

Laboratorio di Reti e Sistemi Distribuiti

12: Linking ed ELF file

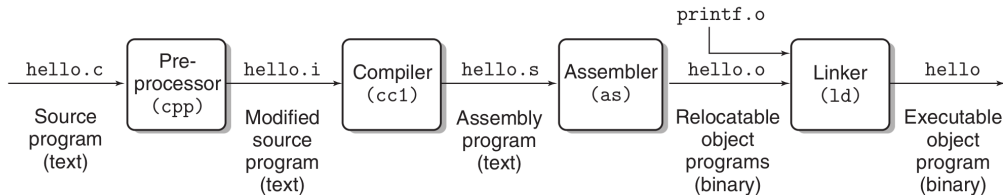
Roberto Marino, PhD¹
`roberto.marino@unime.it`

¹Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

Last Update: 28th March 2025

Compiler Driver

Abbiamo visto durante la lezione precedente che `gcc` (compiler driver) è il programma front-end che gestisce l'intero processo di compilazione in un compilatore come `gcc` (GNU Compiler Collection). **gcc** non è quindi solo il compilatore vero e proprio, ma un driver che coordina diverse fasi del processo di compilazione



Esplicitare le fasi di compilazione

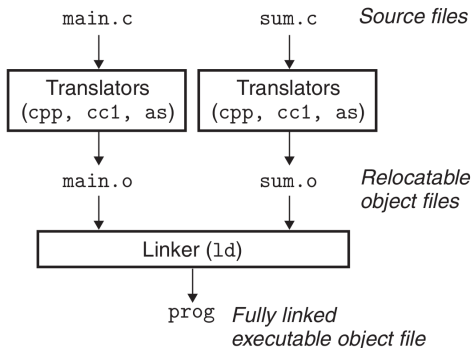
Se vogliamo invocare manualmente le varie fasi (compilazione granulare):

```
1 gcc -E main.c -o main.i      # Solo preprocessing (chiama cpp)
2 gcc -S main.i -o main.s      # Solo compilazione in assembly (chiama cc1)
3 gcc -c main.s -o main.o      # Solo assemblaggio (chiama as)
4 gcc main.o sum.o -o prog      # Solo linking (chiama ld)
```

Linking Statico

Definizione

Il linking statico è un processo in cui tutte le librerie necessarie per un programma vengono copiate direttamente nell'eseguibile finale, rendendolo **indipendente da librerie esterne** al momento dell'esecuzione.



Linking Statico e Dinamico

Durante il linking statico il linker ld combina:

- File oggetto (.o) generati dal compilatore.
- Librerie statiche (.a), che contengono il codice precompilato di funzioni necessarie.
- Simboli e riferimenti per generare l'eseguibile finale.

Differenza principale dal linking dinamico:

- Statico: Il codice delle librerie è incluso direttamente nell'eseguibile.
- Dinamico: Il programma fa riferimento a librerie esterne condivise (.so) al momento dell'esecuzione.

```
1 gcc -static main.o -o prog -L/usr/lib -lmylib # (con libmylib.a)
2 gcc -o prog main.c -L/usr/lib -lmylib # (dinamico, con libmylib.so)
```

Si può usare il comando ldd per verificare se un programma è linkato staticamente o dinamicamente

Remark: Definizione vs Dichiarazione vs Inizializzazione

Definizione

La definizione è il momento in cui una variabile o una funzione viene "creata", ovvero viene allocato **lo spazio in memoria per essa**. Una definizione può includere un'inizializzazione, ma non è obbligatoria.

Inizializzazione

Una inizializzazione è l'atto di **assegnare un valore iniziale** ad una variabile al momento della sua definizione.

Dichiarazione

Una dichiarazione **informa il compilatore dell'esistenza e del tipo di una variabile o funzione**, ma non necessariamente ne crea l'allocazione.

