    |____|    |____|
                  a         b         c


Istruzione 2 :  |1000|    |n.d |    | 1  |
                |____|    |____|    |____|
                  a         b         c

Istruzione 3 :  |1000|    |1001|    | 1  |
                |____|    |____|    |____|
                  a         b         c
```

I blocchi di istruzioni

I blocchi di istruzioni consistono in una o più istruzioni raggruppate tra un carattere di inzio ({) e uno di fine (}), ad esempio:

```
if (a<b) {
          c=a*b;
          a=a+1;
         }
```

In questo esempio il blocco di istruzioni è quello che segue la parola chiave **if**, ovvero:

```
c= a*b;
a= a+1;
```

Il significato di questo spezzone di codice C è:"Se il valore(contenuto) della variabile a è minore di quello della variabile b, allora poni nella variabile c il valore di a moltiplicato per quello di b, ed aumenta il valore della variabile a di 1". Supponiamo a=4 e b=10, allora sarebbe:

```
Blocco codice   |    Descrizione
                |
if (a<b) {      |
          c=a*b;|
          a=a+1;|
         }      |  | 4  |    | 10 |    | n.d|
                |  |____|    |____|    |____|
                |    a         b         c
                |   E'  a minore di b? Risposta: Si
                |   Allora:
                |
                |  Poni in c=a*b, ed aumenta a di 1.
                |
                |  | 11 |    | 10 |    | 40 |
                |  |____|    |____|    |____|
                |    a         b         c
                |
```

Blocchi di funzione.

Un blocco di funzione è composto da uno o più blocchi di istruzioni e da istruzioni combinate in modo da svolgere un compito specifico. Durante lo sviluppo dei programmi, istruzioni e blocchi di istruzioni vengono usati dalle funzioni per creare un singolo modulo per risolvere un singolo aspetto di un problema. Esaminiamo la seguente funzione:

Linea	Codice
1	int area_triangolo(void)
2	{
3	int base,area,altezza;
4	base=10;
5	altezza=5;
6	area=(base*altezza)/2
7	return area;

8	} **Tabella 1.7**

Questa funzione ha una dichiarazione fatta in questo modo:

tipo_valore nome_funzione (tipo_valori_parametri)

La funzione **area_triangolo** restituisce un valore di tipo int, che sta per numero intero rappresentante l'area di un triangolo di base=10. Notiamo ora la linea 3. Precedentemente avevamo parlato di variabili, e in questa linea abbiamo dato il modo per definire una variabile: **tipo_variabile nomevariabile;**

Se le variabili come in questo caso sono dello stesso tipo, basta allora specificarle in questo modo:
tipo_variabile nomevariabile1, nomevariabile2, nomevariabile3,ecc..;

Altresí si definiranno le variabili in quest'altro modo:

```
tipo1_variabile nomevar,.....;
tipo2_variabile nomevar,.....;
.............................;
.............................;
tipoN_variabile nomevar,.....;
```

Ritorneremo sui tipi nel prossimo capitolo, per ora mancano solo da spiegare di questa funzione: l'istruzione return e il parametro void.

Per capire questa istruzione e il parametro void occorre dire che una funzione in un linguaggio di programmazione ha un significato abbastanza simile alla matematica:

```
  f:A->B
Funzione da A in B
```

Una funzione prende dagli altri sottoprogrammi un insieme A di valori e restiuisce un insieme B di valori. Nel caso A sia vuoto (come nel nostro esempio) si specifica void nei parametri della funzione:

```
int area_triangolo(void)
                   ^^^^^^
```

Nel caso B sia vuoto si speficica void prima della definizione della funzione e si ommette l'istruzione return, esempio:

```
void nome_funzione(void)
  {
   int base;
```

```
 base=0;
}
```

Nel caso in cui le nostre funzioni debbano restituire uno valore, occorre dichiarare nel prototipo il tipo del valore che la data funzione restituisce, come fatto nella linea 1 della funzione area_triangolo, ed occorre usare l'istruzione return prima di chiudere il blocco funzione, restituendo il valore elaborato dalla nostra funzione.

3.3 Il primo programma.

