How to…

gcc e make

I passi della compilazione

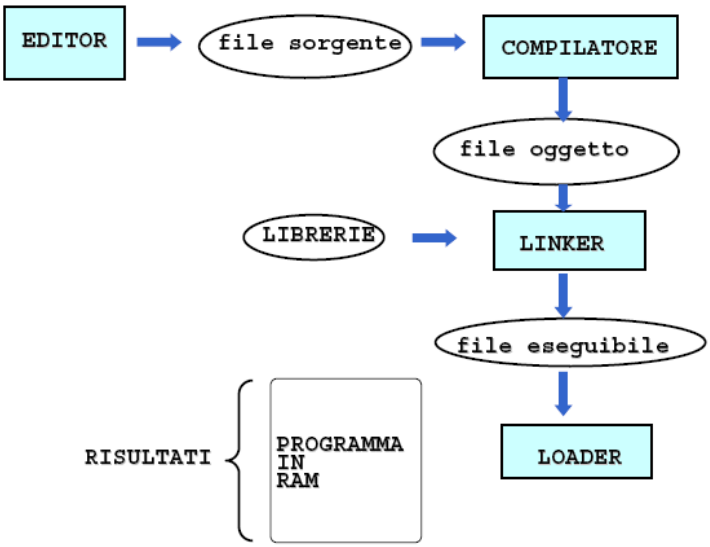
Sia gcc che g++ processano file di input attraverso uno o piu' dei seguenti passi:

1) preprocessing
-rimozione dei commenti
-interpretazioni di speciali direttive per il preprocessore denotate da "#" come:
#include - include il contenuto di un determinato file, Es. #include<math.h>
#define -definisce un nome simbolico o una variabile, Es. #define MAX_ARRAY_SIZE 100

2) compilation
-traduzione del codice sorgente ricevuto dal preprocessore in codice assembly

3) assembly
-creazione del codice oggetto

4) linking
-combinazione delle funzioni definite in altri file sorgenti o definite in librerie con la funzione main() per creare il file eseguibile.



Pre-compilazione

Il **preprocessor** e' un programma che viene attivato dal compilatore nella fase precedente alla compilazione, detta di **precompilazione**.

Il **preprocessor** legge un sorgente C e produce in output un altro sorgente C, dopo avere espanso in linea le macro, incluso i file e valutato le compilazioni condizionali o eseguito altre direttive.

Una direttiva inizia sempre con il carattere pound '#' eventualmente preceduto e/o seguito da spazi. I token seguenti definiscono la direttiva ed il suo comportamento.
Una direttiva al preprocessor puo' comparire in qualsiasi punto del sorgente in compilazione ed il suo effetto permane fino alla fine del file.

Espansione in linea delle macro

Definire una macro significa *associare una stringa ad un identificatore*.
Ogni volta che il preprocessore C incontra l'identificatore cosi' definito, esegue la sua sostituzione in linea, con la stringa ad esso associata.
La definizione delle macro avviene per mezzo della direttiva #define
Esempio:

```
  #define MAX   100
  #define STRING_ERR "Rilevato errore !\n"
```

Le macro possono essere definite anche in forma paramentrica; in tal caso la sostituzione dei parametri formali con quelli attuali avviene in modo testuale durante la fase di espansione della macro.
Esempio:

```
#define NUM_SCRITTORI 100

void main(){…

For(int i=0;i<NUM_SCRITTORI;i++) …

}
```

Questo consente di avere costanti con un nome (per esempio NUM_SCRITTORI rappresenta il numero di processi scrittori da allocare) usate in tutto il sorgente, definite in un solo posto e aggiornate automaticamente dappertutto non appena il valore cambia.

Se un parametro formale e' preceduto dal carattere pound **#**, il suo valore attuale e' espanso testualmente come stringa.
Esempio:

```
#define DEBUG_OUT(expr)       fprintf(stderr, #expr " = %g\n", (float)(expr))
...
DEBUG_OUT(x*y+z);     /* espansione:
                       * fprintf(stderr, "x*y+z" " = %g\n", (float)(x*y+z))
                       */
```

E' anche possibile annullare la definizione di una macro con la direttiva #undef. Di solito #undef e' impiegata per assicurarsi che una funzione sia definita come tale, piuttosto che come macro. Un altro possibile impiego di #undef e' per la gestione della compilazione condizionale.

Esempio:

```
#undef DEBUG
```

Tipicamente sono predefinite 5 macro:

- **__LINE__** Valore decimale del numero della linea corrente del sorgente.
- **__FILE__** Stringa del nome del file in corso di compilazione.
- **__DATE__** Stringa della data di compilazione (formato Mmm dd yyyy).
- **__TIME__** Stringa dell'ora di compilazione (formato hh:mm:ss).
- **__STDC__** Contiene il valore 1 se il compilatore e' conforme allo standard ANSI.

Di solito il compilatore accetta nella linea di comando delle opzioni per definire e cancellare delle macro analogamente a quanto viene eseguito con #define e #undef.
Tipicamente tali opzioni sono **-Dmacro** o **-Dmacro=def** per definire una macro e **-Umacro** per eliminare la definizione.
Le opzioni **-D** e **-U** vengono eseguite prima di cominciare l'attivita' di preprocessing sul sorgente.

Uso ed esempi

Usare ifndef per stampare un messaggio di debug

```
#define DEBUG

#ifdef DEBUG

      Printf(“MESSAGGIO DI DEBUG”);

#endif
```

Visto che DEBUG è diverso da 0 le la linea del printf viene compilata ed aiuta nel debug del programma. Successivamente omettendo #define DEBUG 1 le linee non verranno compilate e quindi eseguite.

Inclusione di file (#include)

Le definizioni ricorrenti delle macro, le dichiarazioni dei prototype di funzione e delle variabili esterne, di solito vengono scritte, una volta per tutte, in files tradizionalmente chiamati header ed aventi normalmente estensione .h.

Il preprocessore C, tramite la direttiva **#include**, puo' ricercare il file indicato in alcune directory standard o definite al momento della compilazione ed espanderlo testualmente in sostituzione della direttiva.

Considerato che nel C ogni funzione, variabile, macro **deve** essere definita o dichiarata prima del suo utilizzo, risulta evidente che ha senso includere gli header file all'inizio, cioe' nella **testata** (e da qui deriva il nome di header), del file sorgente.

La direttiva #include puo' essere impiegata in due forme:

```
#include <nomefile>
#include "nomefile"
```

Nel 1° caso il nomefile viene ricercato in un insieme di directory standard definite dall'implementazione ed in altre che sono specificate al momento della compilazione.
Nel caso della versione Unix del compilatore, le directory standard di ricerca degli header potrebbero essere /usr/include, /usr/local/include, ...
Nel 2° caso il nomefile viene ricercato nella directory corrente e poi, se non e' stato trovato, la ricerca continua nelle directory standard e in quelle specificate al momento della compilazione come nel 1° caso.

N.B. - Nel caso che un header venga modificato, e' necessario ricompilare tutti i sorgenti che lo includono.

Compilazione condizionale

Il preprocessore C al verificarsi di alcune condizioni puo' includere o escludere parti del codice sorgente alla compilazione.
Le direttive che indicano al preprocessore la **compilazione condizionata** sono riportate di seguito:

#if **espressione_costante_intera**
#ifdef **identificatore**
#ifndef **identificatore**
#else
#elif **espressione_costante_intera**
#endif

dove #if, #ifdef, #ifndef testano la condizione. Se risulta verificata, viene incluso per la compilazione il codice dalla riga successiva alla direttiva, fino ad incontrare una delle direttive #else, #elif o #end.

In particolare #if testa l'espressione_costante_intera e se risulta diversa da zero, la condizione e' considerata verificata positivamente. L'espressione_costante_intera non puo' comprendere costanti di tipo enumerativo, operatori di cast e sizeof.

#ifdef considera superata la condizione se e' definito identificatore.
#ifdef identificatore equivale a #if defined (identificatore) o #if defined identificatore
#ifndef considera superata la condizione se non e' definito identificatore.
#ifndef identificatore equivale a #if !defined (identificatore) o #if !defined identificatore

La parte di codice successiva a #else viene passata al compilatore nel caso cha la #if, #ifdef, #ifndef non sia stata soddisfatta.
La direttiva #elif equivale ad #else #if tranne il fatto di non aumentare di un livello di annidamento l'intera #if.
La direttiva #endif chiude la #if, #ifdef, #ifndef del corrispondente livello di annidamento.
Esempio:

```
#ifdef DEBUG
  fprintf(stderr, "Linea di Debug %d\n", (int)__LINE__);
#endif
```

Compilazione

Come passo intermedio, gcc trasforma il vostro codice in Assembly. Per farlo deve capire cosa intendevate fare analizzando il vostro codice. Se avete commesso degli errori di sintassi ve lo dirà e la compilazione fallirà. Di solito la gente confonde questo passo con l'intero processo di compilazione. Ma c'è ancora molto lavoro da fare per gcc.

Assembly

`as` trasforma il codice Assembly in codice oggetto. Il codice oggetto non può ancora essere lanciato sulla CPU, ma ci si avvicina molto. L'opzione `-c` trasforma un file .c in un file oggetto con estensione .o Se lanciamo

```
gcc -c game.c
```

creiamo automaticamente un file che si chiama game.o Qui siamo incappati in un punto importante. Possiamo prendere un qualsiasi file .c e creare un file oggetto da esso. Come vedremo più avanti potremo combinare questi file oggetto in un eseguibile nella fase di Link. Andiamo avanti col nostro esempio. Poiché stiamo programmando un gioco di carte abbiamo definito un mazzo di carte come un `deck_t`, creeremo una funziona per mischiare il mazzo. Questa funzione prenderà un puntatore a un tipo deck e lo riempie con delle carte messe a caso. Tiene traccia di quali carte sono già state aggiunte con l'array 'drawn'. Questo arraydi DECKSIZE elementi ci impedisce di duplicare un valore di carta.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "deck.h"

static time_t seed = 0;
```

```
void shuffle(deck_t *pdeck)
{
  /* Tiene traccia di che numeri sono stati usati */
  int drawn[DECKSIZE] = {0};
  int i;

  /* Inizializzazione da fare una volta di rand */
  if(0 == seed)
  {
    seed = time(NULL);
    srand(seed);
  }
  for(i = 0; i < DECKSIZE; i++)
  {
    int value = -1;
    do
    {
      value = rand() % DECKSIZE;
    }
    while(drawn[value] != 0);

    /* segna il valore come usato */
    drawn[value] = 1;

    /* codice di debug */
    printf("%i\n", value);
    pdeck->card[i] = value;
  }
  pdeck->dealt = 0;
  return;
}
```

Salvate questo file come `shuffle.c`. Abbiamo inserito del codice di debug in questo codice in modo che, quando viene lanciato, scriva i valori delle carte che genera. Questo non aggiunge niente alle funzionalità del programma, ma è cruciale adesso che non possiamo vedere cosa succede. Visto che stiamo appena cominciando il nostro gioco non abbiamo altro modo di sapere se la nostra funzione fa quello che vogliamo. Con il comando printf potete vedere esattamente cosa sta succedendo in modo che quando passeremo alla prossima fase sapremo che il mazzo viene mescolato bene. Dopo che saremmo soddisfatti del buon funzionamento potremo togliere la linea dal nostro codice. Questa tecnica di debugging può sembrare spartana, ma fa quello che deve fare con il minimo sforzo. Discuteremo di debugger più sofisticati in seguito.

Notate due cose.

1. Passiamo il parametro per indirizzo, e lo si capisce dall'operatore '&' (indirizzo di). Questo passa l'indirizzo in memoria della variabile alla funzione, in modo che la funzione possa cambiare la variabile stessa. È possibile programmare con variabili globali, ma dovrebbero essere usate molto raramente. I puntatori sono una parte importante del C e dovreste imparare a usarli molto bene.
2. Stiamo usando una chiamata a funzione da un nuovo file .c. Il Sistema Operativo cerca sempre una funzione di nome 'main' e inizia l'esecuzione da lí. `shuffle.c` non ha una funzione 'main' e quindi non può essere trasformato in un eseguibile indipendente. Dobbiamo attaccarlo ad un altro programma che abbia una 'main' e che chiami la funzione 'shuffle'.

Lanciate il comando

```
gcc -c shuffle.c
```

e accertatevi che crei un nuovo file chiamato `shuffle.o`. Editate il file game.c e, alla linea 7, dopo la dichiarazione della variabile `deck` di tipo deck_t, aggiungete la linea

```
shuffle(&deck);
```

Ora, se tenteremo di creare un eseguibile allo stesso modo di prima otterremo un errore

```
gcc -o game game.c
```

```
/tmp/ccmiHnJX.o: In function `main':
/tmp/ccmiHnJX.o(.text+0xf): undefined reference to `shuffle'
collect2: ld returned 1 exit status
```

La compilazione è andata a buon fine perché la sintassi era corretta. La fase di link è fallita perché non abbimo detto al compilatore dove fosse la funzione 'shuffle'. Cos'è la fase di *link* e come diciamo al compilatore dove trovare questa funzione?

Link

Il linker, `ld`, prende il codice oggetto precedentemente creato da `as` e lo trasforma in un eseguibile col comando

```
gcc -o game game.o shuffle.o
```

Questo unisce i due codici oggetto insieme e crea l'eeguibile `game`.

Il linker trova la funzione `shuffle` nell'oggetto shuffle.o e la include nell'eseguibile. La cosa veramente interessante dei file oggetto è il fatto che se volessimo usare di nuovo quella funzione, tutto ciò che dovremmo fare sarebbe includere il file "deck.h" e unire il file oggetto `shuffle.o` nel nuovo file eseguibile.

Il riutilizzo di codice, come in questo caso, viene fatto spesso. Non abbiamo scritto la funzione `printf` che abbiamo chiamato prima come funzione di debug, il linker ha trovato la sua definizione nel file che abbiamo incluso con `#include <stdlib.h>` e l'ha unita al codice oggetto contenuto nella libreria C (/lib/libc.so.6). In questo modo possiamo usare le funzioni di qualcun'altro, sapendo che funzionano, e preoccuparci solo di risolvere i nostri problemi. Questo è il motivo per cui i file header solitamente contengono solo le definizioni delle funzioni e non il codice vero e proprio. Normalmente si creano file oggetto o librerie che il linker metterà nell'eseguibile. Un problema potrebbe sorgere con il nostro codice poiché non abbiamo messo nessuna definizione di funzione nel nostro header. Cosa possiamo fare per essere sicuri che tutto funzioni bene?

Altre Due Opzioni Importanti

L'opzione `-Wall` abilita tutti i tipi di avvertimenti sulla sintassi del linguaggio per essere sicuri che il nostro codice sia corretto e il più portabile possibile. Quando usiamo questa opzione e compiliamo il nostro codice vediamo qualcosa come:

```
game.c:9: warning: implicit declaration of function `shuffle'
```

Questo ci permette di sapere che abbiamo ancora un po' di lavoro da fare. Dobbiamo aggiungere una linea al nostro header dove diamo al compilatore tutte le informazionisulla nostra funzione `shuffle` in modo che possa fare tutti i controlli di cui ha bisogno. Sembra una difficoltà inutile, ma separa la definizione dall'implementazione e ci permette di usare la nostra funzione dove vogliamo semplicemente includendo il nostro nuovo header nel codice oggetto. Metteremo questa riga nel file deck.h