La variabile è **dichiarata**, ma è **definita** (spazio allocato) in un altro file oggetto:

Dichiarazione di una variabile senza definizione

```
extern int a;
```

La funzione è dichiarata ma la sua implementazione non ancora (altrove nello stesso file oggetto o in un altro file:

Dichiarazione di una funzione senza definizione

```
int funzione(int x);
```

Remark: variabili static

Le variabili `static` vengono allocate una sola volta e rimangono in memoria per l'intera esecuzione del programma. Ciò significa che il loro valore persiste tra le chiamate alla funzione in cui sono dichiarate o tra le varie parti del programma se dichiarate a livello globale.

a livello di funzione

Sono visibili (scope) solo all'interno della funzione in cui sono definite

a livello di file oggetto

Sono visibili (scope) solo all'interno del file oggetto, **non sono risolvibili esternamente**

Cosa fa nel dettaglio il linker?

I **linker statici**, come il programma `ld` di Linux, prendono in ingresso un insieme di file oggetto rilocabili e di argomenti della riga di comando e generano in uscita un file oggetto eseguibile completamente risolto che può essere caricato ed eseguito. I file oggetto rilocabili in ingresso consistono in varie sezioni di codice e dati, dove ogni sezione è una sequenza contigua di byte. Le **istruzioni** si trovano in una sezione, le **variabili globali** inizializzate in un'altra sezione e le **variabili non inizializzate** in un'altra ancora.

Per generare un eseguibile vengono finalizzati due processi: **la risoluzione dei simboli** e **la rilocazione**

Risoluzione dei simboli e rilocalizzazione

Risoluzione dei simboli

I file oggetto definiscono e fanno riferimento a simboli, dove ogni simbolo corrisponde a una **funzione**, a una variabile **globale** o a una variabile **statica** (cioè qualsiasi variabile C dichiarata con l'attributo **static**). Lo scopo della risoluzione dei simboli è quello di associare a ogni riferimento di simbolo esattamente una definizione di simbolo.

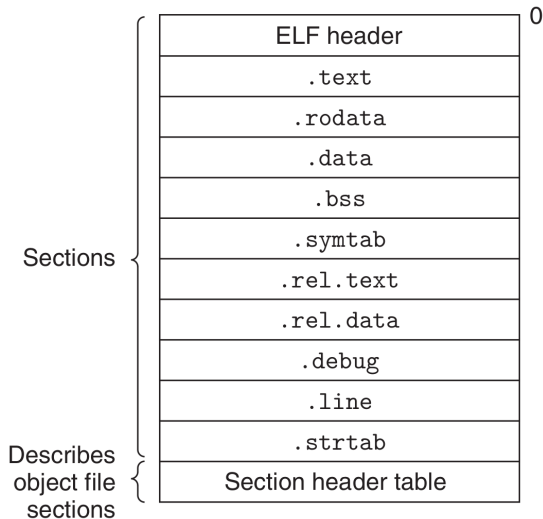
Rilocalizzazione

I compilatori e gli assembleri generano sezioni di codice e di dati **che iniziano all'indirizzo 0**. Il linker ricolloca queste sezioni associando una posizione di memoria a ogni definizione di simbolo e modificando tutti i riferimenti a tali simboli in modo che puntino a questa posizione di memoria.

Quanti tipi di file oggetto esistono?

- ❶ **File oggetto rilocabili:** Contengono codice e dati in forma tale da poter essere combinati con altri file oggetto rilocabili a creare un file oggetto eseguibile
- ❷ **File oggetto eseguibili:** Contengono codice e dati in forma tale da poter esser copiati in memoria centrale ed eseguiti
- ❸ **File oggetto condivisibili (shared):** tipo speciale di file oggetto che possono essere caricati in memoria e linkati dinamicamente durante il caricamento (librerie dinamiche) o durante l'esecuzione (vedi funzione `dlopen()`)

Executable and Linkable Format (ELF) Rilocabile



Definizione

Il formato **ELF (Executable and Linkable Format)** è uno standard per file eseguibili, oggetto, librerie condivise e core dump, ampiamente utilizzato nei sistemi Unix-like, tra cui Linux. A differenza di un ELF eseguibile, un file rilocabile (tipicamente associato ai file oggetto (.o), non ha segmenti (program headers), perché non è ancora pronto per essere caricato in memoria.

Un file ELF rilocabile è organizzato in tre parti principali:

- ❶ **ELF Header:** Contiene metadati sul file (architettura, tipo, offset delle sezioni, ecc.).
- ❷ **Sezioni (Sections):** Aree che contengono codice, dati, simboli, informazioni di rilocalizzazione, ecc.
- ❸ **Header delle Sezioni (Section Headers):** Una tabella che descrive ogni sezione presente nel file.

