Object File in Linux

Eseguibili, Librerie, Formato ELF, Rilocazione e Dynamic Linking

Roberto Marino, PhD

Università di Messina
CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

April 22, 2026



UniMe
1548

Filo logico della lezione

Le domande a cui risponderemo

1. Cosa è un **simbolo**?
2. Che **tipi di object file** esistono?
3. Come è fatta la **symbol table**?
   Cosa significa LOCAL, GLOBAL, WEAK?
4. Come è fatto un **file ELF**?
   Sezioni vs Segmenti.
5. Cosa è la **rilocazione**?
6. Gli indirizzi rilocati sono **virtuali o fisici**?
7. Differenza tra **symbol resolution** e **relocation**.
8. Come si chiama il processo eseguito dalla **MMU**?

Il filo conduttore

Tutto parte da un problema pratico:

Come fa il compilatore a produrre codice eseguibile, proveniente da più file sorgente, se non sa ancora dove andranno in memoria le funzioni e i dati?

Prerequisito

Conoscenza base del C e del concetto di compilazione separata:
ogni `.c` $\rightarrow$ un `.o` indipendente,
il linker li assembla insieme.

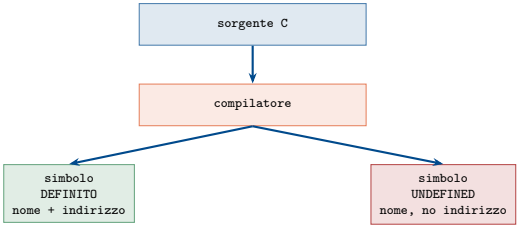
Cosa è un simbolo?

Definizione

Un simbolo è un **nome associato a un indirizzo**.

Ogni volta che nel codice C definisci qualcosa con un nome — una funzione, una variabile globale — il compilatore crea un simbolo che associa quel nome all'indirizzo di memoria dove quella cosa si trova.

```
1  int x = 42;          // simbolo "x"
2                       // -> indirizzo della variabile
3  void foo(void) {}    // simbolo "foo"
4                       // -> indirizzo della funzione
```



Due tipi di simbolo

- **Definito**: "il nome `foo` si trova a questo indirizzo, in questo file"
- **Undefined**: "uso il nome `printf` ma non so dove sta — cercalo tu, linker"

Perché servono i simboli?

I simboli sono il **linguaggio comune** tra compilatore, assembler e linker.

Il compilatore produce *simboli*, il linker li risolve in *indirizzi concreti*.

Tipi di object file ELF

Il campo `e_type` nell'ELF header li distingue

`ET_REL` — Relocatable (`.o`)

Output dell'assembler, input del linker.
Indirizzi **non definitivi** — placeholder.
Simboli esterni marcati come `UNDEF`.
Non ha Program Header Table.
Non eseguibile direttamente.

`ET_EXEC` — Executable (non-PIE)

Indirizzi virtuali **fissi** e hardcoded.
ASLR **non può** randomizzare la base.
Prodotto con `gcc -no-pie`.

`ET_DYN` — Shared object / PIE

Copre **due** casi distinti:

- libreria condivisa (`.so`)
- eseguibile PIE (default su Linux moderno)

`ET_CORE` — Core dump

Snapshot della memoria di un processo crashato.
Analizzabile con `gdb ./app core`.

```
# Vedere il tipo di un file ELF
readelf -h /bin/ls | grep Type
file /bin/ls

# ET_EXEC (non-PIE)
gcc -no-pie hello.c -o hello_exec
# ET_DYN  (PIE, default)
gcc         hello.c -o hello_pie
```

Riepilogo

`ET_REL`	`.o` relocatable
`ET_EXEC`	eseguibile fisso
`ET_DYN`	`.so` oppure PIE
`ET_CORE`	core dump

Come è fatta la symbol table?

