

# **LABORATORIO DI RETI E SISTEMI DISTRIBUITI**

## **HANDS-ON 9**

*Tabella dei simboli e compilazione modulare*

**Antonio Mastrolembro Ventura**

**543644**

April 4, 2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Definizione del problema</b>	<b>2</b>
<b>3</b>	<b>Esercizio 1 - Tipi di variabili e tabella dei simboli</b>	<b>3</b>
3.1	Metodologia e risultati . . . . .	3
3.1.1	Codice in analisi . . . . .	3
3.1.2	Compilazione senza linking . . . . .	3
3.1.3	Tabella dei simboli . . . . .	4
3.1.4	Segmenti text, data, bss . . . . .	5
<b>4</b>	<b>Esercizio 2 - Compilazione modulare</b>	<b>6</b>
4.1	Metodologia e risultati . . . . .	6
4.1.1	Compilazione con make . . . . .	7
4.1.2	Tabella dei simboli . . . . .	7
<b>5</b>	<b>Conclusioni</b>	<b>9</b>

# 1 Introduzione

Con il seguente hands-on andremo ad approfondire il funzionamento del processo di compilazione di un programma in linguaggio C analizzandone la relativa tabella dei simboli in due scenari differenti.

Durante la fase di compilazione di un programma in C, il codice sorgente viene tradotto in linguaggio macchina attraverso una serie di fasi ben definite: pre-processamento, compilazione, assemblaggio e linking.

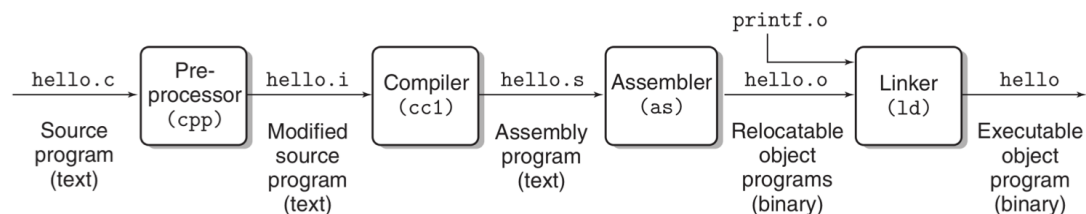


Figure 1: Fasi della compilazione

Dopo la fase di assemblaggio, il file oggetto contiene una tabella dei simboli, che è un elenco di variabili, funzioni e riferimenti presenti nel programma. Questa tabella è fondamentale perché permette di identificare le funzioni e le variabili globali dichiarate nel programma, determinare se un simbolo è definito nel file corrente o se è esterno e verificare le sezioni di memoria in cui sono collocati i simboli (text, data, bss, ecc...).

## 2 Definizione del problema

Per comprendere meglio il funzionamento del compilatore, andremo a svolgere due esercizi: nel primo, dovremo analizzare la tabella dei simboli di un file oggetto generato dalla fase di assemblaggio di un file C, mentre nel secondo esercizio andremo a vedere l'utilizzo di un make file per effettuare una compilazione modulare; anche in questo caso analizzeremo la tabella dei simboli.

## 3 Esercizio 1 - Tipi di variabili e tabella dei simboli

Per il primo esercizio andremo a compilare, senza la fase di linking, un file C ed andremo ad analizzarne la tabella dei simboli.

### 3.1 Metodologia e risultati

#### 3.1.1 Codice in analisi

```
1 #include <stdio.h>
2
3 int a;
4 static int b = 10;
5
6 void funzione() {
7     static int c = 5;
8     int d = 20;
9     c++;
10    d++;
11 }
12
13 int main() {
14     funzione();
15     return 0;
16 }
```

Questo sarà il codice oggetto della nostra analisi.

#### 3.1.2 Compilazione senza linking

```
$ gcc -c exercisel.c -o exercisel.o
```

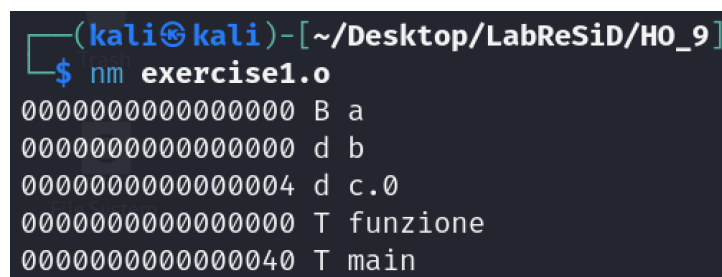
Con questo comando, il compilatore GCC compilerà il file sorgente chiamato `exercisel.c` senza eseguire il linking, questo genererà un file oggetto `exercisel.o`. Il flag `-c` indica proprio il fatto che il file sorgente passerà per il preprocessing, la compilazione e l'assemblaggio.

