Strumenti di sviluppo

G. Lettieri

8 Marzo 2023

Per ottenere un programma eseguibile a partire da dei file sorgenti è necessario passare attraverso diversi strumenti, di cui vogliamo ora capire lo scopo e il funzionamento interno.

Nel seguito faremo riferimento al piccolo esempio di programma misto contenuto nelle Figure 1, 2 e 3. Gli strumenti `g++`, `as`, ... possono essere usati direttamente solo su macchine con architettura Intel/AMD64. Per i Mac Silicon si veda l'appendice.

1 Il preprocessore

Il preprocessore si preoccupa di interpretare le direttive che si trovano nelle linee che iniziano con `#`, espandere eventuali *macro* ed eliminare commenti e spazi superflui.

La direttiva più comune è **`#include`**, che può essere seguita da un nome di file tra doppi apici o tra parentesi angolate. Quando il preprocessore incontra questa direttiva la sostituisce con tutto il contenuto del file. Nel caso di doppi apici, il preprocessore cerca il file prima nella stessa directory in cui si trova il file corrente e (se non lo trova), in una serie di directory di sistema (per es., `/usr/include`). Nel caso di parentesi angolate il preprocessore cerca solo nelle directory di sistema.

Le macro possono essere definite con la direttiva **`#define`**. Per esempio, **`#define`** `PI 3.14` definisce la macro `PI` e le assegna il testo `3.14`. Da questa linea in poi il preprocessore sostituirà ogni occorrenza della parola `PI` con `3.14`.

Mentre le macro sono essenziali in C, in C++ si tende ad evitarle in quanto il linguaggio dispone di costrutti migliori per molti dei loro utilizzi più tipici. Sono però ancora usate in alcuni casi, come in quello della *compilazione condizionale* illustrato alle righe 6–10. L'idea è che quando è definita la macro `DEBUG` il codice deve contenere la definizione di una funzione `debug()` (che nell'esempio è stata lasciata vuota per motivi di spazio, ma che in generale potrebbe inviare il messaggio sul terminale o scriverlo in un file di log). Se la macro `DEBUG` non è definita, la definizione della funzione `debug()` non appare nel file e, invece, viene definita una *macro* `debug()` la cui sostituzione è una stringa vuota (si noti la **`#define`** alla riga 9 di Figura 1). A questo punto si può far apparire la funzione `debug()` durante lo sviluppo e farla sparire completamente nel

```
1  #include "lib.h"
2
3  long var1 = 8;
4  long var2 = 4;
5
6  #ifdef DEBUG
7  void debug(const char *msg) {}
8  #else
9  #define debug(msg)
10 #endif
11
12 int main()
13 {
14         // un commento
15         var1 =    foo(var2);
16         debug("questo e' un messaggio di debug");
17         return var1;
18 }
```

Figura 1: file main.cpp

```
1  long foo(long);
2  void bar();
```

Figura 2: file lib.h

```
1  .data
2  foovar1:
3          .quad 5
4  foovar2:
5          .quad 6
6  .bss
7  foovar3:
8          .quad 0
9  .text
10 .global _Z3fool
11 _Z3fool:
12         pushq %rbp
13         movq %rsp, %rbp
14         movq %rdi, %rax
15         movabsq foovar3, %rax
16         addq foovar1, %rax
17         addq foovar2(%rip), %rax
18         leave
19         ret
```

Figura 3: file foo.s

```
1  # 1 "main.cpp"
2  # 1 "<built-in>"
3  # 1 "<command-line>"
4  # 1 "/usr/include/stdc-predef.h" 1 3 4
5  # 1 "<command-line>" 2
6  # 1 "main.cpp"
7  # 1 "lib.h" 1
8  long foo(long);
9  void bar();
10 # 2 "main.cpp" 2
11 long var1 = 8;
12 long var2 = 4;
13 int main()
14 {
15  var1 = foo(var2);
16  ;
17  return var1;
18 }
```

Figura 4: L'output di `g++ -E main.cpp` (con alcune righe vuote eliminate per questioni di spazio).

programma finale, semplicemente definendo o non definendo la macro `DEBUG`. La definizione può essere fatta anche nel momento in cui si compila, passando l'opzione `-D DEBUG` al compilatore. Con `-D DEBUG=`*qualcosa* è anche possibile assegnare un valore alla macro. Nel preprocessore GNU il test alla riga 6 di Figura 1 risulta vero anche se la macro non ha valore, purché sia definita, quindi è sufficiente `-D DEBUG`.

Con il comando `g++ -E main.cpp` possiamo osservare l'output del preprocessore (Figura 4). Si noti che la linea 1 di Figura 1 è stata sostituita dal contenuto del file `lib.h` alle linee 8 e 9 di Figura 4 (sono state aggiunte anche le linee 1–7 e 10, che servono al compilatore per emettere eventuali errori di sintassi in maniera più precisa). È sparito il commento che si trovava alla linea 14 di Figura 1 e gli spazi all'interno di ciascuna riga sono stati ridotti all'essenziale. Inoltre, sono completamente sparite le righe 6–10 e l'invocazione della funzione `debug()` alla riga 16 di Figura 1 (nella corrispondente riga 16 di Figura 4 è rimasto solo il punto e virgola). In Figura 5, invece, si vede l'output del preprocessore quando è stata definita la macro `DEBUG`. Rispetto alla Figura 4 notiamo che sono rimaste visibili la definizione della funzione `debug()` alla riga 13 e la sua invocazione alla riga 16.