```
void shuffle(deck_t *pdeck);
```

Questo ci libererà del messaggio di avvertimento.

Un'altra opzione comune del compilatore è l'ottimizzazione `-O#` (per esempio -O2). Questo dice al compilatore il livello di ottimizzazione che vogliamo. Il compilatore conosce un sacco di trucchi per far andare più veloce il nostro codice. Per un programma piccolo come il nostro non noterete alcuna differenza, ma per programmi più grandi potreste ottenere buoni aumenti di velocità. Lo vedete dappertutto, quindi dovreste sapere cosa significa.

Altri flag che possono tornare utili

Sebbene GCC abbia centinaia di flag che si possono usare, perlopi`u molti non vengono comunemente usati o vengono inclusi in blocchi da altri (per esempio `-Wall` `

e un alias per una ventina di flag pi`u specifici). Qui ti riporto alcune di quelle ancora non viste ma che incontrerai frequentemente.

- `-D<MACRO NAME>` (nota che non c'`e uno spazio tra la `D` e il nome della macro) equivale ad inserire nei file sorgente da compilare la riga

  ```
  #define <MACRO_NAME>
  ```

  In pratica e' utilizzato per attivare oppurtuni flag in compilazione, come nel seguente esempio:

```
g++ -DDEBUG ciao.cpp -o ciao
```

che attiver`a eventuali controlli presenti nel codice del tipo

```
#ifdef DEBUG ...(codice C++)... #endif
```

Molto utile.

- `-S` invece di generare il compilato genera un file di testo con dentro l'assembly del programma. Quando hai pi`u file non esegue il linking. Utile se sai leggere l'assembly e hai tempo da perdere.

- `-E`fa girare solo il preprocessore (`cpp`, C Pre-Processor), quindi include il codice sorgente degli header, sostituisce le macro ecc. Utile se vuoi capire come sono sono andate a finire le tue macro.

- `-static` linka al tuo sorgente tutte le librerie che il tuo programma utilizzer`a, compresa una copia di tutte quelle di sistema che gli servono. Il file eseguibile diventa enorme ma ha il grande pregio di funzionare anche dove mancano le librerie che servono o dove ci sono versioni differenti. Utile se il programma viene distribuito in vari formati (sorgente, eseguibile, autinstallante ecc.) . Attenzione a non abusarne.

- `-b <machine>` ebbene s´ı, potete usare il vostro Pentium per compilare un programma per Digital Alpha o processori RISC, basta che abbiate le librerie necessarie a bordo. Ricordiamoci che il compilatore alla fine `e solo un traduttore da un linguaggio ad un altro!

- `-v` mostra tutte le singole azioni che sta compiendo il compilatore. Istruttivo.

- `-pg` genera codice aggiuntivo che insieme al programma `gprof` permette di studiare il tempo speso dal vostro software nelle varie funzioni, in modo da capire dove sono i colli di bottiglia. Studia `gprof` prima.

- `-B<directory>` aggiunge la directory indicata al PATH dove cercare librerie, include, eseguibili, dati e il compilatore stesso. Utile se si ha fretta e non vuoi specificare ogni singola voce.

- `-mcpu=i386 -mcpu=i486 -mpcu=i585 -mcpu=i686 -mcpu=pentium`

  Specifica quale cpu si ha a disposizione per creare codice macchina che utilizzi le funzionalit`a speciali del proprio processore. Utile, ma attenzione a distribuire in giro l'eseguibile: quello che funziona su un pentium non `e detto che funzioni su un 486.

Debugging

Come sappiamo tutti, il fatto che il codice venga compilato non significa che funzioni nel modo che vogliamo. Potete verificare che tutti i numeri vengano usati una sola volta lanciando

```
game | sort - n | less
```

e controllando che non ne manchino. Cosa facciamo se c'è un problema? Come troviamo l'errore? Potete controllare il codice con un debugger. La maggior parte delle distribuzioni rendono disponibile il classico gdb. Se la linea di comando vi spaventa come lo fa a me, KDE offre una buona interfaccia con KDbg. Ci sono anche altre interfacce, e sono molto simili. Per iniziare il debugging, scegliete File->Executable e trovate il vostro programma `game`. Quando premete F5 o selezionate Execution->Run dal menu dovreste vedere l'output su un'altra finestra. Cosa succede? Non vediamo niente nella finestra. Non preoccupatevi, non è KDbg che non funziona. Il problema nasce dal fatto che non abbiamo messo informazioni di debug nel nostro codice eseguibile, quindi Kdbg non può dirci cosa sta succedendo internamente. L'opzione del compilatore `-g` inserisce le

informazioni richieste nei file oggetto. Dovete compilare i file oggetto (estensione .o) con questa opzione, quindi il comando diventa

```
gcc -g -c shuffle.c game.c
gcc -g -o game game.o shuffle.o
```

Questo inserisce degli agganci nell'eseguibile per permettere a gdb e KDbg di capire cosa sta succedendo. Il debugging è un'abilità importante, vale il tempo che dedicherete a imparare a usarla bene. Il modo in cui i debugger aiutano i programmatori è la loro capacità di mettere dei 'Breakpoint' nel codice sorgente. Provate a inserirne uno facendo click col tasto destro sulla linea che chiama la funzione `shuffle`. Un piccolo cerchio rosso dovrebbe apparire vicino alla riga. Ora, quando premete F5 il programma ferma la sua esecuzione a quella linea. Premete F8 per entrare *dentro* la funzione shuffle. Hey, ora state guardando il codice dentro `shuffle.c`! Possiamo controllare l'esecuzione passo per passo e vedere cosa sta realmente accadendo. Se passate il puntatore su una variabile locale potete vedere cosa contiene. Carino. Molto meglio di quelle istruzioni `printf`, no?

Compilare un programma di un solo file

Per prima cosa verifichiamo se sul sistema e presente il compilatore che ci serve; digitiamo con la shell il comando

```
g++ --version
```

Se la risposta `e un numero del tipo 3.4.2 allora quella `e la versione del compilatore che avete a disposizione. Tenete a mente che le informazioni sugli errori di compilazione o il programma eseguibile che otterrete compilando, sono dipendenti dalla versione del compilatore.

A questo punto abbiamo il compilatore, per cui facciamolo girare su un nostro programma, per esempio (salviamo il codice sorgente che segue con il nome `ciao.cpp`[1]):

```
#include <iostream> using namespace std; int
main() {

   cout << "Ciao.\n";


}
```

Alla riga di comando, nella directory di `ciao.cpp` scriviamo

```
g++ ciao.cpp
```

se non appare alcun messaggio vuole dire che tutto è andato correttamente, e nella stessa directory troveremo un nuovo file `a.out` che possiamo eseguire:

```
a.out
```

(o `./a.out` se la directory corrente non `e presente nel PATH, come spesso accade). Il programma svolge il suo compito mostrando la scritta

```
Ciao. .
```

IL COMANDO 'MAKE' IN AMBIENTE LINUX

Il programma *make* e il concetto di *makefile* fecero la loro prima comparsa in ambiente Unix nel 77 ad opera di **Stuart Feldman** presso i laboratori Bell. Questo strumento ha avuto una tale importanza nella storia dello sviluppo del software che nel 2003 l'ACM (Association for Computing Machinery) ha insignito Feldman dell'*ACM Software System Award.* In questo articolo effettueremo i primi passi con make, vedremo a cosa serve e come creare i nostri makefile.

A cosa servono make ed i makefile

Esistono vari tool che implementano il concetto di *makefile* ma dato che l'implementazione GNU è la più diffusa e portabile concentreremo su di essa la nostra attenzione. Semplificando molto i makefile sono file di testo che consentono di specificare e gestire le dipendenze che intercorrono tra un certo gruppo di file. Una volta definite tali dipendenze esse vengono processate dal programma *make* che si occupa di eseguire le istruzioni contenute nel makefile. Storicamente i makefile sono stati inventati per coadiuvare il lavoro dei programmatori (principalmente C/C++) minimizzando i tempi di compilazione dei programmi e gestendo in maniera automatica o semiautomatica le dipendenze tra i vari moduli. In realtà GNU *make* mette a disposizione moltissime funzionalità che consentono di utilizzarlo negli ambiti più disparati.

Per funzionare make ha bisogno di una shell su cui girare. Sotto Windows possiamo utilizzare MinGW/MSYS (Maggiori dettagli su MinGW e MSYS li potete trovare nell'articolo "Istallare MinGW, MSYS e Insight"). Gli utenti GNU/Linux in genere hanno l'utility make istallata di default ed hanno a disposizione varie shell come Bourne Shell, C Shell, Korn Shell etc.

Il primo makefile

Cominciamo con un esempio pratico, vediamo come compilare un semplice programma C utilizzando un makefile. Di seguito è riportato il nostro programma C:

```
/* file: main.c */

#include <stdio.h>

int main(int argc, char *argv[])

{

  printf("Ciao mondo!\n");
  return 0;
}
```

Immagina di sviluppare ulteriormente questo programma, aggiungendo feature, funzioni, strutture, etc. Ti accorgerai presto che per testarlo lo avrai ricompilato moltissime volte, probabilmente richiamando più volte un comando come:

```
# > gcc -o main.exe main.c -Wall -O1
```

Per evitare di scrivere tutte le volte lo stesso comando potremmo creare uno script, usare comandi shell come "!gcc" o creare un makefile. Già in un caso così semplice il makefile è la scelta migliore. Di seguito viene riportato un semplice makefile che consente di compilare il nostro programma. Si noti che la quarta linea è indentata rispetto alla terza con un carattere *tab*. Se si usano spazi invece che il tab make riporterà un errore. Copiate il contenuto del makefile che vedete qua sotto all'interno di un file chiamato "Makefile" (notate la maiuscola) ed assicuratevi che la quarta linea cominci con un tab.

```
# file: Makefile
all:
    gcc -o main.exe main.c -Wall -O1
```

Le linee che cominciano con il carattere **#** sono dei commenti. Una volta salvato il makefile nella stessa direcotory in cui avete il programma main.c sopra descritto, basterà posizionarsi nella stessa directory utilizzando una shell e lanciare il comando "make".

```
#> make
gcc -o main.exe main.c -Wall -O1

#> main
Ciao mondo!
```

Se il nostro makefile si chiama "Makefile" (nota la maiuscola) possiamo lanciare semplicemente make senza parametri. Infatti in questo caso make cercherà di default un file chiamato Makefile nella directory corrente. Se il nostro makefile ha un nome differente, ad esempio "dipendenze.make" dovremo lanciare make utilizzando l'opzione "-f" come nell'esempio:

```
#> make -f dipendenze.make
```

```
gcc -o main.exe main.c -Wall -O1

#> main
Ciao mondo!
```

Makefile Components

- **Comments**

  Comments are any text beginning with the pound (#) sign. A comment can start anywhere on a line and continue until the end of the line. For example:

  ```
  # $Id: slides,v 1.2 1992/02/14 21:00:58 reggers Exp $
  ```

- **Macros**

  Make has a simple macro definition and substitution mechanism. Macros are defined in a Makefile as = pairs. For example:

  ```
  MACROS=  -me
  PSROFF=  groff -Tps
  DITROFF= groff -Tdvi
  CFLAGS= -O -systype bsd43
  ```

  There are lots of default macros -- you should honor the existing naming conventions. To find out what rules/macros make is using type:

  ```
  % make -p
  ```

  NOTE: That your environment variables are exported into the make as macros. They will override the defaults.

  You can set macros on the make command line:

  ```
  % make "CFLAGS= -O" "LDFLAGS=-s" printenv
     cc -O printenv.c -s -o printenv
  ```

Il Makefile ed i *target* del make

Per funzionare make ha bisogno che voi scriviate un file chiamato **Makefile** in cui siano descritte le relazioni fra i vostri files ed i comandi per aggiornarli. Quando il make viene invocato esegue le istruzioni contenute nel Makefile.
Una idea base che bisogna capire del make e' il concetto di target . Il primo target in assoluto e' il Makefile stesso. se si lancia il make senza aver preparato un Makefile si ottiene il seguente risultato

```
make
make: No targets specified and no makefile found.  Stop.
```

Quello che segue e' un semplice Makefile in cui sono stati definiti tre target e tre azioni corrispondenti:

```
# Un esempio di Makefile
```

```
one:
        @echo UNO!

two:
        @echo DUE!

three:
        @echo E TRE!
```

La definizione di un target inizia sempre all'inizio della riga ed seguito da **:** . Le azioni (in questo caso degli output su schermo) seguono le definizioni di ogni target e, anche se in questo esempio sono singole, possono essere molteplici. La prima riga, che inizia con **#,** e' un commento.

Per utilizare i target invochiamoli sulla riga di comando del make:

```
make one
UNO!
make one two three
UNO!
DUE!
E TRE!
```

Se non si invoca nessun target nella linea di comando, make assume come default il primo che trova nel Makefile:

```
make
UNO!
```

IMPORTANTE: le linee in cui si specificano le azioni corrispondenti ad ogni target (Es. @echo UNO!) devono iniziare con un separatore <TAB>!
Il seguente Makefile non e' valido perche' la riga seguente la definizione del target non inizia con un separatore <TAB>:

```
# Un esempio di Makefile mal scritto

one:
  @echo UNO!

make one
Makefile:4: *** missing separator.  Stop.
```

Le righe di azione devo iniziare invariabilmente con un separatore <TAB>, **NON POSSONO ESSERE UITLIZZATI DEGLI SPAZI!**

Dipendenze
E' possibile definire delle dipendenze fra i target all' interno del Makefile

```
# Un esempio di Makefile con dipendenze

one:
        @echo UNO!

two:    one
        @echo DUE!

three:  one two
        @echo E TRE!

all:    one two three
```

```
@echo TUTTI E TRE!
```

Si noti come i target vengono elaborati in sequenza:

```
make three
UNO!
DUE!
E TRE!
make all
UNO!
DUE!
E TRE!
TUTTI E TRE!
```

Macro e variabili ambiente
E' possibile definere delle Macro all' interno del Makefile

```
#Definiamo la Macro OBJECT

OBJECT=PIPPO

one:
        @echo CIAO $(OBJECT)!


make

CIAO PIPPO!
```

Possiamo ridefinire il valore della macro OBJECT direttamente sulla riga di comando, senza alterare il Makefile!

```
make OBJECT=pippa

CIAO pippa!
```

- **Continuation of Lines**

  Use a back slash (\). This is important for long macros and/or rules.

- **Conventional Macros**

  There are lots of default macros (type "make -p" to print out the defaults). Most are pretty obvious from the rules in which they are used:

```
AR = ar
GFLAGS =
GET = get
ASFLAGS =
MAS = mas
AS = as
FC = f77
CFLAGS =
CC = cc
LDFLAGS =
LD = ld
LFLAGS =
```

```
LEX = lex
YFLAGS =
YACC = yacc
LOADLIBS =
MAKE = make
MAKEARGS = 'SHELL=/bin/sh'
SHELL = /bin/sh
MAKEFLAGS = b
```

- **Special Macros**

  Before issuing any command in a target rule set there are certain special macros predefined.

  1. $@ is the name of the file to be made.
  2. $? is the names of the changed dependents.

  So, for example, we could use a rule

```
printenv: printenv.c
        $(CC) $(CFLAGS) $? $(LDFLAGS) -o $@
```

  alternatively:

```
printenv: printenv.c
        $(CC) $(CFLAGS) $@.c $(LDFLAGS) -o $@
```

  There are two more special macros used in implicit rules. They are:

  3. $< the name of the related file that caused the action.
  4. $* the prefix shared by target and dependent files.

Il Makefile puo' accedere alle variabili ambiente:

```
# Usiamo una variabile ambiente

OBJECT=$(TERM)

one:
        @echo CIAO $(OBJECT)!
```

```
make

CIAO xterm!
```

- **Makefile Target Rules**

  The general syntax of a Makefile Target Rule is

```
target [target...] : [dependent ....]
[ command ...]
```

Items in brackets are optional, ellipsis means one or more. Note the tab to preface each command is required.

The semantics is pretty simple. When you say "make target" make finds the target rule that applies and, if any of the dependents are newer than the target, make executes the com- mands one at a time (after macro substitution). If any dependents have to be made, that happens first (so you have a recursion).

A make will terminate if any command returns a failure sta- tus. That's why you see rules like:

```
clean:
        -rm *.o *~ core paper
```

Make ignores the returned status on command lines that begin with a dash. eg. who cares if there is no core file?

Make will echo the commands, after macro substition to show you what's happening as it happens. Sometimes you might want to turn that off. For example:

```
install:
        @echo You must be root to install
```

- **Example Target Rules**

For example, to manage sources stored within RCS (sometimes you'll need to "check out" a source file):

```
SRCS=x.c y.c z.c

$(SRCS):
        co $@
```

To manage sources stored within SCCS (sometimes you'll need to "get" a source file):

```
$(SRCS):
        sccs get $@
```

Alternativley, to manage sources stored within SCCS or RCS let's generalize with a macro that we can set as required.

