Una introduzione a GCC

per i compilatori GNU gcc e g++

Brian Gough
Prefazione di Richard M. Stallman

I dati di catalogazione per questo libro sono disponibili nella British Library.

Prima pubblicazione: Marzo 2004 (7/3/2004).

Pubblicato da Network Theory Limited.

15 Royal Park
Bristol
BS8 3AL
United Kingdom

Email: info@network-theory.co.uk

ISBN 0-9541617-9-3

Maggiori informazioni circa questo libro sono disponibili su
http://www.network-theory.co.uk/gcc/intro/

Immagine di copertina da uno schema di una pila hardware rapida ed energeticamente efficiente[1]. Immagine creata con il sistema di disegno elettronico libero Electric VLSI da Steven Rubin di Static Free Software (www.saticfreesoft.com). Static Free Software fornisce il supporto per Electric all'industria del disegno elettronico.

Il sorgente Texinfo di questo manuale può essere ottenuto da:
http://www.network-theory.co.uk/gcc/intro/src/

Traduzione in lingua italiana di Andrea Montagner (dgumo@tiscali.it)

Versione 1.0 (22 aprile 2008)

1 “A Fast and Energy-Efficient Stack” di J. Ebergen, D. Finchelstein, R. Kao, J. Lexau e R. Hopkins.

Indice generale

Prefazione

Questa prefazione è stata gentilmente offerta da Richard M. Stallman, l'autore principale di GCC e il fondatore del Progetto GNU.

Questo libro è una guida introduttiva a GCC, GNU Compiler Collection [Collezione di compilatori GNU]. Vi descriverà l'uso di GCC come strumento di programmazione. GCC è uno strumento di programmazione, ciò è vero – ma è anche qualcosa più. Fa parte di una campagna ventennale per la libertà degli utenti di computer.

Noi tutti vogliamo del buon software, ma cosa significa per il software essere “buono”? Funzionalità adeguate e affidabilità sono ciò che si intende essere *tecnicamente* buono, ma non basta. Il buon software deve essere anche *eticamente* buono: deve rispettare la libertà degli utenti.

Come utenti del software, dovreste avere il diritto di farlo girare come meglio credete, il diritto di studiarne il codice sorgente e quindi di modificarlo come meglio vi aggrada, il diritto di redistribuirne copie agli altri e il diritto di pubblicare una versione modificata in modo da contribuire alla costruzione della comunità. Quando un programma rispetta la vostra libertà in questo modo, noi possiamo definirlo *software libero*. Prima di GCC esistevano altri compilatori per C, Fortran, Ada, ecc..., ma non erano software liberi: non potevate usarli in libertà. Ho scritto GCC in modo da poter usare un compilatore senza gettare via la nostra libertà.

Un compilatore da solo non basta per usare un sistema di computer, avete bisogno di un sistema operativo completo. Nel 1983 tutti i sistemi operativi per computer moderni erano non-liberi. Per porvi rimedio nel 1984 iniziai lo sviluppo del sistema operativo GNU, un sistema simil-Unix che sarebbe stato software libero. Sviluppare GCC era una parte dello sviluppo di GNU.

A partire dai primi anni '90 il sistema operativo quasi terminato venne completato con l'aggiunta di un kernel, Linux, che era divenuto software libero nel 1992. Il sistema operativo frutto della combinazione GNU/Linux ha raggiunto l'obiettivo di rendere possibile l'uso del computer in libertà. Ma la libertà non è sicura automaticamente e noi dobbiamo darci da fare per difenderla. Il movimento del Software Libero ha bisogno del vostro aiuto.

Richard M Stallman
Febbraio 2004

1. Introduzione

Lo scopo di questo libro è quello di spiegare l'uso dei compilatori GNU C e C++, `gcc e g++`. Dopo la sua lettura dovreste comprendere come si compila un programma e come si usano le opzioni essenziali dei compilatori per l'ottimizzazione e la correzione degli errori. Questo libro non cerca di insegnare proprio i linguaggi C e C++, dal momento che ciò può essere trovato in molti altri luoghi (v. [Ulteriori letture], pag. 93).

I programmatori esperti che hanno familiarità con altri sistemi, eccetto che con nuovi compilatori GNU, possono saltare le prime sezioni dei capitoli "*Compilazione di un programma C*", "*Uso del preprocessore*" e "*Compilazione di un programma C++*". I restanti capitoli e sezioni dovrebbero fornire una discreta panoramica sulle caratteristiche di GCC per quelli che già sanno come usare altri compilatori.

1.1. Breve storia di GCC

L'autore originario di GNU C Compiler (GCC o *Compilatore C GNU*) è Richard Stallman, il fondatore del Progetto GNU.

Il progetto GNU è iniziato nel 1984 per realizzare un completo sistema operativo simil-Unix che fosse software libero in modo da favorire la libertà e la collaborazione tra gli utenti e i programmatori di computer. Qualsiasi sistema operativo simil-Unix necessita di un compilatore C e, siccome all'epoca non esistevano compilatori liberi, il Progetto GNU iniziò a crearne uno da zero. L'opera venne finanziata dalle donazioni versate da privati e società alla Free Software Foundation, un'organizzazione non a scopo di lucro istituita per sostenere l'attività del Progetto GNU.

La prima versione di GCC venne alla luce nel 1987. Questa fu una grossa innovazione, trattandosi del primo compilatore ANSI C ottimizzante rilasciato come software libero. Sino da allora GCC è divenuto uno degli strumenti più importanti nello sviluppo del software libero.

Una importante revisione del compilatore apparve con la serie 2.0 nel 1992, che aggiunse la capacità di compilare C++. Nel 1997 fu creata una branca sperimentale del compilatore (EGCS) per migliorare l'ottimizzazione e il supporto C++. In seguito a ciò, EGCS venne adottato come nuova linea principale dello sviluppo di GCC e queste funzionalità divennero ampiamente disponibili nella versione 3.0 di GCC nel 2001.

Con il tempo GCC fu esteso per supportare molti linguaggi aggiuntivi, compresi Fortran, ADA,

Java e Objective-C. L'acronimo GCC è attualmente utilizzato per indicare la “GNU Compiler Collection” [*Collezione di Compilatori GNU*]. Il suo sviluppo è guidato dal *GCC Steering Committee*, un gruppo composto dai rappresentanti delle comunità di utenti GCC nell'industria, nella ricerca e nell'accademia.

1.2. Caratteristiche principali di GCC

Questa sezione descrive alcune delle più importanti caratteristiche di GCC.

Prima di tutte, GCC è un compilatore portabile – gira attualmente sulle principali piattaforme disponibili al giorno d'oggi ed è in grado di produrre risultati per molti tipi di processore. In aggiunta ai processori impiegati nei personal computer supporta anche microcontrollori, DSP e CPU a 64-bit.

GCC non è soltanto un compilatore nativo: può anche effettuare la compilazione incrociata [*cross-compilation*] di ogni programma, producendo file eseguibili per un differente sistema partendo da quello usato dallo stesso GCC. Ciò consente di compilare software per sistemi integrati che non sono in grado di far girare un compilatore. GCC è scritto in C con una forte attenzione alla portabilità e può compilare se stesso in modo da poter essere adattato a nuovi sistemi con facilità.

GCC possiede molteplici *front-end* [programmi “frontali” che fanno da interfaccia semplificata tra il programma vero e proprio e l'utente] dei linguaggi per poter trattare linguaggi differenti. Programmi in qualsiasi linguaggio possono essere compilati, o compilati in modo incrociato, per qualsiasi architettura. Per esempio, un programma ADA può essere compilato per un microcontrollore oppure un programma C per un supercomputer.

GCC ha una progettazione modulare che gli consente l'inclusione del supporto per nuovi linguaggi e architetture. Aggiungere un nuovo front-end di linguaggio a GCC abilita all'uso di quel linguaggio su qualsiasi architettura a condizione che siano disponibili gli strumenti per l'esecuzione (come le librerie). Allo stesso modo l'integrazione del supporto per nuove architetture lo rende fruibile con tutti i linguaggi.

Infine - e cosa molto importante – GCC è software libero, distribuito sotto la GNU General Public License (GNU GPL[1] o Licenza Pubblica Generale GNU) Ciò significa che avete la libertà di usare e modificare GCC come con tutto il software GNU. Se avete bisogno del supporto per un nuovo tipo di CPU, per un nuovo linguaggio, o una nuova funzionalità, potete aggiungerli da voi stessi, oppure potete pagare qualcuno che sviluppi GCC al posto vostro. Potete inoltre pagare qualcuno per la correzione di un errore se ciò è importante per la vostra attività.

Inoltre avete la libertà di condividere qualsiasi miglioria da voi apportata a GCC. Come conseguenza di questa libertà, potete anche far uso dei miglioramenti a GCC apportati da altri. Le molte caratteristiche offerte da GCC oggi mostrano come questa libertà di collaborazione agisca a vantaggio vostro e di chiunque altro usi GCC.

(1) Per dettagli leggete il file di licenza “COPYING” distribuito con GCC.

1.3. Programmazione in C e C++

C e C++ sono linguaggi che permettono l'accesso diretto alla memoria del computer. Storicamente sono stati impiegati per la scrittura del software di sistema a basso livello e di applicazioni dove le prestazioni elevate o il controllo sull'uso delle risorse sono critici. Tuttavia è richiesta un'attenzione scrupolosa per assicurare che l'accesso alla memoria avvenga correttamente per evitare la corruzione di altre strutture di dati. Questo libro descrive tecniche che vi aiuteranno a scoprire potenziali errori durante la compilazione, sebbene il rischio nell'utilizzo di linguaggi come C o C++ non possa essere mai eliminato.

In aggiunta a C e C++ il Progetto GNU fornisce anche altri linguaggi ad alto livello come GNU Common Lisp (`gcl`), GNU Smalltalk (`gst`), il linguaggio ad estensione GNU Scheme (`guile`) e il compilatore GNU per Java (`gcj`). Questi linguaggi non consentono all'utente di accedere direttamente alla memoria, eliminando la possibilità di errori d'accesso alla stessa: sono un alternativa più sicura a C e C++ per molte applicazioni.

1.4. Convenzioni usate in questo manuale

Questo manuale contiene molti esempi che possono essere digitati con la tastiera. Un comando inserito nel terminale appare come questo:

```
$ comando
```

seguito dai suoi dati in uscita. Per esempio:

```
$ echo   Ciao mondo
Ciao mondo
```

Il primo carattere della linea è il *prompt* o *invito* del terminale e non deve essere digitato. Il segno del dollaro '$' si usa come invito standard in questo manuale, sebbene alcuni sistemi possano utilizzare un carattere diverso.

Quando un comando in un esempio è troppo lungo per contenerlo entro una singola linea, esso viene mandato a capo e quindi indentato nelle linee successive, come questo:

```
$ echo   un esempio di linea che è troppo lunga per essere
     contenuta in questo manuale
```

Quando viene inserito con la tastiera, l'intero comando dovrebbe essere digitato di seguito.

I file sorgenti di esempio usati in questo manuale possono essere scaricati dal sito internet dell'editore[(2)] oppure inseriti manualmente utilizzando un qualsiasi editor di testo come, ad esempio, `emacs`, l'editor GNU standard. Gli esempi dei comandi di compilazione impiegano `gcc` e `g++` come nomi dei compilatori GNU C e C++ e `cc` per riferirsi agli altri. I programmi di esempio dovrebbero funzionare con qualsiasi versione di GCC. Nel testo qualsiasi opzione della linea di comando disponibile solo nelle versioni recenti di GCC viene segnalata.

(2) V. `http://www.network-theory.co.uk/gcc/intro/`

Gli esempi danno per scontato l'utilizzo di un sistema operativo GNU (potrebbero esserci delle minime diversità nell'emissione dei dati in altri sistemi). Alcuni messaggi non essenziali e prolissi di emissione dipendente dal sistema (come percorsi piuttosto lunghi) per brevità sono stati modificati negli esempi. I comandi per impostare le variabili ambientali usano la sintassi della shell standard GNU (`bash`) e dovrebbero funzionare con qualsiasi versione della Bourne shell.

2. Compilazione di un programma C

Questo capitolo descrive come si compilano i programmi C utilizzando `gcc`. I programmi possono essere compilati partendo da un singolo file sorgente o da molteplici file sorgenti e possono ricorrere a librerie di sistema e file di intestazione (*header files*).

La compilazione è quel processo di conversione di un programma dal *codice sorgente* (in un linguaggio di programmazione come C o C++) al *codice macchina*, cioè la sequenza di 1 e 0 usata per controllare l'unità di elaborazione (CPU) del computer. Tale codice macchina viene poi conservato in un file noto come *file eseguibile*, talvolta chiamato *file binario*.

2.1. Compilazione di un semplice programma C

Il classico programma d'esempio per il linguaggio C è *`CiaoMondo`*. Qui c'è il codice sorgente della nostra versione del programma:

```
#include <stdio.h>
int
main (void)
{1\
printf (  Ciao Mondo!\n  );
return 0;
}
```

Presumeremo che il codice sorgente sia conservato in un file chiamato '`ciao.c`'. Per compilare questo file con `gcc`, useremo il comando seguente:

```
$ gcc -Wall ciao.c -o ciao
```

Ciò compila il codice sorgente di '`ciao.c`' in codice macchina e lo salva nel file eseguibile '`ciao`'. Il file emesso in codice macchina viene specificato usando l'opzione '`-o`'. Tale opzione viene solitamente inserita come ultimo argomento nella linea di comando. Se omessa, il risultato viene scritto in un file standard chiamato '`a.out`'.

Notate che se esiste già un file con lo stesso nome di quello eseguibile nella medesima directory, questo verrà sovrascritto.

L'opzione '`-Wall`' attiva tutti gli avvertimenti [*warning*] del compilatore più comunemente usati (si raccomanda di utilizzare sempre questa opzione!). Esistono molte altre opzioni di avvertimento che tratteremo negli ultimi capitoli, ma '`-Wall`' è la più importante. GCC non genererà alcun avvertimento a meno che questo non venga abilitato. Gli avvertimenti del compilatori costituiscono un aiuto essenziale nella scoperta dei problemi mentre si programma in C e C++.

In tal caso il compilatore non emette nessun avvertimento con l'opzione '`-Wall`' dal momento che il programma è totalmente valido. Il codice sorgente che non produce avvertimenti si dice che "*compila pulito*" [*it compiles cleanly*].

Per avviare il programma, battete il nome di percorso dell'eseguibile come nell'esempio:

```
$ ./ciao
Ciao Mondo!
```

Tutto ciò carica il file eseguibile in memoria e costringe la CPU ad iniziare l'esecuzione delle istruzioni in esso contenute. Il percorso `./` si riferisce alla directory corrente, in modo che `./ciao` carica ed avvia il file eseguibile '`ciao`' collocato nella directory corrente.

2.2. Scoperta degli errori in un semplice programma

Come menzionato prima, gli avvertimenti [*warning*] del compilatore costituiscono un aiuto fondamentale durante la programmazione in C e C++. Per dimostrare ciò, il programma seguente contiene un piccolo errore: utilizza la funzione `printf` in modo scorretto, specificando un formato a virgola mobile '%f' per un valore intero:

```
#include <stdio.h>

int
main (void)
{
     printf (  Due più due fa %f\n  ,4);
     return 0;
}
```

Questo errore non è manifesto di primo acchito, ma può essere individuato dal compilatore se l'opzione '`-Wall`' è stata attivata.

Compilando il programma qui sopra, '`sbaglio.c`', con l'opzione di avvertimento '`-Wall`' si ottiene il seguente messaggio

```
$ gcc -Wall sbaglio.c -o bad
sbaglio.c: In function 'main':
sbaglio.c:6: warning: double, format different type arg (arg 2)
```

Ciò indica che una stringa di formato è stata utilizzata in modo scorretto nella linea 6 del file '`sbaglio.c`'. Il messaggio generato da GCC ha sempre la forma *file:numero-riga:messaggio*. Il compilatore distingue tra *messaggi di errore* [*error messages*], che impediscono la riuscita della compilazione, e *messaggi di avvertimento* [*warning messages*] che segnalano possibili problemi (ma non impediscono la compilazione del programma).

In questo caso, il corretto specificatore di formato avrebbe dovuto essere '%d' (gli specificatori di formato permessi per printf possono essere rintracciati in qualsiasi generico libro sul C, come il *GNU C Library Reference Manual*, (v. 'Ulteriori letture', pag. 93).

Senza l'opzione di avvertimento '`-Wall`' il programma sembra compilare in modo pulito, ma produce risultati sbagliati:

```
$ gcc sbaglio.c -o sbaglio
$ ./sbaglio
Due più due fa 2.585495          (risultato errato)
```

L'errato specificatore di formato determina un risultato scorretto, perché alla funzione `printf` viene passato un numero intero al posto di uno in virgola mobile. I numeri interi e in virgola mobile vengono conservati nella memoria in formati differenti, e generalmente occupano una quantità diversa di byte, cosa che conduce a risultati spuri. Il reale risultato mostrato sopra può essere diverso a seconda degli specifici piattaforma ed ambiente.

Evidentemente è molto pericoloso sviluppare un programma senza controllare gli avvertimenti del compilatore. Se esiste qualche funzione non utilizzata correttamente, questa può bloccare il programma oppure produrre risultati sbagliati. L'attivazione dell'opzione di avvertimento '`-Wall`' farà intercettare molti dei più comuni errori che si commettono nella programmazione in C.

2.3. Compilazione di molteplici file sorgente

Un programma può essere suddiviso in numerosi file. Ciò lo rende più semplice da scrivere e da comprendere, specialmente in caso di grandi programmi (permette anche di compilare in modo indipendente le singole parti).

Nell'esempio seguente spezzeremo il programma *CiaoMondo* in tre file: '`main.c`', '`ciao_fn.c`' ed il file di intestazione '`ciao.h`'. Ecco qui il programma principale '`main.c`':

```
#include "ciao.h"
int
main (void)
{
ciao ("mondo");
return 0;
}
```

L'originale chiamata alla funzione di sistema `printf` nel precedente programma '`ciao.c`' è stato sostituito da una chiamata ad una nuova funzione esterna `ciao` che noi definiremo in un file separato '`ciao_fn.c`'.

Il programma principale comprende anche il file di intestazione '`ciao.h`' che conterrà la dichiarazione della funzione `ciao`. La dichiarazione viene usata per assicurare che i tipi degli argomenti ed il valore di ritorno corrispondano correttamente tra la chiamata della funzione e la definizione della stessa. Non ci serve più includere il file di intestazione di sistema '`stdio.h`' in '`main.c`' per dichiarare la funzione `printf` in quanto il file '`main.c`' non la chiama direttamente.