La compilazione condizionale è spesso usata anche per creare programmi che si possano adattare a diversi sistemi operativi e/o architetture. Per questo scopo il preprocessore parte con un certo numero di macro già definite. Per esempio, sui sistemi Linux è sempre definita la macro `linux`. La lista di tutte le macro

```
1  # 1 "main.cpp"
2  # 1 "<built-in>"
3  # 1 "<command-line>"
4  # 1 "/usr/include/stdc-predef.h" 1 3 4
5  # 1 "<command-line>" 2
6  # 1 "main.cpp"
7  # 1 "lib.h" 1
8  long foo(long);
9  void bar();
10 # 2 "main.cpp" 2
11 long var1 = 8;
12 long var2 = 4;
13 void debug(const char *msg) {}
14 int main()
15 {
16  var1 = foo(var2);
17  debug("questo e' un messaggio di debug");
18  return var1;
19 }
```

Figura 5: L'output di `g++ -D DEBUG -E main.cpp` (con alcune righe vuote eliminate per questioni di spazio).

definite può essere ottenuta passando le due opzioni `-E` e `-dM` al comando `g++` (è necessario comunque il nome di un file sorgente esistente, anche se vuoto).

2 Il compilatore

Il compilatore C++ (o C) riceve un unico file in ingresso e produce un unico file assembler in uscita. È uno strumento molto sofisticato che richiederebbe da solo un intero corso. Tramite gli esercizi di traduzione da C++ ad assembler, in cui ci sostituiamo al compilatore, possiamo farci un idea di come deve operare.

La cosa che più ci interessa in questo momento è osservare che il compilatore vede esclusivamente il contenuto del file di Figura 4. L'inclusione del file `lib.h` serve a fare in modo che il compilatore veda la dichiarazione della funzione `foo`, che è l'unica cosa di cui ha bisogno per poter tradurre la linea 17.

Per semplicità traduciamo a mano il file `main.cpp` ottenendo il file in Figura 6. Per semplificare ulteriormente il contenuto dei vari file oggetto, abbiamo anche definito direttamente l'etichetta `_start` al posto di `main`.

```
1  .data
2  .global var1, var2
3  var1:
4          .quad   8
5  var2:
6          .quad   4
7  .text
8  .global _start
9  _start:
10         pushq   %rbp
11         movq    %rsp, %rbp
12         movq    var2(%rip), %rax
13         movq    %rax, %rdi
14         call    _Z3fool
15         movq    %rax, var1(%rip)
16         movq    var1(%rip), %rax
17         popq    %rbp
18         ret
```

Figura 6: file main.s

3 L'assemblatore

Nel nostro esempio l'assemblatore entra in gioco sia per tradurre l'output prodotto dal compilatore, sia per tradurre il file `foo.s` che è scritto direttamente in assembler. Anche l'assemblatore lavora esclusivamente osservando un file alla volta.

Lo scopo dell'assemblatore è di generare il contenuto di *sezioni* della memoria del programma. Una sezione è denominata **`.text`** ed è destinata a contenere il codice del programma, e un'altra è denominata **`.data`** ed è destinata a contenere i dati globali. Una terza sezione, denominata **`.bss`**, è destinata a contenere dati globali che devono essere inizializzati con zero. Quest'ultima sezione può essere considerata una ottimizzazione, in quanto i file oggetto e l'eseguibile finale possono limitarsi a dire quanto deve essere grande, senza realmente occupare spazio nel file. Ci possono essere altre sezioni, anche definite dall'utente.

L'assemblatore lavora senza sapere (in genere) a quale indirizzo le sezioni verranno poi caricate. Questo comporta che in molti casi la sua traduzione è incompleta e deve essere completata successivamente, dal collegatore o dal caricatore. L'assemblatore genera anche una serie di tabelle contenenti le informazioni necessarie al collegatore (o caricatore) per completare la traduzione: la *tabella delle sezioni*, la *tabella dei simboli* (locali e globali) e la *tabella di rilocazione*.

Per ogni sezione l'assemblatore mantiene un *contatore*, inizialmente pari a zero. Mentre legge il file sorgente, l'assemblatore fa riferimento sempre ad una

sezione corrente, che è l'ultima che è stata nominata (o `.text`, se non è stata nominata nessuna). Ogni comando, come **`.quad`** `5` oppure **`movq`** `%`**`rsp`**`,%`**`rbp`**, corrisponde ad una richiesta di aggiungere un certo numero di byte nella sezione corrente (avanzando di conseguenza il contatore). Il comando **`.quad`** `5` aggiunge 8 byte contenenti la rappresentazione binaria di 5, mentre il comando **`movq`** `%`**`rsp`**`,%`**`rbp`** aggiunge i byte necessari a codificare questa istruzione in linguaggio macchina. Si noti che ogni comando può far parte di qualunque sezione: il risultato è comunque l'aggiunta dei corrispondenti byte alla sezione.

La definizione di una etichetta, come alle linee 2, 4 e 7 e 11 di Figura 3, corrisponde alla richiesta di aggiungere un nuovo simbolo alla tabella dei simboli. Per ogni simbolo l'assemblatore deve sapere il nome, la sezione a cui appartiene e il valore. La linea 2 aggiunge il simbolo di nome "`foovar1`" appartenente alla sezione **`.data`** (che è quella corrente). Il valore di un simbolo è il valore del contatore della sezione corrente nel momento in cui il simbolo viene introdotto: rappresenta, dunque, l'offset rispetto alla base della sezione del primo byte che verrà aggiunto dopo il simbolo. Nel caso di `foovar1` si tratta dunque dell'offset del quad 5 introdotto alla linea 3. L'effetto complessivo è quello di dare un nome ad una certa locazione di memoria (anche se l'indirizzo di questa locazione non è ancora noto).