```
$ emacs primo.c
```

Una volta avviato emacs, scriviamo il nostro primo programma C:

```
/* Primo programma in C */
#include<stdio.h>
int main(void) {
printf("Ciao Mondo!\n");
return 0;
}
```

Esempio 1.0

La prima cosa che notiamo nell'esempio soprastante sono i simoboli `/*` e `*/`. Il primo ha il significato di **apri commento** e il secondo di **chiudi commento**, vediamone un ulteriore esempio:

```
/*
            Questo è un commento insensato:
             secondo voi, chi è più bella:
            Laetitia Casta, Cindy Crawford o Megan Gale ? */
```

Il compilatore ignora tutto ciò, che trova tra `/*` e `*/`, queste coppie di caratteri sono chiamate deliminatori di commento. I commenti, durante la stesura del codice, sono molto utili sia per ricordarci cosa fa il programma che nella fase di debug. Si può infatti vedere facilmente come si comporta il programma, commentando una parte di esso e poi ricompilandolo.

Una cosa importante da notare è che ogni commento aperto con /* deve essere chiuso con */, infatti in ANSI C non è lecito scrivere commenti di questo tipo:

```
/*
Il programma analizza
/* le derivate ennesime */
```

```
*/
```

Notiamo ora la seconda linea dell'esempio 1.0, che è una direttiva al preprocessore. Infatti il codice di un programma scritto in C può includere alcune istruzioni dirette al compilatore. Queste istruzioni sono dette **direttive al preprocessore**. In parole povere, tutte le linee che iniziano con il carattere # sono direttive al preprocessore, la nostra è appunto:

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

Il significato specifico di tale linea di programma è il seguente: *"Cerca su disco (alla posizione* `/usr/include`*) il file stdio.h e inserisci in questo punto del programma tutto ciò, che trovi su quel file"*. I file *.h vengono anche chaimati header in quanto fanno riferimento aa una data libreria. Il file `stdio.h`**(stdio = standard input output)** si usa in tutti i programmi, che richiedono una qualche forma di immissione o emissione dati (leggi I/O, input/output dei dati), ovvero quasi la totalità dei programmi. Il file `stdio.h` permette di includere nel programma le funzioni di input/output, a cui il programma può fare riferimento, in questo esempio la routine di I/O a cui si fa riferimento è la `printf`. Osservate il seguente esempio:

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

Apparentemente tra la prima e la seconda linea di codice, non v'è nessuna differenza, se escludiamo le virgolette, ma dal punto di vista del compilatore la differenza è sostanziale.

Nel primo caso il file header, è posto nella directory standard degli header, `/usr/include`, invece nel secondo è posto nella directory corrente. In generale possiamo osservare che:

- Le *parentesi angolari*, che precedono e seguono un file header indicano al preprocessore di cercare il file header richiesto nella directory standard.
- Le *virgolette*, invece, indicano al preprocessore di cercare il file header nella directory corrente e poi nella directory standard.

Vediamo un altro esempio:

```
#include "/home/deneb/myio.h"
```

In quest'ultimo caso, ho specificato esattamente il percorso dove trovare l'header myio.h.

Andando avanti con l'esempio 1.0 troviamo la funzione `int main(void)` . Come dovreste aver già capito questa funzione restituisce un valore intero e prende un valore nullo:

```
                    _______________
          void   |               |
```

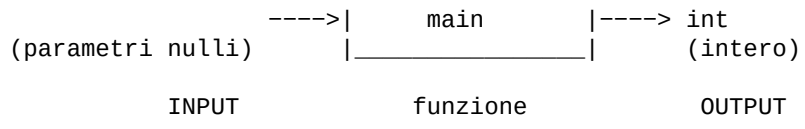
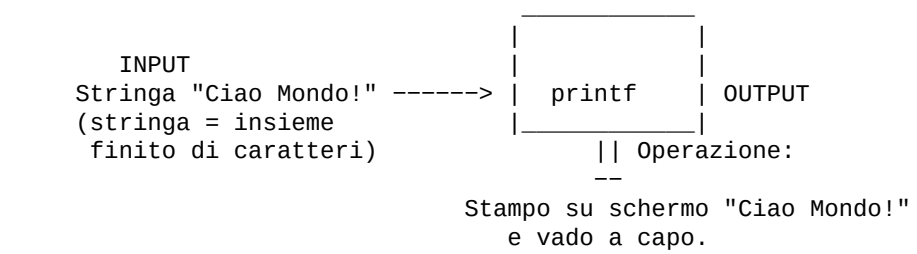


Figura 1.7

Si noti che all'interno della funzione `int main(void)` abbiamo la chiamata ad un'altra funzione:

```
printf("Ciao, Mondo !");
```

Questa è una funzione di I/O per cui richiede l'header , ed ha il compito di stamapare i caratteri compresi tra `"` e `"` sullo schermo. Il carattere `\n` (**newline**) fa in modo che quando a fine stringa si vada a capo. Nella `printf`, come in ogni altra funzione, tutto ciò che è compreso tra (e) costituisce argomento della funzione stessa:



Dopo la `printf`, restituiamo il valore **EXIT_SUCCESS** in quanto supponiamo che le operazioni siano state svolte in maniera corretta.

Macro di uscita

Macro	**Valore**
`EXIT_SUCCESS`	0
`EXIT_FAILURE`	1 **Tabella 1.8**

Osservate la Tabella 1.8. Essa presenta due macro definite dalla libreria GNU, e rappresentanti i valori di uscita di un programma nel caso venga eseguito correttamente o nel caso vi siano errori. In Linux, è buona pratica nella funzione principale *(leggi main)* specificare un valore di uscita per il programma. Per definizione se il programma ha eseguito tutte le operazioni correttamente uscirà con valore 0, altresí con valore 1. Le macro `EXIT_SUCCESS` e `EXIT_FAILURE` sono state create per migliorare la leggibiltà del codice, infatti le istruzioni:

```
return 1;        return EXIT_FAILURE;
```

sono equivalenti.

3.4 Ancora sulle funzioni

Nell'esempio 1.0 abbiamo visto una funzione, che richiama un'altra funzione, ma per far ciò la funzione deve essere *visibile* alla funzione che la richiama motivo per cui l'header è posto all'inizio del programma e non in fondo. In generale un programma C è cosí composto:

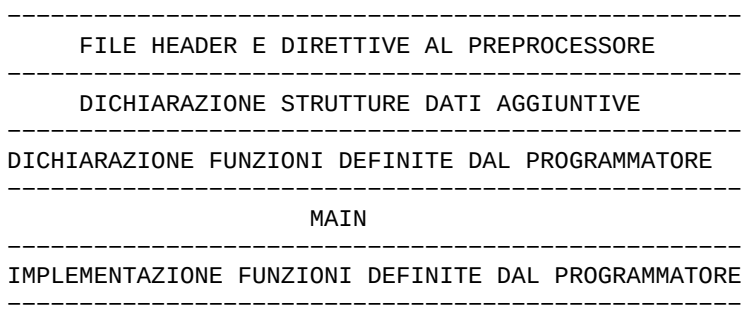


Figura 1.8

La Figura 1.8 da una descrizione generica della struttura di un programma C,vedremo nel corso del libro ogni singolo aspetto sia delle strutture dati che delle direttive al preprocessore, le altre cose le abbiamo già iniziate a vedere e implementare.

L'esecuzione di ogni programma, al di là della sua struttura, inizia dal main e prosegue discendendo per ogni funzione e istruzione in esso contenuta fino a terminare il blocco funzione del main stesso. Esempio:

```
/* File Header */

#include <stdio.h>

/* Dichiarazione funzioni definite dal programmatore */

int quadrato(int valore);