La dichiarazione in '`ciao.h`' è una singola linea che specifica il prototipo della funzione `ciao`:

```
void ciao (const char * name);
```

La definizione della stessa funzione `ciao` è contenuta nel file '`ciao_fn.c`':

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

void
ciao (const char * nome)
{
printf ("Ciao, %s!\n", nome);
}
```

Questa funzione stampa il messaggio "`Ciao,` *nome*`!`" utilizzando il suo argomento come valore di *nome*.

Tra l'altro, la differenza tra le due forme `#include` *`FILE`*`.h` e `#include <`*`FILE`*`.h>` del comando **#include** consiste nel fatto che il compilatore cerca '*`FILE`*`.h`' nella directory corrente prima di guardare in quelle di sistema dei file di intestazione. Il comando include `#include <`*`FILE`*`.h>` cerca i file di intestazione di sistema, ma non nella directory corrente, salvo istruzioni diverse.

Per compilare questi file sorgenti con `gcc`, utilizzate il comando seguente:

```
$ gcc -Wall main.c ciao_fn.c -o nuovociao
```

In tal caso noi impieghiamo l'opzione '`-o`' per specificare un diverso file in uscita per l'eseguibile, '`nuovociao`'. Osservate che il file di intestazione '`ciao.h`' non è indicato nell'elenco dei file nella linea di comando. La direttiva `#include <ciao.h>` nei file sorgenti istruisce il compilatore ad includerlo automaticamente nelle posizioni opportune.

Per avviare il programma, battete il nome del percorso dell'eseguibile:

```
$ ./nuovociao
```

```
Ciao Mondo!
```

Tutte le parti del programma sono state raccolte in un singolo file eseguibile che produce il medesimo risultato di un eseguibile creato dal singolo file sorgente utilizzato in precedenza.

2.4. Compilare i file in modo indipendente

Se un programma viene conservato in un unico file, qualsiasi modifica ad ogni singola funzione richiede la ricompilazione dell'intero programma per produrre un nuovo eseguibile. La ricompilazione di grossi file sorgenti può essere molto dispendiosa in termini di tempo.

Quando i programmi sono registrati in file sorgenti indipendenti, soltanto i file che sono stati modificati necessitano di essere ricompilati dopo la modifica del file sorgente. Con questo approccio i file sorgenti vengono compilati separatamente e poi *collegati* (o "*linkati*") assieme – un processo a due fasi: nella prima, un file viene compilato senza creare un eseguibile. Il risultato viene chiamato *file oggetto* (*object file*) e possiede l'estensione '`.o`' quando si usa `GCC`.

Nella seconda fase i file oggetto vengono fusi assieme da un programma separato chiamato *linker* (o *combinatore*). Quest'ultimo mette insieme tutti i file oggetto per creare un singolo eseguibile.

Un file oggetto contiene del codice macchina in cui ogni riferimento ad indirizzi di memoria delle funzioni (o delle variabili) in altri file viene lasciato indefinito. Ciò consente ai file sorgenti di essere compilati senza riferimenti diretti a ciascuno di essi. Il combinatore (o *linker*) mette tali indirizzi mancanti quando produce l'eseguibile.

2.4.1. Creazione di file oggetto dai file sorgenti

L'opzione a linea di comando '`-c`' viene utilizzata per compilare un file sorgente in un file oggetto. Per esempio, il comando seguente compilerà il file sorgente '`main.c`' in un file oggetto:

```
$ gcc -Wall -c main.c
```

Tutto ciò genera il file oggetto '`main.o`' contenente il codice macchina della funzione `main`. Esso possiede un riferimento alla funzione esterna `ciao`, ma in questa fase il corrispondente indirizzo di memoria viene lasciato indefinito nel file oggetto (sarà indicato più tardi durante il *linkaggio*).

Il corrispondente comando per la compilazione della funzione `ciao` nel file sorgente '`ciao_fn.c`' è:

```
$ gcc -Wall -c ciao_fn.c
```

Questo produce il file oggetto '`ciao_fn.o`'.

Notate che in questo caso non serve usare l'opzione '`-o`' per indicare il nome del file in uscita. Quando si compila con '`-c`' il compilatore crea automaticamente un file oggetto il cui nome è uguale a quello del file sorgente, salvo '`.o`' al posto dell'estensione originale.

Non c'è bisogno di mettere il il file di intestazione '`ciao.h`' nella linea di comando dal momento che viene incluso automaticamente dalle istruzioni `#include` di '`main.c`' e di '`ciao_fn.c`'.

2.4.2. Creazione di eseguibili da file oggetto

Il passo conclusivo per la creazione di un file eseguibile è l'impiego di `gcc` per mettere insieme [*linkaggio* o collegamento] i file oggetto e per definire gli indirizzi mancanti delle funzioni esterne. Per combinare i file oggetto basta semplicemente elencarli nella linea di comando:

```
$ gcc main.o ciao_fn.o -o ciao
```

Questa è una delle poche occasioni in cui no serve usare l'opzione di avvertimento '`-Wall`', dal momento che i singoli file sorgenti sono stati già compilati in codice oggetto con successo. Una volta che i file sorgenti sono stati compilati, il *linkaggio* (o collegamento) è un processo non ambiguo che può riuscire o meno (fallisce solo se ci sono dei riferimenti che non possono essere risolti).

Per eseguire la fase di collegamento `gcc` utilizza il *linker* `ld`, che è un programma distinto.. Nei sistemi GNU si usa il *linker* GNU `ld`. Altri sistemi potrebbero usare il *linker* GNU insieme a `GCC` oppure potrebbero avere i propri *linker*. Il *linker* stesso verrà trattato più avanti (v. Capitolo 11, Come funziona il compilatore, pag. 83). Avviando il *linker*, gcc crea u file eseguibile partendo dai file oggetto.

Il file eseguibile risultante può essere ora fatto partire:

```
$ ./ciao
Ciao, mondo!
```

Produce lo stesso risultato della versione del programma nella sezione precedente che usa un solo file sorgente.

2.4.3. L'ordine dei collegamenti dei file oggetto

Nei sistemi simil-Unix il comportamento tradizionale dei compilatori e dei *linker* è quello di cercare funzioni esterne da sinistra a destra nei file oggetto specificati nella linea di comando. Ciò comporta che il file oggetto con la definizione di una funzione dovrebbe presentarsi dopo ciascun file che invoca tale funzione.

In tal caso, il file '`ciao_fn.o`' contenente la funzione `ciao` dovrebbe essere essere indicato dopo '`main.o`' stesso, dal momento che `main` chiama `ciao`:

```
$ gcc main.o ciao_fn.o -o hello     (ordine corretto)
```

Con alcuni compilatori o *linker* l'ordine inverso potrebbe causare un errore,

```
$ cc ciao_fn.o main.o -o ciao
main.o: In function `main':
```

```
main.o(.text+0xf): undefined reference to `ciao'
```

perché non esistono file oggetto contenenti `ciao` dopo '`main.o`'.

Molti compilatori e *linker* recenti cercheranno tutti i file oggetti, senza preoccuparsi dell'ordine, ma, poiché non tutti i compilatori si comportano così, è preferibile seguire la convenzione di ordinare i file oggetto da sinistra a destra.

E' bene tenerlo in mente se incontrerete problemi inattesi con i riferimenti indefiniti e tutti i file oggetto appariranno essere presenti nella linea di comando.

2.5. Ricompilazione e ricollegamento

Per mostrare come possono essere compilati i file sorgenti scriveremo il programma principale '`main.c`' e lo modificheremo per stampare un saluto a tutti al posto che al mondo:

```
#include "ciao.h"
int
main (void)
{
  ciao ("a tutti"); /* cambiato rispetto a "mondo" */
  return 0;
}
```

Il file '`main.c`' aggiornato può essere ora ricompilato con il comando seguente:

```
$ gcc -Wall -c main.c
```

Ciò produce un nuovo file oggetto '`main.o`'. Non serve creare un nuovo file oggetto per '`ciao_fn.c`' dal momento che questo file insieme a quelli da lui dipendenti, come i file di intestazione, non sono stati modificati.

Il nuovo file oggetto può essere collegato nuovamente con la funzione `ciao` per generare un nuovo file eseguibile:

```
$ gcc main.o ciao_fn.o -o ciao
```

L'eseguibile risultante '`ciao`' adesso utilizza la nuova funzione `main` per produrre il seguente risultato:

```
$ ./ciao
Ciao a tutti!
```

Notate che soltanto il file 'main.c' è stato ricompilato e poi *rilinkato* con l'esistente file oggetto per la funzione `ciao`. Se invece fosse stato modificato il file '`ciao_fn.c`', avremmo potuto ricompilarlo per creare un nuovo file oggetto '`ciao_fn.o`', collegando quest'ultimo con il file

'`main.o`' esistente [1].

In genere il collegamento o *linkaggio* è più rapido della compilazione – in un grosso progetto con molti file sorgenti, la ricompilazione soltanto di quelli che sono stati modificati può consentire un risparmio significativo. Il processo di ricompilare solo i file modificati in un progetto può essere automatizzato utilizzando *GNU Make* (v. [Ulteriori letture], pagina 93).

2.6. Il collegamento con librerie esterne

La libreria è una collezione di file oggetto precompilati che `possono` essere collegati ai programmi. L'utilizzo più comune delle librerie è quello di fornire funzioni di sistema, come ad esempio la funzione della radice quadrata `sqrt` che si trova nella libreria `math` del C.

Le librerie vengono normalmente conservate in file di archivio con estensione '.a', a cui ci si riferisce come *librerie statiche* (o *static libraries*). Esse vengono create partendo da file oggetto con uno strumento separato, l'archiviatore GNU `ar`, e vengono utilizzate dal *linker* per risolvere i riferimenti alle funzioni durante la compilazione. Vedremo più avanti come creare librerie ricorrendo al comando `ar` (v. Capitolo 10 - Strumenti collegati al compilatore [pag. 75]). Per semplicità, questa sezione tratta solo le librerie statiche (il collegamento dinamico durante l'esecuzione utilizzando le librerie condivise [*shared libraries*] verrà descritto nel prossimo capitolo.

Le librerie standard di sistema si trovano di solito nelle directory '`/usr/lib`' e '`/lib`' [2]. Per esempio, la libreria C `math` viene conservata normalmente nel file '`/usr/lib/libm.a`' nei sistemi simil-Unix. Le relative dichiarazioni di prototipo per le funzioni di questa libreria sono date dal file di intestazione '`/usr/include/math.h`'. La stessa libreria C standard viene conservata in '`/usr/lib/libc.a`' e contiene le funzioni specificate nello standard ANSI/ISO C, come '`printf`' (questa libreria viene collegata ad ogni programma C, salvo diverse indicazioni).

Qui c'è un programma di esempio che effettua una chiamata alla funzione esterna `sqrt` nella libreria matematica '`libm.a`':

```
#include <math.h>

#include <stdio.h>



int

main (void)

{
```

1 Se il prototipo di una funzione è stato cambiato, è necessario modificare e ricompilare tutti gli altri file sorgenti che lo utilizzano.

2 Nei sistemi che supportano eseguibili sia a 64 che a 32 bit, le versioni a 64 bit delle librerie si troveranno spesso conservate in '`/usr/lib64`' e in '`/lib64`', mentre quelle a 32 bit in '`/usr/lib`' e '`/lib`'.

```
  double x = sqrt (2.0);

  printf ("La radice quadrata di 2.0 vale %f\n", x);

  return 0;

}
```

Provare a creare un eseguibile da questo unico file sorgente provoca un errore del compilatore nella fase di collegamento (*linking*):

```
$ gcc -Wall calc.c -o calc
/tmp/ccbR6Ojm.o: In function `main':
/tmp/ccbR6Ojm.o(.text+0x19): undefined reference
  to `sqrt'
```

Il problema consiste nel riferimento alla funzione `sqrt` che non può essere risolto senza la libreria matematica esterna '`libm.a`'. La funzione `sqrt` non è definita nel programma o nella libreria base '`libc.a`' ed il compilatore non collega il file '`libm.a`' a meno che non sia esplicitamente selezionato. Tra l'altro, il file riportato nel messaggio di errore '`/tmp/ccbR6Ojm.o`' è un file oggetto temporaneo creato dal compilatore partendo da '`calc.o`' per procedere nel processo di collegamento.

Per abilitare il compilatore a collegare la funzione `sqrt` al programma principale '`calc.o`' dobbiamo fornirgli la libreria '`libm.a`'. Un modo ovvio ma ridondante per fare ciò è quello di indicarla espressamente nella linea di comando:

```
$ gcc -Wall calc.c /usr/lib/libm.a -o calc
```

La libreria '`libm.a`' contiene dei file oggetto per tutte le funzioni matematiche, come `sin`, `cos`, `exp`, `log`, e `sqrt`. Il collegatore (*linker)* fruga tra questi per trovare il file oggetto contenente la funzione `sqrt`.

Una volta che è stato rintracciato il file oggetto della funzione `sqrt`, il programma `main` può essere collegato e viene prodotto un eseguibile completo:

```
$ ./calc
La radice quadrata di 2.0 vale 1.414214
```

Il file eseguibile comprende il codice macchina della funzione principale (*main*) e della funzione `sqrt`, copiate dai corrispondenti file oggetto della libreria '`libm.a`'.

Per evitare di dover specificare lunghi percorsi nella linea di comando, il compilatore mette a disposizione una opzione di scorciatoia '`-l`' per collegarsi alle librerie. Per esempio il comando seguente:

```
$ gcc -Wall calc.c -lm -o calc
```

equivale all'originario comando precedente che utilizza l'intero nome di libreria

'`/usr/lib/libm.a`'.

In genere l'opzione '`-lNAME`' proverà a collegare i file oggetto con un file di libreria '`-l.NAME.a`' nelle directory standard delle librerie. Possono essere indicate delle directory aggiuntive ricorrendo alle opzioni a linea di comando e alle variabili ambientali che tratteremo brevemente. Un grosso programma usa normalmente molte opzioni '`-l`' per collegare librerie come quelle matematiche, grafiche e di rete.

2.6.1. Ordine di collegamento delle librerie

La sequenza delle librerie nella linea di comando segue la stessa convenzione dei file oggetto: esse vengono ricercate da sinistra a destra – una libreria contenente la definizione di una funzione dovrebbe comparire dopo qualsiasi file sorgente od oggetto che la utilizzino. Ciò comprende le librerie indicate con la scorciatoia '`-l`', come viene mostrato nel comando seguente:

```
$ gcc -Wall calc.c -lm -o calc          (ordine corretto)
```

Con alcuni compilatori l'ordine contrario (collocando l'opzione '`-l`' prima del file che la utilizza) potrebbe restituire un errore,

```
$ cc -Wall -lm calc.c -o calc           (ordine sbagliato)
main.o: In function `main':
main.o(.text+0xf): undefined reference to `sqrt'
```

perché non esiste una libreria o un file oggetto contenente `sqrt` dopo '`calc.c`'. L'opzione '`-lm`' dovrebbe comparire dopo il file '`calc.c`'.

Quando vengono utilizzate diverse librerie, la medesima convenzione dovrebbe essere seguita per le librerie tra di loro. Una libreria che chiama una funzione esterna definita in un'altra libreria dovrebbe comparire dopo di questa.

Per esempio, un programma '`data.c`' che utilizza la libreria di Programmazione Lineare GNU '`libglpk.a`', che a sua volta usa la libreria matematica '`libm.a`', dovrebbe essere compilata come segue

```
$ gcc -Wall data.c -lglpk -lm
```

dal momento che i file oggetto in '`libglpk.a`' impiegano funzioni definite in '`libm.a`'.

Come per i file oggetto, molti degli attuali compilatori cercheranno tutte le librerie, senza preoccuparsi dell'ordine. Comunque, dal momento che non tutti i compilatori si comportano così, è preferibile seguire la convenzione di disporre le librerie da sinistra a destra.

2.7. Uso dei file di intestazione delle librerie

Quando si utilizza una libreria è essenziale includere i file di intestazione appropriati per dichiarare gli argomenti della funzione e restituire valori di tipo corretto. Senza dichiarazioni gli

argomenti di una funzione possono essere passati con il tipo sbagliato, causando risultati alterati.

L'esempio seguente mostra un altro programma che effettua una chiamata di funzione alla libreria matematica del C. In questo caso la funzione `pow` viene utilizzata per calcolare il cubo di due (2 elevato alla potenza di 3):

```
#include <stdio.h>

int
main (void)
{
  double x = pow (2.0, 3.0);
  printf ("Due al cubo corrisponde a %f\n", x);
  return 0;
}
```

Comunque il programma contiene un errore – l'istruzione `#include` per '`math.h`' è assente, cosicché il prototipo `double pow (double x, double y)`, lì definito, non verrà visto dal compilatore.

La compilazione del programma senza alcuna opzione di avvertimento produrrà un file eseguibile che darà risultati errati:

```
$ gcc badpow.c -lm
$ ./a.out
Due al cubo corrisponde a 2.851120     (risultato errato: dovrebbe essere 8)
```

I risultati sono errati perché gli argomenti ed il valore di ritorno della chiamata a `pow` vengono passati con i tipi sbagliati[3]. Ciò può essere individuato attivando l'opzione di avvertimento '`-Wall`':

```
$ gcc -Wall badpow.c -lm
badpow.c: In function `main':
badpow.c:6: warning: implicit declaration of
  function `pow'
```

Questo esempio mostra nuovamente l'importanza dell'utilizzo dell'opzione di avvertimento '`-Wall`' per la scoperta di problemi seri che altrimenti potrebbero facilmente passare inosservati.

(3) Il risultato reale mostrato qui sopra potrebbe essere diverso in base agli specifici piattaforma e ambiente.

3. Opzioni di compilazione

Questo capitolo descrive altre opzioni del compilatore di uso comune disponibili in GCC. Tali opzioni controllano le caratteristiche come i percorsi di ricerca utilizzati per la localizzazione delle librerie e dei file di inclusione, l'uso degli avvertimenti aggiuntivi e della diagnosi, le macro del preprocessore e i dialetti del linguaggio C.

3.1. Definizione dei percorsi di ricerca

Nel capitolo precedente abbiamo visto come collegare un programma con funzioni alla libreria matematica del C '`libm.a`' utilizzando l'opzione-scorciatoia '`-lm`' ed il file di intestazione '`math.c`'.

Un problema comune quando si compila un programma impiegando i file di intestazione è l'errore:

```
FILE.h : No such file or directory
```

Questo capita se il file di intestazione non è presente nelle directory dei file di inclusione standard utilizzate da `gcc`. Un problema analogo si può verificare per le librerie:

```
/usr/bin/ld: cannot find library
```

Ciò accade quando una libreria da collegare non è presente nelle directory standard delle librerie usate da `gcc`.

Normalmente `gcc` cerca i file di intestazione nelle seguenti directory:

```
/usr/local/include/
/usr/include/
```

L'elenco delle directory dei file di intestazione è indicato spesso come *percorso di inclusione* (*include path*), mentre quello delle directory delle librerie come il *percorso di ricerca delle librerie* (*library search path*) o *percorso di collegamento* (*link path*).

Le directory di questi percorsi vengono esaminate in ordine, dalla prima all'ultima, nei due elenchi citati [(1)]. Per esempio, un file di intestazione (*header file*) trovato in

(1) I percorsi standard di ricerca possono anche comprendere directory aggiuntive dipendenti dal sistema o specifiche del sito, oltre a quelle della installazione stessa di `GCC`. Per esempio, in piattaforme a 64 bit le directory addizionali

'/usr/local/include' ha la precedenza su di un file con lo stesso nome in '/usr/include'. Allo stesso modo una libreria trovata in '/usr/local/lib' ha la precedenza su una libreria con lo stesso nome in '/usr/lib'.

Quando si installano librerie aggiuntive in altre directory è necessario estendere i percorsi di ricerca per permettere alle librerie di essere rintracciate. Le opzioni '-I' e '-L' del compilatore aggiungono nuove directory all'inizio rispettivamente del percorso di inclusione e del percorso di ricerca delle librerie.

3.1.1. Esempio di percorso di ricerca

Il seguente programma di esempio utilizza una libreria che potrebbe essere installata come pacchetto addizionale in un sistema – la Libreria GNU di Gestione di Base Dati (GDBM o *GNU Database Management Library*). La Libreria GDBM conserva coppie di valori-chiave in un file DBM, un tipo di file di dati che consente di mantenere ed indicizzare dei valori mediante una chiave o *key* (una sequenza arbitraria di caratteri). Qui c'è il programma di esempio 'dbmain.c' che crea un file DBM contenente una chiave 'testkey' di valore 'testvalue':

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

int
main (void)
{
  GDBM_FILE dbf;
  datum key = { "testkey", 7 };     /* key, length */
  datum value = { "testvalue", 9 }; /* value, length */

  printf ("Storing key-value pair... ");
  dbf = gdbm_open ("test", 0, GDBM_NEWDB, 0644, 0);
  gdbm_store (dbf, key, value, GDBM_INSERT);
  gdbm_close (dbf);
  printf ("done.\n");
  return 0;
}
```

Il programma impiega il file di intestazione 'gdbm' e la libreria 'libgdbm'. Se la libreria è stata installata nella posizione consueta '/usr/local/lib' con il file di intestazione in '/usr/local/include', allora il programma potrà essere compilato con il seguente semplice comando:

```
$ gcc -Wall dbmain.c -lgdbm
```

'lib64' normalmente possono anche essere oggetto d ricerche.

Entrambe le directory fanno parte dei normali percorsi di intestazione e collegamento di `gcc`.

Altrimenti se GDBM è stato installata in un posto diverso, il tentativo di compilazione del programma produrrà il seguente errore:

```
$ gcc -Wall dbmain.c -lgdbm
dbmain.c:1: gdbm.h: No such file or directory
```

Per esempio, se la versione 1.8.3 del pacchetto GDBM è installata nella directory '`/opt/gdbm-1.8.3`' la posizione del file di intestazione dovrebbe essere

```
/opt/gdbm-1.8.3/include/gdbm.h
```

che non fa parte del normale percorso delle intestazioni di `gcc`. L'aggiunta della directory corretta al percorso delle intestazioni con l'opzione da linea di comando '`-I`' permette al programma di essere compilato, ma non collegato:

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include dbmain.c -lgdbm
/usr/bin/ld: cannot find -lgdbm
collect2: ld returned 1 exit status
```

La directory che contiene la libreria non è ancora assente dal percorso di collegamento. Può venire aggiunta a questo percorso utilizzando la seguente opzione:

```
-L/opt/gdbm-1.8.3/lib/
```

La successiva linea di comando consente di compilare e collegare il programma:

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include
   -L/opt/gdbm-1.8.3/lib dbmain.c -lgdbm
```

Ciò produce l'eseguibile finale collegato alla libreria GDBM. Prima di vedere come si lancia tale eseguibile daremo un rapido sguardo alle variabili d'ambiente che influiscono sulle opzioni '`-I`' e '`-L`'.

Prestate attenzione a non mettere mai dei percorsi assoluti dei file di intestazione nei comandi `#include` nei vostri codici sorgenti in quanto ciò impedirebbe di compilare il programma in altri sistemi. l'opzione '`-I`' o la variabile INCLUDE_PATH descritte di seguito dovrebbe essere sempre impiegate per impostare il percorso di inclusione dei file di intestazione.

3.1.2. Le variabili d'ambiente

I percorsi di ricerca dei file di ricerca e delle librerie possono essere controllati nella shell anche tramite le variabili ambientali. Queste potrebbero venire impostate automaticamente in ogni sessione utilizzando il file di autenticazione (*login file*) corretto, come ad esempio '`.bash_profile`'.

Le directory supplementari possono essere aggiunte al percorso delle inclusioni utilizzando la variabile ambientale `C_INCLUDE_PATH` (per i file di intestazione del C) o

CPLUS_INCLUDE_PATH (per i file di intestazione del C++). Per esempio, i seguenti comandi aggiungeranno `/opt/gdbm-1.8.3/include' al percorso delle inclusioni quando si compilano i programmi in C:

```
$ C_INCLUDE_PATH=/opt/gdbm-1.8.3/include
$ export C_INCLUDE_PATH
```

Tale directory sarà ricercata dopo qualsiasi directory specificata nella linea di comando con l'opzione '-I' e prima delle consuete directory predefinite `/usr/local/include' e `/usr/include'. Il comando export della shell è necessario per rendere disponibile la variabile ambientale ai programmi esterni alla shell stessa, come il compilatore – serve solo una volta per ogni variabile in ciascuna sessione di shell e può essere impostato nell'appropriato file di accesso al sistema.

