Program Library HOWTO

David A. Wheeler

Il presente HOWTO per programmatori discute come utilizzare librerie di programma in ambiente Linux. Questo include librerie statiche, librerie condivise e librerie caricate dinamicamente.

Traduzione a cura di Riccardo Vianello <r_vianello@toglimi.hotmail.com>, revisione a cura di Alessio Rolleri <aroller@toglimi.libero.it>.

Sommario

1. Introduzione

Il presente HOWTO per programmatori discute come creare ed utilizzare librerie di programma in ambiente Linux utilizzando l'insieme di strumenti GNU. Una "libreria di programma" consiste semplicemente in un file contenente codice compilato (e dati) che viene successivamente incorporato in un programma; le librerie di programma consentono ai programmi di essere più modulari, più veloci da ricompilare e più semplici da aggiornare. Le librerie di programma possono essere divise in tre categorie: librerie statiche, librerie condivise e librerie a caricamento dinamico (DL, dall'inglese "dynamically loaded").

Questo articolo discute inizialmente le librerie statiche, le quali vengono installate in un programma eseguibile prima che il programma stesso possa essere mandato in esecuzione. Vengono successivamente discusse le librerie condivise, che vengono caricate all'avvio del programma e condivise tra i programmi. Infine, si discutono le librerie caricate dinamicamente (DL), le quali possono essere caricate ed utilizzate in ogni momento durante l'esecuzione di un programma. Le librerie dinamiche non corrispondono in realtà ad un differente formato di libreria (sia le librerie statiche che quelle condivise possono essere utilizzate come librerie a caricamento dinamico); piuttosto, la differenza sta in come le librerie dinamiche vengono utilizzate dai programmatori. L'HOWTO si conclude con una sezione contenente ulteriori esempi ed una sezione con riferimenti ad altre fonti di informazione.

La maggior parte dei programmatori intenzionati a sviluppare librerie dovrebbe creare librerie condivise, dal momento che queste consentono agli utenti di aggiornare le loro librerie separatamente dalle applicazioni che le utilizzano. Le librerie caricate dinamicamente sono utili, ma richiedono un certo lavoro in più per essere utilizzate e molti programmi non necessitano della flessibilità offerta da questo tipo di libreria. Al contrario, l'aggiornamento di librerie statiche risulta di gran lunga più complesso, tanto che un loro utilizzo generale risulta difficile da raccomandare. Detto questo, ogni categoria presenta dei vantaggi specifici; i pregi di ciascun tipo di libreria sono illustrati nella sezione dedicata. Gli sviluppatori che utilizzano il C++ e le librerie caricate dinamicamente dovrebbero inoltre consultare il "C++ dlopen mini-HOWTO".

Vale la pena di notare che alcuni utilizzano il termine DLL (dynamically *linked* libraries, cioè librerie *collegate* dinamicamente) per riferirsi alle librerie condivise, altri usano il termine DLL per indicare qualunque libreria che venga utilizzata come una libreria a caricamento dinamico e alcuni altri intendendo un tipo di libreria che corrisponde ad entrambi i significati. Indipendentemente da quale significato venga scelto, il presente HOWTO tratta le DLL in ambiente Linux.

Per quanto riguarda il formato di eseguibili e librerie, questo HOWTO discute unicamente il formato ELF (Executable and Linking Format), utilizzato attualmente dalla quasi totalità di distribuzioni Linux. L'insieme di strumenti GNU gcc può in realtà gestire formati di librerie diversi da ELF; in particolare la maggior parte di distribuzioni Linux può ancora utilizzare l'obsoleto formato a.out. In ogni caso, tali formati esulano dalla portata del presente articolo.