```
SRCS=x.c y.c z.c
# GET= sccs get
GET= co

$(SRCS):
        $(GET) $@
```

For example, to construct a library of object files

```
lib.a: x.o y.o z.o
        ar rvu lib.a x.o y.o z.o
        ranlib lib.a
```

Alternatively, to be a bit more fancy you could use:

```
OBJ=x.o y.o z.o
AR=ar

lib.a: $(OBJ)
```

```
        $(AR) rvu $@ $(OBJ)
        ranlib $@
```

Since AR is a default macro already assigned to "ar" you can get away without defining it (but shouldn't).

If you get used to using macros you'll be able to make a few rules that you can use over and over again. For example, to construct a library in some other directory

```
INC=../misc
OTHERS=../misc/lib.a

$(OTHERS):
        cd $(INC); make lib.a
```

Alcuni target standard

Esistono alcuni target standard usati da programmatori Linux e GNU. Fra questi:

- **install**, viene utilizzato per installare i file di un progetto e puo' comprendere la creazione di nuove directory e la assegnazione di diritti di accesso ai file.
- **clean**, viene utilizzato per rimuovere dal sistema i file oggetto (*.o), i file core, e altri file tempornei creati in fase di compilazione
- **all**, di solito utilizzato per richiamare altri target con lo scopo di costruire l'intero progetto.

Aggiungiamo il target **clean** al nostro Makefile:

```
OBJECTS=main.o myfunc.o

CC=g++

CFLAGS=-g -Wall
LIBS=-lm

PROGRAM_NAME=prova



$(PROGRAM_NAME):$(OBJECTS)
        $(CC) $(CFLAGS) -o $(PROGRAM_NAME) $(OBJECTS) $(LIBS)

        @echo " "
        @echo "Compilazione completata!"
        @echo " "

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

Invocare il target clean comporta la cancellazione di tutti i file oggetto e del file core.

```
make clean
rm -f *.o
rm -f core
```

- **Make Dependencies**

It's pretty common to have source code that uses include files. For example:

```
% cat program.c

#include
#include        "defs.h"
#include        "glob.h"
        etc....
main(argc,argv)
        etc...
```

The implicit rule only covers part of the source code depen- dency (it only knows that program.o depends on program.c). The usual method for handling this is to list the dependen- cies separately;

```
        etc...
        $(CC) $(CFLAGS) -c $*.c
        etc...
program.o: program.c defs.h glob.h
```

Usually an implicit rule and a separate list of dependencies is all you need. And it ought to be easy enough to figure out what the dependencies are.

However, there are a number of nice tools around that will automatically generate dependency lists for you. For example (trivial):

```
DEPEND= makedepend $(CFLAGS)
        etc...
# what are the source dependencies

depend: $(SRCS)
        $(DEPEND) $(SRCS)

        etc....
# DO NOT DELETE THIS LINE -- ....

printenv.o: /usr/include/stdio.h
```

These tools (mkdepend, mkmkf, etc.) are very common these days and aren't too difficult to use or understand. They're just shell scripts that run cpp (or cc -M, or etc.) to find out what all the include dependencies are. They then just tack the dependency list onto the end of the Makefile.

Compiliamo con make

Supponiamo di voler compilare il seguente codice C++ composto da tre moduli (main.cpp, myfunc.cpp e myfunc.h) usando il comando make.

```
// main.cpp
#include<iostream>
#include"myfunc.h"

int main() {
  int a=6;
  int b=3;
```

```
  cout<<"a="<<a<<", b="<<b<<endl;
  cout<<"a/b="<<div(a,b)<<endl;
  cout<<"a*b="<<mul(a,b)<<endl;
  cout<<"a^b="<<pot(a,b)<<endl;

  return 0;
}
```

```
// myfunc.cpp
#include<math.h>

int div(int a, int b) {
  return a/b;
};

int mul(int a, int b) {
  return a*b;
};

float pot(float a, float b) {
  return pow(a,b);
}
```

```
    // myfunc.h

    int div(int a, int b);
    int mul(int a, int b);
    float pot(float a, float b);
```

Un semplice Makefile si presenta cosi':

```
OBJECTS=main.o myfunc.o
CFLAGS=-g -Wall
LIBS=-lm
CC=g++
PROGRAM_NAME=prova

$(PROGRAM_NAME):$(OBJECTS)
        $(CC) $(CFLAGS) -o $(PROGRAM_NAME) $(OBJECTS) $(LIBS)
        @echo " "
        @echo "Compilazione completata!"
        @echo " "
```

Il make ricompilera' il *target* **prova** se i files da cui questo *dipende* (gli OBJECTS **main.o** e **myfunc.o**) sono stati modificati dopo che prova e' stato modificato l'ultima volta oppure non esistono. Il processo di ricompilazione avverra' secondo la regola descritta nell' *azione del target* e usando le *Macro* definite dall' utente (CC, CFLAGS, LIBS).
Per compilare usiamo semplicemente

```
make
g++    -c -o main.o main.cpp
g++    -c -o myfunc.o myfunc.cpp
g++ -g -Wall -o prova main.o myfunc.o -lm

Compilazione completata!
```

Se modifichiamo solo un modulo, per esempio myfunc.cpp, il make effettuera' la compilazione di questo file solamente.

```
make
g++    -c -o myfunc.o myfunc.cpp
g++ -g -Wall -o prova main.o myfunc.o -lm
```

```
Compilazione completata!
```

Un esempio di progetto gestito con make

Vediamo adesso un makefile un po' più complesso che gestisce un progetto fatto di 3 header e 3 file C:

```c
/* main.c */

#include <stdio.h>

#include "somma.h"

#include "modulo.h"


int main(int argc, char *argv[])

{

  vettore_t vect1 = { 10.0f, 20.0f, 30.0f };
  vettore_t vect2 = { 50.0f, 60.0f, 70.0f };
  vettore_t v;
  v = somma(vect1, vect2);
  printf("v = %f, %f, %f\n", v.x, v.y, v.z);
  printf("modulo = %f\n", modulo(v));
  return 0;

}
```

```c
/* somma.c */

#include "vettore.h"



vettore_t somma(vettore_t v1, vettore_t v2) {
 vettore_t tmp;
 tmp = v1;
 tmp.x+=v2.x;
 tmp.y+=v2.y;
 tmp.z+=v2.z;
 return tmp;
}
```

```
/* modulo.c */

#include <math.h>

#include "vettore.h"


float modulo(vettore_t v) {
  return sqrt(v.x*v.x + v.y*v.y + v.z*v.z);

}
```

```
/* somma.h */

#include "vettore.h"

vettore_t somma(vettore_t v1, vettore_t v2);
```

```
/* modulo.h */

#include "vettore.h"

float modulo(vettore_t v);
```

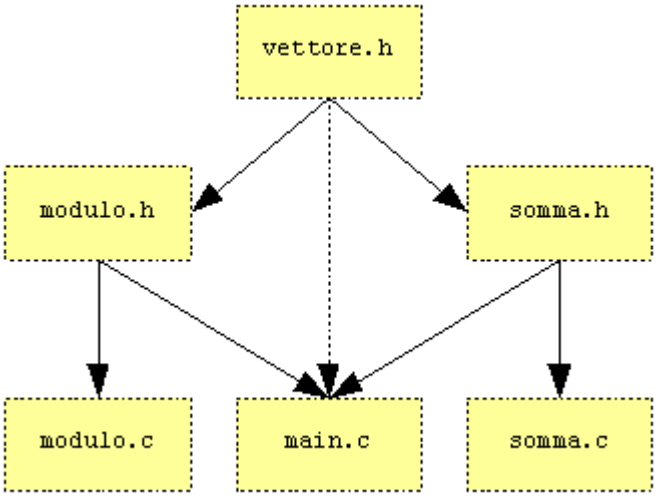
```
/* vettore.h */

#ifndef VETTORE_T_DEFINITO

#define VETTORE_T_DEFINITO


typedef struct {
  float x,y,z;
} vettore_t;
```

```
#endif
```

Ovviamente per un programma così semplice normalmente non c'è bisogno di usare tutti questi file ma in questo caso ciò è utile alla spiegazione, vedremo meglio perché.

Il file principale del nostro progetto è "main.c" in cui utilizziamo le funzioni *somma* e *modulo* implementate nei rispettivi file C. Per utilizzare le funzioni *somma* e *modulo* in "main.c" includiamo i file "somma.h" e "modulo.h". Ciò significa in gergo "makefile-ese" che "main.c" dipende da "somma.h" e "modulo.h" perché ogni cambiamento apportato a questi due header ha ripercussioni su "main.c". A loro volta "somma.h" e "modulo.h" dipendono da "vettore.h" perché lo includono per poter utilizzare il tipo "vettore_t" definito in quel file. Ma a sua volta "main.c" dipende da "vettore.h" attraverso "somma.h" e "modulo.h". Infatti se noi modificassimo la definizione di "vettore_t" contenuto in "vettore.h" togliendo ad esempio il campo "z", esso avrebbe ripercussioni su "main.c" che non compilerebbe più dato che utilizza tale campo. Il file "vettore.h" non include nessun altro file per cui non dipende da nessuno.

La seguente figura illustra lo schema delle dipendenze tra i file del nostro progetto:



Senza utilizzare i makefile possiamo compilare il progetto utilizzando il seguente comando:

```
#> gcc -o main.exe main.c somma.c modulo.c -Wall -O1
```

In pratica diciamo al compilatore di ricompilare sempre tutti i file C. Evidentemente man mano che il nostro progetto cresce e include sempre più file, funzioni e dipendenze i tempi di compilazione e di linking cominciano ad appesantire lo sviluppo e ci accorgiamo che non ha senso ricompilare tutti i file C se ho modificato ad esempio solamente "somma.c". Sarebbe più logico compilare solo i file modificati salvando di volta in volta i file C compilati (file oggetto .o).

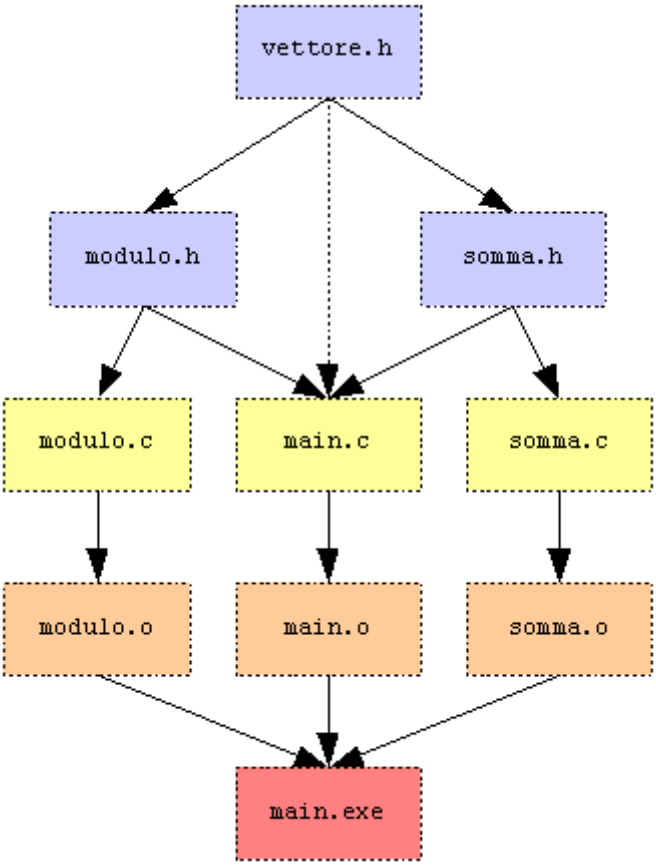
Il seguente makefile consente di fare proprio questo:

```
# main.exe dipende da main.o, somma.o e modulo.o
main.exe: main.o somma.o modulo.o
   gcc -o main.exe main.o somma.o modulo.o # linka i file .o
   strip main.exe # rimpiccolisce l'eseguibile
# main.o dipende da main.c
main.o: main.c
   gcc -c main.c -Wall -O1

# somma.o dipende da somma.c
somma.o: somma.c
   gcc -c somma.c -Wall -O1

# modulo.o dipende da modulo.c
modulo.o: modulo.c
   gcc -c modulo.c -Wall -O1
```

Questo esempio ci mostra in maniera più chiara la sintassi di un make file. In pratica quando make viene lanciato controlla la prima condizone che trova, in questo caso "main.exe: main.o somma.o modulo.o". Nel nostro esempio "main.exe" è detto *target* e "main.o somma.o modulo.o" sono detti *dipendenze* di "main.exe". Se uno dei file elencati nella lista delle dipendenze è più giovane rispetto a "main.exe" (perché creato o modificato dopo quest'ultimo) allora make esegue la lista di comandi elencati subito sotto la condizione. Nota che tale lista di comandi è indentata rispetto alla condizione utilizzando un carattere di tabulatura. Indentare tali comandi con spazi è un errore.
Schema generale di un makefile:

```
# Questo e' un commento
target1:  dipendenza1, dipendenza2 ... dipendenzaN
[  TAB  ] commando1 # i comandi sono indentati utilizzando un TAB
[  TAB  ] commando2          .
          .
          .
[  TAB  ] commandoN

target2:  dipendenza1, dipendenza2 ... dipendenzaN
[  TAB  ] commando1 # i comandi sono indentati utilizzando un TAB
[  TAB  ] commando2          .
          .
          .
[  TAB  ] commandoN

...
```

Il comando make non si limita a controllare la prima dipendenza ma controlla anche le altre dipendenze collegate a questa. Infatti come è facile notare dall'esempio, "main.exe" dipende si da "main.o", "somma.o" e "modulo.o" ma anche loro sono a loro volta target di condizioni che vengono specificate in seguito. In pratica una volta appurato che "main.exe" dipede ad esempio da "somma.o", make va a controllare la condizione "somma.o: somma.c". Se "somma.c" è stato modificato in seguito alla data di ultima modifica di "somma.o" (o se "somma.o "non esiste) allora procede alla compilazione di "somma.c" invocando il comando "gcc -c somma.c -Wall -O1" il quale genera un nuovo file "somma.o" cosa che a sua volta invalida la prima condizione "main.exe: main.o somma.o modulo.o" e quindi rende necessaria l'esecuzione dei comandi specificati da quest'ultima: "gcc -o main.exe main.o somma.o modulo.o" e "strip main.exe". In questo modo ogni qualvolta un file C è modificato make genera il relativo file .o e riesegue il linking generando un nuovo eseguibile.
Il seguente grafico mostra le dipendenze tra i file .exe .o .c e .h computate da make.