Allo stesso modo le directory supplementari possono essere aggiunte al percorso dei collegamenti utilizzando la variabile LIBRARY_PATH. Per esempio, i seguenti comandi aggiungeranno '/opt/gdbm-1.8.3/lib' al percorso dei collegamenti:

```
$ LIBRARY_PATH=/opt/gdbm-1.8.3/lib
$ export LIBRARY_PATH
```

Questa directory verrà ricercata dopo tutte quelle indicate nella linea di comando con l'opzione '-L' e prima delle normali directory predefinite '/usr/local/lib' e '/usr/lib'.

Con le impostazioni sopraindicate delle variabili ambientali il programma 'dbmain.c' può essere compilato senza le opzioni '-I' e '-L',

```
$ gcc -Wall dbmain.c -lgdbm
```

perché i percorsi base ora utilizzano le directory indicate nelle variabili ambientali C_INCLUDE_PATH e LIBRARY_PATH.

3.1.3. I percorsi di ricerca estesi

Seguendo la convenzione standard di Unix per i percorsi di ricerca, diverse directory possono essere specificate contemporaneamente in una variabile ambientale come elenco separato dal segno due punti (:):

```
DIR1 :DIR2 :DIR3 :...
```

Le directory vengono poi ricercate in ordine da sinistra a destra. Un unico punto '.' può essere utilizzato per indicare la directory corrente(2) .

Per esempio, le seguenti definizioni creano dei percorsi base delle inclusioni e dei collegamenti per i pacchetti installati nella directory corrente '.' e le directory 'include' e 'lib' rispettivamente sotto '/opt/gdbm-1.8.3' e '/net':

```
$ C_INCLUDE_PATH=.:/opt/gdbm-1.8.3/include:/net/include
```

(2) La directory corrente può anche essere indicata usando un elemento vuoto dei percorsi. Per esempio, *:DIR1 :DIR2* equivale a *.:DIR1:DIR2*.

```
$ LIBRARY_PATH=.:/opt/gdbm-1.8.3/lib:/net/lib
```

Per specificare molteplici directory dei percorsi di ricerca nella linea di comando, le opzioni '`-I`' e '`-L`' possono essere ripetute. Per esempio, il comando successivo,

```
$ gcc -I. -I/opt/gdbm-1.8.3/include -I/net/include
   -L. -L/opt/gdbm-1.8.3/lib -L/net/lib .....
```

equivale alle precedenti impostazioni delle variabili ambientali.

Quando variabili ambientali e opzioni a linea di comando vengono utilizzate insieme, il compilatore cerca le directory nell'ordine che segue:

1. le opzioni '`-I`' e '`-L`', da sinistra a destra
2. le directory indicate nelle variabili ambientali, come `C_INCLUDE_PATH` e `LIBRARY_PATH`
3. le directory base del sistema

Nell'uso quotidiano le directory vengono solitamente aggiunte ai percorsi di ricerca con le opzioni '`-I`' e '`-L`'.

3.2. Librerie condivise e librerie statiche

Sebbene il precedente programma d'esempio sia stato compilato e collegato con successo, è necessario compiere un'operazione finale prima di essere in grado di caricare ed avviare il file eseguibile.

Se si cerca di far partire l'eseguibile direttamente, in molti sistemi apparirà il seguente errore:

```
$ ./a.out
./a.out: error while loading shared libraries:
libgdbm.so.3: cannot open shared object file:
No such file or directory
```

Avviene ciò perché il pacchetto GDBM fornisce una *libreria condivisa* (*shared library*). Tale tipo di libreria richiede un trattamento speciale: deve essere caricata da disco prima dell'avvio dell'eseguibile.

Le librerie esterne vengono fornite normalmente in due forme: *librerie statiche* (*static libraries*) e *librerie condivise*. Le librerie statiche sono i file '`.a`' visti in precedenza. Quando un programma viene collegato ad una libreria statica, il codice macchina dai file oggetto per qualsiasi funzione esterna utilizzata dal programma viene copiato dalla libreria nell'eseguibile finale.

Le librerie condivise vengono gestite per mezzo di una forma più avanzata di collegamento che rende il file eseguibile più snello. Esse utilizzano l'estensione '`.so`', che sta per *oggetto*

condiviso (*shared object*).

Un file eseguibile collegato ad una libreria condivisa contiene solo una piccola tabella delle funzioni richieste, invece del codice macchina completo dei file oggetto delle funzioni esterne. Prima che il file eseguibile inizi a girare, il codice macchina per le funzioni esterne viene copiato in memoria dal file della libreria condivisa su disco ad opera del sistema operativo (un processo chiamato *collegamento dinamico* o *dynamic linking*).

Il collegamento dinamico rende i file eseguibili più snelli e risparmia spazio nei dischi, perché un'unica copia di una libreria può essere condivisa da più programmi. Molti sistemi operativi forniscono anche un meccanismo di memoria virtuale che consente ad una sola copia di una libreria condivisa di essere utilizzata da tutti i programmi attivi risparmiando sia memoria, sia spazio su disco.

Inoltre le librerie condivise permettono l'aggiornamento di una di esse senza la ricompilazione dei programmi che la utilizzano (purché l'interfaccia alla libreria non cambi).

A causa di questi vantaggi `gcc` compila normalmente in molti sistemi i programmi in modo da usare le librerie condivise, se disponibili. Ogni qualvolta verrà utilizzata una libreria statica '`lib`*`NOME`*`.a`' per il collegamento tramite l'opzione '`-l`*`NOME`*' il compilatore prima cercherà una libreria condivisa alternativa con il nome medesimo e l'estensione '`.so`'.

In questo caso, quando il compilatore ricercherà la libreria '`libgdbm`' nel percorso dei collegamenti, troverà i seguenti due file nella directory '`/opt/gdbm-1.8.3/lib`':

```
$ cd /opt/gdbm-1.8.3/lib
$ ls libgdbm.*
libgdbm.a libgdbm.so
```

Conseguentemente il file oggetto condiviso '`libgdbm.so`' verrà utilizzato di preferenza rispetto alla libreria statica '`libgdbm.a`'.

In ogni caso, quando si avvia un file eseguibile, la sua funzione di caricamento deve trovare la libreria condivisa per poterlo caricare in memoria. Di base il caricatore (*loader*) ricerca le librerie condivise in un insieme predefinito di directory di sistema, come '`/usr/local/lib`' e '`/usr/lib`'. Se la libreria non è posizionata in una di queste directory, deve essere aggiunta al percorso di caricamento(3).

Il modo più semplice per impostare il percorso di caricamento è attraverso la variabile ambientale `LD_LIBRARY_PATH`. Per esempio, i seguenti comandi definiscono il percorso di caricamento su '`/opt/gdbm-1.8.3/lib`' cosicché '`libgdbm.so`' può essere trovata:

```
$ LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib
$ export LD_LIBRARY_PATH
$ ./a.out
```

(3) Notate che la directory contenente la libreria condivisa può, teoricamente, essere conservata ("hard-coded") nell'eseguibile stesso utilizzando l'opzione del collegatore (*linker*) '`-rpath`', ma di solito non si fa perché ciò crea problemi se la libreria viene spostata o l'eseguibile viene copiato in un altro sistema.

```
Storing key-value pair... done.
```

L'eseguibile ora funziona con successo, stampa il suo messaggio e crea un file DBM chiamato '`test`', contenente la copia di valori-chiave '`testkey`' e '`testvalue`'

Per risparmiare battute la variabile ambientale `LD_LIBRARY_PATH` può essere definita una sola volta per ciascuna sessione nel corretto file di accesso al sistema, come ad esempio '`.bash_profile`' nella shell GNU Bash.

Le diverse directory delle librerie condivise possono essere piazzate nel percorso di caricamento, come elenco separato da due punti *`DIR1 :DIR2 :DIR3 :...:DIRN`*. Per esempio, il seguente comando configura il percorso di caricamento per l'utilizzo delle directory '`lib`' sotto '`/opt/gdbm-1.8.3`' e '`/opt/gtk-1.4`':

```
$ LD_LIBRARY_PATH=/opt/gdbm-1.8.3/lib:/opt/gtk-1.4/lib
$ export LD_LIBRARY_PATH
```

Se il percorso di caricamento contiene dati preesistenti, può essere esteso impiegando la sintassi `LD_LIBRARY_PATH=`*`NEWDIRS`*`:$LD_LIBRARY_PATH`. Per esempio, il comando successivo aggiunge la directory '`/opt/gsl-1.5/lib`' al percorso di caricamento sopraindicato:

```
$ LD_LIBRARY_PATH=/opt/gsl-1.5/lib:$LD_LIBRARY_PATH
$ echo $LD_LIBRARY_PATH
/opt/gsl-1.5/lib:/opt/gdbm-1.8.3/lib:/opt/gtk-1.4/lib
```

L'amministratore di sistema può configurare la variabile `LD_LIBRARY_PATH` per tutti gli utenti aggiungendola ad uno script d'accesso base, come '`/etc/profile`'. Nei sistemi GNU un percorso di sistema esteso può essere definito nel file di configurazione del caricatore (*loader*) '`/etc/ld.so.conf`'.

In alternativa, il collegamento statico può essere forzato con l'opzione '`-static`' di `gcc` per evitare l'uso delle librerie condivise:

```
$ gcc -Wall -static -I/opt/gdbm-1.8.3/include/
    -L/opt/gdbm-1.8.3/lib/ dbmain.c -lgdbm
```

Ciò crea un eseguibile collegato alla libreria statica '`libgdbm.a`' che funzionare senza impostare la variabile LD_LIBRARY_PATH o senza mettere le librerie condivise tra le directory predefinite:

```
$ ./a.out
Storing key-value pair... done.
```

Come notato precedentemente, è possibile collegarsi direttamente ai singoli file delle librerie specificando il percorso completo ad esse nella linea di comando. Per esempio la seguente linea di comando collegherà direttamente alla libreria statica '`libgdbm.a`',

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include
    dbmain.c /opt/gdbm-1.8.3/lib/libgdbm.a
```

ed il comando sottostante collegherà con con il file di libreria condivisa '`libgdbm.so`':

```
$ gcc -Wall -I/opt/gdbm-1.8.3/include
    dbmain.c /opt/gdbm-1.8.3/lib/libgdbm.so
```

Nel caso precedente è ancora necessario impostare il percorso di caricamento delle librerie quando si avvia l'eseguibile.

3.3. Gli standard del linguaggio C

Normalmente `gcc` compila i programmi utilizzando il dialetto GNU del linguaggio C, chiamato *GNU C*. Tale dialetto incorpora lo standard ufficiale ANSI/ISO del linguaggio C con diverse utili estensioni GNU, come le funzioni annidate e le matrici di dimensioni variabili. La maggior parte dei programmi ANSI/ISO possono essere compilati con GNU C senza modifiche.

Ci sono diverse opzioni che controllano il dialetto del C usato da `gcc`. Quelle più comunemente utilizzate sono '`-ansi`' e '`-pedantic`'. Gli specifici dialetti del linguaggio C di ogni standard possono essere selezionati tramite l'opzione '`-std`'.

3.3.1. ANSI/ISO

Di tanto in tanto un valido programma ANSI/ISO può essere incompatibile con le estensioni del GNU C. Per risolvere tale questione, l'opzione del compilatore '`-ansi`' disabilita quelle estensioni GNU che confliggono con lo standard ANSI/ISO. Nei sistemi utilizzanti la libreria GNU C (`glibc`) disabilita anche le estensioni alla libreria standard del C. Ciò consente ai programmi scritti in ANSI/ISO C di essere compilati senza effetti indesiderati derivanti dalle estensioni GNU.

Per esempi, c'è un valido programma ANSI/ISO C che utilizza una variabile chiamata `asm`:

```
#include <stdio.h>
int
main (void)
{
  const char asm[] = "6502";
  printf ("the string asm is '%s'\n", asm);
  return 0;
}
```

Il nome della variabile `asm` è valido nello standard ANSI/ISO, ma questo programma non verrà compilato in GNU C perché `asm` è una estensione di una istruzione GNU C (consente di impiegare le istruzioni native in assembly nelle funzioni in C). Di conseguenza non può essere usata come nome di variabile senza emissione di un errore di compilazione:

```
$ gcc -Wall ansi.c
```

```
ansi.c: In function `main':
ansi.c:6: parse error before `asm'
ansi.c:7: parse error before `asm'
```

Invece, utilizzando l'opzione '`-ansi`' si disabilita l'estensione dell'istruzione `asm` e si permette al programma precedente di essere compilato correttamente:

```
$ gcc -Wall -ansi ansi.c
$ ./a.out
the string asm is '6502'
```

Come riferimento, le istruzioni e le macro non standard definite nelle estensioni `GNU C` sono `asm`, `inline`, `typeof`, `unix` e `vax`. Maggiori dettagli si possono trovare nel manuale di riferimento di `GCC` "*Using GCC*" (v. "Ulteriori letture", pagina 93).

Il prossimo esempio mostra l'effetto dell'opzione '-ansi' nei sistemi che usano la libreria `GNU C`, come i sistemi GNU/Linux. Il sottostante programma stampa il valore di pigreco, x = `3.14159...`, dalla definizione `M_PI` del preprocessore nel file di intestazione '`math.h`':

```
#include <math.h>
#include <stdio.h>
int
main (void)
{
  printf("Il valore di pigreco corrisponde a %f\n", M_PI);
  return 0;
}
```

La costante `M_PI` non fa parte della libreria del C ANSI/ISO (deriva dalla versione BSD di Unix). In questo caso, il programma non verrà compilato con l'opzione '`-ansi`':

```
$ gcc -Wall -ansi pi.c
pi.c: In function `main':
pi.c:7: `M_PI' undeclared (first use in this function)
pi.c:7: (Each undeclared identifier is reported only once
pi.c:7: for each function it appears in.)
```

Il programma può essere compilato senza l'opzione '`-ansi`'. In tal caso sia le estensioni del linguaggio e delle librerie sono abilitate in partenza:

```
$ gcc -Wall pi.c
$ ./a.out
Il valore di pigreco corrisponde a 3.141593
```

E' pure possibile compilare il programma usando il C ANSI/ISO abilitando solo le estensioni nella libreria GNU C stessa. Questo risultato può essere ottenuto definendo delle macro speciali,

come _GNU_SOURCE, che abilita le estensioni nella libreria GNU C [4]:

```
$ gcc -Wall -ansi -D_GNU_SOURCE pi.c
$ ./a.out
Il valore di pigreco corrisponde a 3.141593
```

La Libreria GNU C fornisce un certo numero di queste macro (cosiddette *macro di prova delle funzionalità* o *feature test macros*) che consentono il controllo sul supporto delle estensioni POSIX (`_POSIX_C_SOURCE`), BSD (`_BSD_SOURCE`), SVID (`_SVID_SOURCE`), XOPEN (`_XOPEN_SOURCE`) e GNU (`_GNU_SOURCE`).

La macro `_GNU_SOURCE` abilita tutte le estensioni contemporaneamente, con quelle POSIX che assumono la precedenza sulle altre nei casi in cui entrano in conflitto. Maggiori informazioni sulle macro di prova delle funzionalità si possono trovare nel GNU C Library Reference Manual (v. "Ulteriori letture", pagina 93).

3.3.2. ANSI/ISO rigoroso

L'opzione a linea di comando '`-pedantic`' in combinazione con '`-ansi`' farà in modo che `gcc` rifiuti tutte le estensioni GNU C, non solo quelle che sono incompatibili con lo standard ANSI/ISO. Ciò vi aiuta a scrivere dei programmi portabili che seguono lo standard ANSI/ISO.

Qui c'è un programma che utilizza le matrici a dimensione variabile (*variable-size arrays*), un'estensione del GNU C. La matrice `x[n]` viene dichiarata con una lunghezza specificata dalla variabile intera `n`.

```
int
main (int argc, char *argv[])
{
  int i, n = argc;
  double x[n];
  for (i = 0; i < n; i++)
    x[i] = i;
  return 0;
}
```

Questo programma verrà compilato con '`-ansi`', affinché il supporto delle matrici a lunghezza variabile non interferisca con la compilazione di programmi ANSI/ISO validi (è un'estensione per la retrocompatibilità):

```
$ gcc -Wall -ansi gnuarray.c
```

Comunque la compilazione con '`-ansi -pedantic`' restituisce degli avvisi (*warnings*) riguardanti le violazioni dello standard ANSI/ISO:

(4) L'opzione '`-D`' per definire delle macro verrà spiegata in dettaglio nel prossimo capitolo,

```
$ gcc -Wall -ansi -pedantic gnuarray.c
gnuarray.c: In function `main':
gnuarray.c:5: warning: ISO C90 forbids variable-size
  array `x'
```

Fate attenzione che la mancanza di avvisi da '`-ansi -pedantic`' non garantisce la stretta conformità di un programma allo standard ANSI/ISO. Lo standard stesso indica solo un limitato complesso di circostanze che potrebbero produrre dei segnali diagnostici, e quest'ultimi sono ciò che emette '`-ansi -pedantic`'.

3.3.3. La selezione di standard specifici

Lo specifico standard di linguaggio usato da GCC può essere controllato con l'opzione '`-std`'. Sono supportati i seguenti standard del linguaggio C:

'`-std=c89`' o '`-std=iso9899:1990`'

Lo standard originale del linguaggio ANSI/ISO C (ISO/IEC 9899:1990). GCC incorpora le correzioni nelle due ISO Technical Corrigenda allo standard originale.

'`-std=iso9899:199409`'

Il standard del linguaggio ISO C con l'emendamento ISO 1, pubblicato nel 1994. Tale emendamento riguardò principalmente l'internazionalizzazione, tipo l'aggiunta del supporto dei caratteri multibyte alla libreria C.

'`-std=c99`' o '`-std=iso9899:1999`'

Gli standard del linguaggio ISO C, pubblicati nel 1999 (ISO/IEC 9899:1999)

Gli standard dei linguaggio C con le estensioni GNU possono essere selezionati con le opzioni '`-std=gnu89`' e '`-std=gnu99`'.

3.4. Le opzioni di avviso in `-Wall`

Come descritto in precedenza (v. Sezione 2.1 [Compilazione di un semplice programma C], pagina 11), l'opzione di avviso '`-Wall`' abilita gli avvisi per molti errori comuni e dovrebbe essere utilizzata sempre. Essa comprende un ampio numero di altre opzioni d'avviso più specifiche, che possono essere selezionate anche individualmente. Qui c'è un sommario di queste:

'`-Wcomment`' (compresa in '`-Wall`')

Questa opzione segnala dei commenti annidati. I commenti annidati spuntano normalmente quando una sezione di codice contenente commenti viene successivamente decommentata (*commented out*):

```
/* commented out
double x = 1.23 ; /* x-position */
```

```
*/
```

I commenti annidati possono essere una sorgente di confusione: il modo sicuro per decommentare una sezione di codice contenente commenti è quello di circoscriverla con la direttiva di preprocessore `#if 0 ... #endif`:

```
/* commented out */
#if 0
double x = 1.23 ; /* x-position */
#endif
```

'`-Wformat`' (compresa in '`-Wall`')

Questa opzione segnala l'uso scorretto delle stringhe di formato nelle funzioni come `printf` e `scanf`, in cui l'indicatore del formato non concorda con il tipo del corrispondente argomento della funzione.

'`-Wunused`' (compresa in '`-Wall`')

Questa opzione segnala le variabili inutilizzate. Quando una variabile viene dichiarata ma non utilizzata, ciò può dipendere da un'altra variabile che è stata sostituita accidentalmente al suo posto. Se la variabile non è realmente necessaria, può essere rimossa dal codice sorgente.

'`-Wimplicit`' (compresa in '`-Wall`')

Questa opzione segnala funzioni che sono usate senza essere dichiarate. La ragione più comune dell'uso di una funzione senza averla dichiarata è per aver scordato di includere un file di intestazione.

'`-Wreturn-type`' (compresa in '`-Wall`')

Questa opzione segnala funzioni che sono state definite senza un tipo di ritorno ma non dichiarate `void` (vuote). Intercetta pure i comandi `return` vuoti nelle funzioni che non sono state dichiarate `void`.

Per esempio il programma seguente non utilizza un valore di ritorno specifico:

```
#include <stdio.h>
int
main (void)
{
  printf ("ciao mondo\n");
  return;
}
```

L'assenza di un valore di ritorno nel codice soprastante potrebbe essere il risultato di un'omissione accidentale del programmatore (il valore restituito dalla funzione `main` è in realtà il valore di ritorno della funzione `printf` [il numero di caratteri stampati]). Per evitare ambiguità, è preferibile utilizzare un valore esplicito nell'istruzione `return`, che si tratti di una variabile o di una costante, come `return 0`.

L'elenco completo delle opzioni di avviso incluse in '`-Wall`' si può trovare nel Manuale di Riferimento di GCC "*Using GCC*" (v. [Ulteriori letture], pagina 93). Le opzione comprese in '`-Wall`' hanno la caratteristica comune di segnalare sintassi che sono sempre sbagliate o che possono essere facilmente riscritte in maniera corretta e non ambigua. E' questo il motivo per cui sono così utili: ogni avviso prodotto da '`-Wall`' può essere interpretato come indicativo di un problema potenzialmente serio.

3.5. Opzioni di avviso supplementari

GCC mette a disposizione molte altre opzioni di avviso che non sono comprese in '`-Wall`', ma che sono spesso utili. Solitamente queste generano avvisi per un codice che potrebbe tecnicamente essere corretto ma che potrebbe molto facilmente causare problemi. I principi di queste opzioni si basano sull'esperienza di errori comuni. Esse non sono comprese in '`-Wall`' perché indicano solamente un codice possibilmente fonte di problemi o "sospetto".

Dal momento che tali avvisi possono essere emessi per un codice valido, non è necessario compilare quest'ultimo con essi tutte le volte. E' più corretto utilizzarli di tanto in tanto e verificare i risultati, controllando qualcosa di inatteso, o abilitarli per alcuni programmi o file.

'`-W`'

Questa è un'opzione generica simile a '`-Wall`' che segnala una serie di comuni errori di programmazione, come funzioni che possono ritornare senza un valore (note anche come "perdita della fine del corpo di una funzione") e confronti tra valori con segno (*relativi*) e senza segno (*assoluti*). Per esempio, la funzione successiva controlla se un intero senza segno è negativo (cosa ovviamente impossibile):

```
int
foo (unsigned int x)
{
  if (x < 0)
    return 0; /* impossibile */
  else
    return 1;
}
```

Compilando questa funzione con '`-Wall`' non si ottiene un avviso,

```
$ gcc -Wall -c w.c
```

ma ne emette uno con '`-W`':

```
$ gcc -W -c w.c
w.c: In function `foo':
w.c:4: warning: comparison of unsigned
  expression < 0 is always false
```

Nella pratica le opzioni '`-W`' e '`-Wall`' vengono usate normalmente assieme.

'`-Wconversion`'

Questa opzione avvisa circa conversioni dei tipi implicite che potrebbero causare risultati inattesi. Per esempio, l'assegnazione di un valore negativo ad una variabile senza segno, come nel codice seguente,

```
unsigned int x = -1;
```

è tecnicamente consentita dallo standard ANSI/ISO (con l'intero negativo che viene convertito in intero positivo secondo la rappresentazione della macchina) ma potrebbe essere un semplice errore di programmazione. Se dovete eseguire una conversione, potete usare una impostazione esplicita, come `((unsigned int) -1)`, per evitare qualsiasi avviso proveniente da tale opzione. Nelle macchine con complemento a due il risultato della conversione di tipo (*`cast`*) restituisce il numero massimo che può essere rappresentato da un intero assoluto (*unsigned integer*).

'`-Wshadow`'

Questa opzione avvisa circa la ripetizione di un nome di variabile in un contesto (*scope*) dove è già stata dichiarata. Ciò viene definito come **oscuramento** delle variabili (*variable shadowing*) e genera confusione su quale valore corrisponda alla ricorrenza della variabile.

La funzione seguente dichiara una variabile locale `y` che oscura la dichiarazione nel corpo della funzione:

```
double
test (double x)
{
  double y = 1.0;
  {
    double y;
    y = x;
  }
  return y;
}
```

Questo è ANSI/ISO C corretto, dove il valore di ritorno è 1. L'oscuramento (*shadowing*) della

variabile y potrebbe far sembrare (ingiustamente) che il valore di ritorno sia `x`, osservando la linea `y = x` (specialmente in una funzione estesa e complicata).

L'oscuramento può verificarsi anche per i nomi delle funzioni. Per esempio, il programma seguente tenta di definire una variabile `sin` che oscura la funzione standard `sin(x)`.

```
double
sin_series (double x)
{
  /* series expansion for small x */
  double sin = x * (1.0 - x * x / 6.0);
  return sin;
}
```

Questo errore verrà rilevato dall'opzione '`-Wshadow`'.

'`-Wcast-qual`'

Questa opzione segnala i puntatori che sono stati definiti per rimuovere un attributo dei tipi (*type qualifier*), come `const`. Per esempio, la funzione seguente elimina l'attributo `const` dal proprio argomento di inserimento, consentendo di sovrascriverlo:

```
void
f (const char * str)
{
  char * s = (char *)str;
  s[0] = '\0';
}
```

La modifica dei contenuti originali di `str` è una violazione delle proprietà di `const`. L'opzione avviserà circa l'inesatta `definizione` della variabile `str` che consente la modifica della stringa.

'`-Wwrite-strings`'

Questa opzione restituisce implicitamente tutte le stringhe delle costanti definite nel attributo `const` di un programma, generando un avviso nel momento della compilazione se c'è un tentativo di sovrascriverle. Il risultato della modifica di una costante stringa non viene definita nello standard ANSI/ISO e l'utilizzo di costanti stringa scrivibili è sconsigliato nel GCC.

'`-Wtraditional`'

Questa opzione segnala le parti di codice che potrebbero essere interpretate differentemente da un compilatore ANSI/ISO e da uno "tradizionale" pre-ANSI[5]. Qualora il mantenimento

(5) La forma tradizionale del linguaggio C è stata descritta nell'originale manuale di riferimento del C "Il linguaggio di programmazione C" (Prima Edizione) di Kernighan e Ritchie

all'aderenza del software fosse necessaria per individuare se la versione tradizionale o quella ANSI/ISO era stata seguita nel codice originale in base agli avvisi generati da tale opzione.

Le opzioni precedenti generano degli avvisi diagnostici, ma permettono alla compilazione di proseguire e produrre un file oggetto o eseguibile. Nel caso di grossi programmi è preferibile l'intercettazione di tutti gli avvisi fermando la compilazione ogni qualvolta che viene emesso un avviso. L'opzione `'-Werror'` cambia il comportamento normale convertendo gli avvisi in errori, fermando la compilazione ogni volta che si verifica un avviso.

4. L'uso del preprocessore

Questo capitolo descrive l'utilizzo del preprocessore `cpp` del GNU C, che fa parte del pacchetto GCC. Il preprocessore espande le macro nei file sorgenti prima che vengano compilati. Esso viene invocato ogni volta che GCC elabora un programma C o C++[1].

4.1. Definire delle macro

Il programma seguente dimostra l'uso più comune del preprocessore del C. Esso utilizza `#ifdef` condizionale del preprocessore per controllare se è stata definita una macro.

Quando è stata definita una macro, il preprocessore include il codice corrispondente sino al comando di chiusura `#endif`. In questo esempio la macro che viene controllata si chiama `TEST` e la parte condizionale del codice sorgente è un'istruzione `printf` che stampa il messaggio "`Test mode`":

```
#include <stdio.h>
int
main (void)
{
#ifdef TEST
  printf ("Test mode\n");
#endif
  printf ("Running...\n");
  return 0;
}
```

L'opzione di `gcc` '`-DNOME`' definisce una macro `NOME` del preprocessore da linea di comando. Se il programma precedente viene compilato con l'opzione a linea di comando '`-DTEST`', la macro `TEST` verrà definita e il relativo eseguibile stamperà entrambi i messaggi:

```
$ gcc -Wall -DTEST dtest.c
$ ./a.out
```

(1) Nelle versioni recenti di GCC il preprocessore è integrato nel compilatore, sebbene sia fornito anche un comando separato `cpp`.

```
Test mode
Running...
```

Se lo stesso programma viene compilato senza l'opzione '`-D`', allora il messaggio "`Test mode`" viene omesso dal codice sorgente dopo il preprocessamento e l'eseguibile finale non include il codice attinente:

```
$ gcc -Wall dtest.c
$ ./a.out
Running...
```

Le macro in genere non sono definite, a meno che non vengano specificate nella linea di comando con l'opzione '`-D`' o in un file sorgente (o file di intestazione di librerie) con `#define`. Alcune macro vengono definite automaticamente dal compilatore; queste usano normalmente uno spazio dei nomi (*namespace*) riservato, iniziante con un prefisso di due tratti di sottolineatura '`__`'. L'insieme completo delle macro predefinite può essere elencato avviando il preprocessore GNU `cpp` con l'opzione '`-dM`' in un file vuoto:

```
$ cpp -dM /dev/null
#define __i386__ 1
#define __i386 1
#define i386 1
#define __unix 1
#define __unix__ 1
#define __ELF__ 1
#define unix 1
.......
```

Notate che questa lista comprende un esiguo numero di macro dello specifico sistema e definite da `gcc` senza ricorrere al prefisso del doppio sottolineato. Queste macro non standard possono essere disabilitate con l'opzione '`-ansi`' di `gcc`.

4.2. Macro con valori

In aggiunta all'essere definita, una macro po' essere attribuito un valore concreto. Tale valore viene inserito nel codice sorgente in ogni posto dove compare la macro. Il programma seguente usa una macro `NUM` per rappresentare un numero da stampare:

```
#include <stdio.h>
int
main (void)
{
  printf("NUM vale %d\n", NUM);
```

```
  return 0;
}
```

Osservate che le macro non vengono espanse all'interno delle stringhe: solo la ricorrenza di `NUM` fuori della stringa viene sostituita dal preprocessore.

Per definire una macro con un valore, l'opzione a linea di comando '`-D`' può essere usata nella forma '`-D`*`NOME=VALORE`*'. Per esempio, la seguente linea di comando definisce `NUM` con il valore di `100` quando si compila il programma qui sotto:

```
$ gcc -Wall -DNUM=100 dtestval.c
$ ./a.out
NUM vale 100
```

Questo esempio utilizza un numero, ma una macro può assumere dei valori in ogni forma. Qualsiasi sia il valore di una macro, esso viene inserito direttamente nel codice sorgente nel punto in cui si presenta il nome della macro. Per esempio, la seguente definizione espande le chiamate a `NUM` a `2+2` durante il preprocessamento:

```
$ gcc -Wall -DNUM="2+2" dtestval.c
$ ./a.out
NUM vale 4
```

Dopo che il preprocessore ha eseguito la sostituzione NUM ↦ 2+2, ciò equivale a compilare il seguente programma:

```
#include <stdio.h>
int
main (void)
{
  printf("NUM vale %d\n", 2+2);
  return 0;
}
```

Notate che è buona idea circondare le macro con delle parentesi ogni volta che fanno parte di un'espressione. Per esempio, il programma seguente utilizza delle parentesi per garantire le precedenze corrette alla moltiplicazione `10*NUM`:

```
#include <stdio.h>
int
main (void)
{
  printf ("Dieci volte NUM vale %d\n", 10 * (NUM));
  return 0;
}
```

Con tali parentesi si ottengono i risultati attesi dopo aver compilato con la stessa linea di comando come segue:

```
$ gcc -Wall -DNUM="2+2" dtestmul10.c
$ ./a.out
Dieci volte NUM vale 40
```

Senza parentesi, il programma avrebbe generato il valore 22 basandosi sulla formula letterale dell'espressione `10*2+2=22`, in luogo del desiderato valore `10*(2+2)=40`.

Quando una macro viene definita con '`-D`' soltanto, `gcc` usa un valore predefinito pari a `1`. Per esempio, compilando l'originale file di prova con l'opzione '`-DNUM`' si genera un eseguibile che produce i seguenti risultati;

```
$ gcc -Wall -DNUM dtestval.c
$ ./a.out
NUM vale 1
```

Una macro può essere definita con un valore nullo usando le virgolette nella linea di comando, `-DNOME=""`. Tale macro viene ancora trattata come *definita* dalle direttive condizionali tipo `#ifdef`, ma non espande nulla.

Una macro contenente virgolette può essere definita tramite i caratteri delle virgolette munite di ESC [\] della shell. Per esempio, l'opzione a linea di comando `-DMESSAGGIO="\"Ciao, Mondo!\""` definisce una macro `MESSAGGIO` che si estende alla sequenza di caratteri "`Ciao, Mondo!`". Per una spiegazione sui tipi diversi di virgolettature e di caratteri ESC utilizzati nella shell date un occhiata al "*GNU Bash Reference Manual*", [Ulteriori letture], pagina 93.

4.3. Preprocessamento dei file sorgenti

E' possibile riscontrare direttamente l'effetto del preprocessore sui file sorgenti usando l'opzione '`-E`' di `gcc`. Per esempio il file successivo definisce ed usa una macro `TEST`:

```
#define TEST "Ciao, Mondo!"
const char str[] = TEST;
```

Se il file si chiama '`test.c`', l'effetto del preprocessore può essere visto con la seguente linea di comando:

```
$ gcc -E test.c
# 1 "test.c"


const char str[] = "Ciao, Mondo!" ;
```

L'opzione '`-E`' costringe `gcc` ad avviare il preprocessore, a mostrare i risultati espansi ed infine a terminare senza compilazione del codice sorgente. Il valore della macro TEST viene sostituito

direttamente all'uscita, producendo la sequenza di caratteri `const char str[] = "Ciao, Mondo!" ;`.

Il preprocessore inserisce pure linee che registrano il file sorgente ed i numeri di linea nella forma *`# numero-linea "file-sorgente"`*, per aiutare nella fase di ricerca degli errori e permette al compilatore di inviare messaggi d'errore che si riferiscono a questa informazione. Tali linee non influenzano il programma stesso.

L'abilità di esaminare i file sorgenti preprocessati può essere utile per esaminare gli effetti dei file d'intestazione di sistema e per trovare le dichiarazioni delle funzioni. Il programma seguente include il file di intestazione "stdio.h" per ottenere la dichiarazione della funzione `printf`:

```
#include <stdio.h>
int
main (void)
{
  printf ("Ciao, Mondo!\n");
  return 0;
}
```

E' possibile vedere le dichiarazioni dell'incluso file di intestazione preprocessando il file con `gcc -E`:

```
$ gcc -E ciao.c
```

In un sistema GNU ciò produce un risultato simile al seguente:

```
# 1 "ciao.c"
# 1 "/usr/include/stdio.h" 1 3

extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;

extern int fprintf (FILE * __stream,
                    const char * __format, ...) ;
extern int printf (const char * __format, ...) ;

  [ ... additional declarations ... ]

# 1 "ciao.c" 2

int
```

```
main (void)
{
   printf ("Ciao, Mondo!\n");
   return 0;
}
```

I file di intestazione di sistema preprocessati normalmente generano dati in quantità. Questi possono essere essere dirottati verso un file oppure salvati più opportunamente usando l'opzione di `gcc` '`-save-temps`':

```
$ gcc -c -save-temps ciao.c
```

Dopo aver dato questo comando, i dati in uscita del preprocessamento saranno disponibili nel file '`ciao.i`'. L'opzione '`-save-temps`' salva pure i file '`.s`' dell'assembly e i file oggetto '`.o`' oltre ai file '`.i`' del preprocessamento.

5. Compilare per scoprire gli errori

Normalmente un file eseguibile non contiene alcun riferimento al codice sorgente del programma originale, come ad esempio nomi di variabile o numeri di riga: il file eseguibile è semplicemente la sequenza di istruzioni in codice macchina prodotte dal compilatore. Ciò è insufficiente per la caccia agli errori dal momento che non esiste un modo semplice per trovare la causa di un errore se il programma si impianta.

GCC mette a disposizione l'opzione di debug '`-g`' per conservare delle informazioni di ricerca degli errori aggiuntive nei file oggetto ed eseguibili. Queste informazioni di individuazione degli errori permettono di essere ritrovate da una specifica istruzione della macchina ad una corrispondente linea nel file sorgente originale. Essa permette pure l'esecuzione di un programma in modo che sia tracciato in un debugger, come il debugger GNU `gdb` (per maggiori informazioni, v. "*Debugging with GDB:The GNU Source-Level Debugger*", [Ulteriori letture], pagina 93). L'uso di un debugger permette anche di esaminare i valori delle variabili mentre sta girando il programma.

L'opzione di debug funziona conservando i nomi delle funzioni e delle variabili (e tutti i riferimenti ad esse), con i loro corrispondenti numeri di linea del codice sorgente, in una *tabella dei simboli* in file oggetto ed eseguibili.

5.1. Esame dei file core

In aggiunta alla possibilità di far girare un programma con il debugger, un'altra utile applicazione dell'opzione '`-g`' è quella di scoprire le circostanze per cui un programma si blocca.

Quando un programma si chiude in modo anormale il sistema operativo può scrivere un *core file*, normalmente chiamato 'core', che contiene lo stato della memoria del programma al momento del blocco. Combinato con le informazioni derivanti dalla tabella dei simboli prodotta dall'opzione '`-g`', il file *core* può essere utilizzato per trovare la linea dove il programma si è fermato ed i valori delle sue variabili in quel punto.

Ciò è utile sia durante lo sviluppo del software, sia durante la messa in funzione di quest'ultimo: permette infatti di indagare sui problemi nel momento in cui un programma si è bloccato "sul campo".

Qui c'è un semplice programma contenente un errore di invalido accesso alla memoria , che noi useremo per generare un file *core*:

```
int a (int *p);

int
main (void)
{
   int *p = 0;   /* null pointer */
  return a (p);
}

int
a (int *p)
{
  int y = *p;
  return y;
}
```

Il programma cerca di dereferenziare un puntatore nullo `p`, che costituisce una operazione non consentita. Nella maggioranza dei sistemi ciò determina un blocco(1).

Per essere più tardi in grado di scoprire la causa del blocco, dobbiamo compilare il programma con l'opzione '`-g`':

```
$ gcc -Wall -g null.c
```

Notate che un puntatore nullo causerà un problema solo durante l'esecuzione, cosicché l'opzione '`-Wall`' non produce alcun avvertimento.

Far funzionare il file eseguibile in un sistema x86 GNU/Linux costringerà il sistema operativo a terminare il programma in modo anormale:

```
$ ./a.out
Segmentation fault (core dumped)
```

Ogni volta che appare il messaggio '`core dumped`', il sistema operativo dovrebbe produrre un file chiamato '`core`' nella directory corrente(2). Questo file *core* contiene una copia completa delle pagine di memoria usate dal programma al momento in cui è stato terminato. Incidentalmente, il termine *segmentation fault* (errore di segmentazione) si riferisce al fatto che il programma ha tentato di accedere ad un "segmento" di memoria riservato al di fuori dell'area di memoria che era stata allocata ad esso.

Alcuni sistemi sono configurati in partenza in modo da non scrivere i file *core*, dal momento che i file potrebbero essere grandi e potrebbero riempire rapidamente o spazio disponibile su disco in un

(1) Storicamente, un puntatore nullo tipicamente corrisponde alla locazione di memoria 0, che normalmente è riservata al kernel del sistema operativo e non è accessibile da parte dei programmi degli utenti.

(2) Alcuni sistemi, come FreeBSD e Solaris, possono essere anche configurati per scrivere file *core* in specifiche directory, ad es. '`/var/coredumps/`', utilizzando i comandi `sysctl` o `coreadm`.

sistema. Nella shell *GNU Bash* il comando `ulimit -c` controlla la dimensione massima dei file *core*. Se il limite dimensionale è zero, non vengono generati file *core*. Il limite dimensionale corrente può essere mostrato battendo il comando seguente:

```
$ ulimit -c
0
```

Se il risultato è 0, come apparso sopra, allora può essere aumentato con il seguente comando per permettere la scrittura di file *core* di qualsiasi dimensione(3):

```
$ ulimit -c unlimited
```

Osservate che questa impostazione si applica solo alla shell corrente. Per definire il limite per sessioni future il comando dovrebbe essere collocato in un appropriato file di login, come '`.bash_profile`' nella shell GNU Bash.

I file *core* possono essere caricati nel GNU Debugger `gdb` con il comando seguente:

```
$ gdb EXECUTABLE-FILE CORE-FILE
```

Notate che per l'eliminazione degli errori (*debugging*) sono richiesti sia l'eseguibile originale, sia il file *core*: è impossibile, infatti, correggere un file *core* senza l'eseguibile corrispondente. In questo esempio, noi possiamo caricare l'eseguibile ed il file *core* con il comando:

```
$ gdb a.out core
```

Il programma di correzione (*debugger*) immediatamente inizia a stampare le informazioni di diagnostica e mostra un listato della linea dove il programma si è impiantato (linea 13):

```
$ gdb a.out core
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x080483ed in a (p=0x0) at null.c:13
13        int y = *p;
(gdb)
```

La linea finale `(gdb)` è il prompt del GNU Debugger ed indica che a questo punto possono essere dati ulteriori comandi.

Per indagare sulla causa del blocco, evidenziamo il valore del puntatore `p` utilizzando il comando del debugger `print`:

(3) Questo esempio usa il comando `ulimit` della shell GNU Bash. In altri sistemi l'utilizzo del comando `ulimit` può variare oppure avere un nome diverso (la shell `tcsh` usa invece il comando `limit`). Il limite dimensionale dei file *core* può essere anche definito con uno specifico valore in kilobyte.