Un array di strutture, una per ogni simbolo

La struttura `Elf64_Sym`

```c
typedef struct {
  Elf64_Word    st_name;  // indice in .strtab
  unsigned char st_info;  // tipo | binding
  unsigned char st_other; // visibility
  Elf64_Section st_shndx; // sezione di appartenenza
  Elf64_Addr    st_value; // indirizzo o offset
  Elf64_Xword   st_size;  // dimensione in byte
} Elf64_Sym;
```

Dove si trova e come leggerla

```
# Symbol table completa
readelf -s hello.o
nm hello.o

# Solo simboli dinamici (.dynsym)
nm -D ./hello
readelf -s --use-dynamic ./hello

# Dopo strip: .symtab sparisce,
# .dynsym resta intatta
strip ./hello
nm ./hello        # vuoto
nm -D ./hello     # funziona ancora
```

```
# Output di: readelf -s hello.o
Num   Value  Size Type    Bind   Ndx Name
  0: 0000    0    NOTYPE  LOCAL  UND
  1: 0000    0    FILE    LOCAL  ABS hello.c
  2: 0000   22    FUNC    GLOBAL   1 main
  3: 0000    0    NOTYPE  GLOBAL UND printf
```

LOCAL, GLOBAL, WEAK

Il campo *binding* — chi può vedere il simbolo?

LOCAL

Visibile **solo nel file** `.o` corrente.

Nessun altro modulo può vederlo.

Si ottiene con `static` in C.

```
1 static int x = 0;
2 static void helper()
3   { }
```

Due moduli possono avere il proprio `static int x` senza conflitti.

GLOBAL

Visibile a **tutti i moduli**.

Default per funzioni e variabili senza `static`.

Due GLOBAL con lo stesso nome ⇒ **errore** del linker.

```
1 int x = 42;
2 void foo(void) { }
```

Usato per le API pubbliche di librerie e moduli.

WEAK

Come GLOBAL, ma con **priorità inferiore**.

Può essere sovrascritto da un GLOBAL senza errori.

```
1 __attribute__((weak))
2 void hook(void)
3   { /* default */ }
```

Usato per implementazioni di default sostituibili (*plug-in pattern*).

Regole del linker per WEAK

- GLOBAL + WEAK stesso nome ⇒ vince **GLOBAL**, nessun errore

Lettere di `nm`

`T/t`	.text (codice)
`D/d`	.data (inizializzati)
`B/b`	.bss (zero-init)

Keyword C e il linker

Come il codice C controlla il binding dei simboli

LOCAL

`static` — produce LOCAL

```c
/* file scope: LOCAL, invisibile fuori */
static int contatore = 0;
static void helper(void) { }

/* dentro una funzione: durata
   statica, sempre LOCAL */
void foo(void) {
    static int n = 0; /* persiste */
    n++;
}
```

GLOBAL

`extern` — produce UNDEF

```c
/* dichiara senza definire:
   il linker cerchera' la def. */
extern int valore;
extern void funz_esterna(void);
```

`const` globale — va in `.rodata`

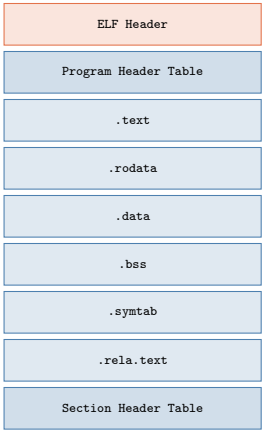
```c
/* segmento R-- (sola lettura)
   condiviso tra tutti i processi */
const int MAX = 1024;
const char MSG[] = "ciao";
/* scriverci causa segfault */
```

Riepilogo binding

`static` (file scope)	LOCAL
`extern`	UNDEF
`static inline`	LOCAL (se emesso)
`inline` (C99)	nessun simbolo
`const` globale	GLOBAL in `.rodata`
`_Thread_local`	GLOBAL in TLS

Come è fatto un file ELF?

Due viste sullo stesso contenuto: sezioni e segmenti



Sezioni vs Segmenti — la distinzione fondamentale

Sono due **modi di guardare gli stessi byte** del file, usati in momenti diversi.

Sezioni (Section Header Table) → per il **linker**:
dividono il file in blocchi con significato preciso (`.text`, `.data`, `.symtab`...). Il linker le usa per assemblare più `.o` insieme.

Segmenti (Program Header Table) → per il **loader**:
raggruppano sezioni con gli stessi permessi di memoria e dicono al kernel: "mappa questi byte all'indirizzo X con permessi R-X".

I dati sono scritti una volta sola sul disco.
Le due tabelle sono solo due indici sullo stesso contenuto.