Gestire le dipendenze dagli header

Ma cosa succede quando un header è modificato? Semplice, tutti i file C che lo includono devono essere ricompilati. Per fare questo modifichiamo il nostro makefile in modo che gestisca anche questa situazione:

```
# main.exe dipende da main.o, somma.o e modulo.o
main.exe: main.o somma.o modulo.o
    gcc -o main.exe main.o somma.o modulo.o
    strip main.exe

# main.o dipende da main.c, somma.h, modulo.h, vettore.h
main.o: main.c somma.h modulo.h vettore.h
    gcc -c main.c -Wall -O1

# somma.o dipende da somma.c, somma.h vettore.h
somma.o: somma.c somma.h vettore.h
    gcc -c somma.c -Wall -O1

# modulo.o dipende da modulo.c, modulo.h, vettore.h
modulo.o: modulo.c modulo.h vettore.h
    gcc -c modulo.c -Wall -O1
```

Non abbiamo fatto altro che aggiungere alle dipendenze di ogni file .o gli header inclusi dal relativo file C. Nota che anche gli header inclusi da tali header vanno messi nella lista, come è il caso di "vettore.h" che non è direttamente incluso da "main.c" ma è incluso in "somma.h" e "modulo.h" (inclusi da "main.c").

In questo modo il nostro nuovo makefile è in grado di capire che se modifichiamo "vettore.h" è necessario ricompilare tutti i file C dato che ognuno di essi lo include indirettamente. L'unica cosa che il programmatore deve fare è tenere aggiornata la lista delle dipendenze. Spesso tenere aggiornata tale lista non è un compito semplice ed eventuali dimenticanze possono facilmente generare errori di linking o crash inaspettati a runtime. Per questo motivo esistono tool che permettono di generare automaticamente tali dipendenze, ad esempio tramite l'utilizzo di *makedepend*, del GCC ("gcc -MM"), di *sed* o script vari.

Le variabili di un makefile

All'interno dei nostri makefile possiamo anche dichiarare delle variabili che ci permettono di mantenere il tutto più leggibile e configurabile, ad es:

```
CFLAGS=-Wall -O1
LDFLAGS=
CC=gcc
PROGRAMMA=vettori
OBJECTS=main.o somma.o modulo.o

$(PROGRAMMA).exe:  $(OBJECTS)
   $(CC) -o $(PROGRAMMA).exe $(OBJECTS) $(LDFLAGS)
   strip $(PROGRAMMA).exe

main.o: main.c somma.h modulo.h vettore.h
   $(CC) -c main.c $(CFLAGS)

somma.o: somma.c somma.h vettore.h
   $(CC) -c somma.c $(CFLAGS)

modulo.o: modulo.c modulo.h vettore.h
   $(CC) -c modulo.c $(CFLAGS)

.PHONY: clean
clean:
   rm $(PROGRAMMA).exe $(OBJECTS) -f
```

Utilizzando le variabili in questo modo possiamo ad esempio configurare i flag di compilazione in maniera molto più veloce e pulita, definire i file oggetto una volta sola, definire in manera più evidente il nome dell'eseguibile ed evitare di ripeterlo più volte etc.

Si noti l'istuzione ".PHONY: clean" alla fine del makefile. Essa dice a make che di seguito è riportato il target "clean" e che tale target non è associato ad un file. In particolare il target "clean" elimina i file .o e l'eseguibile. Per richiamare uno specifico target basta invocare "make nometarget", ad esempio digitare "make clean" ottiene il seguente risultato:

```
#> make clean
rm vettori.exe main.o somma.o modulo.o -f
```

In teoria nulla vieta di chiamare gli altri target (ad es. "make main.o") ma ciò non ha alcuna utilità nel nostro caso.

Deduzione automatica delle regole

Volendo possiamo lasciare a make il compito di dedurre cosa deve fare, ad esempio il seguente makefile compila il nostro progetto usando una sintassi molto più succinta:

```
CFLAGS=-Wall -O1
LDFLAGS=
CC=gcc
PROGRAMMA=vettori
OBJECTS=main.o somma.o modulo.o
INCLUDE=somma.h modulo.h vettori.h

%.o: %.c $(INCLUDE)
    $(CC) -c -o $@ $< $(CFLAGS)

$(PROGRAMMA).exe: $(OBJECTS)
    $(CC) -o $@ $^ $(LDFLAGS)
    strip $(PROGRAMMA).exe

.PHONY: clean
clean:
    rm $(PROGRAMMA).exe $(OBJECTS) -f
```

Rispetto al precedente ha il vantaggio rimanere di dimensioni molto ridotte anche quando il nostro progetto contiene decine di file. Lo svantaggio è che in questo modo non possiamo definire in maniera fine le dipendenze dei nostri file. Infatti utilizzando questo makefile modificare un file header significa ricompilare tutti i file C anche quelli che non ne dipendono.

In questo makefile make riconosce ed usa automaticamente le variabili "CC" e "CFLAGS". Inoltre si noti l'uso delle variabili speciali "$@", "$^" e "$<", che di volta in volta prendono come valore rispettivamente il target, la lista delle dipendenze, la prima dipendenza. Inoltre la regola generica

"%.o: %.c $(INCLUDE)" viene applicata a tutti i file .o specificati nella regola "$(PROGRAMMA).exe: $(OBJECTS)" estraendone automaticamente in nomi con estenzione .c da quelli .o.

Se disponete di una macchina multiprocessore o hyperthreading potete provare ad utilizzare l'opzione "-j" seguita dal numero di processi che volete utilizzare per effettuare la compilazione. Ad esempio il comando "make -j3" alloca tre processi per effettuare la risoluzione delle dipendenze definite nel makefile. In questo caso fate attenzione che le operazioni portate a termine dalle singole regole siano indipendenti tra di loro o che i programmi invocati possano essere eseguiti in simultanea. Se ad esempio più regole invocano uno stesso programma che durante la sua esecuzione scrive in tutte le sue istanze in uno stesso file, la probabilità che tale file venga corrotto da accessi simultanei non sincronizzati è molto alta. Comunque se usate make alla maniera "tradizionale" per compilare i vostri programmi difficilmente incorrerete in questi problemi.

Librerie statiche

Le librerie statiche sono semplicemente una raccolta di comuni file oggetto; per convenzione, i nomi delle librerie statiche terminano con il suffisso "`.a`". Una tale raccolta si crea utilizzando il programma `ar` (dall'inglese *archiver*). Le librerie statiche non sono più utilizzate tanto spesso quanto in passato, per via dei vantaggi che caratterizzano le librerie condivise (descritte in seguito). Ciononostante, esse vengono ancora talvolta utilizzate, storicamente sono venute prima e sono più semplici da illustrare.

L'utilizzo di librerie statiche ne consente il link a programmi eseguibili senza che ne debba essere ricompilato il codice, risparmiando tempo di compilazione. Si noti che, data la maggiore velocità dei compilatori odierni, il tempo di ricompilazione è divenuto meno determinante, così che questa esigenza non è più tanto sentita quanto in passato. Le librerie statiche sono spesso utili agli sviluppatori che vogliano consentire ad altri programmatori di utilizzarle, ma che non siano intenzionati a distribuire il codice sorgente delle librerie stesse (il che può essere un vantaggio per chi vende una libreria, ma ovviamente non lo è per il programmatore che cerchi di utilizzarla). In teoria, la velocità di esecuzione del codice di una libreria statica prodotta nel formato ELF e incorporata in un programma dovrebbe essere leggermente superiore (di un 1-5%) rispetto a quella di una libreria condivisa o caricata dinamicamente, ma nella pratica questo raramente si verifica per via di altri fattori concomitanti.

Per creare una libreria statica, o per aggiungere ulteriori file oggetto ad una libreria statica esistente, si utilizza un comando simile al seguente:

```
ar rcs mia_libreria.a file1.o file2.o
```

Il comando di questo esempio aggiunge il file oggetto `file1.o` e `file2.o` alla libreria statica `mia_libreria.a`, creando `mia_libreria.a` nel caso in cui quest'ultima non sia già presente. Per ulteriori informazioni riguardo alla creazione di librerie statiche si veda `ar(1)`.

Una volta creata una libreria statica, la si vorrà probabilmente usare. È possibile utilizzare una libreria statica facendovi riferimento durante il processo di compilazione e link di un programma

eseguibile. Nel caso in cui, per la creazione dell'eseguibile, si stia utilizzando `gcc(1)` è possibile allora utilizzare, al fine di specificare la libreria, l'opzione -l; si faccia riferimento a `info:gcc` per ulteriori informazioni.

Nell'uso di `gcc` si ponga attenzione all'ordine dei parametri; -l è un'opzione del linker, e deve essere di conseguenza indicata DOPO il nome del file che si intende compilare. Questo aspetto differisce sensibilmente dalla normale sintassi che caratterizza le opzioni. Se si posiziona l'opzione -l prima del nome del file, il link può fallire, producendo messaggi di errore piuttosto criptici.

È inoltre possibile usare il linker `ld(1)` direttamente, utilizzandone le opzioni -l e -L, ma nella maggior parte dei casi risulta preferibile utilizzare `gcc(1)` dal momento che l'interfaccia di `ld(1)` ha maggiori probabilità di subire modifiche.

Librerie condivise

Le librerie condivise sono librerie che vengono caricate all'avvio dei programmi. Una volta che una libreria condivisa è stata correttamente installata, tutti i programmi successivamente eseguiti ne faranno automaticamente uso. Il funzionamento è in realtà molto più flessibile e sofisticato di quanto detto, infatti l'approccio usato da Linux permette di:

- aggiornare librerie e al tempo stesso garantire il supporto di programmi che necessitano delle vecchie versioni delle stesse librerie;
- forzare l'uso di specifiche librerie o anche di specifiche funzioni di una libreria, in sostituzione di quelle rese normalmente disponibili, quando viene eseguito un particolare programma;
- fare tutto questo mentre sono in esecuzione programmi che utilizzano le librerie esistenti.

Convenzioni

Affinché le librerie condivise supportino tutte queste caratteristiche è necessario attenersi ad un certo numero di convenzioni e linee guida. Occorre a questo scopo che risulti chiara la differenza tra i nomi con cui è possibile fare riferimento ad una libreria, in particolare i suoi *"soname"* e *"nome vero"* (e in che relazione questi siano tra di loro). Deve inoltre essere chiaro dove queste debbano essere poste nel filesystem.

Nomi delle librerie condivise

Ogni libreria condivisa ha uno speciale nome chiamato "soname". Il soname è caratterizzato dal prefisso "`lib`", dal nome della libreria, dalla particella "`.so`", seguita da un punto e da un numero di versione che viene incrementato ogni qualvolta avvengano delle modifiche all'interfaccia (una eccezione particolare è rappresentata dalle librerie di più basso livello del C, il cui nome non comincia per "`lib`"). Un soname completamente qualificato include come prefisso la directory in cui è posto; in un sistema funzionante al soname completamente qualificato corrisponde semplicemente un link simbolico al "nome vero" della libreria condivisa.

Ogni libreria condivisa ha anche un "nome vero", che corrisponde al nome del file che contiene effettivamente il codice di libreria. Il nome vero aggiunge al soname un punto, un numero di versione secondario, un ulteriore punto e il numero di release. L'ultimo punto ed il numero di

release sono opzionali. Il numero di versione secondario ed il numero di release sono di supporto al controllo di configurazione, consentendo di sapere esattamente quale o quali versioni della libreria siano state installate. Si noti che questi numeri potrebbero non coincidere con quelli utilizzati per descrivere la libreria nella documentazione, anche se quando coincidono le cose certamente si semplificano.

In aggiunta a questi, esiste inoltre il nome utilizzato dal compilatore nel momento in cui fa richiesta di una particolare libreria (in seguito riferito come il "nome per il linker"), il quale coincide semplicemente con il soname privato di qualunque numero di versione.

La chiave della gestione delle librerie condivise consiste nella distinzione fra questi nomi. I programmi, nell'elencare internamente le librerie condivise di cui hanno bisogno, dovrebbero indicarne solo il soname. Al contrario, quando si crea una libreria condivisa, si crea solo la libreria stessa, con uno specifico nome di file (quindi con maggiore dettaglio sulle informazioni relative alla versione). Quando si installa una nuove versione di una libreria, la si copia in una posizione scelta fra un limitato insieme di speciali directory e quindi si esegue il programma `ldconfig(8)`. `ldconfig` esamina i file esistenti e crea i soname come link simbolici ai nomi veri e, allo stesso tempo, aggiorna il file di cache `/etc/ld.so.cache` (descritto più avanti).

`ldconfig` non predispone i nomi per il linker; questo viene tipicamente fatto durante l'installazione della libreria ed il nome per il linker viene semplicemente creato come un link simbolico al "più recente" soname o al più recente nome vero. Raccomanderei la scelta di predisporre il nome per il linker come link simbolico al soname, dal momento che nella maggior parte dei casi se viene aggiornata una libreria la si vorrà probabilmente utilizzare automaticamente quando si esegue il link dei programmi. Ho chiesto a H. J. Lu il motivo per cui `ldconfig` non configuri automaticamente i nomi per il linker. La sua spiegazione è stata sostanzialmente che si potrebbe voler eseguire del codice utilizzando la versione più aggiornata della libreria, ma si potrebbe al contrario volere che lo *sviluppo* fosse collegato ad una versione più vecchia (ed eventualmente non compatibile). Quindi, `ldconfig` non fa assunzioni a proposito di cosa si voglia utilizzare in fase di link dei programmi e, di conseguenza, chi installa una libreria deve specificamente modificare i link simbolici per aggiornare la versione della libreria utilizzata dal linker.

Così, `/usr/lib/libreadline.so.3` è un soname completamente qualificato, che `ldconfig` predisporrebbe come link simbolico ad un qualche nome vero come `/usr/lib/libreadline.so.3.0`. Dovrebbe inoltre essere presente un nome per il linker, `/usr/lib/libreadline.so` che potrebbe essere un link simbolico che fa riferimento a `/usr/lib/libreadline.so.3`.

Posizionamento nel filesystem

Le librerie condivise devono essere poste in qualche locazione nel filesystem. La maggior parte del software open source tende a seguire gli standard GNU; per maggiori informazioni si faccia riferimento alla documentazione disponibile presso info:standards#Directory_Variables. Gli standard GNU raccomandano, per la distribuzione di software accompagnato dai sorgenti, di utilizzare come locazione predefinita delle librerie `/usr/local/lib` (mentre tutti i comandi dovrebbero andare in `/usr/local/bin`). Essi stabiliscono inoltre le convenzioni per la ridefinizione di queste locazioni e per l'attivazione delle procedure di installazione.

Il Filesystem Hierarchy Standard (FHS) discute cosa dovrebbe andare a far parte di una distribuzione e dove (vedasi http://www.pathname.com/fhs). Secondo l'FHS, la maggior parte delle librerie dovrebbero essere installate in `/usr/lib`, tranne le librerie necessarie all'avvio che dovrebbero essere in `/lib`; infine, le librerie che non sono parte del sistema dovrebbero essere in `/usr/local/lib`.