```
(gdb) print p
$1 = (int *) 0x0
```

Ciò mostra che `p` è un puntatore nullo (`0x0`) del tipo '`int *`', cosicché noi sappiamo che dereferenziandolo con l'espressione `*p` ha causato il blocco in questa linea.

5.2. Mostrare un backtrace

Il debugger può anche mostrare le chiamate di funzioni e gli argomenti fino all'attuale punto di esecuzione: ciò si chiama *stack backtrace* e viene mostrato con il comando `backtrace`:

```
(gdb) backtrace
#0 0x080483ed in a (p=0x0) at null.c:13
#1 0x080483d9 in main () at null.c:7
```

In questo caso il backtrace mostra che il blocco alla linea 13 è capitato quando la funzione `a()` era stata chiamata con un argomento di `p=0x0` dalla linea 7 nel `main()`. E' possibile spostarsi in livelli diversi nel tracciamento dello stack ed esaminare le loro variabili, usando i comandi del debugger su e giù.

Una descrizione completa di tutti i comandi disponibili nel gdb si può trovare nel manuale "*Debugging with GDB: The GNU Source-Level Debugger*" (v. [Ulteriori letture], pagina 93).

6. Compilare con l'ottimizzazione

GCC è un compilatore *ottimizzante*. Mette a disposizione un'ampia gamma di opzioni che mirano ad accrescere la velocità, o a ridurre le dimensioni, dei file eseguibili che produce.

L'ottimizzazione è un processo complesso. Per ciascun comando ad alto livello nel codice sorgente ci sono solitamente moltissime combinazioni di istruzioni macchina che possono essere utilizzate per raggiungere il corretto risultato finale. Il compilatore deve considerare queste possibilità e scegliere tra loro.

In genere codice differente deve essere generato da processori differenti, dal momento che usano linguaggi assembly e macchina incompatibili. Ogni tipo di processore ha anche le sue peculiari caratteristiche: alcune CPU offrono un ampio numero di *registri* per conservare i risultati provvisori dei calcoli, mentre altri devono registrare e prelevare i risultati intermedi dalla memoria. In ogni caso deve essere prodotto un codice corretto.

Inoltre, sono necessarie differenti quantità di tempo per istruzioni differenti in base a come queste sono ordinate. GCC considera tutti questi fattori e tenta di produrre l'eseguibile più veloce per un dato sistema mentre compila con l'ottimizzazione.

6.1. Ottimizzazione a livello di sorgente

La prima forma di ottimizzazione usata da GCC avviene a livello del codice sorgente e non richiede alcuna conoscenza delle istruzioni macchina. Esistono svariate tecniche di ottimizzazione a livello sorgente. Questa sezione ne descrive due tipi comuni: la *common subexpression elimination* (*eliminazione delle subespressioni comuni*) e la *function inlining* (*messa in linea delle funzioni*).

6.1.1. Common subexpression elimination

Un metodo di ottimizzazione a livello sorgente che è semplice da comprendere implica l'elaborazione di una espressione nel codice sorgente con minori istruzioni, riutilizzando risultati già calcolati. Per esempio, la seguente assegnazione:

```
x = cos(v)*(1+sin(u/2)) + sin(w)*(1-sin(u/2))
```

può essere riscritta con una variabile temporanea `t` per eliminare una valutazione extra non necessaria del termine `sin(u/2)`:

```
t = sin(u/2)
x = cos(v)*(1+t) + sin(w)*(1-t)
```

Questa riscrittura è detta *common subexpression elimination* (CSE o *eliminazione delle subespressioni comuni*) e viene eseguita automaticamente quando l'ottimizzazione è attivata[1].

I valori temporanei introdotti dal compilatore durante la *common subexpression elimination* vengono utilizzati solo internamente e non inficiano le reali variabili. Il nome della variabile temporanea '`t`' mostrata sopra è usato soltanto come esempio.

L'*eliminazione delle subespressioni comuni* è potente poiché contemporaneamente aumenta la velocità e riduce la grandezza del codice.

6.1.2. Function inlining

Un altro tipo di ottimizzazione a livello dei sorgenti, detta *function inlining* o *messa in linea*, aumenta l'efficienza delle funzioni chiamate frequentemente.

Ogni volta che viene usata una funzione, una certa quantità di tempo extra è richiesta dalla CPU per tirare fuori la chiamata: essa deve memorizzare gli argomenti della funzione nei registri e locazioni di memoria corretti, saltare all'inizio della funzione (portando le appropriate pagine di memoria virtuale nella memoria fisica o nella cache della CPU, se necessario), iniziare l'esecuzione del codice e poi ritornare al punto originale dell'esecuzione quando la chiamata di funzione è completa. Questo lavoro aggiuntivo viene denominato *function-call overhead* (sovraccarico della chiamata di funzione). La *messa in linea delle funzioni* elimina questo sovraccarico rimpiazzando le chiamate ad una funzione con il codice della funzione stessa (noto come collocamento del codice *in linea*). In molti casi il sovraccarico di chiamate di funzioni è una trascurabile frazione del tempo totale di esecuzione di un programma. Esso può divenire significante solo quando ci sono delle funzioni contenenti relativamente poche istruzioni e tali funzioni contano per una sostanziosa frazione del tempo di esecuzione: in tal caso il sovraccarico diviene poi una consistente proporzione del tempo totale di esecuzione.

L'*inlining* è sempre vantaggioso se c'è un solo punto di chiamata ad una funzione. Senza dubbio è anche meglio se l'invocazione di una funzione richiede più istruzioni (memoria) che spostare il corpo della funzione in linea. Questa è una situazione comune per semplici funzioni accessori in C++, che possono trarre grossi benefici dalla messa in linea. Per di più l'*inlining* può ulteriormente facilitare l'ottimizzazione, come la *common subexpression elimination*, fondendo diverse funzioni separate in una sola grande funzione.

La seguente funzione `sq(x)` è un tipico esempio di una funzione che potrebbe trarre beneficio dall'essere allineata. Essa calcola x^2, il quadrato del suo argomento x:

```
double
sq (double x)
{
```

(1) I valori temporanei introdotti dal compilatore durante la *common subexpression elimination* vengono utilizzati solo internamente e non inficiano le reali variabili. Il nome della variabile temporanea '`t`' mostrata sopra è usato soltanto come esempio.

```
  return x * x;
}
```

Questa funzione è piccola, cosicché il sovraccarico di chiamate è paragonabile con il tempo impiegato ad eseguire la singola moltiplicazione riportata dalla funzione stessa. Se la funzione viene utilizzata all'interno di un ciclo, come in quello sottostante, allora il sovraccarico delle chiamate di funzione potrebbe divenire sostanziale:

```
for (i = 0; i < 1000000; i++)
  {
     sum += sq (i + 0.5);
  }
```

L'ottimizzazione con l'*inlining* sostituisce il ciclo interno del programma con il corpo della funzione, dando luogo al seguente codice:

```
for (i = 0; i < 1000000; i++)
  {
    double t = (i + 0.5); /* temporary variable */
    sum += t * t;
  }
```

L'eliminazione della chiamata di funzione e l'esecuzione della moltiplicazione in linea permette al ciclo di funzionare con la massima efficienza.

GCC sceglie le funzioni per la messa in linea utilizzando un numero di euristica, come la funzione che è breve al punto giusto. Trattandosi di un'ottimizzazione, l'*inlining* viene eseguita solo con ciascun file oggetto. La parola-chiave `inline` può essere usata per richiedere esplicitamente che una specifica opzione possa essere messa in linea ogni volta che è possibile, comprendendo il suo uso in altri file(2). Il manuale di riferimento di GCC "*Using GC*"' fornisce i dettagli completi della parola-chiave `inline` e del suo uso con i qualificatori `static` ed `extern` per controllare il collegamento delle funzioni messe esplicitamente in linea (v. [Ulteriori letture], pagina 93).

6.2. Compromessi velocità-spazio

Mentre alcune forme di ottimizzazione, come la *common subexpression elimination*, sono in grado contemporaneamente di aumentare la velocità e ridurre la dimensione di un programma, altri tipi di ottimizzazione producono un codice più veloce al prezzo di un aumento di dimensioni dell'eseguibile. Tale scelta tra velocità e memoria è indicata come *compromesso tra velocità e spazio* (*speed-space tradeoff*). Le ottimizzazioni con un compromesso tra velocità e spazio possono essere usate anche per rendere più snello un eseguibile al costo di farlo girare più lentamente.

(2) In tal caso, la definizione della funzione in linea deve essere resa disponibile agli altri file (in un file d'intestazione, per esempio).

6.2.1. Lo srotolamento del ciclo

Un primo esempio di ottimizzazione con compromesso velocità-spazio è lo *srotolamento del ciclo* (*loop unrollling*). Questa forma di ottimizzazione incrementa la velocità dei cicli eliminando la condizione "fine del ciclo" (*end of loop*) in ciascuna iterazione. Per esempio, il ciclo seguente da `0` a `7` testa la condizione `i < 8` in ciascuna iterazione:

```
for (i = 0; i < 8; i++)
  {
    y[i] = i;
  }
```

Alla fine del ciclo questa verifica sarà stata effettuata 9 volte e una grossa frazione del tempo di esecuzione sarà stata spesa testando la condizione. Un modo più efficiente per scrive e lo stesso codice è semplicemente quello di sviluppare il ciclo (*unroll the loop*) ed eseguire le assegnazioni direttamente:

```
y[0] = 0;
y[1] = 1;
y[2] = 2;
y[3] = 3;
y[4] = 4;
y[5] = 5;
y[6] = 6;
y[7] = 7;
```

Questa forma di codice non richiede alcuna verifica e si esegue alla massima velocità. Poiché ciascuna assegnazione è indipendente, ciò consente al compilatore di utilizzare il calcolo parallelo nei processori che lo supportano. Lo sviluppo del ciclo è una ottimizzazione che incrementa la velocità dell'eseguibile risultante, ma generalmente accresce anche le sue dimensioni (a meno che il ciclo sia molto breve, con una o due iterazioni, per esempio).

Lo sviluppo del ciclo è pure possibile quando il limite superiore del ciclo è sconosciuto, purché le condizioni d'inizio e fine siano gestite correttamente. Per esempio, lo stesso ciclo con un limite superiore arbitrario,

```
for (i = 0; i < n; i++)
  {
    y[i] = i;
  }
```

può essere riscritto dal compilatore come segue:

```
for (i = 0; i < (n % 2); i++)
  {
```

```
      y[i] = i;
    }
  for ( ; i + 1 < n; i += 2) /* no initializer */
    {
      y[i] = i;
      y[i+1] = i+1;
    }
```

Il primo ciclo gestisce il caso `i = 0` dove n è dispari ed il secondo ciclo tratta le rimanenti iterazioni. Notate che il secondo ciclo non usa un inizializzatore (*initializer*) nel primo argomento dell'istruzione `for`, dal momento che continua dove finisce il primo ciclo. Le assegnazioni nel secondo ciclo possono essere rese parallele ed il numero totale delle verifiche viene ridotto di un fattore 2 (approssimativamente). Fattori più alti si possono raggiungere srotolando di più assegnazioni dentro il ciclo al costo di una dimensione maggiore del codice.

6.3. La pianificazione

Il livello più basso di ottimizzazione è la *pianificazione* o *scheduling*, in cui il compilatore stabilisce l'ordine migliore delle singole istruzioni. La maggioranza delle CPU permettono l'avvio dell'esecuzione di una o più nuove istruzioni prima che siano terminate le altre. Molte CPU supportano anche il *pipelining*, in cui molteplici istruzioni vengono eseguite in parallelo sulla stessa CPU.

Quando è abilitata la pianificazione, le istruzioni devono essere sistemate in modo che i loro risultati divengano disponibili per le successive istruzioni al momento giusto, e per permettere la massima esecuzione in parallelo. La pianificazione migliora la velocità di un eseguibile senza incrementarne le dimensioni, ma richiede memoria aggiuntiva e tempo nel processo stesso di compilazione (a seguito della sua complessità).

6.4. Livelli di ottimizzazione

Per controllare il tempo di compilazione, l'utilizzo della memoria da parte del compilatore e i compromessi tra velocità e spazio del risultante eseguibile, GCC offre un insieme di livelli generali di ottimizzazione, numerati da `0` a `3`, così come le opzioni individuali per specifici tipi di ottimizzazione.

Il livello di ottimizzazione si sceglie con l'opzione a linea di comando '`-O`*`LIVELLO`*', dove *`LIVELLO`* è un numero da `0` a `3`. Gli effetti dei differenti livelli di ottimizzazione sono descritti sotto:

'`-O0`' o nessuna opzione '`-O`' (di base)

In questo livello di ottimizzazione GCC non effettua nessuna ottimizzazione e compila il codice sorgente nel modo più diretto possibile. Ciascun comando nel codice sorgente viene convertito

direttamente nelle corrispondenti istruzioni nel file eseguibile, senza ritocchi. Questa è la migliore opzione da utilizzare quando si correggono gli errori di un programma.

L'opzione '`-O0`' equivale a non specificare nessuna opzione '`-O`'.

'`-O1`' o '`-O`'

Questo livello attiva le forme più comuni di ottimizzazione che non richiede nessun compromesso velocità-spazio. Con tale opzione gli eseguibili risultanti dovrebbero essere più piccoli e veloci rispetto a quelli con '`-O0`'. Le opzioni più onerose, come la pianificazione delle istruzioni, non vengono utilizzate in questo livello.

La compilazione con l'opzione '`-O1`' può spesso occupare meno tempo della compilazione con '`-O0`', a causa delle ridotte quantità di dati che devono essere elaborate dopo le semplici ottimizzazioni.

'`-O2`'

Questa opzione attiva ulteriori ottimizzazioni in aggiunta a quelle usate da '`-O1`'. Tali ottimizzazioni aggiuntive comprendono la pianificazione delle istruzioni. Vengono usate soltanto le ottimizzazioni che non richiedono compromessi velocità-spazio e, quindi, l'eseguibile non dovrebbe aumentare di dimensioni. Il compilatore ci metterà di più per compilare i programmi e richiederà più memoria rispetto a '`-O1`'. Questa opzione è in genere la scelta migliore per dispiegare un programma, perché fornisce la massima ottimizzazione senza aumentare le dimensioni dell'eseguibile. E' il normale livello di ottimizzazione per i rilasci dei pacchetti GNU.

'`-O3`'

Questa opzione attiva ottimizzazioni più onerose, come la messa in linea della funzione (*function inlining*), in aggiunta a tutte le ottimizzazioni dei livelli inferiori '`-O1`' e '`-O2`'. Il livello do ottimizzazione '`-O3`' può aumentare la velocità dell'eseguibile risultante, ma può anche aumentarne le dimensioni. Sotto determinate circostanze in cui queste ottimizzazioni non sono convenienti, tale opzione potrebbe realmente rendere più lento un programma.

'`-funroll-loops`'

Questa opzione attiva il *loop unrolling*, lo srotolamento dei cicli, ed è indipendente dalle altre opzioni di ottimizzazione. Incrementerà la dimensione dell'eseguibile. Se questa opzione produce un risultato vantaggioso o meno, va valutato caso per caso.

'`-Os`'

Questa opzione seleziona le ottimizzazioni che riducono le dimensioni di un eseguibile. L'obbiettivo di tale opzione è quello di produrre il più piccolo eseguibile possibile per sistemi limitati nella memoria o nello spazio su disco. In alcuni casi un eseguibile più snello girerà anche più velocemente a causa del miglior utilizzo della cache.

E' importante ricordare che il beneficio dell'ottimizzazione ai livelli più alti deve essere

soppesato con i costi. I costi dell'ottimizzazione comprendono una maggiore complessità nella correzione degli errori e un aumento delle richieste di tempo e memoria durante la compilazione. Per molti scopi è sufficiente usare '`-O0`' per la correzione degli errori e '`-O2`' per lo sviluppo ed impiego.

6.5. Esempi

Il seguente programma verrà utilizzato per mostra gli effetti dei diversi livelli di ottimizzazione:

```
#include <stdio.h>

double
powern (double d, unsigned n)
{
  double x = 1.0;
  unsigned j;

  for (j = 1; j <= n; j++)
    x *= d;

  return x;
}

int
main (void)
{
  double sum = 0.0;
  unsigned i;

  for (i = 1; i <= 100000000; i++)
    {
       sum += powern (i, i % 5);
    }

  printf ("sum = %g\n", sum);
  return 0;
}
```

Il programma `main` contiene un ciclo che chiama la funzione `powern`. Tale funzione calcola la potenza n-esima di un numero a virgola mobile attraverso ripetute moltiplicazioni (è stato scelto perché è adatto sia alla messa in linea che per lo srotolamento dei cicli). Il tempo di esecuzione di un programma può essere misurato usando il comando `time` nella shell GNU Bash.

Qui ci sono alcuni risultati del programma soprastante, compilato con un Intel Celeron a 566 MHz con cache di primo livello da 16 KB e una cache di secondo livello da 128 KB, usando GCC 3.3.1 in un sistema GNU/Linux:

```
$ gcc -Wall -O0 test.c -lm
$ time ./a.out
real    0m13.388s
user    0m13.370s
sys     0m0.010s

$ gcc -Wall -O1 test.c -lm
$ time ./a.out
real    0m10.030s
user    0m10.030s
sys     0m0.000s

$ gcc -Wall -O2 test.c -lm
$ time ./a.out
real    0m8.388s
user    0m8.380s
sys     0m0.000s

$ gcc -Wall -O3 test.c -lm
$ time ./a.out
real    0m6.742s
user    0m6.730s
sys     0m0.000s

$ gcc -Wall -O3 -funroll-loops test.c -lm
$ time ./a.out
real    0m5.412s
user    0m5.390s
sys     0m0.000s
```

Il dato rilevante dei risultati per confrontare la velocità degli eseguibili risultanti è il tempo '`user`' che restituisce il tempo reale della CPU speso nel funzionamento del processo. Le altre righe, '`real`' e '`sys`', registrano il reale tempo totale di funzionamento del processo (compresi i tempi in cui altri processi stavano utilizzando la CPU) ed il tempo speso in attesa di chiamate del sistema operativo. Sebbene sopra venga mostrato per ogni caso un solo funzionamento, i programmi di verifica delle prestazioni (*benchmark*) sono stati eseguiti diverse volte per confermare i risultati.

Dai risultati si può vedere che aumentando i livelli di ottimizzazione con '`-O1`', '`-O2`' e '`-O3`' si determina un incremento di velocità rispetto al codice non ottimizzato, compilato con '`-O0`'. L'opzione aggiuntiva '`-funroll-loops`' produce un'ulteriore accelerazione. La velocità del programma complessivamente è più che raddoppiata, passando dal codice non ottimizzato al massimo livello di ottimizzazione. Notate che per un piccolo programma come questo qui possono esistere considerevoli variazioni tra sistemi e versioni dei compilatori. Per esempio, in un sistema Intel Pentium 4M Mobile a 2.0 GHz la tendenza dei risultati utilizzando la stessa versione di GCC è simile eccetto che le prestazioni con '`-O2`' sono leggermente peggiori rispetto a quelle con '`-O1`'. Questo dimostra un punto importante: non necessariamente l'ottimizzazione può rendere in ogni caso il programma più veloce.

6.6. L'ottimizzazione e la ricerca degli errori

Con GCC è possibile usare l'ottimizzazione in combinazione con l'opzione per la ricerca degli errori '`-g`'. Molti altri compilatori non permettono ciò.

Quando si utilizzano insieme il *debugging* e l'ottimizzazione, la riorganizzazione interna estrapolata dall'ottimizzatore può rendere difficoltoso vedere come sta funzionando quando si esamina un programma ottimizzato. Per esempio, le variabili temporanee vengono spesso eliminate e l'ordine delle istruzioni può variare.

Comunque, quando un programma si blocca in modo inatteso, qualsiasi informazione per la caccia agli errori è meglio di nessuna, cosicché l'uso dell'opzione '`-g`' è raccomandato per i programmi ottimizzati, sia nello sviluppo che nell'impiego. L'opzione '`-g`' per la caccia agli errori viene normalmente abilitata per i rilasci dei pacchetti GNU, insieme con l'opzione di ottimizzazione '`-O2`'.

6.7. L'ottimizzazione e gli avvisi del compilatore

Quando l'ottimizzazione è attiva, GCC può produrre degli avvisi aggiuntivi che non appaiono quando si compila senza l'ottimizzazione.

Come parte del processo di ottimizzazione, il compilatore esamina l'uso di tutte le variabili ed i loro valori iniziali (ciò viene definito *analisi del flusso dei dati* o *data-flow analysis*). Esso forma la base per altre strategie di ottimizzazione, come la pianificazione delle istruzioni. Un effetto collaterale dell'analisi dei flussi di dati è che il compilatore può scoprire l'utilizzo di variabili non inizializzate.

L'opzione '`-Wuninitialized`' (che è compresa in '`-Wall`') segnala le variabili che vengono lette senza essere state inizializzate. Funziona solo quando il programma viene compilato con l'ottimizzazione che abilita l'analisi dei flussi di dati. La funzione seguente contiene un esempio di una tale variabile:

```
int
sign (int x)
{
  int s;

  if (x > 0)
    s = 1;
  else if (x < 0)
    s = -1;

  return s;
}
```

La funzione lavora perfettamente con la maggior parte degli argomenti, ma ha un errore quando `x` è `0`: in questo caso il valore di ritorno della variabile s risulterà indefinito.

Compilare il programma soltanto con l'opzione '`-Wall`' non produce alcun avviso, poiché l'analisi dei flussi di dati non viene utilizzata senza l'ottimizzazione:

```
$ gcc -Wall -c uninit.c
```

Per produrre un avviso, il programma deve essere compilato con '`-Wall`' e l'ottimizzazione contemporaneamente. Nella pratica c'è bisogno del livello di ottimizzazione '`-O2`' per avere buoni avvisi:

```
$ gcc -Wall -O2 -c uninit.c
uninit.c: In function `sign':
uninit.c:4: warning: `s' might be used uninitialized
  in this function
```

Questo scopre correttamente la possibilità che la variabile s venga usata senza essere stata definita.

Notate che mentre GCC abitualmente troverà la maggior parte delle variabili non inizializzate, ciò lo fa utilizzando l'euristica che di tanto in tanto salta alcuni casi complicati o avvisa in modo sbagliato in altri. Nell'ultimo caso, è spesso possibile riscrivere le linee interessate in maniera più semplice che rimuovere gli avvisi e migliora la leggibilità del codice sorgente.

7. Compilare un programma C++

Questo capitolo descrive come usare GCC per compilare programmi scritti in C++ e le opzioni a linea di comando specifiche di questo linguaggio.

Il compilatore C++ GNU messo a disposizione da GCC è un vero compilatore C++: compila codice sorgente C++ direttamente in linguaggio assembly. Alcuni altri "compilatori" C++ sono dei traduttori che convertono i programmi C++ in C e poi compilano il risultante programma C utilizzando un esistente compilatore C. Un vero compilatore C++, come GCC, è in grado di fornire un supporto migliore per la segnalazione degli errori, per l'eliminazione degli stessi e per l'ottimizzazione.

7.1. Compilare un semplice programma C++

La procedura per compilare un programma C++ è la stessa per un programma C, solo che usa il comando `g++` al posto di `gcc`. Entrambi i compilatori fanno parte della GNU Compilers Collection.

Qui c'è una versione del programma *`Ciao Mondo`* scritta in C++ per dimostrare l'impiego di `g++`:

```
#include <iostream>

int
main ()
{
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

Il programma può essere compilato con la seguente linea di comando:

```
$ g++ -Wall ciao.cc -o ciao
```

L'interfaccia frontale C++ di GCC usa molte delle stesse opzioni del compilatore C `gcc`. Supporta anche alcune opzioni aggiuntive per controllare le funzionalità del linguaggio C++ che

saranno descritte in questo capitolo. Notate che al codice sorgente C++ dovrebbe essere attribuita una delle valide estensioni dei file C++ '`.cc`', '`.cpp`', '`.cxx`' o '`.C`' piuttosto che l'estensione '`.c`' utilizzata per i programmi C.