Se si deve implementare un'applicazione che deve essere portata su molti sistemi, in alternativa all'uso diretto degli strumenti di Linux, può essere preso in considerazione, al fine di compilare ed installare librerie, l'utilizzo di GNU libtool (http://www.gnu.org/software/libtool/libtool.html). GNU libtool è uno script di supporto alla produzione di librerie che maschera la complessità nell'utilizzo di librerie condivise (riguardo, ad esempio, alla creazione ed installazione delle stesse) dietro un'interfaccia consistente e portabile. Sotto Linux, GNU libtool è implementato sulla base degli strumenti e delle convenzioni descritte nel presente HOWTO. Per le librerie caricate dinamicamente è possibile utilizzare differenti strumenti che ne incapsulano le funzionalità dietro un'interfaccia portabile. GNU libtool include uno di questi strumenti, chiamato "libtdl". In alternativa, è possibile utilizzare la libreria glib (da non confondersi con glibc) con il suo supporto portabile al caricamento dinamico di moduli. È possibile reperire ulteriori informazioni riguardo a glib presso http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html. Ancora una volta, sotto Linux questa funzionalità è implementata utilizzando i costrutti descritti in questo HOWTO. Se si sta effettivamente sviluppando il codice in ambiente Linux si vorranno probabilmente avere a disposizione le informazioni contenute nel presente articolo.

La copia di riferimento di questo HOWTO è disponibile presso http://www.dwheeler.com/program-library, e fa parte del Linux Documentation Project (http://www.linuxdoc.org). È Copyright (C) 2000 di David A. Wheeler e se ne fornisce licenza d'uso secondo i termini della General Public License (GPL); si rimanda alla sezione conclusiva per ulteriori informazioni.

2. Librerie statiche

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

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

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

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

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

Una volta creata una libreria statica, la si vorrà probabilmente usare. È possibile utilizzare una libreria statica facendovi riferimento durante il processo di compilazione e link di un programma eseguibile. Nel caso in cui, per la creazione dell'eseguibile, si stia utilizzando `gcc(1)` è possibile allora utilizzare, al fine di specificare la libreria, l'opzione -l; si faccia riferimento a `info:gcc` per ulteriori informazioni.

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

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

3. Librerie condivise

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

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

3.1. Convenzioni

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

3.1.1. Nomi delle librerie condivise

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

Ogni libreria condivisa ha anche un "nome vero", che corrisponde al nome del file che contiene effettivamente il codice di libreria. Il nome vero aggiunge al soname un punto, un numero di versione secondario, un ulteriore punto e il numero di release. L'ultimo punto ed il numero di release sono opzionali. Il numero di versione secondario ed il numero di release sono di supporto al controllo di configurazione, consentendo di sapere esattamente quale o quali versioni della libreria siano state installate. Si noti che questi numeri potrebbero non coincidere con quelli utilizzati per descrivere la libreria nella documentazione, anche se quando coincidono le cose certamente si semplificano.

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

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

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

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

3.1.2. Posizionamento nel filesystem

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

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

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

3.2. Come le librerie vengono utilizzate

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

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

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

La ricerca attraverso tutte queste directory all'avvio del programma risulterebbe gravemente inefficiente, di conseguenza in realtà si utilizza un meccanismo di cache. Il normale comportamento del programma `ldconfig(8)` consiste nel leggere il file `/etc/ld.so.conf`, configurare gli appropriati link simbolici nelle directory (così che questi seguiranno le convenzioni standard) e infine scrivere una cache nel file `/etc/ld.so.cache` che viene quindi utilizzato dagli altri programmi. Questo velocizza enormemente l'accesso alle librerie. La conseguenza è che `ldconfig` deve essere eseguito ogni volta che una DLL viene aggiunta, quando una DLL viene rimossa o quando cambia l'insieme delle directory in cui effettuare la ricerca delle librerie; spesso quando viene installata una libreria uno dei compiti effettuati dai gestori di pacchetti consiste nell'esecuzione di ldconfig. All'avvio di un programma, quindi, il caricatore dinamico in realtà utilizza il file `/etc/ld.so.cache` e carica quindi le librerie di cui necessita.