### 3.1.3 Tabella dei simboli

A questo punto, il nostro file oggetto rilocabile `exercise1.o` contiene una tabella dei simboli che elenca tutti i simboli definiti ed utilizzati nel codice sorgente. Per analizzare questa tabella utilizziamo il comando `nm` che ci permette appunto di visualizzare i simboli presenti in un file oggetto.

```
$ nm exercise1.o
```

Ed avremo il seguente output:



```
(kali㉿kali)-[~/Desktop/LabReSiD/H0_9]
$ nm exercise1.o
0000000000000000 B a
0000000000000000 d b
0000000000000004 d c.0
0000000000000000 T funzione
0000000000000040 T main
```

Figure 2: Tabella dei simboli

L'output è composto da tre colonne che possiamo comprendere tramite la pagina man del comando `nm`:

- Un **indirizzo** (non definitivo in quanto verrà assegnato definitivamente solo in fase di linking). Un valore che rappresenta la posizione relativa del simbolo all'interno del file oggetto.
- Il **tipo del simbolo**. Nel nostro caso abbiamo "B" che rappresenta un simbolo definito nella sezione "BSS" (blocchi di memoria allocati ma non inizializzati), "d" che indica una variabile inizializzata nella sezione dati (data section) e "T" che indica un simbolo definito nella sezione `.text`.
- Il **nome del simbolo**. Corrisponde al nome della funzione o della variabile così come è stato dichiarato nel codice sorgente.

Mettendo insieme queste tre colonne e, confrontando con il nostro codice sorgente, vediamo subito che ogni simbolo e nome corrisponde effettivamente a quanto definito da noi nel programma `exercise1.c`.

### 3.1.4 Segmenti text, data, bss

Per vedere ancora più nel dettaglio l'organizzazione della memoria nel file oggetto, possiamo aggiungere o eliminare variabili nel codice sorgente e osservare come queste modifiche influenzano le dimensioni delle diverse sezioni di memoria. Utilizziamo il comando

```
$ size --format=GNU exercise1.o
```

per analizzare la dimensione dei segmenti.

```
(kali@kali)-[~/Desktop/LabReSiD/H0_9]
$ size --format=GNU exercise1.o
      text      data      bss      total filename
         88         88         4        180 exercise1.o
```

Figure 3: Segmenti text, data, bss

Le cifre si riferiscono alla dimensione in byte di ciascun segmento all'interno del file oggetto. Cosa succede se aggiungiamo alcune variabili?

```
1 int z = 1;
2
3 void test() {
4 }
```

Output del comando nm:

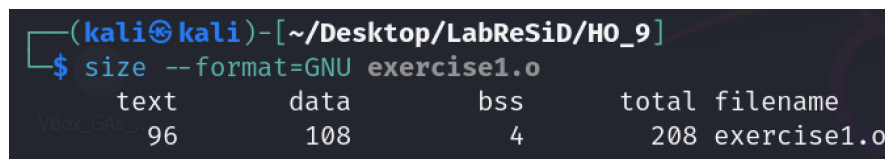
```
(kali@kali)-[~/Desktop/LabReSiD/H0_9]
$ nm exercise1.o
0000000000000000 B a
0000000000000000 d b
0000000000000008 d c.0
0000000000000000 T funzione
0000000000000048 T main
0000000000000040 T test
0000000000000004 D z
```

Figure 4: Tabella dei simboli aggiornata

Come possiamo vedere adesso abbiamo due nuovi simboli: "T" riferito a "test" che è la nostra funzione e "D" riferito a "z" ovvero la variabile inizializzata.

Rispetto a prima la "D" è maiuscola in quanto la variabile `z` è dichiarata come globale, ovvero accessibile anche da altri file oggetto durante il linking.

Analogamente, possiamo verificare la dimensione dei segmenti e come essa sia cambiata. Utilizziamo lo stesso comando della Figura 3:



```
(kali㉿kali)-[~/Desktop/LabReSiD/H0_9]
$ size --format=GNU exercise1.o
   text      data       bss      total  filename
   96       108         4       208  exercise1.o
```

Figure 5: Segmenti text, data, bss aggiornati

Come possiamo vedere, le dimensioni dei segmenti text e data sono cambiati rispetto a prima: text è aumentato perché abbiamo aggiunto la funzione `test()` e data è aumentato per includere la variabile `z`, che è stata inizializzata con un valore (1).