L'assemblatore lavora concettualmente in due "passate" (letture del file sorgente): una prima passata per raccogliere tutte le definizioni dei simboli e una seconda per effettuare la traduzione vera e propria. Questo perché una istruzione (come un salto in avanti) può riferire un simbolo che è stato definito più avanti nel file.

Alla linea 15 vediamo un caso in cui l'assemblatore non può completare la traduzione: l'istruzione ha bisogno dell'indirizzo completo di `foovar3`, ma l'assemblatore non lo conosce (perché non sa dove verrà caricata la sezione **`.bss`**). In questo caso l'assemblatore usa temporanemente l'indirizzo zero, ma aggiunge una nuova entrata alla *tabella di rilocazione*. Questa tabella contiene le istruzioni che permetteranno al collegatore di inserire l'indirizzo `foovar3` una volta noto. Analogamente per le istruzioni alle righe 16 e 17.

3.1 Il formato ELF

Vediamo ora in concreto cosa l'assemblatore GNU produce su Linux dopo aver assemblato il file `foo.s`. Il risultato è un file oggetto in formato ELF (Executable and Linking Format), uno standard adottato da molti sistemi Unix e che è in grado di contenere sia file oggetto, sia eseguibili, sia librerie dinamiche (che non vedremo).

Per assemblare il file `foo.s` usiamo il comando

```
as -o foo.o foo.s
```

Ottenendo il file `foo.o`. I file in Unix e (Linux) sono sempre solo sequenze di byte il cui significato dipende da varie convenzioni. Possiamo esaminare il contenuto di qualunque file con il comando `hexdump` (Figura 7). Ogni riga

```
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  e0 01 00 00 00 00 00 00  |................|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 08 00 07 00  |....@.....@.....|
00000040  55 48 89 e5 48 89 f8 48  a1 00 00 00 00 00 00 00  |UH..H..H........|
00000050  00 48 03 04 25 00 00 00  00 48 03 05 00 00 00 00  |.H..%....H......|
00000060  c9 c3 05 00 00 00 00 00  00 00 06 00 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000090  00 00 00 00 03 00 01 00  00 00 00 00 00 00 00 00  |................|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 03 00  |................|
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 00 00 03 00 04 00  00 00 00 00 00 00 00 00  |................|
000000d0  00 00 00 00 00 00 00 00  01 00 00 00 00 00 03 00  |................|
000000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000000f0  09 00 00 00 00 00 03 00  08 00 00 00 00 00 00 00  |................|
00000100  00 00 00 00 00 00 00 00  11 00 00 00 00 00 04 00  |................|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000120  19 00 00 00 10 00 01 00  00 00 00 00 00 00 00 00  |................|
00000130  00 00 00 00 00 00 00 00  00 66 6f 6f 76 61 72 31  |.........foovar1|
00000140  00 66 6f 6f 76 61 72 32  00 66 6f 6f 76 61 72 33  |.foovar2.foovar3|
00000150  00 5f 5a 33 66 6f 6f 6c  00 00 00 00 00 00 00 00  |._Z3fool........|
00000160  09 00 00 00 00 00 00 00  01 00 00 00 03 00 00 00  |................|
00000170  00 00 00 00 00 00 00 00  15 00 00 00 00 00 00 00  |................|
00000180  0b 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
00000190  1c 00 00 00 00 00 00 00  02 00 00 00 02 00 00 00  |................|
000001a0  04 00 00 00 00 00 00 00  00 2e 73 79 6d 74 61 62  |..........symtab|
000001b0  00 2e 73 74 72 74 61 62  00 2e 73 68 73 74 72 74  |..strtab..shstrt|
000001c0  61 62 00 2e 72 65 6c 61  2e 74 65 78 74 00 2e 64  |ab..rela.text..d|
000001d0  61 74 61 00 2e 62 73 73  00 00 00 00 00 00 00 00  |ata..bss........|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000220  20 00 00 00 01 00 00 00  06 00 00 00 00 00 00 00  | ...............|
00000230  00 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000240  22 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |"...............|
00000250  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000260  1b 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000270  00 00 00 00 00 00 00 00  60 01 00 00 00 00 00 00  |........`.......|
00000280  48 00 00 00 00 00 00 00  05 00 00 00 01 00 00 00  |H...............|
00000290  08 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00  |................|
000002a0  26 00 00 00 01 00 00 00  03 00 00 00 00 00 00 00  |&...............|
000002b0  00 00 00 00 00 00 00 00  62 00 00 00 00 00 00 00  |........b.......|
000002c0  10 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002d0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000002e0  2c 00 00 00 08 00 00 00  03 00 00 00 00 00 00 00  |,...............|
000002f0  00 00 00 00 00 00 00 00  72 00 00 00 00 00 00 00  |........r.......|
00000300  08 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000310  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000320  01 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  |................|
00000330  00 00 00 00 00 00 00 00  78 00 00 00 00 00 00 00  |........x.......|
00000340  c0 00 00 00 00 00 00 00  06 00 00 00 07 00 00 00  |................|
00000350  08 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00  |................|
00000360  09 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
00000370  00 00 00 00 00 00 00 00  38 01 00 00 00 00 00 00  |........8.......|
00000380  21 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |!...............|
00000390  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000003a0  11 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000003b0  00 00 00 00 00 00 00 00  a8 01 00 00 00 00 00 00  |................|
000003c0  31 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |1...............|
000003d0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000003e0
```

Figura 7: L'output di `hexdump -C foo.o`.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          480 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         8
  Section header string table index: 7
```

Figura 8: L'output di `readelf -h foo.o` (intestazione).