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

3.3. Variabili di ambiente

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

3.3.1. LD_LIBRARY_PATH

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

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

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

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

3.3.2. LD_DEBUG

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

```
export LD_DEBUG=files
programma_da_eseguire
```

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

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

3.3.3. Altre variabili di ambiente

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

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

3.4. Creare una libreria condivisa

Creare una libreria condivisa è facile. Innanzitutto, si devono creare i file oggetto che andranno a far parte della libreria condivisa utilizzando le opzioni -fPIC o -fpic di `gcc`. Le opzioni -fPIC e -fpic abilitano la generazione di codice non dipendente dalla posizione (*"position independent code"*), un requisito per le librerie condivise; si veda oltre per le differenze fra le due opzioni. Il soname viene passato attraverso l'opzione -Wl di gcc. L'opzione -Wl inotra opzioni al linker (in questo caso -soname è quindi un'opzione per il linker); le virgole dopo -Wl non sono un errore di stampa e non si dovrebbero mai includere spazi (a meno di indicarli tramite una sequenza di escape) nel corpo di questa opzione. Si crea quindi una libreria condivisa utilizzando questo formato:

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

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

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

Ci sono alcuni punti degni di nota:

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

"-rdynamic" in alternativa a "-Wl,export-dynamic", ma in base alla documentazione del formato ELF non è sempre garantito il funzionamento dell'opzione "-rdynamic" di `gcc` su sistemi non Linux.

Durante lo sviluppo, esiste il potenziale problema di modificare una libreria che è utilizzata anche da molti altri programmi -- e che non si voglia che altri programmi utilizzino la libreria "di sviluppo", tranne solamente un particolare programma tramite il quale si effettuano procedure di test. Un'opzione di link che si potrebbe usare è l'opzione "rpath" di ld, che specifica il percorso di ricerca delle librerie a tempo di esecuzione per il particolare programma che si sta compilando. Da `gcc`, è possibile definire tale opzione specificandola nel modo seguente:

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

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

3.5. Installare ed utilizzare una libreria condivisa

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

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

```
ldconfig -n directory_con_librerie_condivise
```

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

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

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

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

Solitamente è possibile aggiornare le librerie senza troppe preoccupazioni; se ci sono state variazioni a livello di API, si suppone che il creatore della libreria ne abbia cambiato il soname. In questo modo, differenti versioni di una singola libreria possono coesistere in uno stesso sistema e quella corretta viene selezionata per ogni programma. Comunque, se un programma smette di funzionare in seguito all'aggiornamento di una libreria che ha mantenuto lo stesso soname, è possibile forzarlo ad utilizzare la vecchia versione di libreria facendo una copia della vecchia libreria da qualche parte, rinominando il programma (ad esempio con il vecchio nome seguito da ".orig"), e quindi sostituendolo con un breve script ("wrapper") che riassegna la libreria da utilizzare prima di chiamare il vero programma (precedentemente rinominato). Si può porre la vecchia libreria in una particolare locazione, se preferibile, anche se le convenzioni sulla numerazione permettono, in generale, la coesistenza di versioni differenti in una medesima directory. Lo script potrebbe avere un aspetto simile al seguente:

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

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

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

```
ldd /bin/ls
```

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

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

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

3.6. Librerie incompatibili

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

1. il comportamento di una funzione cambia così da non corrispondere più alle specifiche originali,
2. ci sono variazioni nelle strutture dati esportate (un'eccezione: aggiungere attributi opzionali in fondo a strutture può essere accettabile a condizione che tali strutture vengano allocate unicamente all'interno della libreria stessa),

3. viene rimossa una funzione precedentemente esportata,
4. l'interfaccia di una funzione esportata viene modificata.

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

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

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

Data la lunga lista, gli sviluppatori di librerie in C++ dovranno pianificare lo sviluppo con particolare attenzione se vorranno minimizzare gli aggiornamenti che ne possano compromettere la compatibilità a livello binario. Fortunatamente, nei sistemi di tipo Unix (Linux incluso) si possono caricare ed utilizzare contemporaneamente differenti versioni di una stessa libreria, così che, anche se con qualche penalizzazione in termini di occupazione dello spazio disco, gli utenti possono continuare ad eseguire "vecchi" programmi che richiedono le vecchie librerie.