## 4 Esercizio 2 - Compilazione modulare

Per il secondo esercizio andremo a vedere la compilazione di un programma modulare suddiviso in più file sorgente e ne analizzeremo le tabelle dei simboli.

### 4.1 Metodologia e risultati

Il programma è composto da quattro file principali: `header.h`, `file1.c`, `file2.c` e `main.c`. Il file `header.h` contiene la dichiarazione di una variabile e funzioni esterne, mentre `file1.c` e `file2.c` contengono le implementazioni delle funzioni (dichiarate in `header.h`) che modificano e stampano il valore della variabile dichiarata in `header.h`. Infine, `main.c` è il file che contiene la funzione `main()`, che chiama le funzioni definite in `file1.c` e `file2.c`.

Per compilare il programma, utilizziamo un `Makefile`, che gestisce la compilazione separata dei singoli file sorgente, la creazione dei relativi file oggetto e l'assemblaggio finale dell'eseguibile.

### 4.1.1 Compilazione con make

Per compilare il programma utilizziamo il comando `make`:

```
(kali㉿kali)-[~/Desktop/LabReSiD/H0_9/Ex2]  
$ make  
gcc -g -Wall -c main.c  
gcc -g -Wall -c file1.c  
gcc -g -Wall -c file2.c  
gcc -g -Wall -o programma main.o file1.o file2.o
```

Figure 6: Comando make

### 4.1.2 Tabella dei simboli

A questo punto possiamo analizzare le tabelle dei simboli di ogni programma. Per farlo possiamo digitare il comando `make check`, definito all'interno del Makefile:

```
(kali㉿kali)-[~/Desktop/LabReSiD/H0_9/Ex2]  
$ make check  
  
=== Simboli file1.o ===  
nm file1.o  
0000000000000000 T modifica_var_extern  
                  U printf  
0000000000000000 D var_extern  
0000000000000004 d var_static  
  
=== Simboli file2.o ===  
nm file2.o  
                  U printf  
0000000000000000 T stampa_var_extern  
                  U var_extern  
  
=== Simboli main.o ===  
nm main.o  
0000000000000000 T main  
                  U modifica_var_extern  
                  U stampa_var_extern
```

Figure 7: Tabelle dei simboli



### **Simboli file1.o:**

- T - modifica\_var\_extern: capiamo che si tratta di una funzione, il simbolo T si riferisce appunto che è definita nella sezione `.text`;
- U - printf: "U" sta per undefined, quindi in questo caso ci riferiamo a una funzione esterna (printf) definita appunto nella libreria standard di C;
- D - var\_extern: il simbolo "D" indica che si trova nella sezione `.data` ed è una variabile globale inizializzata;
- d - var\_static: il simbolo "d" indica che si trova nella sezione `.data` ed è una variabile statica inizializzata.

### **Simboli file2.o:**

- T - stampa\_var\_extern: una funzione definita nella sezione `.text`;
- U - var\_extern: abbiamo incontrato questa variabile in file1.o, stavolta la vediamo con il simbolo "U" in quanto appunto la sua definizione è presente nel file precedente.

### **Simboli main.o:**

- T - main: indica una funzione definita nella sezione `.text` (la funzione chiamata main);
- U - modifica\_var\_extern: anche in questo caso, il simbolo "U" di undefined accanto al nome della funzione in quanto la sua definizione avviene in file1.o;
- U - stampa\_var\_extern: lo stesso vale per questa funzione, definita però in file2.o.

## 5 Conclusioni

In questi due esercizi abbiamo analizzato il processo di compilazione di un programma in C andando ad osservare in particolare le tabelle dei simboli. Abbiamo visto come il compilatore gestisce variabili e funzioni, e come questi simboli vengano classificati nelle diverse sezioni di memoria, come `.text`, `.data` e `.bss`.

Nel primo esercizio, abbiamo analizzato un singolo file sorgente, osservando come le variabili globali, statiche e locali vengano rappresentate nella tabella dei simboli. Inoltre, l'aggiunta di nuove variabili e funzioni ha influito sulle dimensioni dei segmenti di memoria, come mostrato dal comando `size`.

Nel secondo esercizio, abbiamo visto il funzionamento della compilazione modulare tramite un `Makefile`. In questo caso, analizzando le tabelle dei simboli, abbiamo osservato come i simboli esterni siano gestiti tra i vari file oggetto. Il simbolo `U` (undefined) ha permesso di identificare le funzioni e le variabili dichiarate ma non definite nel file corrente.