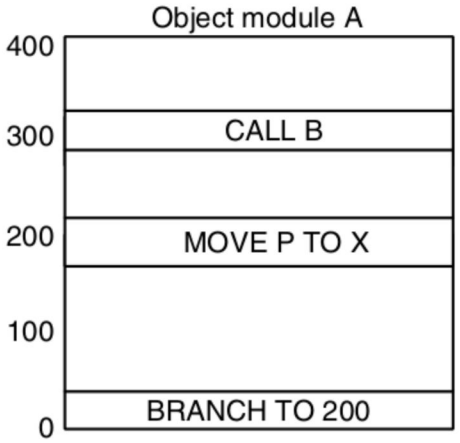
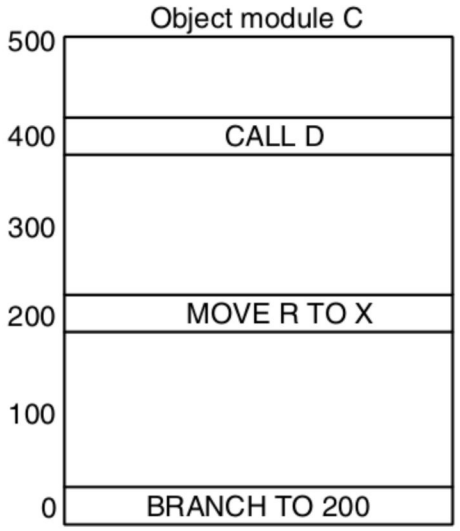
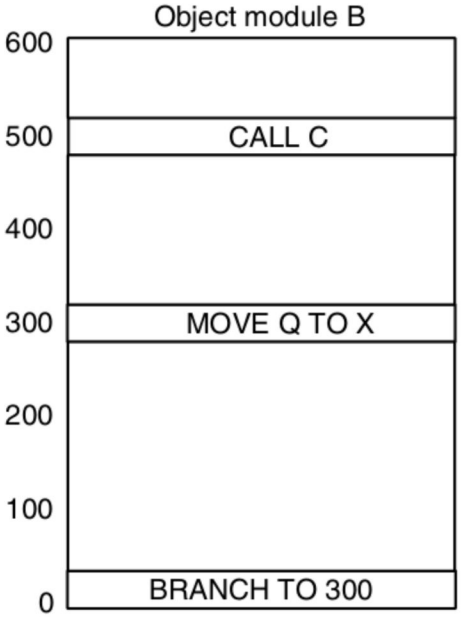
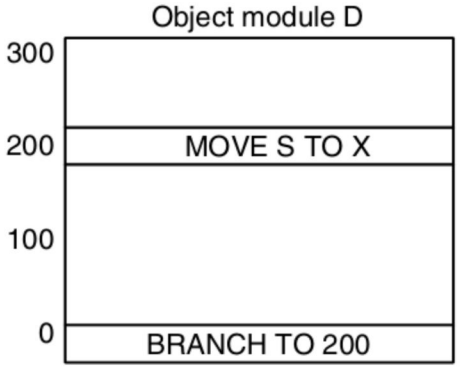
Collegamento (linking)