```
There are 8 section headers, starting at offset 0x1e0:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 000022 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 000160 000048 18   I  5   1  8
  [ 3] .data             PROGBITS        0000000000000000 000062 000010 00  WA  0   0  1
  [ 4] .bss              NOBITS          0000000000000000 000072 000008 00  WA  0   0  1
  [ 5] .symtab           SYMTAB          0000000000000000 000078 0000c0 18      6   7  8
  [ 6] .strtab           STRTAB          0000000000000000 000138 000021 00      0   0  1
  [ 7] .shstrtab         STRTAB          0000000000000000 0001a8 000031 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Figura 9: L'output di `readelf -WS foo.o` (tabella delle sezioni).

mostra 16 byte del file, in esadecimale. La prima colonna contiene l'offset del primo byte della riga (in esadecimale), mentre le 16 colonne successive mostrano i byte in questione. L'ultima colonna (tra barre verticali) mostra il carattere ASCII che corrisponde ad ogni byte della riga, quando possibile. Il carattere "." indica che il byte corrispondente non contiene un codice ASCII stampabile.

Nel caso di file ELF abbiamo altri strumenti per ispezionarne il contenuto. Quelli più comunemente disponibili sono `readelf` e `objdump`. Il file ELF inizia con una intestazione standard, che possiamo leggere con l'opzione `-h` di `readelf` (Figura 8). Tra le varie informazioni, ci interessa lo "start of section headers". Questo dice che la tabella delle sezioni si trova nel file a un offset di 480 byte. Possiamo chiedere a `readelf` di mostrarci il contenuto della tabella delle sezioni (Figura 9). Per ogni sezione abbiamo il nome, il tipo, l'indirizzo a cui deve essere caricata in memoria (Address), l'offset a cui la sezione inizia all'interno del file (Off) e la sua dimensione (Size). Delle colonne successive

```
foo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_Z3fool>:
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 89 f8              mov    %rdi,%rax
   7: 48 a1 00 00 00 00 00  movabs 0x0,%rax
   e: 00 00 00
  11: 48 03 04 25 00 00 00  add    0x0,%rax
  18: 00
  19: 48 03 05 00 00 00 00  add    0x0(%rip),%rax        # 20 <_Z3fool+0x20>
  20: c9                    leaveq
  21: c3                    retq
```

Figura 10: L'output di `objdump -d foo.o` (disassemblato).

```
Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     4: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    3 foovar1
     5: 0000000000000008     0 NOTYPE  LOCAL  DEFAULT    3 foovar2
     6: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    4 foovar3
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    1 _Z3fool
```

Figura 11: L'output di `readelf -s foo.o` (tabella dei simboli).

notiamo i flag (Flg) che fornisce ulteriori informazioni su come caricare la sezione. Notiamo subito che tutti i campi Address sono zero, perché l'assemblatore non sa ancora a che indirizzo le sezioni debbano essere caricate. Inoltre, solo le sezioni con il flag A vanno effettivamente caricate, mentre le altre servono ad altri scopi.

Osserviamo per esempio la sezione **`.text`**. Il tipo PROGBITS dice che il contenuto della sezione è deciso dal programma e non dallo standard ELF. La sezione inizia all'offset 0x40 nel file. In Figura 7 vediamo che a questo offset troviamo i byte 0x55, 0x48, 0x89, ... Si tratta della traduzione in linguaggio macchina delle istruzioni che abbiamo scritto nella sezione **`.text`** in Figura 3. Possiamo verificarlo usando il comando `objdump`, che ci permette di *disassemblare* il contenuto della sezione **`.text`** di un file ELF (Figura 10). L'output è su più colonne: la prima contiene l'offset all'interno della sezione, seguito da due punti; seguono i byte che si trovano a partire da quell'offset e infine l'interpretazione di quei byte come istruzione assembler (si noti che l'istruzione è ottenuta senza guardare il sorgente). La sezione va caricata (flag A) e deve essere eseguibile (flag X).

La sezione 5 è la tabella dei simboli, che possiamo osservare in Figura 11. È una delle sezioni che non va caricata (niente flag A) e serve solo al procedimento di costruzione dell'eseguibile. Le colonne importanti sono il valore (Value), lo scopo (Bind), la sezione a cui il simbolo appartiene (Ndx) e il suo nome (Name).

```
Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     4: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    3 var1
     5: 0000000000000008     0 NOTYPE  GLOBAL DEFAULT    3 var2
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT    1 _start
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z3fool
```

Figura 12: L'output di `readelf -s main.o` (tabella dei simboli).

```
Relocation section '.rela.text' at offset 0x160 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000009  000300000001 R_X86_64_64       0000000000000000 .bss + 0
000000000015  00020000000b R_X86_64_32S      0000000000000000 .data + 0
00000000001c  000200000002 R_X86_64_PC32     0000000000000000 .data + 4
```

Figura 13: L'output di `readelf -r foo.o` (tabella di rilocazione).

I simboli il cui tipo è SECTION servono in realtà a contenere l'indirizzo, ancora ignoto, dell'inizio delle sezioni **`.text`** (Ndx=1), **`.data`** (Ndx=3) e **`.bss`** (Ndx=4). Per gli altri simboli il campo Value contiene l'offset all'interno della sezione. Si noti, per esempio, il Value=8 per il simbolo `foovar2`. Tutti i simboli sono marcati come LOCAL (campo Bind): vuol dire che il collegatore non li userà per risolvere i riferimenti indefiniti. Il simbolo `foo` è marcato come GLOBAL, perché lo abbiamo dichiarato tale (linea 10 di Figura 3).