L'ELF Header si trova all'inizio del file e definisce:

- **Magic number (0x7F 'E' 'L' 'F')** per identificare il formato (aprite a.out con hexyl!)
- **Classe:** (32/64 bit).
- **Tipo:** Rilocabile/Non Rilocabile.
- **Architettura:** (es. x86, ARM).
- Offset delle sezioni e della section header table.

Sezioni principali di un ELF rilocabile

Un ELF rilocabile (file oggetto) contiene diverse sezioni, tra cui:

- `.text`: Contiene il codice macchina (istruzioni) generato dal compilatore.
- `.data`: Variabili globali inizializzate.
- `.bss`: Variabili globali non inizializzate (occupa spazio solo a runtime).
- `.rodata`: Dati di sola lettura (es. stringhe costanti).
- `.symtab`: (Symbol iTable) Tabella dei simboli, che include: funzioni definite (global/extern), Variabili globali, Simboli non risolti (riferimenti esterni).
- `.rel.text` e `.rel.data`: Tabelle di rilocazione che indicano al linker come modificare i riferimenti a simboli esterni o indirizzi assoluti.

Tabella dei simboli

La symbol table (sezione .symtab) è una struttura dati negli ELF che tiene traccia di:

- Funzioni e variabili globali definite nel file oggetto (anche dette nonstatic).
- Simboli esterni (riferimenti a funzioni/variabili definiti altrove, external).
- Simboli locali (funzioni e variabili definite static)

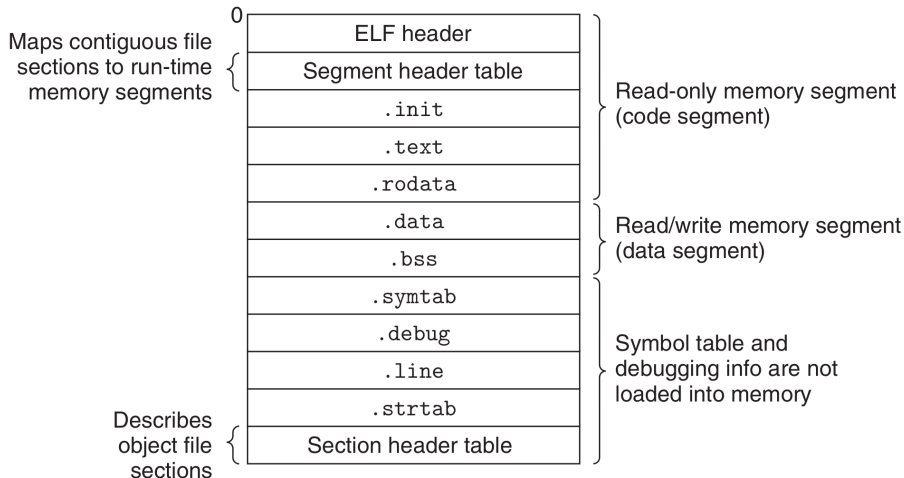
Supponiamo di avere il seguente codice:

```
1 extern int external_var; // Simbolo esterno (non definito qui)
2 int global_var = 42;    // Simbolo globale definito
3 void foo() {}           // Funzione definita
```

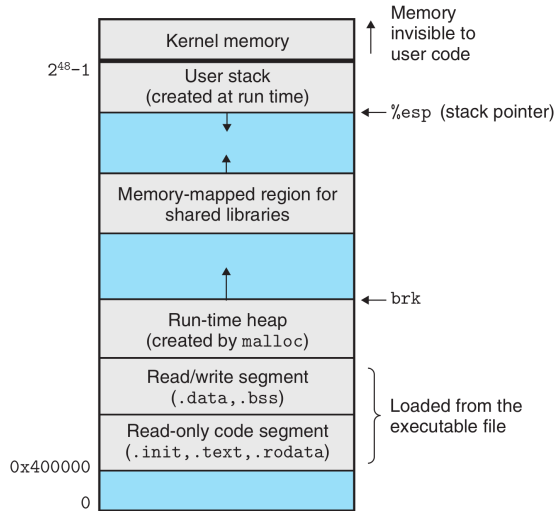
Dopo la compilazione possiamo usare **readelf -s prog** per ottenere:

1	Num:	Value	Size	Type	Bind	Vis	Ndx	Name
2	0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	
3	1:	00000000	0	FILE	LOCAL	DEFAULT	ABS	example.c
4	2:	00000000	0	SECTION	LOCAL	DEFAULT	1	.text
5	3:	00000000	11	FUNC	GLOBAL	DEFAULT	1	foo
6	4:	00000000	4	OBJECT	GLOBAL	DEFAULT	3	global_var
7	5:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	external_var

Executable ELF



Run-time memory image



Linker Memory map

La si ottiene con il comando `ld --verbose` ed è diversa per ogni sistema operativo e per ogni architettura. Serve per istruire il linker sugli offset di rilocalizzazione.

Propongo un esempio didattico di mappa della memoria:

```
1 .text    0x1000    : { *(.text) }      # Codice a 0x1000
2 .data    0x2000    : { *(.data) }      # Dati a 0x2000
```

Supponiamo ora di avere due file asm da dover **assemblare e rilocalizzare**. `main.asm` → `main.o`

```
1 ; File: main.asm
2 CALL somma      ; Chiama la funzione "somma"
3 MOV  AX, x      ; Accede alla variabile "x"
```

`math.asm` → `math.o`

```
1 ; File: math.asm
2 x DB 5          ; Definisce la variabile "x"
3 somma:          ; Definisce la funzione "somma"
4     ADD AX, BX
5     RET
```

Risoluzione dei nomi e rilocalizzazione

Dopo la **risoluzione dei nomi** il codice diventa:

```
1 CALL 0x1000; Sostituito "somma" con l'indirizzo 0x1000
2 MOV AX, [0x2000]; Sostituito "x" con l'indirizzo 0x2000
```

Dopo la **rilocalizzazione** (somma di un offset di rilocalizzazione: es. 0x5000) il codice diventa:

```
1 CALL 0x7000      ; Chiama "somma" all'indirizzo 0x7000
2 MOV  AX, [0x6000] ; Accede a "x" all'indirizzo 0x6000
```

Risoluzione dei nomi e rilocalizzazione

```
objdump -d main.o
```

```
1 main.o:      file format elf64-x86-64
2
3 Disassembly of section .text:
4 0000000000000000 <_start>:
5   0:   e8 00 00 00 00          call    5 <_start+0x5>  # Chiama 'somma
   ' (indirizzo non ancora risolto)
6   5:   8b 05 00 00 00 00      mov     eax, [rip+0x0]  # Accede a 'x'
   (indirizzo non ancora risolto)
```

Risoluzione dei nomi e rilocalizzazione

```
objdump -d math.o
```

```
1 math.o:      file format elf64-x86-64
2
3 Disassembly of section .text:
4 0000000000000000 <somma>:
5     0:    01 d8          add     eax, ebx
6     2:    c3             ret
7
8 Disassembly of section .data:
9 0000000000000000 <x>:
10    0:    05 00 00 00      mov     eax, 0x0  # Valore di 'x' = 5
```

Risoluzione dei nomi e rilocalizzazione

Dopo `ld main.o math.o -o prog` e disassemblando con `objdump` l'ELF eseguibile `prog`:

```
1 Disassembly of section .text:
2 0000000000401000 <_start>:
3   401000:    e8 0b 00 00 00      call    401010 <somma>    # Risolto:
        call a 0x401010
4   401005:    8b 05 05 10 00 00    mov     eax, [rip+0x1005] # Risolto: x
        a 0x402000
5
6 0000000000401010 <somma>:
7   401010:    01 d8              add     eax, ebx
8   401012:    c3                ret
9
10 Disassembly of section .data:
11 0000000000402000 <x>:
12  402000:    05 00 00 00      mov     eax, 0x0    # x = 5
```

Indirizzo_di_destinazione = Indirizzo_della_prossima_istruzione + Offset_relativo

Dopo la generazione dell'ELF eseguibile il programma è pronto per il loading e la presa in carico da parte dello scheduler.

Concetti introdotti (fondamentali!!):

- 1 Compilazione granulare
- 2 ELF file
- 3 Linking statico e dinamico
- 4 Risoluzione dei simboli e rilocazione
- 5 Tool *nix per l'analisi statica (objdump, readelf, ld, ldd, nm, size, ar)