Non esiste un reale conflitto fra questi due documenti; gli standard GNU raccomandano un comportamento predefinito per gli sviluppatori di codice sorgente, mentre l'FHS raccomanda il comportamento per chi distribuisce i programmi (che in maniera selettiva ridefinisce il comportamento prestabilito nel codice sorgente, di solito per mezzo del sistema di gestione dei pacchetti della distribuzione). Nella pratica tutto questo funziona bene: il codice sorgente "più aggiornato" (ed eventualmente bacato!) che si è scaricato dalla rete si installa automaticamente nella directory "locale" (`/usr/local`), e, una volta che il codice ha raggiunto uno stadio maturo, i gestori dei pacchetti possono banalmente ridefinire il comportamento predefinito per posizionare il codice in una locazione standard per la distribuzione. Si noti che se una libreria invoca programmi che possono essere richiamati unicamente da librerie, tali programmi dovrebbero essere posti in `/usr/local/libexec` (che diventa `/usr/libexec` in una distribuzione). Una complicazione è rappresentata dal fatto che i sistemi derivati da distribuzioni Red Hat non includono `/usr/local/lib` nel percorso predefinito per la ricerca delle librerie; per ulteriori informazioni si veda anche la discussione che segue a proposito di `/etc/ld.so.conf`. L'insieme delle directory comunemente utilizzate include `/usr/X11R6/lib` per le librerie del sistema X-windows. Si noti che `/lib/security` viene utilizzato per i moduli PAM (Pluggable Authentication Modules), ma questi sono di solito gestiti come librerie a caricamento dinamico (anche queste discusse più avanti).

Come le librerie vengono utilizzate

Nei sistemi basati sulle GNU `glibc`, inclusi quindi tutti i sistemi Linux, l'avvio di un eseguibile binario in formato ELF attiva l'esecuzione del caricatore di programma. Nei sistemi Linux, questo caricatore ha nome `/lib/ld-linux.so.X` (dove X è il numero di versione). Tale caricatore, a sua volta, localizza e carica in memoria tutte le librerie condivise utilizzate dal programma.

La lista delle directory su cui effettuare la ricerca è contenuta nel file `/etc/ld.so.conf`. Molte distribuzioni derivate da Red Hat non includono normalmente `/usr/local/lib` nel file `/etc/ld.so.conf`. Personalmente lo considero un baco e aggiungere `/usr/local/lib` in `/etc/ld.so.conf` rappresenta un tipico "rimedio" necessario per eseguire molti programmi su sistemi derivati da Red Hat.

Se si vuole forzare l'utilizzo di poche specifiche funzioni in alternativa a quelle normalmente rese disponibili da una libreria, ma mantenere valido il resto della libreria stessa, si possono inserire i nomi di queste librerie sostitutive (file `.o`) in `/etc/ld.so.preload`; queste librerie di "preloading" avranno la precedenza su quelle standard. Questo file di preloading viene tipicamente utilizzato per le correzioni di emergenza alla configurazione del sistema; una distribuzione di solito non includerà un simile file quando viene rilasciata.

La ricerca attraverso tutte queste directory all'avvio del programma risulterebbe gravemente inefficiente, di conseguenza in realtà si utilizza un meccanismo di cache. Il normale comportamento del programma `ldconfig(8)` consiste nel leggere il file `/etc/ld.so.conf`, configurare gli appropriati link simbolici nelle directory (così che questi seguiranno le convenzioni standard) e infine scrivere una cache nel file `/etc/ld.so.cache` che viene quindi utilizzato dagli altri

programmi. Questo velocizza enormemente l'accesso alle librerie. La conseguenza è che `ldconfig` deve essere eseguito ogni volta che una DLL viene aggiunta, quando una DLL viene rimossa o quando cambia l'insieme delle directory in cui effettuare la ricerca delle librerie; spesso quando viene installata una libreria uno dei compiti effettuati dai gestori di pacchetti consiste nell'esecuzione di ldconfig. All'avvio di un programma, quindi, il caricatore dinamico in realtà utilizza il file `/etc/ld.so.cache` e carica quindi le librerie di cui necessita.

Ad ogni modo, FreeBSD utilizza nomi di file leggermente diversi per questa cache. Sotto FreeBSD, la cache per il formato ELF è `/var/run/ld-elf.so.hints` e la cache per il formato a.out è `/var/run/ld.so.hints`. Questi file sono comunque aggiornati da `ldconfig(8)`, di conseguenza questa differenza di collocazione nel filesystem dovrebbe assumere una qualche importanza solo in rare, "esotiche", situazioni.

Variabili di ambiente

Diverse variabili d'ambiente permettono di controllare il processo di gestione delle librerie condivise ed esistono variabili d'ambiente che consentono di modificarne il funzionamento predefinito.

LD_LIBRARY_PATH

È possibile utilizzare, per una specifica esecuzione di un programma, una libreria differente. Sotto Linux, la variabile d'ambiente LD_LIBRARY_PATH costituisce una sequenza di directory, separate da doppi punti, dove le librerie dovrebbero essere inizialmente cercate, prima che venga cioè preso in esame l'insieme delle directory di sistema; questo risulta utile quando si sta sottoponendo a dubug una nuova libreria o quando si voglia utilizzare una libreria non standard per uno scopo particolare. La variabile d'ambiente LD_PRELOAD elenca le librerie condivise con funzioni che si sostituiscono a quelle predefinite, allo stesso modo di quanto avviene per `/etc/ld.so.preload`. L'utilizzo di queste variabili è implementato nel caricamento delle librerie da `/lib/ld-linux.so`. Si deve inoltre notare che, per quanto LD_LIBRARY_PATH funzioni per molte delle varianti di Unix, non funziona per tutte; per esempio, questa funzionalità è disponibile sotto HP-UX ma come variabile d'ambiente SHLIB_PATH, mentre sotto AIX la variabile è LIBPATH (con la medesima sintassi, una lista separata da doppi punti).

LD_LIBRARY_PATH risulta comoda per lo sviluppo e le operazioni di test, ma non dovrebbe venire modificata nel corso di una procedura di installazione al fine di essere utilizzata dai comuni utenti; si veda "Why LD_LIBRARY_PATH is Bad" al link http://www.visi.com/~barr/ldpath.html per una illustrazione dei motivi. Ciononostante, oltre ad essere utile per lo sviluppo e le operazioni di test, l'uso di questa variabile permette talvolta di aggirare problemi che non potrebbero essere risolti diversamente. Se non si desidera intervenire sulla variabile d'ambiente LD_LIBRARY_PATH, sotto Linux si può eventualmente invocare direttamente il caricatore di programma passandogli degli argomenti. Per esempio, il seguente comando utilizza il PERCORSO fornito in sostituzione al contenuto della variabile LD_LIBRARY_PATH ed avvia l'ESEGUIBILE indicato:

```
/lib/ld-linux.so.2 --library-path PERCORSO ESEGUIBILE
```

L'esecuzione di ld-linux.so senza argomenti fornisce ulteriori informazioni sul suo utilizzo, ma, ancora una volta, non è consigliabile ricorrere a questo metodo se non per operazioni di debug.

LD_DEBUG

Un'altra variabile d'ambiente utilizzata dal caricatore C di GNU è LD_DEBUG. Questa variabile attiva le funzioni dl* così che forniscano un'informazione piuttosto dettagliata sulle operazioni che vengono eseguite. Per esempio:

```
export LD_DEBUG=files
programma_da_eseguire
```

visualizza l'elaborazione di file e librerie indicando quali dipendenze vengono individuate e quali oggetti condivisi vengono caricati ed in che ordine. Impostando LD_DEBUG come "bindings" visualizza informazioni sul collegamento dei simboli, impostandolo come "libs" visualizza i percorsi dove le librerie vengono ricercate e impostandolo come "versions" indica le dipendenze fra le versioni.

Impostare LD_DEBUG come "help" e provare poi ad eseguire un qualche programma fa sì che vengano elencate le opzioni ammesse. Ancora una volta, l'uso di LD_DEBUG non fa parte delle normali operazioni, ma può risultare comodo nel debug.

Altre variabili di ambiente

Esiste in realtà un certo numero di ulteriori variabili d'ambiente che controllano il processo di caricamento; i nomi di tali variabili cominciano con i prefissi LD_ o RTLD_. La maggior parte di queste si utilizzano nel debug di basso livello del processo di caricamento o per l'implementazione di particolari comportamenti. Queste variabili sono per lo più scarsamente documentate; se si ha necessità di conoscerne le caratteristiche, il modo migliore di imparare qualcosa è leggere il codice sorgente del caricatore (che fa parte della distribuzione del compilatore `gcc`).

Permettere il controllo a livello utente sul caricamento di librerie a collegamento dinamico sarebbe disastroso per programmi con setuid/setgid se non venissero prese adeguate precauzioni. Di conseguenza, nel funzionamento del caricatore GNU (che carica il resto del programma all'avvio dello stesso), se il programma è setuid o setgid queste variabili (e altre variabili simili) vengono ignorate o fortemente limitate nei loro effetti. Il caricatore determina se un programma è setuid o setgid controllandone gli attributi; se l'uid e l'euid differiscono, o se il gid e l'egid differiscono, il caricatore presume che si stia trattando di un programma con setuid/setgid (o discendente di uno che lo sia) e quindi limita fortemente le possibilità di controllarne il collegamento. Leggendo il codice sorgente della libreria GNU `glibc` è possibile verificarlo; in particolare si vedano ad esempio i file `elf/rtld.c` e `sysdeps/generic/dl-sysdep.c`. Questo significa che facendo coincidere uid e gid con l'euid e l'egid e quindi chiamando un programma, queste variabili avranno un effetto completo. Altri sistemi Unix gestiscono questa situazione in modo differente, ma per la stessa ragione: un programma con setuid/setgid non dovrebbe essere indebitamente influenzato dalla configurazione delle variabili d'ambiente.

Creare una libreria condivisa

Creare una libreria condivisa è facile. Innanzitutto, si devono creare i file oggetto che andranno a far parte della libreria condivisa utilizzando le opzioni -fPIC o -fpic di `gcc`. Le opzioni -fPIC e -fpic abilitano la generazione di codice non dipendente dalla posizione (*"position independent code"*), un requisito per le librerie condivise; si veda oltre per le differenze fra le due opzioni. Il soname viene

passato attraverso l'opzione -Wl di gcc. L'opzione -Wl inotra opzioni al linker (in questo caso -soname è quindi un'opzione per il linker); le virgole dopo -Wl non sono un errore di stampa e non si dovrebbero mai includere spazi (a meno di indicarli tramite una sequenza di escape) nel corpo di questa opzione. Si crea quindi una libreria condivisa utilizzando questo formato:

```
gcc -shared -Wl,-soname,mio_soname \
    -o nome_della_libreria elenco_dei_files elenco_delle_librerie
```

Ecco un esempio in cui si creano due file oggetto (`a.o` e `b.o`) e successivamente si crea una libreria condivisa che li contiene entrambi. Si noti che questa modalità di compilazione comprende le informazioni di debug (-g) e genererà eventuali warning (-Wall); tale modalità non rappresenta un requisito nella creazione di una libreria condivisa, ma è una pratica consigliata. La compilazione genera i file oggetto (utilizzando -c), ed include la necessaria opzione -fPIC:

```
gcc -fPIC -g -c -Wall a.c
gcc -fPIC -g -c -Wall b.c
gcc -shared -Wl,-soname,libmialibreria.so.1 \
    -o libmialibreria.so.1.0.1 a.o b.o -lc
```

Ci sono alcuni punti degni di nota:

- Non si sottoponga a strip la libreria risultante, e non si utilizzi l'opzione di compilazione -fomit-frame-pointer a meno che non sia proprio inevitabile. La libreria risultante funzionerà, ma queste operazioni rendono i debugger sostanzialmente inutili.
- Si usino -fPIC o -fpic nella generazione del codice. La scelta fra -fPIC e -fpic nella generazione del codice è una questione legata all'architettura della piattaforma per cui si sviluppa. Scegliere -fPIC funziona sempre, ma può produrre codice di maggiori dimensioni rispetto a -fpic (un metodo mnemonico per ricordarlo è che PIC è scritto con caratteri più grandi e quindi può produrre codice più grande). Utilizzare l'opzione -fpic generalmente produce codice di dimensioni inferiori e più veloce, ma con limitazioni dipendenti dalla piattaforma, quali il numero di simboli globalmente visibili o la dimensione stessa del codice. Il linker comunicherà se il progetto rientra in queste limitazioni all'atto di creare la libreria condivisa. Nel dubbio, io scelgo -fPIC, che funziona sempre.
- In alcuni casi, la chiamata a `gcc` per creare i file oggetto richiede anche di includere l'opzione "-Wl,-export-dynamic". Normalmente, la tabella dinamica dei simboli contiene solo i simboli utilizzati da oggetti dinamici. Questa opzione (nel momento in cui si crea un file in formato ELF) aggiunge tutti i simboli alla tabella dinamica dei simboli (si veda `ld(1)` per ulteriori informazioni). È necessario utilizzare questa opzione quando esistono "dipendenze inverse", vale a dire, quando una libreria a collegamento dinamico contiene dei simboli non risolti che per convenzione devono essere definiti nei programmi che intendono caricare queste librerie. Affinché le "dipendenze inverse" funzionino, il programma principale deve rendere i propri simboli disponibili dinamicamente. Si noti che, nel caso in cui si stia lavorando esclusivamente con sistemi Linux, si potrebbe usare "-rdynamic" in alternativa a "-Wl,export-dynamic", ma in base alla documentazione del formato ELF non è sempre garantito il funzionamento dell'opzione "-rdynamic" di `gcc` su sistemi non Linux.

Durante lo sviluppo, esiste il potenziale problema di modificare una libreria che è utilizzata anche da molti altri programmi -- e che non si voglia che altri programmi utilizzino la libreria "di sviluppo", tranne solamente un particolare programma tramite il quale si effettuano procedure di test. Un'opzione di link che si potrebbe usare è l'opzione "rpath" di ld, che specifica il percorso di

ricerca delle librerie a tempo di esecuzione per il particolare programma che si sta compilando. Da gcc, è possibile definire tale opzione specificandola nel modo seguente:

```
-Wl,-rpath,$(DEFAULT_LIB_INSTALL_PATH)
```

Se si utilizza questa opzione nel creare il programma che utilizza la libreria non è necessario preoccuparsi di LD_LIBRARY_PATH (si veda anche oltre) a parte verificare che non crei conflitti, o utilizzare altre tecniche per nascondere la versione di sviluppo della libreria al resto del sistema.

Installare ed utilizzare una libreria condivisa

Una volta creata una libreria condivisa, la si vorrà installare. L'approccio semplice consiste nel copiare la libreria in una delle directory standard (ad esempio, /usr/lib) ed eseguire ldconfig(8).

Innanzitutto, sarà necessario aver creato da qualche parte la libreria condivisa. Successivamente si dovranno creare i necessari link simbolici, in particolare un link dal soname al nome vero (come anche da un soname privo di versione, vale a dire, un soname che termina in ".so" per gli utenti che non specificano alcun numero di versione). L'approccio più semplice consiste nell'eseguire:

```
ldconfig -n directory_con_librerie_condivise
```

Infine, nel compilare i programmi, si dovrà informare il linker di tutte le librerie condivise e statiche che si vogliono utilizzare. Si usino a questo scopo le opzioni -l e -L.