Le sezioni principali

Sezione	Flag	Contenuto
`.text`	AX	Codice macchina
`.rodata`	A	Costanti, stringhe
`.data`	AW	Var. globali inizializzate
`.bss`	AW	Var. zero-init (**0 byte nel file!**)
`.symtab`	—	Symbol table (rimovibile)
`.dynsym`	A	Simboli per `ld.so`
`.rela.text`	—	Relocation entries
`.dynamic`	AW	Info per il dynamic linker
`.got`	AW	Global Offset Table
`.plt`	AX	Procedure Linkage Table

Flags per il runtime

A = Allocate (mappato in memoria)
W = Write (scrivibile)
X = Execute (eseguibile)

`.bss`: zero byte nel file

```c
int buf[1000000]; // .bss
// -> 0 byte nel file .o
// -> 4 MB allocati a runtime
// Il loader usa mmap anonima
// (gia' azzerata dal kernel)
```

`.symtab` vs `.dynsym`

`.symtab`: tutti i simboli, usata da `nm`, `gdb`, `objdump`.
Rimossa da `strip`.

`.dynsym`: sottoinsieme minimo per `ld.so` a runtime.
Non rimovibile.

```
readelf -S ./hello   # sezioni
readelf -l ./hello   # segmenti
```

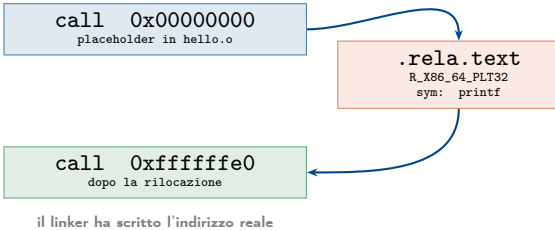
Cosa è la rilocazione?

Il problema

La compilazione separata crea un problema: quando l'assembler produce un `.o`, **non sa ancora**:

1. a quale indirizzo sarà caricato quel modulo
2. dove si trovano i simboli definiti altrove

La soluzione: lascia dei **placeholder** (spesso 0x00000000) e annota in una tabella (`.rela.text`) "qui c'e' un buco, e questo è il simbolo che ci va".



La struttura di una relocation entry

```
1 typedef struct {
2   Elf64_Addr   r_offset; // byte da patchare
3   Elf64_Xword  r_info;   // simbolo + tipo
```

Rilocazione statica vs dinamica

Statica (link-time): il linker risolve tutti i simboli interni e patcha il binario. Nessun overhead a runtime.

Dinamica (load/runtime): simboli che vengono da `.so` non possono essere risolti a link-time. Le entry `.rela.plt`

Gli indirizzi rilocati sono virtuali o fisici?

Il linker, il loader e `ld.so` lavorano **esclusivamente** con indirizzi virtuali.

Non sanno nulla degli indirizzi fisici.

La traduzione da virtuale a fisico è compito esclusivo della **MMU** (Memory Management Unit), in modo trasparente e invisibile al software.



Symbol resolution vs Relocation

Due fasi distinte del lavoro del linker

1. Symbol Resolution

Risponde alla domanda:
"dove si trova questo simbolo?"

Il linker scansiona tutti i `.o` e costruisce una tabella globale che associa ogni nome al modulo che lo definisce.

È una fase puramente *anagrafica*:
nessun byte viene modificato.
Si stabilisce solo *chi* definisce *cosa*.

Esempio

`main`	→ definito in `main.o`
`foo`	→ definito in `foo.o`
`printf`	→ definito in `libc`

2. Relocation

Risponde alla domanda:
"ora che so dove si trova,
scrivo l'indirizzo nel codice"

Solo dopo aver risolto tutti i simboli, il linker conosce gli indirizzi definitivi e può sostituire i placeholder nel codice binario con i valori corretti.

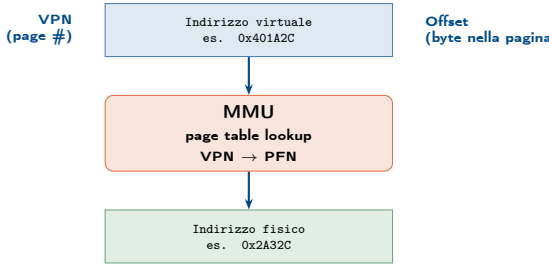
Come si chiama il processo eseguito dalla MMU?

Address Translation

La MMU traduce ogni indirizzo **virtuale** in un indirizzo **fisico** in hardware, in modo trasparente e automatico ad ogni accesso alla memoria.

Il meccanismo sottostante si chiama **paging**: memoria virtuale e fisica sono divise in blocchi di dimensione fissa chiamati **pagine** (tipicamente 4096 byte su x86-64).

La MMU mantiene una struttura dati chiamata **page table** che mappa ogni pagina virtuale alla corrispondente pagina fisica.



Page fault

Avviene Se la pagina virtuale non ha una corrispondente pagina fisica in RAM