In Figura 12 osserviamo anche la tabella dei simboli di `main.o`, per notare un altro caso che si può presentare: il simbolo numero 7 (`_Z3fool`) risulta non definito (valore UND nella colonna Ndx). Questo avviene perché il file di Figura 6 riferisce il simbolo `_Z3fool` (riga 14) ma non lo definisce.

Osserviamo ora l'istruzione all'offset 7 in Figura 10. Questa traduce l'istruzione alla riga 15 di Figura 3, che contiene il riferimento a `foovar3` di cui, come abbiamo detto, l'assemblatore non conosce l'indirizzo. Possiamo osservare che l'assemblatore ha usato zero al posto dell'indirizzo di `foovar3`. In Figura 13 vediamo invece la tabella delle rilocazioni che l'assemblatore ha generato. Nel formato ELF c'è una tabella di rilocazione diversa per ogni sezione che ne ha bisogno. La tabella stessa si trova in una sezione del file e la possiamo vedere in Figura 9, riga **`.rela.text`**. La colonna Inf ci dice, in questo caso, che la tabella è relativa alla sezione 1 (cioè la sezione **`.text`**).

Ogni entrata della tabella di rilocazione fornisce le indicazioni per il collegatore (nel suo ruolo di *link editor*) su come e dove scrivere gli indirizzi. Il collegatore è in grado di eseguire semplici calcoli che, in genere, comportano di sommare il valore di un simbolo con una costante (*Addend*), fare qualche altra semplice operazione e scrivere il risultato ad un certo offset nella sezione. L'operazione da svolgere è codificata nel campo Type e i possibili tipi sono specificati dallo standard ELF.

- La prima riga in Figura 13 dice che all'offset 0x9 della sezione **`.text`** è

```
There are 8 section headers, starting at offset 0x1f0:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 000023 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 000158 000060 18   I  5   1  8
  [ 3] .data             PROGBITS        0000000000000000 000063 000010 00  WA  0   0  1
  [ 4] .bss              NOBITS          0000000000000000 000073 000000 00  WA  0   0  1
  [ 5] .symtab           SYMTAB          0000000000000000 000078 0000c0 18      6   4  8
  [ 6] .strtab           STRTAB          0000000000000000 000138 00001a 00      0   0  1
  [ 7] .shstrtab         STRTAB          0000000000000000 0001b8 000031 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Figura 14: L'output di `readelf -WS main.o` (tabella delle sezioni).

necessario scrivere il valore di **`.bss`** `+ 0`. Questo non è altro che l'indirizzo di `foovar3`, e l'offset 0x9 corrisponde agli 8 byte nulli all'interno dell'istruzione che si trova all'offset 7 in Figura 10.

- La seconda riga chiede di fare una operazione leggermente diversa, in quanto è relativa all'istruzione alla riga 16 di Figura 3, che non usa **`movabsq`** e dunque riserva soltanto 32 bit per il campo destinato a contenere l'indirizzo di `foovar1`. In questo caso il collegatore dovrà scrivere il valore di `.data + 0` (che è appunto l'indirizzo di `foovar1`) come un numero su 32 bit, con segno (operazione `32S`). Il collegatore darà un errore se l'indirizzo dovesse risultare non rappresentabile su 32 bit.

- La terza riga della tabella di rilocazione di Figura 13 contiene un caso ancora diverso. La riga è relativa all'istruzione alla riga 17 di Figura 3, che usa un indirizzamento relativo a **`%rip`**. Si noti che qui servirebbe l'offset tra l'indirizzo di `foovar2` (**`.data`** `+ 8`) e l'instruction pointer dell'istruzione successiva, che inizia all'offset 0x20. Il collegatore, però, non è in grado di decodificare le istruzioni e non sa dove queste iniziano e terminano. La soluzione adottata prevede che il collegatore, tramite l'operazione `PC32`, esegua una differenza con il *campo Offset* (0x1c in questo caso) e che l'assemblatore sottragga dall'Addend costante la differenza (anch'essa costante, e a lui nota) tra l'offset e l'inizio dell'istruzione successiva (4 in questo caso). Questo è il motivo per cui l'assemblatore ha usato un Addend 4 (vale a dire, $8 - 4$) invece di 8. Anche in questo caso il collegatore darà errore se la differenza non è rappresentabile su 32 bit.

Per completezza, le Figure 14 e 15 mostrano la tabella delle sezioni e la tabelle di rilocazione del file `main.o`

4 Il collegatore

Il collegatore ha tre compiti:

```
Relocation section '.rela.text' at offset 0x158 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000007  000500000002 R_X86_64_PC32     0000000000000008 var2 - 4
00000000000f  000700000004 R_X86_64_PLT32    0000000000000000 _Z3fool - 4
000000000016  000400000002 R_X86_64_PC32     0000000000000000 var1 - 4
00000000001d  000400000002 R_X86_64_PC32     0000000000000000 var1 - 4
```

Figura 15: L'output di `readelf -r main.o` (tabella di rilocazione).

- decidere dove caricare ogni sezione;
- risolvere tutti i riferimenti ai simboli non definiti;
- eseguire tutte le rilocazioni.

Il collegatore è il primo che vede il programma nella sua interezza: riceve infatti la lista di tutti i file oggetto da collegare.

In una prima fase, il collegatore cerca di ottenere un'unica sezione **`.text`**, **`.data`**, **`.bss`** e un'unica tabella dei simboli, mettendo insieme le rispettive sezioni di ogni file oggetto. Il risultato di questa fase è una nuova tabella delle sezioni e una nuova tabella dei simboli.

Per creare l'unica sezione **`.text`** il collegatore si limita a concatenare le sezioni **`.text`** dei file oggetto, nell'ordine in cui gli sono stati passati. Lo stesso vale per la sezione **`.data`**. Per la sezione **`.bss`** si limita a sommare le dimensioni delle varie sezioni **`.bss`** dei file di ingresso. Per creare l'unica tabella dei simboli esamina in ordine tutte le tabelle dei simboli dei file oggetto e le unisce. Durante questa operazione deve:

- aggiustare gli offset dei simboli dei file dal secondo in poi, sommandovi le dimensioni delle sezioni dei file precedenti;
- controllare che uno stesso simbolo GLOBAL non sia definito due volte (altrimenti termina con un errore);
- creare una lista di tutti i simboli non definiti.

Una volta esaminati tutti i file, deve controllare che ogni simbolo non definito in un certo file sia stato definito in qualche altro file, cioè che ciascuno dei simboli nella lista dei non definiti si trovi nella lista dei simboli complessivi, marcato come GLOBAL. Se questo non accade, il collegatore termina con un errore.

A questo punto il collegatore può assegnare un indirizzo ad ogni sezione e, di conseguenza, calcolare il valore definitivo di ogni simbolo.

Fatto ciò, può procedere a consultare le tabelle di rilocazione di tutti i file oggetto, mettendo in atto le istruzioni che vi trova.

Il contenuto di tutte le sezioni è ormai pronto e il collegatore deve solo creare la *tabella di caricamento*, che dirà al caricatore come e dove caricare le sezioni in memoria, ogni volta che il programma dovrà essere eseguito.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4000b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          720 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         2
  Size of section headers:           64 (bytes)
  Number of section headers:         8
  Section header string table index: 7
```

Figura 16: L'output di `readelf -h prog` (intestazione).

```
Elf file type is EXEC (Executable file)
Entry point 0x4000b0
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  LOAD           0x000000 0x0000000000400000 0x0000000000400000 0x0000f5 0x0000f5 R E 0x1000
  LOAD           0x0000f5 0x00000000004010f5 0x00000000004010f5 0x000020 0x000028 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data .bss
```

Figura 17: L'output di `readelf -Wl prog` (tabella di caricamento).

4.1 Il formato ELF per gli eseguibili

Torniamo al nostro esempio e supponiamo che `main.o` e `foo.o` siano stati collegati insieme per ottenere il file `prog`. Anche questo è un file ELF e ne possiamo esaminare l'intestazione (Figura 16). Notiamo che sono ora presenti dei *program headers* (dopo 64 byte nel file) e che il campo Entry Point Address ha un valore diverso da zero.

I program headers compongono la tabella di caricamento, che possiamo osservare in Figura 17. La tabella contiene una riga per ogni *segmento* dove, nel gergo ELF, un segmento corrisponde ad un insieme di sezioni. La prima riga dice che all'offset zero nel file si trova un segmento grande 0xf5 (FileSiz) che deve essere caricato all'indirizzo 0x400000 (VirtAddr o PhysAddr, che sono sempre uguali) e reso leggibile (Flg R) ed eseguibile (Flg E). Si tratta, come vediamo più sotto, del segmento che contiene la sezione **`.text`**. La seconda riga dice che all'offset 0xf5 nel file si trova un segmento grande 0x20 (FileSiz) che deve essere caricato all'indirizzo 0x4010f5 (VirtAddr o PhysAddr) e reso leggibile (Flg R) e scrivibile (Flg W). Inoltre, dopo la copia, ulteriori byte devono essere azzerati

```
There are 8 section headers, starting at offset 0x2d0:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000004000b0 0000b0 000045 00  AX  0   0  1
  [ 2] .data             PROGBITS        0000000004010f5 0000f5 000020 00  WA  0   0  1
  [ 3] .bss              NOBITS          0000000000401115 000115 000008 00  WA  0   0  1
  [ 4] .note.gnu.gold-version NOTE            0000000000000000 000118 00001c 00      0   0  4
  [ 5] .symtab           SYMTAB          0000000000000000 000138 000108 18      6   4  8
  [ 6] .strtab           STRTAB          0000000000000000 000240 00004a 00      0   0  1
  [ 7] .shstrtab         STRTAB          0000000000000000 00028a 000043 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Figura 18: L'output di `readelf -WS prog` (tabella delle sezioni).

```
Symbol table '.symtab' contains 11 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000401105     0 NOTYPE  LOCAL  DEFAULT    2 foovar1
     2: 000000000040110d     0 NOTYPE  LOCAL  DEFAULT    2 foovar2
     3: 0000000000401115     0 NOTYPE  LOCAL  DEFAULT    3 foovar3
     4: 000000000040111d     0 NOTYPE  GLOBAL DEFAULT    2 _end
     5: 0000000000401115     0 NOTYPE  GLOBAL DEFAULT    2 __bss_start
     6: 0000000000401115     0 NOTYPE  GLOBAL DEFAULT    2 _edata
     7: 00000000004000d3     0 NOTYPE  GLOBAL DEFAULT    1 _Z3fool
     8: 00000000004000b0     0 NOTYPE  GLOBAL DEFAULT    1 _start
     9: 00000000004010fd     0 NOTYPE  GLOBAL DEFAULT    2 var2
    10: 00000000004010f5     0 NOTYPE  GLOBAL DEFAULT    2 var1
```

Figura 19: L'output di `readelf -s prog` (tabella dei simboli).

fino ad arrivare alla dimensione di 0x28 (MemSiz). Si tratta del segmento che contiene le sezioni **`.data`** e **`.bss`**, come scritto in fondo alla Figura.

Anche se non servono più una volta ottenuto l'eseguibile, il collegatore ha lasciato nel file ELF anche la tabella delle sezioni e la tabella dei simboli complessive. Possiamo osservarle nelle Figure 18 e 19. Notiamo come la dimensione delle sezioni **`.text`** e **`.data`** sia la somma delle dimensioni delle corrispondenti sezioni nei file `main.o` e `foo.o`. Notiamo anche come i simboli abbiano ora tutti il loro valore definitivo. Possiamo vedere che il campo Entry Point Address di Figura 16 è esattamente il valore del simbolo `_start`.

Si noti che il collegatore ha aggiunto dei simboli propri (`__bss_start`, `_edata` ed `_end`, che marcano l'inizio o la fine di varie parti del programma.

Infine, osserviamo il disassemblato del file `prog` (Figura 20), che contiene il risultato finale di tutte le operazioni di rilocazione eseguite dal collegatore. In particolare osserviamo come nella funzione `foo`, all'indirizzo 0x4000da, l'istruzione **`movabsq`** `foovar3,%`**`rax`** sia finalmente completa con l'indirizzo `foovar3`, che è 0x40115 come possiamo confermare dalla tabella dei simboli in Figura 19.