L'eseguibile risultante può essere avviato nell'esatto stesso modo della versione in C, digitando semplicemente il suo nome di file:

```
$ ./ciao
Ciao, mondo!
```

L'eseguibile produce lo stesso risultato della versione C del programma, usando `std::cout` invece della funzione C `printf`. Tutte le opzioni usate nei comandi di `gcc` nei precedenti capitoli si applicano a `g++` senza modifiche, così come le procedure per la compilazione ed il collegamento di file e librerie (usando naturalmente `g++` invece di `gcc`). Una naturale differenza è che l'opzione '`-ansi`' richiede l'osservanza del C++ standard, al posto del C standard, quando viene usata con `g++`.

Notate che i programmi che usano i file oggetto C++ devono essere sempre collegati con `g++` per fornire le appropriate librerie C++. Il tentativo di collegare un file oggetto con il compilatore C gcc produrrà alcuni errori "`undefined reference`" riguardanti le funzioni della libreria standard C++:

```
$ g++ -Wall -c ciao.cc
$ gcc ciao.o       (dovremmo usare g++)
ciao.o: In function `main':
ciao.o(.text+0x1b): undefined reference to `std::cout'
.....
ciao.o(.eh_frame+0x11):
  undefined reference to `__gxx_personality_v0'
```

Il collegamento del medesimo file oggetto con `g++` fornisce tutte le librerie C++ necessarie e produrrà un eseguibile funzionante:

```
$ g++ ciao.o
$ ./a.out
Ciao, mondo!
```

Un punto che qualche volta crea confusione è che attualmente `gcc` compila il codice sorgente C ++ quando rileva un'estensione dei file C++, ma poi non è in grado di collegare i file oggetto risultanti.

```
$ gcc -Wall -c ciao.cc    (succede anche con il C++)
$ gcc ciao.o
ciao.o: In function `main':
ciao.o(.text+0x1b): undefined reference to `std::cout'
```

Per evitare tale problema è meglio utilizzare coerentemente `g++` per i programmi C++ e `gcc` per

i programmi C.

7.2. Uso della libreria standard C++

Un'implementazione della libreria standard C++ è fornita come parte di GCC. Il programma seguente utilizza la classe `string` della libreria standard per reimplementare il programma *Ciao Mondo*:

```
#include <string>
#include <iostream>

using namespace std;

int

main ()
{
  string s1 = "Ciao,";
  string s2 = "Mondo!";
  cout << s1 + " " + s2 << endl;
  return 0;
}
```

Il programma può essere compilato e fatto funzionare impiegando gli stessi comandi come sopra:

```
$ g++ -Wall ciaostr.cc
$ ./a.out
Ciao, Mondo!
```

Notate che per rispetto allo standard C++, i file di intestazione per la libreria stessa non utilizzano un'estensione di file. Le classi nella libreria sono definite anche nello spazio dei nomi (*namespace*) `std`, cosicché serve la direttiva `using namespace std` per accedervi, a meno che venga utilizzato dall'inizio alla fine il prefisso `std::` (come visto nella sezione precedente).

7.3. Modelli

I modelli (o *template*) forniscono la capacità di definire classi C++ che supportano tecniche di *programmazione generica*. I modelli possono essere considerati come un potente genere di facilitazione delle macro. Quando una classe o funzione da modello viene utilizzata con una specifica classe o tipo, come `float` o `int`, il corrispondente codice da modello viene compilato con sostituzione di quel tipo nei posti giusti.

7.3.1. Uso dei modelli della libreria standard C++

La libreria standard C++ '`libstdc++`' fornita con GCC offre un ampio insieme di generiche classi contenitore come le liste e le code in aggiunta a generici algoritmi, come quelli di ordinamento. Queste classi in origine facevano parte della Standard Template Library (STL), che era un pacchetto separato, ma ora sono integrate nella libreria standard C++ stessa.

Il programma seguente dimostra l'uso della libreria dei modelli creando un elenco di stringhe con il modello `list<string>`:

```
#include <list>
#include <string>
#include <iostream>

using namespace std;

int
main ()
{
  list<string> list;
  list.push_back("Ciao");
  list.push_back("Mondo");
  cout << "List size = " << list.size() << endl;
  return 0;
}
```

Non sono richieste opzioni speciali per utilizzare le classi dei modelli nella libreria standard. Le opzioni a linea di comando per compilare questo programma sono gli stessi di prima:

```
$ g++ -Wall string.cc
$ ./a.out
List size = 2
```

Notate che gli eseguibili creati da `g++` usando la libreria standard C++ saranno collegate alla libreria condivisa '`libstdc++`', che è fornita quale parte dell'installazione base di GCC. Esistono diverse versioni di questa libreria: se distribuite eseguibili che usano la libreria standard C++, dovete assicurarvi che il ricevente abbia una versione compatibile di '`libstdc++`', oppure collegate il vostro programma staticamente utilizzando l'opzione a linea di comando '`-static`'.

7.3.2. Fornitura dei vostri modelli personali

Oltre alle classi di modelli fornite dalla libreria standard C++, potete definire dei vostri modelli personali. Il metodo raccomandato per utilizzare modelli con `g++` è di seguire il modello di

compilazione delle inclusioni (*inclusion compilation model*), in cui le definizioni del modello sono collocate in file di intestazione. Questo è il metodo usato dalla libreria standard C++ fornita con lo stesso GCC. I file d'intestazione possono essere poi inclusi con '`#include`' in ogni file sorgente dove sono necessari.

Per esempio, il seguente file di modello crea una semplice classe `Buffer<T>` che rappresenta un buffer circolare contenente oggetti di tipo `T`.

```
#ifndef BUFFER_H
#define BUFFER_H

template <class T>
class Buffer
{
public:
  Buffer (unsigned int n);
  void insert (const T & x);
  T get (unsigned int k) const;
private:
  unsigned int i;
  unsigned int size;
  T *pT;
};

template <class T>
Buffer<T>::Buffer (unsigned int n)
{
   i = 0;
   size = n;
   pT = new T[n];
};

template <class T>
void
Buffer<T>::insert (const T & x)
{
   i = (i + 1) % size;
   pT[i] = x;
```

```
};

template <class T>
T
Buffer<T>::get (unsigned int k) const
{
   return pT[(i + (size - k)) % size];
};

#endif /* BUFFER_H */
```

Il file contiene sia la definizione della classe che le definizioni delle appartenenti funzioni. Questa classe è fornita solo a scopo dimostrativo e non dovrebbe essere considerata un esempio di buona programmazione. Notate che l'uso di *include guards*, che controllano la presenza della macro `BUFFER_H`, assicura che le definizioni nel file d'intestazione siano verificate una volta sola, se il file viene incluso più volte nello stesso contesto.

Il programma sottostante usa la classe a modelli `Buffer` per creare n buffer di misura 10, che conserva i valori in virgola mobile `0,25` e `1,0` nel buffer:

```
#include <iostream>
#include "buffer.h"

using namespace std;

int
main ()
{
  Buffer<float> f(10);
  f.insert (0.25);
  f.insert (1.0 + f.get(0));
  cout << "valore memorizzato = " << f.get(0) << endl;
  return 0;
}
```

Le definizioni della classe a modelli e le sue funzioni sono incluse nel file sorgente del programma con '`#include "buffer.h"`' prima del loro uso. Il programma può essere poi compilato utilizzando la seguente linea di comando:

```
$ g++ -Wall tprog.cc
$ ./a.out
```

```
valore memorizzato = 1.25
```

Nei punti dove vengono utilizzate le funzioni modello nel file sorgente, `g++` compila la definizione appropriata dal file di intestazione e colloca la funzione compilata nel corrispondente file oggetto.

Se una funzione modello viene usata diverse volte in un programma, essa sarà conservata in più di un file oggetto. Il GNU Linker fa in modo che solo una copia venga piazzata nell'eseguibile finale. Altri collegatori potrebbero segnalare errori "*multiply defined symbol*" quando si imbattono in più di una copia di una funzione modello (un metodo di lavoro con questi collegatori viene descritto di seguito.

7.3.3. Chiamata esplicita dei modelli

Per raggiungere il completo controllo sulla compilazione dei modelli con `g++` è possibile richiedere una esplicita chiamata di ciascuna ricorrenza di un modello, utilizzando l'opzione '`-fno-implicit-templates`'. Tale metodo non è necessario quando si usa il GNU Linker (è un'alternativa al modello della compilazione per inclusione nei sistemi con collegatori (*linker*) che non sono in grado di eliminare le definizioni duplicate delle funzioni modello nei file oggetto.

Con questo approccio le funzioni modello non vengono più compilate nel punto dove vengono usate in conseguenza dell'opzione '`-fno-implicit-templates`'. Invece, il compilatore cerca una istanza esplicita del modello che usa la parola-chiave `template` con un tipo specifico per forzare la sua compilazione (questa è una estensione GNU del comportamento standard). Tali istanze sono tipicamente collocate in un file sorgente, che poi viene compilato per creare un file oggetto contenente tutte le funzioni modello richieste da un programma. Ciò assicura che ciascun modello appaia in un unico file oggetto e che sia compatibile con i collegatori che non sono in grado di eliminare le definizioni duplicate nei file oggetto.

Per esempio, il seguente file '`templates.cc`' contiene una esplicita istanza della classe `Buffer<ifloat>` usata dal programma '`tprog.cc`' visto sopra:

```
#include "buffer.h"
template class Buffer<float>;
```

L'intero programma può essere compilato e collegato utilizzando una istanza esplicita con i seguenti comandi:

```
$ g++ -Wall -fno-implicit-templates -c tprog.cc
$ g++ -Wall -fno-implicit-templates -c templates.cc
$ g++ tprog.o templates.o
$ ./a.out
stored value = 1.25
```

Il codice oggetto per tutte le funzioni modello è contenuto nel file '`templates.o`'. Non esiste un codice oggetto per le funzioni modello nel file '`tprog.o'` quando viene compilato con l'opzione '`-fno-implicit-templates`'.

Se il programma viene modificato per utilizzare tipi aggiuntivi, successivamente si possono aggiungere ulteriori istanze degli oggetti Buffer contenenti valori `double` ed `int`:

```
#include "buffer.h"
template class Buffer<float>;
template class Buffer<double>;
template class Buffer<int>;
```

Lo svantaggio di un'istanza esplicita è che bisogna conoscere che tipi di modello sono richiesti dal programma. Per un programma sofisticato ciò potrebbe essere difficile da determinare in anticipo. Comunque ogni istanza di modello persa può essere verificata nel momento del collegamento ed aggiunta nell'elenco delle istanze esplicite notando che le funzioni sono indefinite.

L'istanziazione esplicita può essere usata anche per produrre librerie di funzioni modello precompilate creando un file oggetto contenente tutte le istanze di una funzione modello (come nel soprastante file '`templates.cc`'). Per esempio, il file oggetto creato dalle istanze di modello precedenti contiene il codice macchina necessario alla classe Buffer con i tipi '`float`', '`double`' e '`int`', e potrebbe essere distribuito in una libreria.

7.3.4. La parola-chiave `export`

Nel momento della scrittura, GCC non supporta la nuova parola-chiave C++ `export` (GCC 3.3.2).

Questa parola-chiave era stata proposta come modo per separare l'interfaccia dei modelli dalla loro implementazione. Tuttavia essa aggiunge la sua intrinseca complessità al processo di compilazione, che può in pratica sottrarre qualsiasi vantaggio.

La parola-chiave `export` non è ampiamente utilizzata e molti altri compilatori non la supportano neppure. Il modello di compilazione ad inclusione precedentemente descritto è raccomandato come il modo più semplice e portabile per usare i modelli.

8. Opzioni specifiche delle piattaforme

GCC mette a disposizione un insieme di opzioni specifiche delle piattaforme per i diversi tipi di CPU. Tali opzioni controllano funzionalità quali le modalità hardware di virgola mobile e l'uso di speciali istruzioni per differenti CPU. Esse si possono selezionare con l'opzione '`-m`' nella linea di comando e funzionano con tutte le interfacce frontali dei linguaggi di GCC, come `gcc` e `g++`.

La sezione seguente descrive alcune delle opzioni disponibili per le comuni piattaforme. Un elenco completo di tutte le opzioni specifiche delle piattaforme si può trovare nel manuale di riferimento di GCC, "*Using GCC"* (v. [Ulteriori letture], pag. 93). Il supporto per i nuovi processori viene aggiunto a GCC non appena sono disponibili, perciò alcune delle opzioni descritte in questo capitolo potrebbero mancare nelle versioni più datate di GCC.

8.1. Le opzioni per x86 Intel e AMD

Le funzionalità delle largamente utilizzate famiglie x86 Intel e AMD di processori (386, 486, Pentium, ecc.) si possono controllare tramite le opzioni GCC specifiche delle piattaforme.

In queste piattaforme, GCC produce del codice eseguibile che in partenza è compatibile con tutti i processori della famiglia x86 (riferendosi al 386). Comunque è pure possibile compilare per uno specifico processore per ottenere migliori prestazioni[1].

Per esempio, le versioni recenti di GCC hanno uno specifico supporto per i processori più nuovi come il Pentium 4 e l'AMD Athlon. Questi si possono selezionare con la seguente opzione per il Pentium 4,

```
$ gcc -Wall -march=pentium4 ciao.c
```

e per l'Athlon:

```
$ gcc -Wall -march=athlon ciao.c
```

Un elenco completo dei tipi di CPU supportate si può trovare nel manuale di riferimento di GCC.

Il codice prodotto con una specifica opzione '`-march=`*`CPU`*" sarà più veloce ma non girerà in altri processori della famiglia x86. Se progettate di distribuire i file eseguibili per l'utilizzo con tutti i processori Intel e AMD, essi dovrebbero essere compilati senza l'opzione '`-march`'.

(1) Si parla anche di 'mirare' ("*targeting*") ad uno specifico processore.

In alternativa, l'opzione '`-mcpu=`*`CPU`*' offre un compromesso fra velocità e portabilità: genera un codice che è ritagliato per uno specifico processore, in termini di pianificazione delle istruzioni, ma non usa alcuna istruzione che non sia disponibile in altre CPU della famiglia x86. Il codice risultante sarà compatibile con tutte le CPU ed avrà un vantaggio come velocità nella CPU specificata in '`-mcpu`'. Gli eseguibili prodotti da '`-mcpu`' non potranno raggiungere le stesse prestazioni di '`-march`' ma nella pratica potrebbero risultare più convenienti.

AMD ha evoluto l'insieme di istruzioni x86 a 32-bit ad uno a 64-bit, chiamato x86-64, che è implementato nei suoi processori AMD64[(2)]. Nei sistemi AMD64 GCC produce normalmente codice a 64-bit: invece l'opzione '-m32' permette di generare codice a 32-bit.

Il processore AMD64 ha diversi modelli di memoria differenti per i programmi che girano in modalità a 64-bit. Il modello base è il modello a piccolo codice (*small code model*), che consente codici e dati di dimensione fino a 2 GB. Il modello a codice intermedio (o *medium code model*) permette dimensioni dei dati illimitate e può essere scelto con '`-mcmodel=medium`'. Esiste anche un modello a codice grande (o *large code model*') che supporta una dimensione illimitata del codice in aggiunta a quella illimitata dei dati: attualmente in GCC non è implementato dal momento che il modello a codice intermedio è sufficiente per tutti gli impieghi pratici (eseguibili con grandezze superiori ai 2 GB non se ne sono incontrati nella pratica).

Uno speciale modello di codice per kernel (o *kernel code model*) '`-mcmodel=kernel`' viene messo a disposizione per il codice a livello di sistema, come per il kernel di Linux. Un punto importante da notare è che di base nell'AMD64 esiste un'area di memoria a 128-bit allocata al di sotto del puntatore dello stack per dati temporanei, detta "zona rossa" o "*red zone*", che non è supportata dal kernel di Linux. La compilazione del kernel di Linux nell'AMD64 richiede le opzioni '`-mcmodel=kernel -mno-red-zone`'.

8.2. Opzioni per DEC Alpha

Il processore DEC Alpha ha delle impostazioni di base che massimizzano le prestazioni in virgola mobile alle spese del supporto completo delle funzionalità aritmetiche IEEE.

Il supporto per l'aritmetica dell'infinito e il gradual underflow (numeri denormalizzati) non è abilitato nella configurazione di partenza del processore DEC Alpha. Le operazioni che producono infiniti o *underflow* causeranno eccezioni in virgola mobile (note anche come *trappole* o *traps*) e faranno in modo che il programma si interrompa, a meno che il sistema operativo intercetti e gestisca le eccezioni (cosa che è, in genere, inefficiente). Lo standard IEEE specifica che queste operazioni dovrebbero produrre risultati speciali per rappresentare le quantità nel formato numerico IEEE.

In molti casi il comportamento base del DEC Alpha è accettabile dal momento che la maggioranza dei programmi non produce infiniti o *underflow*. Per le applicazioni che richiedono queste funzionalità, GCC fornisce l'opzione '-mieee' per abilitare il pieno supporto all'aritmetica IEEE.

(2) Intel ha aggiunto il supporto a questo insieme di istruzioni come "Intel 64-bit enhancements" nelle sue CPU Xeon.

Per dimostrare la differenza tra i due casi, il seguente programma divide 1 per 0:

```
#include <stdio.h>


  int
  main (void)
  {
    double x = 1.0, y = 0.0;
    printf ("x/y = %g\n", x / y);
    return 0;
  }
```

In aritmetica IEEE il risultato di 1/0 è inf (*infinito* o *infinity*). Se il programma viene compilato per il processore Alpha con le impostazioni base, esso genera un'eccezione, che ferma il programma:

```
$ gcc -Wall alpha.c
$ ./a.out
Floating point exception (in un processore Alpha)
```

L'uso dell'opzione '`-mieee`' assicura la piena aderenza a IEEE: la divisione `1/0` attualmente produce il risultato `inf` ed il programma continua a funzionare con successo:

```
$ gcc -Wall -mieee alpha.c
$ ./a.out
x/y = inf
```

Notate che i programmi che generano eccezioni in virgola mobile girano più lentamente quando sono compilati con '`-mieee`' perché le eccezioni vengono gestite via software piuttosto che hardware.

8.3. Opzioni per SPARC

Nel complesso SPARC dei processori l'opzione '`-mcpu=`*`CPU`*' genera del codice specifico per il processore. Le opzioni valide per *`CPU`* sono `v7`, `v8` (SuperSPARC), `Sparclite`, `Sparclet` e `v9` (UltraSPARC). Il codice prodotto con una specifica opzione '`-mcpu`' non girerà negli altri processori della famiglia SPARC, a parte quando è supportato dalla retrocompatibilità del processore stesso.

Nei sistemi UltraSPARC a 64-bit le opzioni '`-m32`' e '`-m64`' controllano la generazione di codice per ambienti a 32-bit e a 64-bit. L'ambiente a 32-bit selezionato con '`-m32`' utilizza tipi di `int`, `long` e puntatori con dimensione a 32 bits. Invece l'ambiente a 64-bit selezionato con '`-m64`' un tipo di `int` a 32-bit e tipi di `long` e puntatori a 64-bit.