4. Librerie caricate dinamicamente

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

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

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

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

4.1. dlopen()

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

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

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

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

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

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

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

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

4.2. dlerror()

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

4.3. dlsym()

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

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

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

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

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

4.4. dlclose()

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

4.5. Esempio di libreria a caricamento dinamico

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

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

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

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
    if (!handle) {
```

```
        fputs (dlerror(), stderr);
        exit(1);
    }

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

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

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

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

5. Miscellanea

5.1. Il comando nm

Il comando `nm(1)` può mostrare la lista dei simboli in una data libreria. Funziona sia con librerie statiche che condivise. Per la libreria indicata `nm(1)` può elencare i nomi dei simboli definiti, il valore di ciascun simbolo ed il corrispondente tipo. È inoltre in grado di indicare dove il simbolo era definito nel codice sorgente (tramite nome del file e numero di linea), se questa informazione è disponibile nella libreria stessa (si veda a questo proposito l'opzione -l).

Il tipo associato al simbolo richiede qualche ulteriore spiegazione. Il tipo è visualizzato tramite una lettera; una lettera minuscola significa che il simbolo è locale, mentre una lettera maiuscola significa che il simbolo è globale (a collegamento esterno). Solitamente i tipi associabili ad un simbolo comprendono: T (una normale definizione nella sezione di codice), D (sezione dati inizializzata), B (sezione dati non inizializzata), U (non definito; il simbolo è utilizzato dalla libreria, ma non è definito dalla libreria stessa), e W (debole; se anche un'altra libreria dovesse definire questo simbolo, tale definizione avrebbe priorità su questa).

Se si conosce il nome di una funzione, ma non ci si ricorda in quale libreria fosse definita, si può utilizzare l'opzione -o di `nm` (che visualizza il nome del file all'inizio di ogni linea) assieme ad un grep per trovare il nome della libreria. Gli utenti di bash, ad esempio, possono ricercare la funzione "cos" in tutte le librerie in `/lib`, in `/usr/lib` comprese le sue immediate sottodirectory e in `/usr/local/lib` con il seguente comando:

```
nm -o /lib/* /usr/lib/* /usr/lib/*/* \
      /usr/local/lib/* 2> /dev/null | grep 'cos$'
```

Informazioni molto più dettagliate su `nm` si possono trovare nella corrispondente documentazione "info" installata localmente sotto: info:binutils#nm.

5.2. Le funzioni costruttore e distruttore di una libreria

Le librerie dovrebbero esportare le procedure di inizializzazione e terminazione utilizzando gli attributi di funzione `__attribute__((constructor))` ed `__attribute__((destructor))` di `gcc`. Si veda a questo proposito la documentazione di `gcc`. Le funzioni costruttore vengono chiamate prima del ritorno dalla chiamata a dlopen (o prima che venga eseguita la funzione main() se la libreria viene caricata all'avvio del programma). Le funzioni distruttore vengono eseguite prima del ritorno della chiamata a dlclose (o dopo exit() o al termine dell'esecuzione di main() se la libreria viene caricata all'avvio del programma). I prototipi C per queste funzioni sono:

```
void __attribute__((constructor)) mia_init(void);
void __attribute__((destructor)) mia_fini(void);
```

Le librerie condivise non dovrebbero essere compilate facendo uso delle opzioni "-nostartfiles" o "-nostdlib" di `gcc`. Se questo avvenisse le procedure di costruzione/distruzione non verrebbero chiamate (a meno che non si applichino particolari accorgimenti).