/* La funzione duplica prende un valore intero
e ne restituisce il quadrato */

int main(void)
{
int a;
a=quadrato(25);
printf("Valore del quadrato di 25: %d\n", a);
return EXIT_SUCCESS;
}

/* Implementazione della funzione quadrato */

int quadrato(int valore)
```

```
{
int quad;
quad=valore*valore;
return quad;
}
```

Esempio 1.1

Spieghiamo il programma schematilizzandolo:

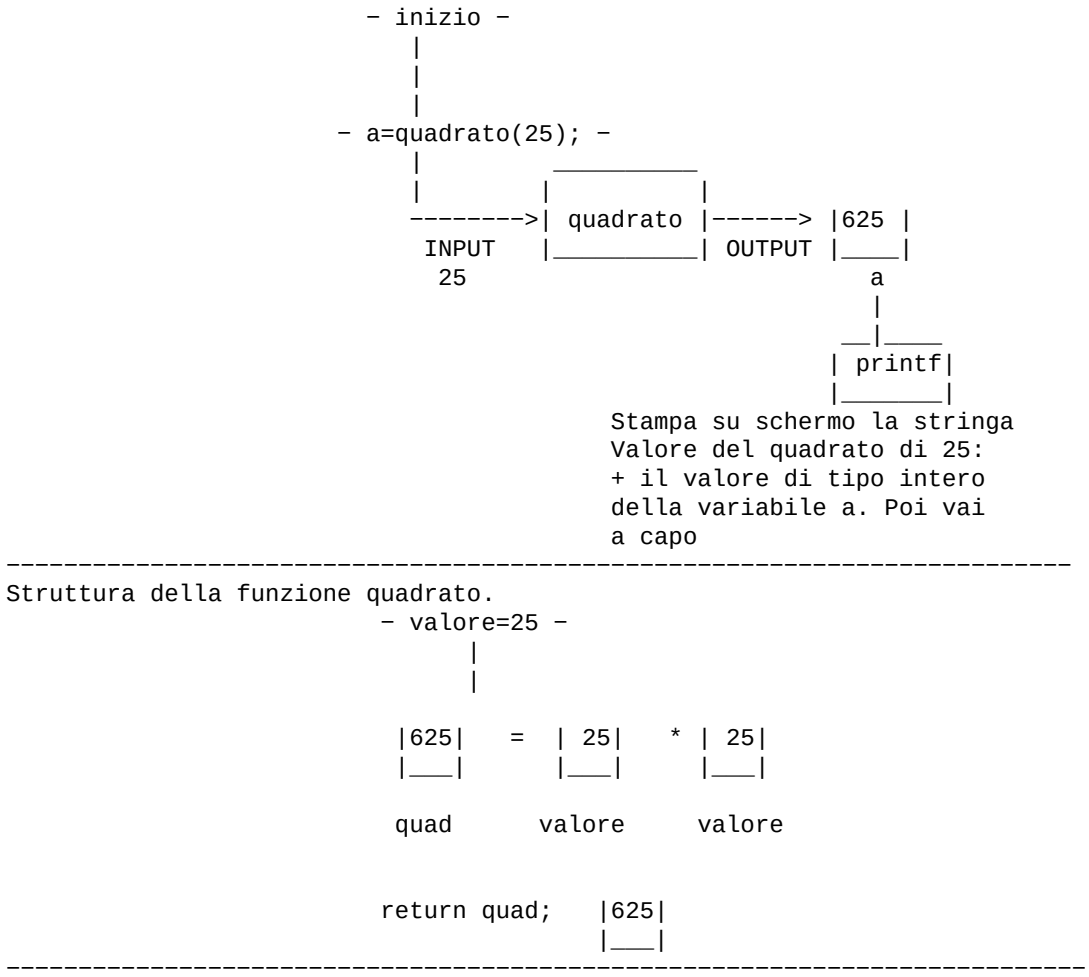


Figura 1.9

La prima cosa che notiamo nell'esempio 1.1, è la dichiarazione della funzione quadrato di tipo intero che ha come parametro una variabile intera e restituisce un valore intero, rappresentato dal quadrato dell'input stesso. Bisogna ricordarsi, che le funzioni vengono viste in C come scatole nere, tutte le variabili dichiarate al loro interno non hanno visibilità all'esterno, ovvero se io usassi la variabile `quad` all'esterno della funzione quadrato si verificherebbero errori, a meno che tale variabile non venga dichiarata nuovamente localmente. Durante l'esecuzione, come è visibile dallo schema, alla variabile a viene assegnato il valore della funzione quadrato e poi stampato con la:

```
printf("Valore del quadrato di 25: %d", a);
```

Questa significa :*stampa la stringa **Valore del quadrato di 25** e aggiungi ad essa il valore di tipo intero della variabile a*. Il simbolo %d sta appunto ad indicare il *valore di tipo intero della variabile a*. Per cui l'output su video del programma è:

```
Valore del quadrato di 25: 625
```

Abbiamo ora finito questo paragrafo e presentiamo alcuni esercizi, che siete già in grado di svolgere.

Esercizi.

1. Modificare l'esempio 1.0 in modo che l'output sia:

   ```
   Ciao,
   quando
   ti ciucci il calzino?
   ```

   Usando prima un'unica chiamata a printf, e poi invece tre chiamate diverse a printf.
2. Riprendere l'esempio di funzione area_triangolo, modificandola in questo modo:

   ```
   int area_triangolo(int base, int altezza)
   ```

   Stampare nella funzione principale l'area triangolo, avendo assegnato i valori base=10; e altezza=25;.
3. Modificare l'esempio 1.1, il corpo della funzione principale (main) affinche alla funzione quadrato sia passata una variabile anziche il valore 10. Tale variabile deve precedentemente dichiarata e inizializzata a 10.
4. Indicare riga per riga gli errori presenti nel seguente programma:

   ```
   #include "stdio.h"
   */ Li troverai gli errori? */
   int func2(int valore1, int valore2);

   Main()
   {
   int b;
   a=func(b);
   b=4;b=b+a;
   a=func2(a,b);
   return EXIT_SUCCESS;
   }
   void func(int value)
   {
   int c;
   c=value*4;
   return 4;
   }
   int func2(int valore1, int valore2)
   {
   ```

```
int c;
c= valore1+valore2;
}
/* Gli hai trovati gli errori? /*
```

3.5 Direttive al preprocessore

da fare.

4. ChangeLog

```
25-01-2000 Giorgio Zoppi <deneb@linux.it>
        * Revisione Parziale del testo, esclusi paragrafi 2.4, 2.5.
        * Aggiunto tutorial su automake e autoconf
27-01-2000 Giorgio Zoppi <deneb@linux.it>
        * Aggiunta introduzione agli editor di testo
          a cura di Marco Abis <abis@programmazione.it>
29-01-2000 Giorgio Zoppi <deneb@linux.it>
        * Aggiunta scrivere le man page
31-01-2000 Monica O. <ozean@tiscalinet.it>
        * Aggiunto nuovo logo
11-02-2000 Giorgio Zoppi <deneb@linux.it>
        * Spiegato il primo programma
        * Aggiunta mailing list
        * Aggiunta prima parte sui makefiles un po più decenti
        * Aggiunta prima parte sul RPM.
        * Corretti alcuni errori di spelling
```

5. TO DO (Da fare)

```
DA FARE:
* Aggiungere ulteriori macro alle tabelle dell'autoconf, ed dell'automake
* Completare gli esempi sui makefile , comprese le estensioni GNU
  (qualche volontario?)
* Rifare tutti gli schizzi ascii, in disegni fatti con GIMP
* Revisione e correzione bozze su Emacs
* Iniziare tutorial su Emacs
* Iniziare tutorial sul DATA Display Debugger
* Ampliare parte sul RPM.
* elenco di risorse on line e non inerenti gli strumenti di
  programmazione descritti(libri e siti, tanto per intenderci)
* Spiegare i tipi di dati
* Spiegare le direttive al preprocessore
```