8.4. Opzioni per POWER/PowerPC

Nei sistemi che utilizzano la famiglia dei processori POWER/PowerPC l'opzione '`-mcpu=`*`CPU`*' seleziona la produzione di codice per specifici modelli di CPU. I possibili valori di *CPU* comprendono '`power`', '`power2`', '`powerpc`', '`powerpc64`' e '`common`' in aggiunta agli altri più specifici numeri di modello. Il codice generato con l'opzione '`-mcpu=common`' girerà su ognuno dei processori. L'opzione '`-maltivec`' abilita l'uso delle istruzioni di elaborazione vettoriale Altivec, se è disponibile l'appropriato supporto hardware.

I processori POWER/PowerPC comprendono una istruzione composita "moltiplicazione e somma" *a*x+b*, che esegue le due operazioni contemporaneamente per speditezza: questa è chiamata moltiplicazione e addizione fuse insieme, e viene utilizzata normalmente da GCC. A causa di differenze nel modo in cui vengono arrotondati i valori intermedi, il risultato di un'istruzione fusa potrebbe non essere esattamente il medesimo di quello con le due operazioni eseguite separatamente. Nei casi in cui viene richiesta una rigida aritmetica IEEE, l'impiego delle istruzioni combinate può essere disabilitato con l'opzione '`-mno-fused-madd`'.

Nei sistemi AIX l'opzione '`-mminimal-toc`' diminuisce il numero di dati che GCC inserisce nella *tabella dei contenuti* (TOC o *Table Of Contents*) globale negli eseguibili per evitare gli errori "TOC overflow" al momento del collegamento (link time). L'opzione '`-mxl-call`' rende il collegamento dei file oggetto di GCC con quelli dei compilatori XL della IBM. Per le applicazioni che fanno uso dei *thread* di POSIX, AIX richiede sempre l'opzione '`-pthread`' compilando, anche quando il programma girerà solo nella modalità a singolo thread.

8.5. Supporto multi-architettura

Numerose piattaforme sono in grado di eseguire codice da più di una architettura. Per esempio, le piattaforme a 64-bit come AMD64, MIPS64, Sparc64 e Power64, supportano l'esecuzione di codice sia a 32-bit che a 64-bit. Allo stesso modo, i processori ARM supportano sia il codice ARM sia un codice più compatto chiamato "Thumb". GCC può essere predisposto per supportare architetture multiple in queste piattaforme. Normalmente il compilatore genererà file oggetto a 64-bit, ma dando l'opzione '`-m32`' produrrà un file oggetto a 32-bit per la corrispondente architettura(3).

Notate che tale supporto per architetture multiple dipende dalla disponibilità delle corrispondenti librerie. Nelle piattaforme a 64-bit che supportano gli eseguibili a 64-bit che a 32-bit, le librerie a 64-bit spesso sono collocate nelle directory '`lib64`' piuttosto che in quelle '`lib`', come per esempio in '`/usr/lib64`' e in '`/lib64`'. Le librerie a 32-bit si trovano poi nelle normali directory '`lib`' come in altre piattaforme. Ciò permette alle librerie a 32-bit ed a 64-bit con lo stesso nome di coesistere nello stesso sistema. Altri sistemi, come l'IA64/Itanium, usano le directory '`/usr/lib`' e '`/lib`' per le librerie a 64-bit. GCC conosce questi percorsi ed utilizza quello appropriato quando compila il codice a 64-bit o 32-bit.

(3) Le opzioni '`-maix64`' e '`-maix32`' si usano con AIX.

9. Individuazione dei problemi

GCC fornisce diverse opzioni di aiuto e di diagnostica per assistere alla individuazione dei problemi durante il processo di compilazione. Tutte le opzioni descritte in questo capitolo funzionano sia con `gcc` che con `g++`.

9.1. Aiuto sulle opzioni a linea di comando

Per ottenere un breve riepilogo delle varie opzioni a linea di comando, GCC fornisce una opzione di aiuto che mostra un sommario delle opzioni di alto livello della linea di comando di GCC:

```
$ gcc --help
```

Per far apparire un elenco completo delle opzioni di gcc e dei suoi programmi associati, come GNU Linker e GNU Assembler, utilizzate la soprastante opzione di aiuto con l'opzione prolisso (*verbose*) '`-v`':

```
$ gcc -v --help
```

La lista completa delle opzioni prodotta da questo comando è estremamente lunga: potreste desiderare di scorrerla in pagine con il comando more o di redirigerne i risultati verso un file per riferimento:

```
$ gcc -v --help 2>&1 | more
```

9.2. I numeri di versione

Potete trovare il numero di versione di gcc utilizzando l'opzione della versione:

```
$ gcc --version
gcc (GCC) 3.3.1
```

Il numero di versione è importante quando si investiga su dei problemi di compilazione, dal momento che versioni più vecchie di GCC potrebbero essere prive di alcune funzioni che un programma usa. Il numero di versione ha la forma *versione-maggiore.versione-minore* oppure *versione-maggiore.versione-minore.versione-micro*, dove l'aggiunto terzo numero di versione "micro" (come mostrato sopra) viene utilizzato per i rilasci successivi di correzione di errori in una

serie di rilasci.

Si possono trovare maggiori dettagli sulla versione usando '`-v`':

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i686/3.3.1/specs
Configured with: ../configure --prefix=/usr
Thread model: posix
gcc version 3.3.1
```

Ciò comprende le informazioni sui build flag del compilatore stesso ed il file installato di configurazione , '`specs`'.

9.3. Compilazione prolissa

L'opzione '`-v`' può essere utilizzata anche per mostrare informazioni dettagliate sull'esatta sequenza dei comandi utilizzati per compilare e collegare un programma. Qui c'è un esempio che fa vedere la compilazione prolissa (*verbose compilation*) del programma *Ciao Mondo*:

```
$ gcc -v -Wall hello.c
Reading specs from /usr/lib/gcc-lib/i686/3.3.1/specs
Configured with: ../configure --prefix=/usr
Thread model: posix
gcc version 3.3.1
 /usr/lib/gcc-lib/i686/3.3.1/cc1 -quiet -v -D__GNUC__=3
 -D__GNUC_MINOR__=3 -D__GNUC_PATCHLEVEL__=1
 ciao.c -quiet -dumpbase ciao.c -auxbase ciao -Wall
 -version -o /tmp/cceCee26.s
GNU C version 3.3.1 (i686-pc-linux-gnu)
 compiled by GNU C version 3.3.1 (i686-pc-linux-gnu)
GGC heuristics: --param ggc-min-expand=51
 --param ggc-min-heapsize=40036
ignoring nonexistent directory "/usr/i686/include"
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/include
 /usr/lib/gcc-lib/i686/3.3.1/include
 /usr/include
```

```
End of search list.
 as -V -Qy -o /tmp/ccQynbTm.o /tmp/cceCee26.s
GNU assembler version 2.12.90.0.1 (i386-linux)
using BFD version 2.12.90.0.1 20020307 Debian/GNU
Linux
/usr/lib/gcc-lib/i686/3.3.1/collect2
 --eh-frame-hdr -m elf_i386 -dynamic-linker
 /lib/ld-linux.so.2 /usr/lib/crt1.o /usr/lib/crti.o
 /usr/lib/gcc-lib/i686/3.3.1/crtbegin.o
 -L/usr/lib/gcc-lib/i686/3.3.1
 -L/usr/lib/gcc-lib/i686/3.3.1/../../.. /tmp/ccQynbTm.o
 -lgcc -lgcc_eh -lc -lgcc -lgcc_eh
 /usr/lib/gcc-lib/i686/3.3.1/crtend.o
 /usr/lib/crtn.o
```

I risultati prodotti da '`-v`' possono essere utili ogni qualvolta si presenti un problema durante lo stesso processo di compilazione. Essi mostrano i percorsi completi delle directory utilizzati per la ricerca dei file di intestazione e delle librerie, i simboli predefiniti del preprocessore e i file oggetto e le librerie usate per il collegamento.

10. Strumenti collegati al compilatore

Questo capitolo descrive un numero di strumenti che sono utili in combinazione con GCC. Questi comprendono l'archiviatore GNU `ar`, per la creazione di librerie, e i programmi GNU per la profilazione e per la prova di copertura, `gprof` e `gcov`.

10.1. Creazione di una libreria con l'archiviatore GNU

L'archiviatore (*archiver*) GNU `ar` combina una collezione di file oggetto in un singolo file di archivio, altrimenti noto come *libreria* (*library*). Un file di archivio è semplicemente un modo conveniente per distribuire un grande numero di file oggetto tra loro collegati (come descritto in precedenza nella Sezione 2.6 [Collegarsi con le librerie esterne], pagina 18).

Per mostrare l'uso dell'archiviatore GNU creeremo una piccola libreria '`libciao.a`' contenente due funzioni ciao e arrivederci.

Il primo file oggetto verrà generato dal codice sorgente della funzione `ciao` nel file '`ciao_fn.c`' visto poco prima.

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

void
ciao (const char * name)
{
  printf ("Ciao, %s!\n", name);
}
```

Il secondo file oggetto verrà prodotto dal file sorgente '`arrivederci_fn.c`', che contiene la nuova funzione `arrivederci`:

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

void
```

```
arrivederci (void)
{
  printf ("Arrivederci!\n");
}
```

Entrambe le funzioni utilizzano il file d'intestazione '`ciao.h`', ora con un prototipo per la funzione `arrivederci()`:

```
void ciao (const char * name);
void arrivederci (void);
```

Il codice sorgente può essere compilato nei file oggetto '`ciao_fn.o`' e '`arrivederci_fn.o`' usando i comandi:

```
$ gcc -Wall -c ciao_fn.c
$ gcc -Wall -c arrivederci_fn.c
```

Tali file oggetto possono essere in una libreria statica utilizzando la seguente linea di comando:

```
$ ar cr libciao.a ciao_fn.o arrivederci_fn.o
```

L'opzione '`cr`' sta per "crea e rimpiazza"[(1)]. Se la libreria non esiste, per prima cosa viene creata. Se la libreria è già esistente, in essa ogni file originale con lo stesso nome viene rimpiazzato dai nuovi file indicati nella linea di comando. Il primo argomento '`libciao.a`' è il nome della libreria. I restanti argomenti sono i nomi dei file oggetto che devono essere copiati nella libreria.

L'archiviatore `ar` fornisce anche una opzione '`t`' "tabella dei contenuti" per elencare i file oggetto di una libreria esistente:

```
$ ar t libciao.a
ciao_fn.o
arrivederci_fn.o
```

Notate che quando viene distribuita una libreria, i file d'intestazione delle funzioni e variabili pubbliche che essa fornisce dovrebbero essere anche resi disponibili in modo che l'utente finale possa includerli ed ottenere i prototipi corretti.

Ora possiamo scrivere un programma che faccia uso delle funzioni contenute nella libreria appena creata:

```
#include "ciao.h"

int
main (void)
{
  ciao ("tutti");
```

(1) Osservate che `ar` non richiede un prefisso '`-`' per le sue opzioni.

```
    arrivederci ();
    return 0;
}
```

Questo file può essere compilato tramite la seguente linea di comando, come descritto nella Sezione 2.6 [Collegamenti con le librerie esterne] a pagina 18, supponendo che la libreria '`libciao.a`' sia posta nella directory corrente:

```
$ gcc -Wall main.c libciao.a -o ciao
```

Il programma `main` è collegato ai file oggetto trovati nel file della libreria '`libciao.a`' per produrre l'eseguibile finale.

L'opzione scorciatoia '`-l`' per collegare le librerie può anche essere utilizzata per collegare il programma senza la necessità di specificare esplicitamente l'intero nome del file della libreria:

```
$ gcc -Wall -L. main.c -lciao -o ciao
```

L'opzione '`-L.`' serve per aggiungere la directory corrente al percorso di ricerca della libreria. L'eseguibile risultante può essere avviato come il solito:

```
$ ./ciao
Ciao, tutti!
Arrivederci!
```

Esso mostra i dati in uscita di entrambe le funzioni `ciao` e `arrivederci` definite nella libreria.

10.2. Uso del profilatore `gprof`

Il profilatore GNU `gprof` è uno strumento utile a misurare le prestazioni di un programma: registra il numero di chiamate a ciascuna funzione e la quantità di tempo lì impiegato, su una base approssimativa-ooooo. Le funzioni che consumano una grossa frazione del tempo di esecuzione possono essere identificate con semplicità in base ai risultati di `gprof`. Gli sforzi per velocizzare un programma dovrebbero concentrarsi per prima cosa su quelle funzioni che primeggiano nel tempo totale di esecuzione.

Useremo `gprof` per esaminare le prestazioni di un piccolo programma numerico che calcola le lunghezze delle sequenze che si presentano in matematica nella irrisolta *congettura di Collatz*[2]. La congettura di Collatz implica sequenze definite dalla regola:

$$x_n \leftarrow \begin{cases} x_n/2 & se\ x_n\ è\ pari \\ 3x_n+1 & se\ x_n\ è\ dispari \end{cases}$$

La sequenza viene ripetuta a partire da un valore iniziale x_0 fino a che termina con il valore `1`. In

(2) American Mathematical Monthly, Volume 92 (1985), 3-23

ossequio alla congettura, tutte le congetture fanno terminare casualmente: il programma in basso mostra le sequenze più lunghe come incrementi di x_0. Il file sorgente '`collatz.c`' contiene tre funzioni: `main`, `nseq` e `step`:

```
#include <stdio.h>
/* Calcola la lunghezza delle sequenze di Collatz */
unsigned int
step (unsigned int x)
{
  if (x % 2 == 0)
    {
      return (x / 2);
    }
  else
    {
      return (3 * x + 1);
    }
}

unsigned int
nseq (unsigned int x0)
{
  unsigned int i = 1, x;
  if (x0 == 1 || x0 == 0)
    return i;

  x = step (x0);

  while (x != 1 && x != 0)
    {
      x = step (x);
      i++;
    }

  return i;
}
```

```
int
main (void)
{
  unsigned int i, m = 0, im = 0;

  for (i = 1; i < 500000; i++)
    {
      unsigned int k = nseq (i);

      if (k > m)
        {
           m = k;
           im = i;
           printf ("lunghezza sequenza = %u for %u\n", m, im);
        }
    }

  return 0;
}
```

Per utilizzare la profilatura, il programma deve essere compilato e collegato con l'opzione di profilatura "`-pg`":

```
$ gcc -Wall -c -pg collatz.c
$ gcc -Wall -pg collatz.o
```

Ciò crea un eseguibile *strumentizzato* o *instrumented* che contiene istruzioni aggiuntive che registrano il tempo speso per ciascuna funzione.

Se il programma consiste in più di un file sorgente, allora l'opzione '`-pg`' dovrebbe essere utilizzata quando si compila ciascun file sorgente, ed usata nuovamente quando si collegano i file oggetto per creare l'eseguibile finale (come mostrato sopra). Scordarsi di collegare con l'opzione '`-pg`' è un errore comune che impedisce alla profilatura di registrare qualche utile informazione.

L'eseguibile deve essere avviato per creare i dati di profilatura:

```
$ ./a.out
```

(viene mostrato il risultato normale del programma)

Mentre sta girando l'eseguibile strumentalizzato, i dati di profilatura vengono scritti silenziosamente nel file '`gmon.out`' della directory corrente. Essi possono essere analizzati con `gprof` dando il nome dell'eseguibile come argomento:

```
$ gprof a.out
Flat profile:
Each sample counts as 0.01 seconds.
 %     cumul.    self             self   total
time seconds seconds     calls us/call us/call name
68.59    2.14    2.14 62135400    0.03    0.03 step
31.09    3.11    0.97   499999    1.94    6.22 nseq
 0.32    3.12    0.01                          main
```

La prima colonna di dati mostra che il programma occupa la maggior parte del suo tempo (quasi il 70%) nella funzione `step` e il 30% in `nseq`. Conseguentemente gli sforzi per ridurre il tempo di esecuzione del programma dovrebbe concentrarsi sulla prima. In confronto il tempo impegnato dalla stessa funzione `main` è del tutto trascurabile (meno dell'1%).

Le altre colonne dei dati in uscita forniscono informazioni sul numero totale di chiamate a funzioni effettuate e sul tempo speso per ciascuna funzione. Risultati aggiuntivi che analizzano i tempi di esecuzione in modo più particolareggiato sono prodotti pure da `gprof` ma qui non vengono mostrati. Dettagli completi si possono trovare nel manuale "*GNU `gprof`-The GNU Profiler*", di Jay Fenlason e Richard Stallman.

10.3. Prova di copertura con `gcov`

Lo strumento GNU `gcov` per la prova di copertura (*coverage testing*) analizza il numero di volte che viene eseguita ogni linea di un programma durante il funzionamento. Ciò rende possibile la scoperta di aree di codice non utilizzate o che non vengono sfruttate durante la prova. Se combinate con le informazioni di profilazione provenienti da `gprof`, quelle della prova di copertura consentono agli sforzi per velocizzare un programma di concentrarsi su specifiche linee del codice sorgente.

Useremo il sottostante programma d'esempio per mostrare `gcov`. Questo programma cicla gli interi da `1` a `9` e prova la loro divisibilità con l'operatore di modulo (%).

```
#include <stdio.h>
int
main (void)
{
  int i;
  for (i = 1; i < 10; i++)
    {
      if (i % 3 == 0)
        printf ("%d divisibile per 3\n", i);
      if (i % 11 == 0)
```

```
          printf ("%d divisibile per 11\n", i);
      }
    return 0;
}
```

Per attivare la prova di copertura, il programma deve essere compilato con le seguenti opzioni:

```
$ gcc -Wall -fprofile-arcs -ftest-coverage cov.c
```

Ciò crea un eseguibile *strumentizzato* (*instrumented*) contenente istruzioni aggiuntive, le quali registrano il numero di volte che ciascuna linea del programma viene eseguita. L'opzione '`-ftest-coverage`' aggiunge delle istruzioni per contare il numero di volte che le singole linee vengono eseguite, mentre '`-fprofile-arcs`' incorpora il codice di strumentizzazione per ciascuna ramificazione del programma. La strumentizzazione delle diramazioni registra con quale frequenza vengono imboccati percorsi differenti attraverso le istruzioni '`if`' ed altre condizioni. L'eseguibile deve essere poi avviato per creare i dati di copertura:

```
$ ./a.out
3 divisibile per 3
6 divisibile per 3
9 divisibile per 3
```

I dati provenienti dall'esecuzione vengono scritti rispettivamente in diversi file con le estensioni '`.bb`', '`.bbg`' e '`.da`' nella directory corrente. Questi dati possono essere analizzati usando il comando `gcov` ed il nome di un file sorgente:

```
$ gcov cov.c
 88.89% of 9 source lines executed in file cov.c
Creating cov.c.gcov
```

Il comando `gcov` produce una versione annotata del file sorgente originale, con l'estensione '`.gcov`', contenente i conteggi del numero di volte che ciascuna linea è stata eseguita:

```
        #include <stdio.h>

        int
        main (void)
        {
     1    int i;

    10    for (i = 1; i < 10; i++)
            {
     9        if (i % 3 == 0)
     3          printf ("%d is divisible by 3\n", i);
```

```
      9       if (i % 11 == 0)
 ######          printf ("%d is divisible by 11\n", i);
      9     }

      1   return 0;
      1 }
```

I conteggi delle linee si possono vede nella prima colonna dei dati in uscita. Le linee che non sono state eseguite sono segnate con i cancelletti '######'. Il comando 'grep '######' *.gcov' può essere utilizzato per trovare parti di un programma che non sono state usate.

11. Come funziona il compilatore

Questo capitolo descrive più dettagliatamente come GCC trasforma i file sorgente in eseguibili. La compilazione è un processo multifase che comprende vari strumenti, compresi lo stesso compilatore GNU (attraverso le interfacce di `gcc` o `g++`), l'Assemblatore GNU `as` e il collegatore (linker) GNU `ld`. L'insieme degli strumenti utilizzati nel processo di compilazione viene definito come *catena degli strumenti* (*toolchain*).

11.1. Panoramica sul processo di compilazione

La sequenza dei comandi eseguiti con una singola chiamata a GCC consiste nelle fasi seguenti:

- preprocessamento (per espandere le macro)
- compilazione (dal codice sorgente al linguaggio assembly)
- assemblaggio (dal linguaggio assembly al codice macchina)
- collegamento o *linking* (per creare l'eseguibile finale)

Come esempio esamineremo tali fasi della compilazione singolarmente utilizzando il programma *Ciao Mondo* '`ciao.c`':

```
#include <stdio.h>

int
main (void)
{
  printf ("Ciao, mondo!\n");
  return 0;
}
```

Notate che non serve usare ciascun singolo comando descritto in questa sezione per compilare un programma. Tutti i comandi vengono eseguiti internamente da GCC in modo automatico e trasparente, e possono essere visti utilizzando l'opzione '`-v`' descritta in precedenza (v. Sezione 9.3 [La compilazione prolissa], pag. 72). Lo scopo di questo capitolo è quello di fornire una

comprensione di come funziona il compilatore.

Sebbene il programma *Ciao Mondo* sia molto semplice, esso utilizza file di intestazione esterni e librerie, mostrando tutte le fasi fondamentali del processo della compilazione.

11.2. Il preprocessore

Il primo stadio del processo della compilazione consiste nell'uso del preprocessore per espandere le macro e i file di intestazione inclusi. Per svolgere questa fase GCC esegue i seguenti comandi[1]:

```
$ cpp ciao.c > ciao.i
```

Il risultato è un file '`ciao.i`' contenente i codice sorgente con tutte le macro espanse. Convenzionalmente ai file preprocessati viene attribuita l'estensione '`.i`' per i programmi in C e '`.ii`' per quelli in C++. Nella pratica il file preprocessato non viene salvato su disco a meno che non si utilizzi l'opzione '`-save-temps`'.

11.3. Il compilatore

Lo stadio successivo del processo consiste nella compilazione vera e propria del codice sorgente in linguaggio assembly di uno specifico processore. L'opzione a linea di comando '-S' istruisce `gcc` a convertire il preprocessato codice sorgente C nel linguaggio assembly senza creare un file oggetto:

```
$ gcc -Wall -S ciao.i
```

Il linguaggio assembly risultante viene salvato nel file '`ciao.s`'. Qui c'è come appare il linguaggio assembly di *Ciao Mondo* per un processore Intel x86 (i686):

```
$ cat ciao.s
    .file "ciao.c"
    .section .rodata
.LC0:
    .string "Ciao, mondo!\n"
    .text
.globl main
    .type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
```

(1) Come ricordato in precedenza, il preprocessore nelle versioni recenti di GCC è stato integrato nel compilatore. Concettualmente il processo di compilazione è il medesimo rispetto all'avvio del preprocessore come applicazione separata.