5.2.1. Le speciali funzioni _init e _fini (OBSOLETO/PERICOLOSO)

Storicamente sono esistite due particolari funzioni, _init e _fini, utilizzabili nel controllo dell'inizializzazione e terminazione di una libreria. Ad ogni modo, questo meccanismo è oggi obsoleto e l'uso di queste funzioni può portare a risultati non predicibili. Le vostre librerie non ne dovrebbero quindi fare uso; si utilizzino piuttosto gli attributi `constructor` e `destructor` descritti in precedenza.

Se si dovesse lavorare su vecchi sistemi o su vecchio codice che utilizzano _init o _fini, ecco un'illustrazione di come funzionavano: erano definite due speciali funzioni per l'inizializzazione e terminazione di un modulo: _init e _fini. Se una libreria esporta una funzione "_init", questa viene chiamata la prima volta che viene caricata (tramite dlopen() o semplicemente all'avvio del programma, se si tratta di una libreria condivisa). In un programma C, questo significa semplicemente aver definito una qualche funzione chiamata _init. Esiste una corrispondente funzione chiamata _fini, che viene chiamata ogniqualvolta l'uso di una libreria termina (tramite una chiamata a dlclose() che ne porta il conteggio dei riferimenti a zero, o alla normale terminazione del programma). I prototipi C di queste funzioni sono:

```
void _init(void);
void _fini(void);
```

In questo caso, nel compilare il file sorgente in un file ".o" con `gcc`, ci si deve assicurare di aggiungere l'opzione "-nostartfiles". Questo evita che il compilatore C colleghi librerie di avvio di sistema al file ".so". In caso contrario si otterrebbero errori dovuti a definizioni multiple. Si noti che questo è completamente diverso dal compilare un modulo utilizzando gli attributi di funzione indicati. Si ringraziano Jim Mischel e Tim Gentry per il suggerimento di aggiungere questa discussione su _init e _fini e per l'assistenza nel comporla.

5.3. Le librerie condivise possono essere script

Vale la pena di notare che il caricatore GNU permette alle librerie condivise di essere comuni file di testo che utilizzano uno speciale linguaggio di scripting in luogo del consueto formato di libreria. Questo può risultare utile per combinare indirettamente altre librerie. Per esempio, questo è il listato di /usr/lib/libc.so su uno dei miei sistemi:

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
GROUP ( /lib/libc.so.6 /usr/lib/libc_nonshared.a )
```

(Il commento presente nel listato indica che preferibilmente verrà utilizzata la libreria condivisa /lib/libc.so.6, ma che dal momento che alcune funzionalità sono presenti solo nella versione statica /usr/lib/libc_nonshared.a quest'ultima verrà utilizzata nei casi in cui la prima non fosse sufficiente. NDT) Per ulteriori informazioni a questo proposito si rimanda alla documentazione texinfo relativa agli script per il linker ld (ld command language). Informazioni generali si trovano in info:ld#Options and info:ld#Commands, mentre i comandi di uso più comune sono discussi in info:ld#Option Commands.

5.4. Versione dei simboli e script di versione

Tipicamente i riferimenti a funzioni esterne vengono collegati quando necessario e non vengono quindi tutti collegati all'avvio del programma. Se una libreria condivisa non fosse aggiornata, qualche porzione dell'interfaccia richiesta potrebbe mancare; se l'applicazione tentasse di utilizzarla potrebbe quindi improvvisamente ed inaspettatamente fallire.

Una soluzione a questo problema consiste nel controllo di versione dei simboli abbinato a script di versione. Con il controllo di versione dei simboli l'utente può ricevere dei messaggi di avvertimento all'avvio dei programmi quando le librerie in uso dovessero risultare troppo vecchie. È possibile trovare ulteriori informazioni su questo argomento nella discussione degli script di versione contenuta nel manuale di ld e reperibile presso http://www.gnu.org/manual/ld-2.9.1/html_node/ld_25.html.

5.5. GNU libtool

Se si sta sviluppando un'applicazione che dovrà essere portata su diverse piattaforme, si può prendere in considerazione l'uso di GNU libtool (http://www.gnu.org/software/libtool/libtool.html) per la compilazione e l'installazione delle librerie. GNU libtool consiste in uno script generico di supporto all'uso di librerie. Libtool nasconde la complessità d'uso di librerie condivise dietro un'interfaccia consistente e portabile. Libtool fornisce un'interfaccia indipendente dalla piattaforma per creare file oggetto, produrre librerie (statiche e condivise), produrre ed eseguire il debug di eseguibili, installare librerie ed eseguibili. È incluso anche libltdl, che fornisce la portabilità ai i programmi con caricamento dinamico. Per maggiori informazioni si consulti la relativa documentazione presso http://www.gnu.org/software/libtool/manual.html

5.6. Rimuovere i simboli per risparmiare spazio

Tutti i simboli inclusi nei file generati risultano utili per il debug, ma incrementano le dimensioni dei file stessi. Se si dovessero avere problemi di spazio, è possibile eliminarne una parte.

L'approccio migliore consiste nel generare i file oggetto nel modo consueto ed eseguire in primo luogo le necessarie procedure di debug e verifica (che risultano fortemente agevolate dalla presenza dei simboli). Successivamente, una volta completata la verifica del programma, si usi strip(1) per rimuovere i simboli. Il comando strip(1) fornisce un buon grado di controllo su quali simboli eliminare; si consulti la documentazione a riguardo per una dettagliata descrizione.

Un differente approccio consiste nell'uso uso delle opzioni "-S" e "-s" del linker GNU `ld`; "-S" omette dal file prodotto in output le informazioni relative ai simboli di debug (ma non tutti i simboli), mentre "-s" omette tutti i simboli. È possibile attivare queste opzioni attraverso il compilatore `gcc` con "-Wl,-S" e "-Wl,-s". Se eliminare i simboli rappresenta la procedura normalmente applicata e queste opzioni si rivelano sufficienti allo scopo, questo metodo può essere utilizzato liberamente, ma si tratta di un approccio meno flessibile.

5.7. Eseguibili estremamente piccoli

L'articolo Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux (http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html) potrebbe rivelarsi utile. Descrive come sia possibile produrre un eseguibile di dimensioni estremamente ridotte. Parlando francamente, la maggior parte dei trucchi descritti non dovrebbero essere utilizzati nelle normali circostanze in cui generalmente si opera, ma risultano piuttosto istruttivi, illustrando l'effettivo funzionamento del formato ELF.

5.8. C++ vs. C

Vale la pena di ricordare che se si sta scrivendo un programma in C++, e da questo si sta chiamando una funzione di libreria implementata in C, il codice C++ dovrà dichiarare tale funzione come `extern "C"`. In caso contrario il linker non sarà in grado di localizzare la funzione C. Internamente, i compilatori C++ effettuano una "decorazione" (mangle) dei nomi delle funzioni C++ (ad esempio per necessità legate al riconoscimento dei tipi), e devono quindi essere informati del fatto che una determinata funzione deve essere chiamata come funzione C (e quindi priva di decorazione del nome).

Se si sta sviluppando una libreria di programma che potrebbe essere chiamata da C o C++ è raccomandabile includere delle dichiarazioni `extern "C"` nei file di intestazione così da predisporli automaticamente per gli utenti. Queste dichiarazioni possono essere abbinate alle normali direttive `#ifndef` necessarie ad evitare l'inclusione ripetuta di uno stesso file di intestazione. In questo modo il contenuto tipico di un generico file `pippo.h`, utilizzabile sia da C che da C++, avrà un aspetto simile a questo:

```
/* Spiegare qui cosa fa 'pippo' */

#ifndef PIPPO_H
#define PIPPO_H

#ifdef __cplusplus
extern "C" {
#endif

  ... Qui vanno le dichiarazioni delle funzioni esportate ...