Se non si può o non si vuole installare la libreria in una locazione standard (ad esempio se non si dispone dei privilegi per modificare /usr/lib), sarà necessario cambiare approccio. In questo caso, la si dovrà installare da qualche parte e quindi fornire il programma di informazioni sufficienti così che il programma possa localizzare la libreria... ed esistono molti modi per farlo. Nei casi semplici si può utilizzare il flag -L di gcc. Si può utilizzare l'approccio basato su "rpath" (descritto precedentemente), in particolare quando solo uno specifico programma utilizza la libreria che si sta installando in una locazione "non standard". Si può anche regolare il funzionamento dei programmi tramite le variabili d'ambiente. In particolare, si può assegnare opportunamente LD_LIBRARY_PATH, che è una lista di directory separata da doppi punti (:) in cui avviene la ricerca delle librerie condivise prima che vengano prese in considerazioni le usuali directory di installazione. Si si sta utilizzando una shell bash è possibile invocare mio_programma nel modo seguente:

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH  mio_programma
```

Se si vuole utilizzare una libreria sostituendone solo alcune funzioni, è possibile farlo creando un file oggetto e assegnando LD_PRELOAD; le funzioni in questo file oggetto si sostituiranno a quelle già presenti nella libreria (lasciando le altre invariate).

Solitamente è possibile aggiornare le librerie senza troppe preoccupazioni; se ci sono state variazioni a livello di API, si suppone che il creatore della libreria ne abbia cambiato il soname. In questo modo, differenti versioni di una singola libreria possono coesistere in uno stesso sistema e quella corretta viene selezionata per ogni programma. Comunque, se un programma smette di funzionare in seguito all'aggiornamento di una libreria che ha mantenuto lo stesso soname, è

possibile forzarlo ad utilizzare la vecchia versione di libreria facendo una copia della vecchia libreria da qualche parte, rinominando il programma (ad esempio con il vecchio nome seguito da ".orig"), e quindi sostituendolo con un breve script ("wrapper") che riassegna la libreria da utilizzare prima di chiamare il vero programma (precedentemente rinominato). Si può porre la vecchia libreria in una particolare locazione, se preferibile, anche se le convenzioni sulla numerazione permettono, in generale, la coesistenza di versioni differenti in una medesima directory. Lo script potrebbe avere un aspetto simile al seguente:

```
#!/bin/sh
export LD_LIBRARY_PATH=/usr/local/mia_lib:$LD_LIBRARY_PATH
exec /usr/bin/mio_programma.orig $*
```

È comunque raccomandabile non fare affidamento su questa possibilità quando si scrive il proprio codice; si cerchi piuttosto di accertarsi che le proprie librerie siano retrocompatibili o che si sia incrementato il numero di versione nel soname ogni volta che sia stata inserita una incompatibilità. Questo è solo un approccio di "emergenza" adatto ad affrontare problemi che si verificano nel peggiore dei casi.

È possibile visualizzare l'elenco delle librerie condivise utilizzate da un programma usando `ldd(1)`. Ad esempio, si possono elencare le librerie condivise usate da ls digitando il comando:

```
ldd /bin/ls
```

Generalmente verrà mostrato un elenco dei soname da cui il programma dipende assieme alle directory dove questi nomi vengono risolti. Nella quasi totalità dei casi si osserveranno almeno due dipendenze:

- `/lib/ld-linux.so.N` (dove N è 1 o un valore superiore, in genere almeno 2). Questa è la libreria che carica tutte le altre.
- `libc.so.N` (dove N è 6 o più). Questa è la libreria del C. Anche altri linguaggi tendono ad utilizzare la libreria del C (se non altro per implementare le proprie librerie), quindi la maggior parte dei programmi la include.

Attenzione: *non* si esegua `ldd` su un programma di cui non ci si fida. Come chiaramente affermato nel manuale di `ldd(1)`, `ldd` funziona (in alcuni casi) assegnando una particolare variabile d'ambiente (per oggetti in formato ELF si tratta di LD_TRACE_LOADED_OBJECTS) e successivamente eseguendo il programma. Può risultare possibile per un programma forzare l'utente di `ldd` ad eseguire un arbitrario segmento di codice (invece che semplicemente mostrare le informazioni che `ldd` produce). Quindi, per ragioni di sicurezza, non si usi `ldd` su programmi che non ci si fiderebbe ad eseguire.

Librerie incompatibili

Quando una nuova versione di una libreria diventa incompatibile a livello binario con la precedente, il soname deve cambiare. In C esistono quattro principali motivi per cui una libreria cessa di essere compatibile a livello binario:

1. il comportamento di una funzione cambia così da non corrispondere più alle specifiche originali,

2. ci sono variazioni nelle strutture dati esportate (un'eccezione: aggiungere attributi opzionali in fondo a strutture può essere accettabile a condizione che tali strutture vengano allocate unicamente all'interno della libreria stessa),
3. viene rimossa una funzione precedentemente esportata,
4. l'interfaccia di una funzione esportata viene modificata.

Se si possono evitare questi motivi risulta allora possibile mantenere la compatibilità binaria delle librerie. Detto in altri termini, è possibile mantenere compatibile l'interfaccia binaria verso le applicazioni (ABI - Application Binary Interface) se si evitano simili modifiche. Per esempio, si potrebbe voler aggiungere delle nuove funzioni, ma non eliminare quelle vecchie. Si possono aggiungere elementi alle strutture, ma solo accertandosi che i vecchi programmi non saranno sensibili al cambiamento aggiungendoli solo in fondo alle strutture preesistenti, permettendo solo alla libreria (e non alle applicazioni) l'allocazione di tali strutture, rendendo opzionale l'uso dei termini aggiunti (o facendo in modo che sia la libreria ad assegnarli opportunamente) e così via. Attenzione: probabilmente non è possibile espandere delle strutture se gli utenti le stanno utilizzando negli array.

Per il C++ (e altri linguaggi che supportano la compilazione di codice in forma di template e/o meccanismi di risoluzione delle chiamate di metodi determinati in fase compilazione) la situazione è più complessa. Risultano validi tutti gli argomenti già citati ai quali se ne aggiungono numerosi altri. La ragione risiede nel fatto che alcune informazioni vengono inserite nel codice compilato in maniera non direttamente visibile allo sviluppatore, risultando in dipendenze che possono non essere ovvie se non si ha presente come il C++ viene tipicamente implementato. Di fatto, non si tratta di problematiche "nuove", è solo che il codice C++ compilato può farle emergere in modi che possono risultare inaspettati. Quella che segue è una lista (probabilmente incompleta) di cose che non si possono fare in C++ mantenendo la compatibilità binaria,:

1. aggiungere reimplementazioni di funzioni virtuali (a meno che non sia possibile per le applicazioni esistenti continuare a chiamare l'implementazione originale), dato che ClasseBase::funzioneVirtuale() viene valutata in fase di compilazione (e non in fase di link).
2. aggiungere o rimuovere funzioni membro virtuali, dato che questo modificherebbe la dimensione e la struttura della vtbl di ogni sottoclasse.
3. modificare il tipo di un qualunque dato membro o spostare un qualunque dato membro a cui si ha accesso tramite funzioni membro dichiarate inline.
4. modificare l'albero di una gerarchia di classi, eccetto per aggiungere nuove foglie.
5. aggiungere o rimuovere dati membro privati, dato che questo modificherebbe dimensione e struttura di ogni sottoclasse.
6. rimuovere funzioni membro pubbliche o protette a meno che non siano dichiarate inline.
7. rendere inline una funzione membro pubblica o protetta.
8. modificare il comportamento di una funzione inline, a meno che la vecchia versione non continui a funzionare.
9. modificare i privilegi di accesso (vale a dire pubblico, protetto o privato) di una funzione membro in un programma che intenda mantenere una certa portabilità in quanto alcuni compilatori inseriscono i privilegi di accesso nella decorazione del nome di funzione.

Data la lunga lista, gli sviluppatori di librerie in C++ dovranno pianificare lo sviluppo con particolare attenzione se vorranno minimizzare gli aggiornamenti che ne possano compromettere la compatibilità a livello binario. Fortunatamente, nei sistemi di tipo Unix (Linux incluso) si possono

caricare ed utilizzare contemporaneamente differenti versioni di una stessa libreria, così che, anche se con qualche penalizzazione in termini di occupazione dello spazio disco, gli utenti possono continuare ad eseguire "vecchi" programmi che richiedono le vecchie librerie.

Librerie caricate dinamicamente

Le librerie caricate dinamicamente sono librerie che vengono caricate in memoria in momenti successivi all'avvio del programma. Risultano particolarmente utili nell'implementazione di *"plugins"* o moduli, dal momento che permettono di attendere, per il caricamento degli stessi, il momento in cui risultino necessari all'applicazione. Ad esempio, il sistema di autenticazione PAM (Pluggable Authentication Modules) usa librerie a caricamento dinamico per permettere agli amministratori di configurarne e riconfigurarne il funzionamento. Risultano inoltre utili nell'implementazione di interpreti che vogliano occasionalmente compilare il codice in esecuzione e utilizzarne la versione compilata per motivi di efficienza, il tutto senza fermarsi. Per esempio, questo approccio può essere utile nell'implementare un compilatore JIT (just-in-time) o un gioco multi-utente (MUD, multi-user dungeon).

Sotto Linux, le librerie a caricamento dinamico non sono in realtà nulla di particolare dal punto di vista del formato; consistono in comuni file oggetto o comuni librerie condivise, come discusso in precedenza. La principale differenza consiste nel fatto che non vengono automaticamente caricate al momento del collegamento o all'avvio di un programma; esiste invece un'API per aprire una libreria, ricercarvi simboli, gestire errori e chiudere la libreria. Per accedere a questa interfaccia gli utilizzatori del linguaggio C dovranno includere il file `<dlfcn.h>`.

L'interfaccia utilizzata da Linux è essenzialmente la stessa usata sotto Solaris, che chiamerò API "dlopen()". D'altro canto, non tutte le piattaforme supportano questa medesima interfaccia. HP-UX utilizza un meccanismo differente, basato su shl_load(), e le piattaforme Windows usano le DLL, con un'interfaccia completamente differente. Se un'ampia portabilità dovesse far parte dei requisiti, si dovrebbe probabilmente prendere in considerazione l'utilizzo di qualche libreria che, attraverso un'ulteriore livello di astrazione, mascheri le differenze fra le varie piattaforme. Una possibile soluzione è rappresentata dalla libreria `glib`, con il suo supporto al caricamento dinamico di moduli; utilizza le procedure per il caricamento dinamico caratteristiche della piattaforma sottostante per implementare un'interfaccia portabile a queste funzioni. Ulteriori informazioni su glib sono disponibili presso http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html. Dal momento che l'interfaccia di `glib` è bene illustrata dalla sua documentazione non la discuterò ulteriormente in questa sede. Un altro approccio consiste nell'utilizzare `libltdl`, parte di GNU libtool. Se fossero richieste ulteriori funzionalità, si potrebbe allora voler prendere in considerazione l'uso di un Object Request Broker (ORB), caratteristico di CORBA. Se invece si è ancora interessati ad utilizzare direttamente l'interfaccia supportata da Linux e Solaris, si può continuare a leggere.

Gli sviluppatori che utilizzano il C++ e librerie a caricamento dinamico dovrebbero consultare inoltre il "C++ dlopen mini-HOWTO".

dlopen()

La funzione dlopen(3) apre una libreria e la inizializza all'uso. Il prototipo in C di tale funzione è:

```
void * dlopen(const char *nome_del_file, int flag);
```

Se il nome del file inizia con "/" (si tratta cioè di un percorso assoluto), dlopen() proverà ad utilizzarlo direttamente (non verrà quindi effettuata nessuna ricerca per localizzare la libreria). Altrimenti, dlopen() cercherà la libreria con il seguente ordine:

1. in una lista di directory separata da doppi punti nella variabile d'ambiente LD_LIBRARY_PATH.
2. nella lista di librerie specificata in `/etc/ld.so.cache` (che è generata da `/etc/ld.so.conf`).
3. in `/lib`, seguita da `/usr/lib`. Si noti che l'ordine in questo caso specifico è l'inverso di quello utilizzato dal vecchio caricatore per il formato a.out. Nel caricare un programma, il caricatore a.out cercava infatti prima in `/usr/lib` e, successivamente, in `/lib` (si veda la pagina man di `ld.so(8)`). Questo normalmente non dovrebbe fare differenza, dal momento che una stessa libreria dovrebbe essere solo in una o nell'altra directory e che librerie diverse, ma con lo stesso nome sono un disastro che attende solo di verificarsi.

Nella chiamata a dlopen(), il valore di *`flag`* deve essere o RTLD_LAZY, che significa "risolvi i simboli non definiti nel momento in cui del codice facente parte della libreria dinamica viene eseguito", o RTLD_NOW, che significa "risolvi tutti i simboli non definiti prima che dlopen() ritorni e fallisci se questo non fosse possibile". RTLD_GLOBAL può essere opzionalmente combinato all'uno o all'altro valore di *`flag`* (tramite un operazione di OR) stando così ad indicare che i simboli con collegamento esterno definiti nella libreria verranno resi disponibili alle librerie caricate successivamente. Durante il debug è in genere preferibile usare RTLD_NOW; usare RTLD_LAZY può creare errori non immediatamente visibili nel caso in cui esistano riferimenti non risolti. Usare RTLD_NOW rende l'apertura di una libreria leggermente più lenta (ma in seguito la ricerca dei simboli risulta più rapida); se questo dovesse causare problemi a livello di interfaccia utente è comunque possibile passare ad utilizzare RTLD_LAZY in un successivo momento.

Se una libreria dipende da un'altra (ad esempio, X dipende da Y), è necessario aprire prima quella dipendente (nell'esempio, prima Y e poi X).

Il valore restituito da dlopen() è un descrittore (un "handle") che dovrebbe essere considerato come un riferimento da utilizzarsi nelle successive chiamate alle altre funzioni di libreria per il caricamento dinamico. dlopen() restituisce NULL se il tentativo di caricamento non dovesse avere successo, e questa condizione andrebbe verificata. Se una stessa libreria viene caricata più di una volta con dlopen(), viene restituito lo stesso descrittore.

Sulle vecchie piattaforme, nel caso in cui una libreria esporti una procedura chiamata _init, tale funzione viene eseguita prima che dlopen() ritorni. Si può utilizzare questa caratteristica nelle proprie librerie per implementare delle procedure di inizializzazione. Ad ogni modo, una libreria non dovrebbe esportare delle procedure con nome _init e/o _fini. Tali meccanismi sono obsoleti e possono dare luogo a comportamenti indesiderati. Piuttosto, una libreria dovrebbe esportare procedure che utilizzano gli attributi di funzione `__attribute__((constructor))` ed `__attribute__((destructor))` (assumendo che si stia utilizzando gcc). Si veda la Sezione 5.2 per ulteriori informazioni.

dlerror()

Eventuali errori possono essere verificati attraverso una chiamata a dlerror(), la quale restituisce una stringa che descrive l'errore generato dall'ultima chiamata a dlopen(), dlsym(), o dlclose(). Una stranezza consiste nel fatto che dopo una chiamata a dlerror(), successive, ulteriori chiamate a dlerror() restituiranno NULL fino a che un ulteriore errore non si dovesse verificare.

dlsym()

Non esiste motivo di caricare dinamicamente una libreria se poi non la si può utilizzare. La funzione principale per l'uso di una libreria a caricamento dinamico è dlsym(3), che ricerca il valore di un simbolo in una data libreria (precedentemente aperta). Tale funzione è dichiarata come:

```
void * dlsym(void *handle, char *simbolo);
```

in cui "handle" è il valore restituito da dlopen e "simbolo" è una stringa terminata da zero. Se possibile, si eviti di assegnare il risultato di dlsym() ad un puntatore di tipo void*, dato che andrebbe convertito tramite un cast ad ogni utilizzo (e fornirebbe meno informazioni ad altri sviluppatori che dovessero trovarsi ad intervenire sul programma).

dlsym() restituisce NULL come risultato se il simbolo non viene trovato. Se risulta noto a priori che il simbolo non può mai assumere come valore NULL o zero, questo può bastare, ma altrimenti può esistere una potenziale ambiguità: se si ottiene NULL, significa che il simbolo non esiste o che NULL è il valore del simbolo stesso? La soluzione standard consiste nel chiamare prima dlerror() (per annullare ogni precedente condizione di errore), quindi richiedere il simbolo tramite la chiamata a dlsym() ed infine chiamare ancora dlerror() per verificare se si è verificato un errore. Un ipotetico frammento di codice assomiglierebbe al seguente:

```
dlerror(); /* annulla precedenti condizioni di errore */
s = (vero_tipo) dlsym(handle, simbolo_da_cercare);
if ((err = dlerror()) != NULL) {
 /* simbolo non trovato, gestisce l'errore */
} else {
 /* simbolo trovato, s ne contiene il valore */
}
```

dlclose()

L'inverso di dlopen() è dlclose(), che chiude una libreria a caricamento dinamico. La libreria `dl` mantiene un conteggio dei riferimenti alle librerie aperte, quindi una libreria a caricamento dinamico non viene in realtà deallocata fin tanto che dlclose non sia stata chiamata su di essa tante volte quante dlopen è stata chiamata con successo sulla stessa libreria. Quindi non è un problema per un programma caricare la stessa libreria più di una volta. Nelle librerie più vecchie, nel momento in avviene la deallocazione, viene chiamata la funzione _fini (ammesso che sia definita), ma _fini rappresenta un meccanismo obsoleto sul quale non si dovrebbe fare affidamento. Piuttosto, una libreria dovrebbe esportare procedure che utilizzano gli attributi di funzione `__attribute__((constructor))` ed `__attribute__((destructor))`. Si veda la Sezione 5.2 per ulteriori informazioni. Nota: dlclose() restituisce 0 se eseguita con successo, un valore non nullo in caso di errore; alcune pagine di manuale di Linux non fanno menzione di questo particolare.

Esempio di libreria a caricamento dinamico

Ecco un esempio dalla pagina man di dlopen(3). Questo esempio carica la libreria matematica e stampa il coseno di 2.0, controllando eventuali errori ad ogni operazione (come si raccomanda di fare sempre):

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

    int main(int argc, char **argv) {
        void *handle;
        double (*coseno)(double);
        char *errore;

        handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
        if (!handle) {
            fputs (dlerror(), stderr);
            exit(1);
        }

        coseno = dlsym(handle, "cos");
        if ((errore = dlerror()) != NULL)  {
            fputs(errore, stderr);
            exit(1);
        }

        printf ("%f\n", (*coseno)(2.0));
        dlclose(handle);
    }
```

Se questo programma fosse in un file chiamato "pippo.c", si potrebbe compilarlo con il comando:

```
    gcc -o pippo pippo.c -ldl
```

Riunire più librerie in un'unica libreria

Cosa succederebbe se si volesse prima creare delle piccole librerie e poi, in un secondo momento, riunirle in librerie di dimensioni maggiori? In un caso simile, potrebbe risultare utile l'opzione "--whole-archive" di `ld`, che consente di riunire efficacemente dei file `.a` e collegarli in un unico file `.so`.

Ecco un esempio di come utilizzare --whole-archive:

```
gcc -shared -Wl,-soname,libmialib.so.$(VER) -o libmialib.so.$(VER).0 \
$(FILE_OGGETTO) -Wl,--whole-archive $(LIBRERIE_DA_RIUNIRE) \
-Wl,--no-whole-archive $(NORMALI_LIBRERIE)
```

Come messo in evidenza dalla documentazione di `ld`, ci si assicuri di utilizzare alla fine l'opzione --no-whole-archive altrimenti `gcc` cercherà di riunire nella libreria in output anche le librerie standard.

Ulteriori esempi

Quelli che seguono sono altri esempi relativi alle tre modalità descritte (librerie statiche, condivise e a caricamento dinamico). Il file `libhello.c` implementa una semplice libreria con `libhello.h` come file di intestazione. Il file `demo.c` è un semplice file dimostrativo che contiene delle chiamate alla libreria. A questi seguono alcuni script commentati (`script_static` e `script_shared`) che illustrano l'uso della libreria come libreria statica e condivisa. Infine, `demo_dynamic.c` e `script_dynamic` mostrano come utilizzare la libreria condivisa come una libreria a caricamento dinamico.

File libhello.c

```
/* libhello.c - dimostrare l'uso di librerie. */

#include <stdio.h>

void hello(void) {
  printf("Hello, library world.\n");
}
```

File libhello.h

```
/* libhello.h - dimostrare l'uso di librerie. */


void hello(void);
```

File demo.c

```
/* demo.c -- dimostrare l'uso diretto della funzione "hello" */

#include "libhello.h"

int main(void) {
 hello();
 return 0;
}
```

File script_static

```
#!/bin/sh
# Esempio di libreria statica

# Crea il file oggetto della libreria statica, libhello-static.o.
# Uso il nome libhello-static per distinguerlo con chiarezza dagli
# esempi di librerie dinamiche, ma non è in generale necessario
# usare "-static" per i nomi di file oggetto che saranno parte
# di librerie statiche.

gcc -Wall -g -c -o libhello-static.o libhello.c

# Crea la libreria statica.

ar rcs libhello-static.a libhello-static.o

# A questo punto si potrebbe semplicemente copiare
# libhello-static.a da qualche altra parte per poi
# riutilizzarla. Per gli scopi dell'esempio ci si
# limiterà a lasciarla nella presente directory.
```

```
# Compilazione del file di programma demo.

gcc -Wall -g -c demo.c -o demo.o

# Creazione del programma demo; -L. fa sì che "." sia
# compresa nella ricerca durante la creazione del programma.
# Si noti che questo comando implica l'incorporazione del
# file libhello-static.a nel file demo_static.

gcc -g -o demo_static demo.o -L. -lhello-static

# Esecuzione del programma.

./demo_static
```

File script_shared

```
#!/bin/sh
# Esempio di libreria condivisa

# Crea il file oggetto della libreria condivisa, libhello.o.

gcc -fPIC -Wall -g -c libhello.c

# Crea la libreria condivisa.
# Si usi -lc per collegarla alla libreria del linguaggio C,
# dato che libhello dipende dalla libreria del C.

gcc -g -shared -Wl,-soname,libhello.so.0 \
    -o libhello.so.0.0 libhello.o -lc

# A questo punto potremmo semplicemente copiare libhello.so.0.0
# in qualche directory, ad esempio /usr/local/lib.

# Ora dobbiamo chiamare ldconfig per sistemare i link simbolici.

# Definizione del soname. Si potrebbe semplicemente eseguire:
# ln -sf libhello.so.0.0 libhello.so.0
# ma lasciamo che sia ldconfig a determinarlo

/sbin/ldconfig -n .

# Definizione del nome per il linker.
# In condizioni più complesse, ci si dovrebbe accertare
# dell'esistenza di un nome per il linker precedentemente
# definito ed in quel caso decidere se mantenerlo o meno.

ln -sf libhello.so.0 libhello.so

# Compilazione del file di programma demo.

gcc -Wall -g -c demo.c -o demo.o

# Creazione del programma demo.
# -L. aggiunge "." alle directory su cui effettuare la
# ricerca durante la creazione del programma; si noti che
# questo non significa che "." verrà controllata quando
# il programma viene eseguito.

gcc -g -o demo demo.o -L. -lhello
```

```
# Esecuzione del programma. Si noti che è necessario dire al
# programma dove trovare la libreria condivisa, utilizzando
# LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo
```

File demo_dynamic.c

```
/* demo_dynamic.c -- dimostrare il caricamento dinamico e
   l'uso della procedura "hello" */


/* dlfcn.h è necessario per le funzioni di caricamento
   dinamico delle librerie */
#include <dlfcn.h>

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

/* Si noti che non è necessario includere "libhello.h".
   Ad ogni modo occorre specificare alcune informazioni
   correlate; si deve specificare un tipo da associare
   al valore che si ricaverà da dlsym(). */

/* Il tipo "simple_demo_function" descrive una funzione
   che non prende alcun argomento, e non restituisce alcun
   valore: */

typedef void (*simple_demo_function)(void);


int main(void) {
 const char *errore;
 void *modulo;
 simple_demo_function demo_function;

 /* Carica dinamicamente la libreria */
 modulo = dlopen("libhello.so", RTLD_LAZY);
 if (!modulo) {
   fprintf(stderr, "Impossibile aprire libhello.so: %s\n",
           dlerror());
   exit(1);
 }

 /* Ricava il simbolo */
 dlerror();
 demo_function = dlsym(modulo, "hello");
 if ((errore = dlerror())) {
   fprintf(stderr, "Impossibile trovare hello: %s\n", errore);
   exit(1);
 }

 /* Ora chiama la funzione dalla libreria a caricamento
    dinamico */
 (*demo_function)();

 /* Tutto fatto, chiude in modo pulito */
 dlclose(modulo);
 return 0;
}
```

File script_dynamic

```
#!/bin/sh
# Dimostrazione di libreria a caricamento dinamico

# Presuppone che libhello.so e compagnia siano
# stati precedentemente creati (si vedano gli esempi
# precedenti).

# Compila il file programma demo_dynamic.c in un file
# oggetto:

gcc -Wall -g -c demo_dynamic.c

# Crea il programma demo_use.
# Si noti che non è necessario definire dove localizzare le
# librerie a caricamento dinamico dal momento l'unica libreria
# particolare utilizzata dal programma non verrà caricata se
# non dopo l'avvio.
# D'altro canto, è necessario utilizzare l'opzione -ldl per
# includere la libreria che implementa le funzioni per la
# gestione delle librerie a caricamento dinamico.

gcc -g -o demo_dynamic demo_dynamic.o -ldl


# Esecuzione del programma. Si noti che è necessario dire al
# programma dove trovare la libreria a caricamento dinamico,
# utilizzando LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_dynamic
```

Autoconf ed Automake per la portabilità del software

Nei primi due articoli abbiamo visto come si compilano programmi in ambiente Linux o comunque tramite le utility della GNU [BeGa1], e come risolvere le problematiche legate allo sviluppo di software all'interno di un team di sviluppo [BeGa2]. Con questo articolo, invece, affronteremo il problema in antitesi al primo articolo, ovvero come generare, tramite una serie di passaggi intermedi, lo script *configure* che permette al nostro sorgente di poter essere compilato, senza modificare il Makefile, anche su sistemi operativi ed architetture diverse da quelle di sviluppo. Molto spesso, anche tra due sistemi Unix, sia i file di libreria sia i file header sono posti in directory diverse e spesso anche con nomi completamente diversi; si rende quindi necessario uno strumento per facilitare il compito degli sviluppatori che hanno l'esigenza di rendere portabile il proprio software.
Per questo motivo è stato introdotto il tool *autoconf* che permette di automatizzare e gestire la compilazione e la distribuzione di qualsiasi progetto software.

Principio di funzionamento
Come abbiamo visto nel nostro primo articolo, il software OpenSource è di solito accompagnato da uno script *configure* che effettua una serie di analisi sul sistema che si sta utilizzando per generare un Makefile ad hoc per il vostro sistema. In particolare viene testata la presenza del compilatore e di altre utility (flex, bison, awk, sed) di cui si ha bisogno in fase di compilazione. Inoltre provvede a controllare l'esistenza delle librerie statiche o dinamiche e dei file di header utilizzati dal programma, segnalando eventuali anomalie. Lo script *configure* genera quindi il file Makefile sostituendo delle variabili (della forma *@nome_variabile@*) nel file *Makefile.in* (una sorta di template) con i valori che vengono autodeterminati, esaminando le caratteristiche del sistema su cui si sta eseguendo la compilazione:

```
Makefile.in --> ./configure --> Makefile
```

Lo script configure è molto complesso, viene quindi utilizzato il tool *autoconf* per generarlo automaticamente partendo da un file di configurazione. In particolare, autoconf legge le impostazioni dal file *configure.in* in cui, tramite una serie di marco, si descrive ad autoconf le operazioni che lo script *configure* dovrà compiere.

```
configure.in --> autoconf --> configure
```

Anche il nostro Makefile.in dovrà essere adattato e non potrà essere una semplice copia del Makefile originale perché vanno inserite, dove opportuno, le macro che *configure* dovrà sostituire. Nonostante il file Makefile.in possa essere scritto manualmente, il tool *automake* è, come suggerisce il nome, uno script per la generarzione in automatico del *Makefile.in* tramite un file di configurazione chiamato *Makefile.am*:

```
Makefile.am --> automake --> Makefile.in
```

E' importante notare che autoconf e automake non sono necessari per la compilazione del programma ma alla sola generazione dello script *configure*. Inoltre per utilizzare autoconf ed automake è necessaria la presenza del programma *m4* e del *Perl*. Tutti questi tool sono disponibili sui vari siti della GNU (ad esempio *ftp.gnu.org*), e comunque presenti in tutte le distribuzioni Linux.
Nel presente articolo illustreremo prima l'utilizzo di *autoconf* e poi quello di *automake*, basandoci su un semplice programma.

Prepariamo l'ambiente di lavoro
Come esempio utilizzeremo, come sempre, il classico programma "Hello World", però con l'intenzione di farne una distribuzione portabile dal punto di vista della compilazione (**Listato 1**). Come vedete viene utilizzata la funzione *strdup*, e la funzione *exit*; nonostante queste non siano necessarie ci torneranno utili per mostrare l'utilizzo di autoconf ed automake per realizzare un pacchetto portabile sui vari sistemi Unix (e non).

Concentriamoci adesso su quello che è presente in questo programma e che potrebbe portare a dei problemi di portabilità (sembra incredibile che in un programma così piccolo ci si debba preoccupare di questo problema, però vedremo che ci sono due punti che potrebbero causare problemi). Prima di tutto si deve tenere presente che l'header *stdlib.h* contiene la dichiarazione di exit, ed è stato introdotto dall'ANSI C, e quindi sui vecchi sistemi potrebbe non essere presente. Inoltre anche la funzione strdup potrebbe non essere presente su alcuni sistemi (nonostante ormai si trovi nelle librerie di quasi tutti i compilatori, non si tratta di una funzione ANSI).
A questo punto si può modificare il nostro programma aggiungendo delle condizioni per il precompilatore (**Listato 2**): cioè includendo il file stdlib.h solo se presente e dando una versione personalizzata di strdup, se questa non si trova nella libreria del compilatore. Come vedete viene incluso il file *config.h* prima di qualsiasi altra #include, perché all'interno di questo file verranno incluse o meno del definizioni del preprocessore che influiranno sulla compilazione del programma:

```
/* il file stdlib.h è presente */
#define STDC_HEADERS

/* strdup fa parte della libreria */
#define HAVE_STRDUP
```

Quindi, ad esempio, chi vorrà compilare il programma su un sistema dove non è presente la funzione strdup dovrà editare il file config.h e cambiare la direttiva #define in #undef. Ovviamente a questo punto vogliamo fornire anche un Makefile (**Listato 3**).

autoconf
Chi compilerà il programma su un differente sistema dovrà conoscere in anticipo l'esistenza dell'header stdlib.h e della funzione *strdup* ed, in base a questo, modificare manualmente il file config.h commentando le righe in cui vengono definite le suddette variabili. Inoltre potrebbe essere necessario modificare anche il valore della variabile CC che dice al Makefile qual'e' l'eseguibile da utilizzare. Ad esempio su molti sistemi Unix il compilatore C potrebbe non essere il *gcc* della GNU, infatti su Digital si chiama *cc*.
A questo punto entra in gioco autoconf: basandosi sul file configure.in creerà lo script di (auto)configurazione che, a sua volta creerà il *Makefile*.
Abbiamo quindi l'esigenza di creare i due file di configurazione principali: *Makefile.in* e *configure.in*. Per creare il Makefile.in possiamo iniziare basandoci su un normale Makefile al cui interno sostituiremo il valore di alcune variabili con delle macro il cui nome sarà racchiuso fra @, come ad esempio @CC@. Quindi avremo, all'interno del Makefile.in:

```
CC = @CC@
```

in questo modo *configure* sostituira @CC@ con il nome del compilatore trovato sul nostro sistema, in genere *gcc* o *cc*.