```
    subl $8, %esp
    andl $-16, %esp
    movl $0, %eax
    subl %eax, %esp
    movl $.LC0, (%esp)
    call printf
movl $0, %eax
leave
ret
.size main, .-main
.ident "GCC: (GNU) 3.3.1"
```

Notate che il linguaggio assembly contiene una chiamata alla funzione esterna `printf`.

11.4. L'assemblatore

La funzione dell'assemblatore (*assembler*) è quella di convertire il linguaggio assembly in codice macchina e di generare un file oggetto. Quando esistono delle chiamate a funzioni esterne nel file sorgente in assembly, l'assemblatore lascia indefiniti gli indirizzi delle funzioni esterne per il successivo completamento da parte del collegatore. L'assemblatore può essere invocato con la seguente linea di comando:

```
$ as hello.s -o hello.o
```

Come per GCC, il file in uscita viene specificato con l'opzione '`-o`'. Il file risultante '`ciao.o`' contiene le istruzioni per la macchina relative al programma *Ciao Mondo* con un riferimento non definito a `printf`.

11.5. Il collegatore o *linker*

Lo stadio finale della compilazione è il collegamento dei file oggetto per la creazione di un eseguibile. In pratica un eseguibile richiede molte funzioni esterne derivanti dal sistema operativo e dalle librerie C di esecuzione (`crt` o *C run-time libraries*). Di conseguenza i reali comandi per il collegamento, utilizzati internamente da GCC, sono complicati. Per esempio, il comando completo per collegare il programma *Ciao Mondo* è:

```
$ ld -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o
 /usr/lib/crti.o /usr/lib/gcc-lib/i686/3.3.1/crtbegin.o
 -L/usr/lib/gcc-lib/i686/3.3.1 ciao.o -lgcc -lgcc_eh
 -lc -lgcc -lgcc_eh /usr/lib/gcc-lib/i686/3.3.1/crtend.o
 /usr/lib/crtn.o
```

Fortunatamente non serve mai digitare direttamente il comando precedente: l'intero processo di collegamento viene gestito in modo trasparente da `gcc` quando viene chiamato come segue:

```
$ gcc ciao.o
```

Ciò collega il file oggetto '`ciao.o`' alla libreria standard del C e produce un file eseguibile '`a.out`':

```
$ ./a.out
Ciao, mondo!
```

Un file oggetto per un programma in C++ può essere collegato alla libreria standard del C++ nella stessa maniera tramite un singolo comando `g++`.

12. Esaminare i file compilati

Questo capitolo descrive diversi strumenti utili per l'esame dei contenuti dei file eseguibili e di quelli oggetto.

12.1. L'identificazione dei file

Quando un file sorgente è stato compilato come file oggetto o eseguibile, le opzioni utilizzate per compilarlo non sono più scontate. Il comando `file` esamina il contenuto di un file oggetto o di un eseguibile e stabilisce alcune delle sue caratteristiche, come, ad esempio, se è stato compilato con collegamenti dinamici o statici.

Per esempio, qui c'è il risultato del comando `file` su un eseguibile tipo:

```
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386,
  version 1 (SYSV), dynamically linked (uses shared
  libs), not stripped
```

I dati in uscita mostrano che il file eseguibile è collegato dinamicamente ed è stato compilato per i processori Intel 386 e compatibili. Una spiegazione completa dei risultati viene riportata di seguito:

`ELF`

Il formato interno del file eseguibile (ELF significa "Executable and Linking Format"). Altri formati, come COFF "Common Object File Format", vengono utilizzati in qualche sistema operativo più vecchio (es. MS-DOS).

`32-bit`

La dimensione della parola (*word*)(per alcune piattaforme potrebbe essere di 64 bit).

`LSB`

Compilato per una piattaforma con parola con il primo byte meno significante (*Least Significant Byte*), come per i processori Intel e AMD (l'alternativa MSB, *Most Significant*

Byte o byte più significante per primo, viene utilizzata da altri processori, come il Motorola 680x0)[1]. Alcuni processori come Itanium e MIPS supportano sia l'ordinamento LSB sia quello MSB.

`Intel 80386`

Il processore per cui è stato compilato il file eseguibile.

`version 1 (SYSV)`

Questa è la versione del formato interno del file.

`dinamically linked`

L'eseguibile utilizza le librerie condivise o *shared libraries* (`statically linked` indica i programmi collegati staticamente, usando per esempio l'opzione '`-static`').

`not stripped`

L'eseguibile contiene una tabella dei simboli (questa può essere rimossa con il comando `strip`).

Il comando `file` può essere utilizzato con i file oggetto, dando un risultato analogo. Lo standard POSIX[2] dei sistemi Unix definisce il comportamento del comando `file`.

12.2. Esame della tabella dei simboli

Come descritto in precedenza durante la trattazione della caccia agli errori, gli eseguibili e i file oggetto possono contenere una tabella dei simboli (v. Capitolo 5 [Compilare per scoprire gli errori], pag. 45). Questa tabella conserva la posizione delle funzioni e delle variabili in base ai nomi e può essere mostrata con il comando `nm`:

```
$ nm a.out
08048334 t Letext
08049498 ? _DYNAMIC
08049570 ? _GLOBAL_OFFSET_TABLE_
........
080483f0 T main
08049590 b object.11
0804948c d p.3
         U printf@GLIBC_2.0
```

(1) Gli ordinamenti MSB e LSB sono rispettivamente noti anche come *big-endian* e *little-endian* (i termini traggono origine dalla satira di Jonathan Swift "I viaggi di Gulliver", 1727).
(2) POSIX.1 (edizione 2003), IEEE Std 1003.1-2003.

Tra i contenuti della tabella dei simboli i dati in uscita mostrano che l'inizio della funzione `main` ha l'indirizzo (*offset*) esadecimale `080483f0`. La maggioranza dei simboli è per l'uso interno del compilatore e del sistema operativo. Una '`T`' nella seconda colonna indica una funzione definita nel file oggetto, mentre una '`U`' sta a significare una funzione non definita (e che dovrebbe essere risolta collegandovi un altro file oggetto). Una spiegazione completa dei risultati del comando `nm` si può trovare nel manuale *GNU Binutils*.

L'utilizzo più comune del comando `nm` è quello di verificare se una libreria contiene la definizione di una specifica funzione attraverso la ricerca di una voce '`T`' nella seconda colonna a ridosso del nome della funzione.

12.3. Trovare le librerie collegate dinamicamente

Quando un programma è stato compilato utilizzando librerie condivise, esso ha bisogno di caricare quelle librerie dinamicamente al momento dell'esecuzione per chiamare funzioni esterne. Il comando `ldd` esamina un eseguibile e mostra un elenco delle librerie condivise di cui ha necessità. Queste librerie condivise sono indicate come le *dipendenze* dell'eseguibile.

Per esempio, i seguenti comandi dimostrano come scoprire le dipendenze delle librerie condivise nel programma *Ciao Mondo*:

```
$ gcc -Wall ciao.c
$ ldd a.out
libc.so.6 => /lib/libc.so.6 (0x40020000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

I dati in uscita mostrano che il programma *Ciao Mondo* dipende dalla libreria `libc` del C (libreria condivisa, versione 6) e dalla libreria dinamica del caricatore (*loader*) `ld-linux` (libreria dinamica, versione 2).

Se il programma impiega librerie esterne, come ad esempio la libreria matematica (*math library*), anche queste vengono mostrate. Ad esempio, il programma calc (che usa la funzione `sqrt`) genera i seguenti risultati:

```
$ gcc -Wall calc.c -lm -o calc
$ ldd calc
libm.so.6 => /lib/libm.so.6 (0x40020000)
libc.so.6 => /lib/libc.so.6 (0x40041000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

La prima linea mostra che questo programma dipende dalla libreria matematica `libm` (libreria condivisa, versione 6), in aggiunta alla libreria C ed a quella dinamica del caricatore.

Il comando `ldd` può essere usato anche per esaminare le librerie condivise stesse, in modo da seguire la catena delle dipendenze delle librerie condivise.

13. Ricevere aiuto

Se vi imbattete in un problema non trattato da questo manuale, esistono diversi manuali di riferimento che descrivono GCC e gli argomenti relativi ai linguaggi con più dettagli (v. [Ulteriori letture], pag. 93). Tali manuali contengono risposte alle comuni domande e il loro studio attento normalmente porterà ad una soluzione. Se essi non sono chiari, il modo più corretto per ottenere aiuto è di chiedere aiuto ad un collega esperto.

In alternativa esistono molte società e consulenti che offrono supporto commerciale sulle materie della programmazione di GCC su base oraria o d'intervento. Per gli affari questo può essere un modo costoso per ottenere supporto di alta qualità. Un elenco di società di supporto software gratuito può essere trovata nel sito web del Progetto GNU(1). Con il software libero il supporto commerciale è disponibile in un libero mercato: le società di assistenza competono sulla qualità ed il prezzo, e gli utenti non sono legati ad alcuna in particolare. Il supporto per il software proprietario invece è normalmente disponibile presso il venditore originale.

Un supporto commerciale di più alto livello per GCC viene messo a disposizione da società coinvolte nello sviluppo della stessa catena di strumenti del compilatore GNU. Un elenco di queste società può essere trovato nella sezione "*Development Companies*" della pagina web dedicata dall'editore a questo libro(2). Tali ditte sono in grado di fornire servizi come l'estensione di GCC per generare il codice per nuove CPU o la correzione di errori nel compilatore.

(1) http://www.gnu.org/prep/service.html
(2) http://www.network-theory.co.uk/gcc/intro/

Ulteriori letture

La guida assoluta di GCC è il manuale di riferimento ufficiale, "*Using GCC*", pubblicato da GNU Press:

> *Using GCC (for GCC version 3.3.1)* di Richard M. Stallman e la Comunità di Sviluppatori di GCC (pubblicato da GNU Press, ISBN 1-882114-39-6)

Questo manuale è essenziale per chiunque voglia lavorare con GCC perché descrive ogni opzione in dettaglio. Notate che il manuale viene aggiornato quando si rendono disponibili nuove versioni di GCC, cosicché il numero ISBN potrebbe cambiare in futuro.

Se siete nuovi alla programmazione con GCC, vorrete anche imparare ad usare il debugger GNU GDB ed a compilare facilmente grossi programmi con GNU Make. Tali strumenti sono descritti nei seguenti manuali:

> *Debugging with GDB: The Gnu Source-Level Debugger* di Richard M. Stallman, Roland Pesch, Stan Shebs, ed altri (pubblicato da GNU Press, ISBN 1-882114-88-4)
>
> *GNU Make: A Program for Directing Recompilation* di Richard M. Stallman e Roland McGrath (pubblicato da GNU Press, ISBN 1-882114-82-5)

Per la programmazione pratica in C è altresì essenziale avere una buona conoscenza della libreria standard del C. Il seguente manuale documenta tutte le funzioni contenute nella GNU C Library:

> *The GNU C Library Reference Manual* di Sandra Loosemore con Richard M. Stallman ed altri (2 volumi) (pubblicato da GNU Press, ISBN 1-882114-22-1 e 1-882114-24-8)

Assicuratevi di controllare il sito web http://www.gnupress.org/ per le ultime edizioni stampate dei manuali pubblicati da GNU Press. I manuali possono essere acquistati in rete utilizzando una carta di credito sul sito web della FSF[1] oltre ad essere disponibili per ordinazione tramite ISBN presso molte librerie. I manuali pubblicati da GNU Press raccolgono fondi per la Free Software Foundation ed il Progetto GNU.

Informazioni sui comandi di shell, sulle variabili ambientali e sulle regole sugli apici, si possono trovare nel libro che segue:

> *The GNU Bash Reference Manual* di Chet Ramey e Brian Fox (pubblicato da Network

(1) http://order.fsf.org/

Theory Ltd, ISBN 0-9541617-7-7)

Altri manuali GNU citati in questo libro (come *GNU gprof-The GNU Profiler* e *The GNU Binutils Manual*) non erano ancora disponibili su carta stampata all'epoca in cui quest'ultimo è stato pubblicato. Collegamenti a copie in rete si possono trovare nella pagina web dell'editore di questo libro[(2)].

Le pagine web ufficiali del Progetto GNU dedicate a GCC si possono trovare su http://www.gnu.org/software/gcc/. Queste comprendono un elenco di domande ricorrenti (Frequently Asked Questions o FAQ), come pure un archivio per l'individuazione degli errori in GCC e una moltitudine di ulteriori utili informazioni su GCC.

Esistono parecchi libri dedicati agli stessi linguaggi C e C++. Due riferimenti standard sono:

The C Programming Language (ANSI edition) Brian W. Kerighan, Dennis Ritchie (ISBN 0-13110362-8)

The C++ Programming Language (3rd edition) Bjarne Stroustrup (0-20188954-4)

Chiunque utilizzi i linguaggi C e C++ in un contesto professionale dovrebbe procurarsi una copia dei loro standard ufficiali.

Il numero ufficiale dello standard C è ISO/IEC 9899:1990 per quello originale pubblicato nel 1990 e implementato da GCC. Una revisione degli standard C ISO/IEC 9899:1999 (nota come C99) è stata pubblicata nel 1999 ed è per la maggior parte (ma non totalmente) supportata da GCC. Lo standard C++ è l'ISO/IEC 14882.

Lo standard aritmetico IEEE in virgola mobile (IEEE-754) è importante anche per qualsiasi programma che implichi calcoli numerici.

Questi documenti sugli standard sono disponibili presso le principali società di standard. Gli standard del C e del C++ sono a disposizione pure in forma di libri stampati:

The C Standard: Incorporating Technical Corrigendum 1 (pubblicato da Wiley, ISBN 0-470-84573-2)

The C++ Standard (pubblicato da Wiley, ISBN 0-470-84674-7)

Avanzando con l'apprendimento, chiunque utilizzi GCC potrebbe prendere in considerazione di iscriversi alla Association of C and C++ Users (ACCU). La ACCU è un'organizzazione non a scopo di lucro dedicata alla professionalità nella programmazione a tutti i livelli, ed è riconosciuta come un'autorità in questo campo. Maggiori informazioni sono disponibili sul sito web dell'ACCU http://www.accu.org/.

L'ACCU pubblica due giornali sulla programmazione in C e C++, e organizza regolari conferenze. Il canone annuale di associazione rappresenta un buon investimento per i privati o le società che vogliono incoraggiare un più elevato standard di sviluppo professionale all'interno dei

(2) http://www.network-theory.co.uk/gcc/intro

loro gruppi.

Riconoscimenti

Molte persone hanno contribuito a questo libro ed è importante riportare qui i loro nomi:

grazie a Gerald Pfeifer per la sua scrupolosa revisione ed i numerosi suggerimenti per migliorare il libro;

grazie ad Andreas Jaeger per le informazioni sul supporto dell'AMD64 e multipiattaforma, e i molti utili commenti;

grazie a David Edelsohn per le sue informazioni sulle serie di processori POWER/PowerPC;

grazie a Jamie Lokier per le ricerche;

grazie a Stephen Compall per le utili correzioni;

grazie a Gerard Jungman per gli utili commenti;

grazie a Steven Rubin per aver generato lo schema del chip per la copertina con Electric;

e, cosa più importante, grazie a Richard Stallman, fondatore del Progetto GNU, per avere scritto GCC ed averlo reso software libero.

Altri libri della casa editrice

Network Theory pubblica libri dedicati al software libero sotto licenze per documentazione libera. Il nostro attuale catalogo comprende i seguenti titoli:

· Comparing and Merging Files with GNU diff and patch by David MacKenzie, Paul Eggert, and Richard Stallman (ISBN 0-9541617-5- 0) $19.95 (£12.95)

· Version Management with CVS by Per Cederqvist et al. (ISBN 0-9541617-1-8) $29.95 (£19.95)

· GNU Bash Reference Manual by Chet Ramey and Brian Fox (ISBN 0-9541617-7-7) $29.95 (£19.95)

· An Introduction to R by W.N. Venables, D.M. Smith and the R Development Core Team (ISBN 0-9541617-4-2) $19.95 (£12.95)

· GNU Octave Manual by John W. Eaton (ISBN 0-9541617-2-6) $29.99 (£19.99)

· GNU Scientific Library Reference Manual--Second Edition by M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, F. Rossi (ISBN 0-9541617-3-4) $39.99 (£24.99)

· An Introduction to Python by Guido van Rossum and Fred L. Drake, Jr. (ISBN 0-9541617-6-9) $19.95 (£12.95)

· Python Language Reference Manual by Guido van Rossum and Fred L. Drake, Jr. (ISBN 0-9541617-8-5) $19.95 (£12.95)

· The R Reference Manual--Base Package (Volume 1) by the R Development Core Team (ISBN 0-9546120-0-0) $69.95 (£39.95)

· The R Reference Manual--Base Package (Volume 2) by the R Development Core Team (ISBN 0-9546120-1-9) $69.95 (£39.95)

Tutti i titoli si possono ordinare nelle librerie di tutto il mondo.

Le vendite dei manuali finanziano lo sviluppo di altro software e documentazione liberi.

Per dettagli visitate il sito web `http://www.network-theory.co.uk/`

Le organizzazioni del software libero

La GNU Compiler Collection fa parte del Progetto GNU, lanciato nel 1984 per sviluppare un completo sistema operativo simil-Unix che fosse software libero: il sistema GNU.

La Free Software Foudation (FSF o Fondazione per il Software Libero) è un ente benefico esentasse che raccoglie fondi per far proseguire il lavoro del Progetto GNU. E' dedita alla promozione dei vostri diritti di usare, studiare, copiare, modificare e redistribuire programmi di computer. Uno dei modi migliori per aiutare lo sviluppo del software libero è diventare membro associato della Free Software Foundation e pagare regolarmente le quote per sostenere i suoi sforzi – per maggiori informazioni visitate il sito internet della FSF:

Free Software Foundation (FSF)

Stati Uniti -- http://www.fsf.org/

In tutto il mondo esistono molte altre organizzazioni associazionistiche locali per il software libero che supportano gli obbiettivi della Free Software Foundation, tra cui:

Free Software Foundation Europe (FSF Europe)

Europa -- http://www.fsfeurope.org/

Association for Free Software (AFFS)

Regno Unito -- http://www.affs.org.uk/

Irish Free Software Organisation (IFSO)

Irlanda -- http://www.ifso.info/

Association for Promotion and Research in Libre Computing (APRIL)

Francia -- http://www.april.org/

Associazione Software Libero

Italia -- http://www.softwarelibero.it/

Verein zur Förderung Freier Software (FFIS)

Germania -- http://www.ffis.de/

Verein zur Förderung Freier Software

Austria -- http://www.ffs.or.at/

Association Electronique Libre (AEL)

Belgio -- http://www.ael.be/

National Association for Free Software (ANSOL)

Portogallo -- http://www.ansol.org/

Free Software Initiative of Japan (FSIJ)

Japan -- http://www.fsij.org/

Free Software Foundation of India (FSF India)

India -- http://www.fsf.org.in/

La *Foundation for a Free Information Infrastructure (FFII)* è una organizzazione importante in Europa. La FFII non riguarda in modo specifico il software libero, ma lavora per difendere i diritti di tutti i programmatori e gli utenti dei computer contro i monopoli nel campo dell'informatica, come nel caso dei brevetti sul software.

Per maggiori informazioni sulla FFII, o per sostenere il suo lavoro con una donazione, visitate il suo sito web su http://www.ffii.org/.

GNU Free Documentation License

Version 1.2, November 2002

59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document free in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ascii without markup, Texinfo input format, LaTEX input format, sgml or xml using a publicly available dtd, and standard-conforming simple html, PostScript or pdf designed for human modification. Examples of transparent image formats include png, xcf and jpg. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, sgml or xml for which the dtd and/or processing tools are not generally available, and the machine-generated html, PostScript or pdf produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is

precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the

stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the

combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections.

You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C) year your name.

Permission is granted to copy, distribute and/or modify

this document under the terms of the GNU Free

Documentation License, Version 1.2 or any later version

published by the Free Software Foundation; with no
```

Invariant Sections, no Front-Cover Texts, and no
Back-Cover Texts. A copy of the license is included in
the section entitled ``GNU Free Documentation License''.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being list their
titles, with the Front-Cover Texts being list, and
with the Back-Cover Texts being list.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Indice analitico

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

_

-

.

/

#

$