#ifdef  __cplusplus
}
#endif
#endif
```

5.9. Velocizzare l'inizializzazione di codice C++

Gli sviluppatori di KDE hanno notato che l'avvio di applicazioni di grosse dimensioni, scritte in C++ e dotate di interfaccia grafica, può talvolta richiedere un lungo intervallo di tempo, in parte dovuto a numerose riallocazioni. Esistono numerose soluzioni a questo inconveniente. Si veda Making C++ ready for the desktop (by Waldo Bastian) (http://www.suse.de/~bastian/Export/linking.txt) per ulteriori informazioni.

5.10. Linux Standard Base (LSB)

Lo scopo del progetto Linux Standard Base (LSB) consiste nello sviluppare e promuovere un insieme di normative standardizzate che incrementino la compatibilità tra le differenti distribuzioni di Linux e consentano l'esecuzione delle applicazioni su ogni sistema Linux conforme allo standard. La home page del progetto è all'indirizzo http://www.linuxbase.org.

Un interessante articolo che riassume come sviluppare applicazioni conformi allo standard LSB è stato pubblicato da George Kraft IV (Senior software engineer, IBM's Linux Technology Center) nell'ottobre 2002, Developing LSB-certified applications: Five steps to binary-compatible Linux applications (http://www-106.ibm.com/developerworks/linux/library/l-lsb.html?t=gr,lnxw02=LSBapps). Chiaramente, se si desidera che le applicazioni risultino portabili, si dovrà sviluppare del codice che acceda unicamente al livello di interfaccia standardizzato;. LSB fornisce inoltre agli sviluppatori di applicazioni C/C++ alcuni strumenti per la verifica della conformità allo standard; questi strumenti utilizzano alcune possibilità del linker e speciali librerie al fine di effettuare i test necessari. Ovviamente, per effettuare questo tipo di verifica si dovranno installare questi strumenti, che possono essere reperiti tramite il sito web di LSB. Fatto questo, è sufficiente utilizzare "`lsbcc`" come compilatore C/C++ (`lsbcc` crea internamente un ambiente di link che produrrà degli errori nel caso in cui determinate regole di conformità allo standard LSB non fossero soddisfatte):

```
$ CC=lsbcc make mia_applicazione
 (oppure)
$ CC=lsbcc ./configure; make mia_applicazione
```

Il programma `lsbappchk` permette di verificare che l'applicazione utilizzi solo funzioni previste dallo standard LSB:

```
$ lsbappchk mia_applicazione
```

È inoltre necessario attenersi alle linee guida di LSB per quanto concerne i pacchetti di installazione (ad esempio utilizzare il formato RPM v3 e nomi dei pacchetti conformi allo standard; LSB prevede inoltre che il software aggiuntivo debba essere normalmente installato sotto `opt`). Si vedano il suddetto articolo ed il sito internet di LSB per ulteriori informazioni.

5.11. Riunire più librerie in un'unica libreria

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

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

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

Come messo in evidenza dalla documentazione di `ld`, ci si assicuri di utilizzare alla fine l'opzione --no-whole-archive altrimenti `gcc` cercherà di riunire nella libreria in output anche le librerie standard. Si ringrazia Kendall Bennett per aver suggerito l'aggiunta di questa ricetta e per averla fornita.

6. Ulteriori esempi

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

6.1. File libhello.c

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

#include <stdio.h>

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

6.2. File libhello.h

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


void hello(void);
```

6.3. File demo.c

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

#include "libhello.h"

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

```
}
```

6.4. File script_static

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

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

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

# Crea la libreria statica.

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

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

# Compilazione del file di programma demo.

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

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

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

# Esecuzione del programma.

./demo_static
```

6.5. File script_shared

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

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

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

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

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

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

# Ora dobbiamo chiamare ldconfig per sistemare i link simbolici.

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

/sbin/ldconfig -n .

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

ln -sf libhello.so.0 libhello.so

# Compilazione del file di programma demo.

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

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

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

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

LD_LIBRARY_PATH="." ./demo
```

6.6. File demo_dynamic.c

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


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

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

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

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

typedef void (*simple_demo_function)(void);