Resta da scrivere configure.in.

Per la scrittura di questo file ci aiuteremo dal tool *autoscan*. Questo programma esaminerà i file all'interno del nostro progetto per cercare di individuare elementi che possono generare problemi di portabilità e, per questi, generare automaticamente i comandi necessari da inserire in *configure.in*. Lanciando *autoscan* verrà generato il file *configure.scan* che sarà necessario rinominare come *configure.in* per apportargli le necessarie modifiche. Lanciamo quindi questo programma sulla directory in cui si trovano i nostri file, ed otterremo il file configure.scan riportato nel **Listato 4**.
Il file creato è ben commentato (i commenti iniziano con la macro *dnl*), ed i nomi sono intuitivi. E' da notare che ogni configure.in deve obbligatoriamente iniziare con *AC_INIT(nomefile)*, dove nomefile è il nome di un file che è presente nel nostro pacchetto (autoscan ha scelto, giustamente, il nome del sorgente principale), e deve terminare con *AC_OUTPUT(nome makefile)*, che indica il nome del makefile che configure dovrà creare. *autoscan* si è accorto che useremo il compilatore C e quindi configure lo cercherà (*AC_PROG_CC*), e

salverà il nome nella variabile *@CC@*. Inoltre controlla che ci siano gli header standard ANSI (*AC_HEADER_STDC*), e che sia presente nella libreria di sistema la funzione strdup (*AC_CHECK_FUNCS(strdup)*). Fino a questo punto, eravamo stati abbastanza previdenti anche noi, però ci eravamo scordati della keyword *const* che potrebbe non essere riconosciuta da alcuni compilatori (*AC_C_CONST*). Notate che autoscan è molto utile per aiutarci a scovare quei punti problematici per la portabilità del software.
A questo punto possiamo copiare il file generato da autoscan direttamente su configure.in e lanciare autoconf, che provvederà a generare il famoso script *configure*. Per vedere configure al lavoro basterà lanciarlo; noteremo una serie di scritte (familiari a chi ha già compilato un pacchetto GNU), ed alla fine sarà creato il Makefile. Come si potrà controllare, all'interno del Makefile la macro @CC@ è stata sostituita con *gcc*.
Abbiamo visto quindi come sfruttare la variabile @CC@ per conoscere il nome del compilatore presente sul sistema in cui verrà effettuata la compilazione; configure terminerà con un errore nel caso non sia presente il compilatore gcc (essenziale).
Vediamo come sfruttare altri tipi di informazioni, ad esempio la presenza o meno di strdup, della keyword *const*, e degli header standard. Se gli header standard sono presenti configure definirà la variabile STDC_HEADERS, e se è presente strdup, la variabile HAVE_STRDUP (si guardi la guida in linea, in formato info, per le varie macro). Sono gli stessi nomi che avevamo scelto noi (volontariamente, per non dovere riscrivere i file). Diremo a configure di ridirezionare tali informazioni in un file header di configurazione *config.h*, aggiungendo la macro

```
AC_CONFIG_HEADER(config.h)
```

all'interno di configure.in. Tale macro permetterà appunto di creare automaticamente il file *config.h* contenente le direttive del preprocessore che indicano o meno la presenza dei vari elementi cercati con le altre macro.
Tuttavia configure per creare config.h deve avere un template, chiamato *config.h.in*; non entriamo nella sintassi che deve seguire questo file, anche perché per generarlo possiamo utilizzare un'altra utility: *autoheader*, che basandosi sul contenuto di configure.in genererà automaticamente il config.h.in; quindi lanciamo il comando autoheader, e dopo potremo lanciare nuovamente autoconf. Infine, lanciando nuovamente configure (possiamo anche cancellare il vecchio config.h), vedremo che oltre al Makefile verrà creato anche config.h. Se la keyword const non venisse gestita dal compilatore configure provvederebbe a definire const come la stringa vuota. Volendo si potrebbe dare un'occhiata al file config.h per vedere il lavoro dei tool di autoconf.
A questo punto il nostro programma può essere compilato (teoricamente senza problemi) su diversi sistemi Unix (e Windows se si ha il compilatore GNU-WIN32 [Cygnus]), semplicemente lanciando

```
configure
make
```

Ci si deve ricordare di distribuire configure e Makefile.in config.h.in, e non Makefile e config.h (questi saranno generati dinamicamente).

Un passo avanti: automake
Arrivati a questo punto però vorremo fare un ulteriore passo in avanti; effettivamente i vari programmi GNU, dopo make, gestiscono anche il comando *make install* per installare i file di programma nel posto giusto; vorremmo avere un modo automatizzabile per costruire la nostra distribuzione (cioè i file che devono essere distribuiti coi sorgenti, come ad esempio file su cui scriviamo le varie cose rimaste in sospeso, o file temporanei); inoltre, forse per eccessiva pigrizia, non vogliamo dover scrivere il file Makefile.in, per la gestione di tutte le operazioni suddette.
Per complicare leggermente l'esempio useremo un altro sorgente *saluti.c*, che effettuerà il printf (similmente allo scorso articolo).
Per questo esiste *automake*; basandosi sul file Makefile.am genererà il file Makefile.in in questione.

Per prima cosa si deve modificare configure.in con alcune macro di automake (che iniziano con AM): in particolare subito dopo AC_INIT, si deve inserire la macro

```
AM_INIT_AUTOMAKE(hello, 1.0)
```

che specifica il nome del pacchetto e la sua versione, sulla riga successiva al primo comando `AC_INIT`, e sostituire il comando che riguarda la creazione di config.h, cioè la macro `AC_CONFIG_HEADER,` con

```
AM_CONFIG_HEADER(config.h)
```

La scrittura di *Makefile.am* è semplicissima

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c saluti.c saluti.h
```

semplicemente diciamo come si chiama il nostro programma, e quali sono i suoi sorgenti. Prima di proseguire è necessario creare anche il file acconfig.h (che autoheader inserirà nel file config.h.in) includendo le seguenti definizioni:

```
/* nome del pacchetto */
#undef PACKAGE
/* versione */
#undef VERSION
```

acconfig.h viene utilizzato da autoheader per gestire variabili definite dal programmatore (diverse da quelle standard recuperabili da configure.in), e, per un bug, PACKAGE e VERSION di automake.
Fatto questo, si possono lanciare i seguenti comandi:

```
aclocal
autoconf
autoheader
automake --add-missing --foreign
```

Il primo serve per generare delle macro di automake che serviranno ad autoconf, è poi necessario rilanciare autoheader, ed infine automake che tra l'altro aggiungerà altri file, molto utili (evitiamo di entrare nel dettaglio delle opzioni).
Nonostante le dimensioni di Makefile.am viene creato un enorme Makefile.in (la cui lettura potrebbe non essere semplice a chi non conosce approfonditamente make), in cui noterete numerosissime macro nella forma *@nome_variabile@* ed in effetti si hanno a disposizione diverse opzioni aggiuntive. Per cominciare lanciamo il comando

```
configure --prefix=$HOME/usr/local
```

e poi possiamo compilare con

```
make
```

e stavolta possiamo anche lanciare

```
make install
```

ed il programma verrà installato nella directory *$HOME/usr/local* (in mancanza dell'opzione *--prefix*, il default è /usr/local, per la quale però è necessario avere i diritti di root). Si ha inoltre a disposizione l'opzione *make uninstall*, *make clean*, e molte altre classiche dei pacchetti GNU, ed inoltre *make dist* che creerà un file

hello-1.0.tar.gz con una distribuzione del nostro programma sotto forma di sorgenti includendo solo i file necessari e quelli specificati fra i sorgenti nel file Makefile.am. come si vede, adesso si hanno tutte le caratteristiche dei pacchetti GNU, ed inoltre la portabilità dei sorgenti.
Inoltre modificando uno qualsiasi dei file necessari ai tool di autoconf e automake, basterà rilanciare make, affinché vengano lanciati i comandi necessari all'aggiornamento della distribuzione.
Vale la pena ricordare che chi compilerà il programma dovrà avere installato solo il compilatore: non saranno necessari autoconf ed automake; infatti configure è un semplice script di shell.
Fra i sorgenti distribuiti insieme all'articolo troverete 4 directory in cui sono presenti i vari passi dell'applicazione delle utility al programma hello: *fisrt* illustra il programma all'inizio, *second* mostra un tentativo (manuale) di miglioramento, e *autoconf* e *automake*, rispettivamente, mostrano l'applicazione dei tool al programma.
Inoltre viene anche incluso un programma GNU, *java2html* [J2H], che utilizza questi due tool; in particolare la prima versione utilizza solo autoconf, mentre la seconda utilizza anche automake. Si tratta di un programma non tanto complesso, ma che sfrutta diverse caratteristiche di questi tool, in particolare java2html utilizza i tool lex e yacc, e quindi si può vedere come viene gestita la ricerca di tali programmi; inoltre vengono settate alcune variabili basandosi sul valore di altre (all'interno di configure.in).
Dopo questo primo articolo a carattere introduttivo si può passare a leggere la documentazione ufficiale di questi tool, presente in formato info, che però non brilla certo di comprensibilità. Nella documentazione in formato info troverete tutte le possibili macro che effettuano i controlli più svariati ed anche le istruzioni per crearne di personalizzate. Come sempre vale poi la raccomandazione di studiare gli esempi degli altri, e quindi prendere i sorgenti di un tool o programma GNU.

Conclusioni
Come si è visto autoconf ed automake sono dei tool molto potenti che permettono di automatizzare quelle fasi che possono essere critiche nel mantenimento e nella distribuzione dei sorgenti di un programma molto complesso. Quasi tutti i programmi GNU utilizzano questi strumenti anche perché ci sono delle estensioni molto utili per l'internazionalizzazione dei programmi. Lo script configure, nella maggior parte dei casi riesce ad esaminare un sistema, semplicemente utilizzando la shell *sh*.
Durante lo sviluppo si può aggiungere del codice che può risultare problematico per la portabilità; si consiglia quindi di rilanciare periodicamente l'utility autoscan, ed esaminare il file configure.scan in cerca di eventuali nuovi comandi AC.
Inoltre nella directory */usr/share/automake* si possono trovare diversi file utili per distribuire del software, ad esempio *COPYING*, che è una copia della licenza GPL, e *INSTALL*, che è un template per scrivere un file di testo con le istruzioni per la compilazione di un programma distribuito coi sorgenti, che utilizza configure.

Compilare un programma GNU

Compilare un programma GNU è un'operazione abbastanza semplice se si hanno già a disposizione tutte le librerie come prerequisitonecessarie alla compilazione del programma. Una volta ottenuto l'archivio contenente il programma in formato sorgenteda compilare è consigliabile estrarlo nel sotto il percorso /usr/local/src destinato alla compilazione di programmi per uso 'locale'. Per estrarre un programma in formato tar.gz si può fare nel seguente modo:

```
# cd /usr/local/src

# tar -zxvf nome.archivio.tar.gz
```

Prima di procedere a qualsiasi operazione successiva vale la pena spendere un pòo' di tempo nella lettura dei file README ed INSTALL di solito forniti a corredo. La lettura di questi file è indispensabile per capire se il nostro sistema ha tutti i prerequisiti necessari all'installazione del programma.

I programmi GNU non sono di solito forniti con Makefile, è infatti necessario lanciare lo script 'configure' per crearli. Questo script analizza il sistema su cui sta girando, creando un Makefile 'ad-hoc' da poter essere utilizzato sull'a attuale piattaforma. Questo sistema rende un programma GNU, non solo portatile attraverso i vari tipi di Unix, ma anche attraverso i vari tipi di processori.

Prima di eseguire 'configure' e può valere la pena leggere le opzioni che a volte contengono informazioni non contenute nella documentazione, col seguente comando:

```
# ./configure --help | more
```

Se configure termina con successo il programma potrà essere compilato ed installato con i seguenti comandi:

```
# ./configure --prefix=/usr/local/

# make

# make install
```

Risolvere alcuni problemi di compilazione

Tra i tanti problemi di compilazione che possono insorgere, i più frequenti sono quelli dovuti al fatto che il compilatore non riesce a trovare dei file header, o delle librerie. Vediamo come esempio il programma nel Listato 2 che crea una finestra in X-Window utilizzando la libreria GTK (www.gtk.org). Provando a compilare questo programma otteniamo:

```
# gcc gtk-hello.c

gtk-hello.c:1: gtk.h: No such file or directory
```

Il messaggio d'errore è stato generato perché il compilatore non ha trovato l'header "gtk.h" nei suoi percorsi di ricerca. Per risolvere questo problema dobbiamo specificare al compilatore dove cercare

tale file. Se non si conosce la directory dove si trova tale file, è possibile cercarlo con il seguente comando:

```
# find / -xdev -name gtk.h

/usr/include/gtk/gtk.h
```

una volta localizzatoche sappiamo dove si trova il file, utilizzando l'opzione -I, possiamo specificare al compilatore un ulteriore percorso aggiuntivo per la ricerca degli header file

```
#gcc gtk-hello.c -I/usr/include/gtk/

/tmp/cca006371.o: In function Main':

/tmp/cca006371.o(.text+0xc): undefined reference to G_print'

/tmp/cca006371.o(.text+0x1c): undefined reference to Gtk_init'

.

.

.

collect2: ld returned 1 exit status
```

La compilazione del programma è andata bene, ma questa volta è stato il linker 'ld' a segnalare che non è stato possibile 'risolvere' alcuni 'simboli'. Questo perché non abbiamo specificato quali sono le librerie dinamiche da collegare al programma. CiòQuesto si ottiene utilizzando l'opzione -l (elle minuscola), come nel seguente esempio:

```
# gcc gtk-hello.c -I/usr/include/gtk -lgtk -lgdk -lglib -lXext -lX11 -lm

/usr/i586-pc-linux-gnulibc1/bin/ld: cannot open -lXext: No such file or
directory

collect2: ld returned 1 exit status
```

specificare quali librerie dinamiche utilizzare non è a volte sufficiente, occorre anche dire al linker dove cercarle. Per far ciò si utilizza il parametro -L per specificare percorsi di ricerca aggiuntivi per le librerie.

```
# gcc gtk-hello.c -I/usr/include/gtk -lgtk -lgdk -lglib -lXext -lX11 -lm -
L/usr/X11R6/lib -o gtk-hello.exe
```

Al termine del comando l'eseguibile è pronto per essere lanciato con il nome di gtk-hello.exe

Il parametro prefix specifica a configure di creare i Makefile in modo tale che i file eseguibili vengano installati all'interno della gerarchica /usr/local

Per districarsi tra la miriade di messaggi generati da make & make install si puo' usare la seguente sintassi:

```
# make && make install && echo "Tutto OK !"
```

cio' garantisce che il messaggio finale venga stampato solo se le due operazioni precedenti sono andate a buon fine.