```
prog:     file format elf64-x86-64


Disassembly of section .text:

00000000004000b0 <_start>:
  4000b0: 55                    push   %rbp
  4000b1: 48 89 e5              mov    %rsp,%rbp
  4000b4: 48 8b 05 42 10 00 00  mov    0x1042(%rip),%rax        # 4010fd <var2>
  4000bb: 48 89 c7              mov    %rax,%rdi
  4000be: e8 10 00 00 00        callq  4000d3 <_Z3fool>
  4000c3: 48 89 05 2b 10 00 00  mov    %rax,0x102b(%rip)        # 4010f5 <var1>
  4000ca: 48 8b 05 24 10 00 00  mov    0x1024(%rip),%rax        # 4010f5 <var1>
  4000d1: 5d                    pop    %rbp
  4000d2: c3                    retq

00000000004000d3 <_Z3fool>:
  4000d3: 55                    push   %rbp
  4000d4: 48 89 e5              mov    %rsp,%rbp
  4000d7: 48 89 f8              mov    %rdi,%rax
  4000da: 48 a1 15 11 40 00 00  movabs 0x401115,%rax
  4000e1: 00 00 00
  4000e4: 48 03 04 25 05 11 40  add    0x401105,%rax
  4000eb: 00
  4000ec: 48 03 05 1a 10 00 00  add    0x101a(%rip),%rax        # 40110d <foovar2>
  4000f3: c9                    leaveq
  4000f4: c3                    retq
```

Figura 20: L'output di `objdump -d prog` (disassemblato).

5 Il caricatore

Il caricatore ha il compito di caricare in memoria i segmenti del programma, agli indirizzi specificati nella tabella di caricamento, e di inizializzare i registri del processore in modo che il programma possa partire. In particolare, il registro **`%rip`** verrà inizializzato con il valore dell'entry point del programma (Figura 16, campo Entry point address) e il registro **`%rsp`** con un valore prefissato.

In Linux il caricatore è parte del nucleo del sistema.

6 Le librerie (statiche)

Le librerie statiche sono soltanto una collezione di file oggetto. In Unix venivano create con `ar`, un comando per la creazione di archivi di file del tutto generici. Nel tempo il comando `ar` si è specializzato per la creazione delle librerie e oggi sopravvive praticamente soltanto per questo scopo[1]. In particolare, il comando `ar` del progetto GNU si preoccupa di creare automaticamente anche l'indice dei simboli globali definiti dai file oggetto dell'archivio, cosa che un tempo andava fatta invocando un comando separato (`ranlib`).

Supponiamo di avere il file oggetto `foo.o`, ottenuto dalla compilazione di `foo.s` in Figura 3, e un altro file `bar.o`, che contiene la definizione di una

[1] La funzione di "archivio" è invece passata a `tar`, che era la versione di `ar` ottimizzata per i nastri magnetici (*tape* archive).

funzione `bar()`. Possiamo creare una libreria `mialib.a` che contenga entrambi i file oggetto tramite il seguente comando:

```
ar cr mialib.a foo.o bar.o
```

Possiamo esaminare il contenuto della libreria usando gli stessi strumenti che abbiamo usato per esaminare i file oggetto: sia `readelf`, che `objdump`, che `nm` riconoscono i file archivio ed eseguono l'operazione su ciascuno dei file oggetto contenuti. In più, il comando `nm`, con opzione `-s`, mostra anche l'indice dei simboli globali creato automaticamente da `ar`.

Una volta creata la libreria, possiamo passarla al collegatore insieme agli altri file oggetto:

```
ld -o main main.o mialib.a
```

Il collegatore processa i file nell'ordine in cui li abbiamo specificati e costruisce via via la tabella dei simboli complessiva. Quando arriva a processare la libreria, ne estrae soltanto i file oggetto che contengono simboli globali che risultano indefiniti in quel momento. I file così estratti entrano a far parte dei file oggetto dell'eseguibile finale, come se li avessimo specificati direttamente. Nel nostro esempio, dopo aver processato `main.o`, il collegatore si troverà con il simbolo `_Z3fool` indefinito. Passerà dunque a processare `mialib.a` e cercherà `_Z3fool` nell'indice, scoprendo che è definito dal file `foo.o`. Estrarrà dunque tale file dall'archivio e comincerà a processarlo come gli altri. Il file `bar.o`, che si trova nella libreria, ma non contiene simboli globali richiesti dagli altri file oggetto del programma, verrà ignorato. L'effetto è dunque di includere nell'eseguibile solo i file oggetto effettivamente necessari.

Si notino alcune particolari conseguenze del modo di procedere del collegatore. Solo i simboli non definiti vengono cercati: se il file `main.cpp` definisse una sua versione di `foo(long)`, questa verrebbe usata al posto di quella contenuta delle libreria, anche da eventuali altre funzioni della libreria stessa che dovessero usarla. Questa caratteristica (detta *interposing*) è in genere voluta: permette al programmatore di ridefinire alcune funzioni per meglio adattarle ai propri scopi. L'altra conseguenza, invece, è solo una cosa a cui stare attenti: una volta processata, una libreria non viene più ripresa in considerazione. Se dopo `mialib.a` ci fosse un file oggetto che ha bisogno di `bar()`, il simbolo non verrebbe trovato. Per questo motivo è in genere opportuno specificare tutte le librerie per ultime.

6.1 Ricerca automatica delle librerie

Per pura comodità di utilizzo, il collegatore è in grado di cercare automaticamente le librerie in una serie di directory standard (come `/usr/lib`), eventualmente configurabili, similmente a quanto fa il preprocessore per i file da includere. Per sfruttare il meccanismo la nostra libreria deve avere un nome che inizia con `lib`. Se vogliamo copiarla in una delle directory di sistema dobbiamo

inoltre avere i diritti di amministratore (direttamente, o tramite `sudo` o simili). Per esempio, possiamo copiare la nostra directory in `/usr/local/lib`, cambiandole contestualmente il nome:

```
sudo cp mialib.a /usr/local/lib/libmia.a
```

A questo punto è possibile collegare la libreria passando l'opzione `-l` al collegatore, subito seguita dal nome della libreria con il prefisso `lib` rimosso:

```
ld -o main main.o -lmia
```

Se la libreria viene trovata si ottiene lo stesso effetto che avevamo ottenuto passando al collegatore direttamente il percorso `mialib.a`. Per rendere più semplice l'uso della nostra libreria, possiamo anche inserire la dichiarazione delle funzioni `foo(`**`long`**`)` e `bar()` in un file header (per esempio, `mia.h`) che poi copiamo in una delle directory standard del preprocessore (per esempio, `/usr/local/include`). Chi vorrà usare la nostra libreria potrà aggiungere **`#include`** `<mia.h>` nei suoi file e poi *dovrà passare `-lmia` al collegatore*.

Approfondiamo l'ultimo punto, perché è una cosa che confonde alcuni programmatori. Per semplicità viene spesso detto che per usare, per esempio, iostream, si deve scrivere **`#include`** `<iostream>` e poi non serve fare altro. Anche se lo standard C++ permette e, anzi, promuove una implementazione del genere, in realtà, negli strumenti esistenti, non funziona così. Con gli strumenti che abbiamo appena visto (e che sono simili agli altri strumenti esistenti per la compilazione di programmi C e C++), **`#include`** `<iostream>` non è né necessario, né sufficiente per poter utilizzare le funzioni di iostream. Non è necessario perché, come abbiamo visto, tutto ciò che il preprocessore fa quando vede **`#include`** `<iostream>` è di andare a cercare un file di nome `iostream` in alcune directory standard, per poi copiarne il contenuto nel file originale. Ciò che serve al compilatore sono le dichiarazioni contenute nel file, non la direttiva **`#include`**. Scrivere le opportune dichiarazioni direttamente all'interno del file originale produrrebbe lo stesso effetto, anche se non sarebbe facile farlo per una libreria complicata come iostream, che ha bisogno a sua volta di tante altre dichiarazioni precedenti (e che vengono normalmente incluse per effetto di altre direttive **`#include`** all'interno del file `iostream` stesso).

La direttiva **`#include`**, inoltre, non è nemmeno sufficiente. Possiamo facilmente capirlo dal fatto che la direttiva è vista solo dal preprocessore. Il collegatore, che è uno strumento del tutto indipendente, non ha modo di sapere che volevamo usare iostream. Non ci accorgiamo di questo solo perché l'implementazione di iostream è contenuta nella libreria standard del C++ e `g++` passa sempre `-lstdc++` al collegatore, senza bisogno che noi lo specifichiamo. Ciò, però, non accade automaticamente per le librerie *non standard* come la nostra (e come tutte le librerie tranne due o tre). Per questo, se vogliamo usare la nostra libreria, è comunque necessario passare `-lmia` al collegatore, indipendentemente dal fatto di aver scritto **`#include`** `<mia.h>`.

A Comandi per Mac Silicon

Per provare l'esempio su un Mac Silicon è necessario usare gli strumenti per la cross-compilazione. Facciamo riferimento alla VM scaricabile all'url

`https://calcolatori.iet.unipi.it/resources/Debian-CE-Silicon-1.0.zip`

In questa VM sono già installati tutti gli strumenti per la cross-compilazione, con nomi che iniziano con `i686-linux-gnu-`. Per esempio, `ld` può essere invocato tramite il comando `i686-linux-gnu-ld`. Conviene aggiungere le seguenti abbreviazioni al proprio file `.bashrc`:

```
alias iar='i686-linux-gnu-ar'
alias ias='i686-linux-gnu-gcc -c -x assembler-with-cpp -m64'
alias ig++='i686-linux-gnu-g++ -m64'
alias ild='i686-linux-gnu-ld.gold -melf_x86_64'
alias iobjdump='i686-linux-gnu-objdump'
```

Le abbreviazioni saranno disponibili la prossima volta che si avvia la shell (per esempio, aprendo un nuovo terminale). A questo punto si possono ripetere i comandi dell'esempio, ma usando sempre `ig++` al posto di `g++`, etc. Notare che non c'è bisogno di una versione cross di `readelf`, in quanto il formato ELF non dipende dall'architettura.