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

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

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

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

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

```
}
```

6.7. File script_dynamic

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

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

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

gcc -Wall -g -c demo_dynamic.c

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

gcc -g -o demo_dynamic demo_dynamic.o -ldl


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

LD_LIBRARY_PATH="." ./demo_dynamic
```

7. Altre fonti di informazione

Fra le principali fonti di informazione, relative all'uso di librerie, vanno incluse le seguenti:

- "The GCC HOWTO" di Daniel Barlow. In particolare, questo HOWTO discute le opzioni di compilazione necessarie alla creazione di librerie e come effettuare ricerche all'interno di librerie. Comprende informazioni non contemplate dal presente documento e viceversa. Questo HOWTO è reperibile tramite il Linux Documentation Project presso http://www.linuxdoc.org.

- "Executable and Linkable Format (ELF)" del comitato per i Tool Interface Standards (TIS) (si tratta in effetti di un capitolo del Portable Formats Specification Version 1.1 edito dallo stesso comitato). Fornisce informazioni dettagliate sul formato ELF (questo non riguarda in modo specifico Linux o il compilatore GNU gcc). Si veda ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz. Se si ottiene il file dal MIT, si noti che si tratta di un formato insolito; dopo aver decompresso ed estratto l'archivio, si otterrà un file "`.hps`"; è sufficiente eliminare le linee in cima ed in fondo al file e rinominarlo in "`.ps`" per ottenere un file stampabile, in formato Postscript, con la consueta estensione.
- "ELF: From the Programmer's Perspective" di Hongjui Lu. Questo documento fornisce informazioni sul formato ELF specifiche per Linux ed il compilatore GNU `gcc` ed è reperibile presso: ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz.
- La documentazione di `ld` "Using LD, the GNU Linker" descrive il linker GNU in maniera molto più dettagliata di quanto possibile nel presente documento. È disponibile presso: http://www.gnu.org/manual/ld-2.9.1.
- Si dovrebbe inoltre consultare la normale documentazione in formato "info", in particolare per `ld` e `gcc`.

8. Copyright e licenza

Queste condizioni consentono il *mirroring* presso altri siti web, ma per cortesia:

- ci si assicuri che la copia venga automaticamente aggiornata tramite il sito principale,
- si mostri chiaramente la locazione del sito di riferimento, http://www.dwheeler.com/program-library, con un collegamento ipertestuale al sito principale, e
- mi si citi (David A. Wheeler) come l'autore.

I primi due punti principalmente mi proteggono dal dover sentir citare ripetutamente errori obsoleti. Non voglio sentir citare errori che ho corretto un anno fa solo per via del fatto che un vostro mirroring non è gestito in modo appropriato. Con un link al sito di riferimento gli utenti possono accertarsi che la copia sia aggiornata. Sono sensibile ai problemi a cui vanno incontro siti sottoposti a forti requisiti di sicurezza e che quindi non possono fornire una normale connessione ad Internet. Se questo rappresenta il vostro caso cercate almeno di attenervi agli altri punti e tentate periodicamente di far "sgattaiolare" un qualche aggiornamento all'interno del vostro ambiente.

Questa licenza vi consente di modificare il documento, ma non di dichiarare come vostro ciò che non avete scritto (vale a dire, non è consentito il plagio), nè di dichiarare che una versione modificata sia identica all'originale. Modificare il documento non trasferisce interamente a voi i diritti d'autore sull'opera; nei termini di legge relativi al diritto d'autore quest'opera non è di "dominio pubblico". Si veda il testo integrale della licenza per ulteriori dettagli, in particolare si noti che è necessario includere nei file modificati annotazioni evidenti del fatto che tali file siano stati modificati da voi ed in quale data questo sia avvenuto. In caso di dubbi a proposito di cosa la licenza consenta, gradirei essere contattato. Nella maggior parte dei casi la cosa migliore consiste nell'inviare le modifiche a chi si occupa di mantenere la copia principale (attualmente David A. Wheeler), così che le vostre modifiche verranno integrate a quelle di tutti gli altri nella copia ufficiale.