- Da A.S. Tanenbaum, Structured Computer Organization





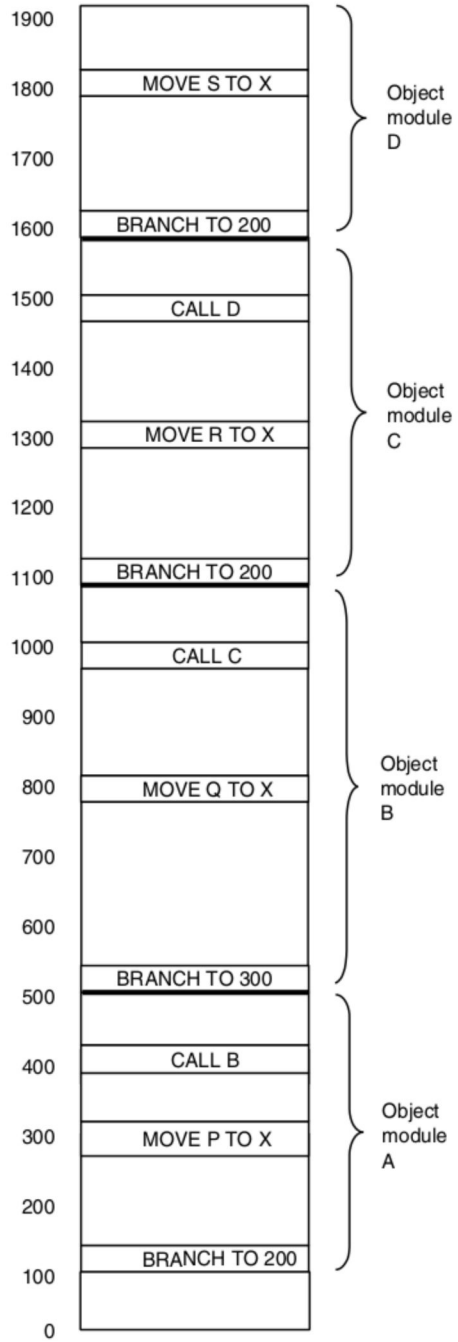


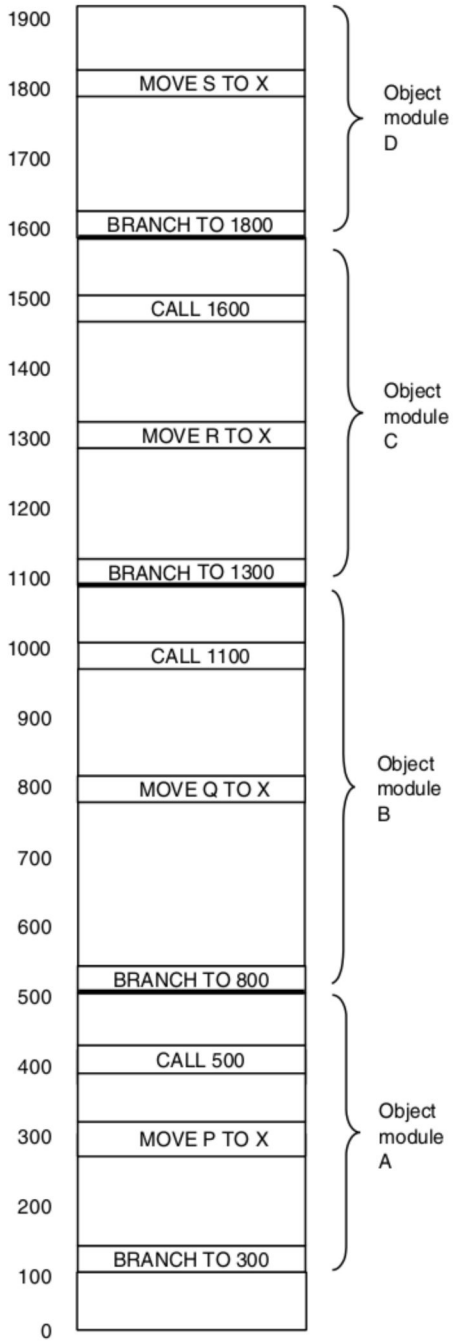


- Diversi moduli oggetto, ciascuno con spazio indirizzi che inizia da 0
- Per il collegamento i moduli hanno una tabella dei simboli esterni.
  Es per il mod A:
  “B al byte 300”

Collegamento (linking statico)

Moduli giustapposti, ma non rilocati e collegati





Dopo rilocazione e collegamenti

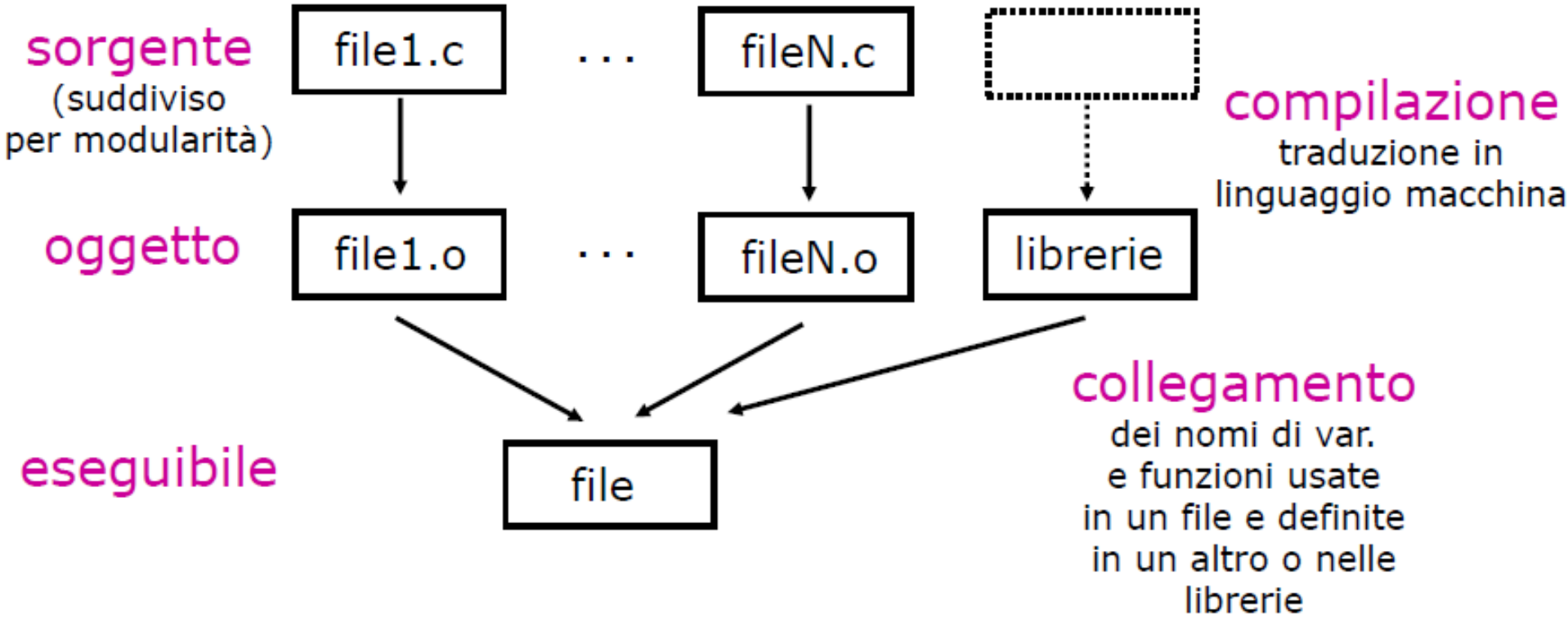
Collegamento (linking)

- Perché serve il collegamento:
  - Per scrivere programmi modulari: isolare un “modulo” che contiene un insieme di funzioni (es. per inserire nodo in una lista, estrarre un nodo, trovare un nodo etc); le funzioni del modulo possono essere chiamate senza conoscerne l’implementazione, che può cambiare
  - Per risparmiare spazio nei file sorgenti (rispetto a copiare e incollare il testo in C)
  - Per risparmiare tempo: evitare di ricompilare le librerie quando compiliamo un programma che la usa (rispetto a #includere il file in C contentente il codice delle funzioni)
  - Risparmiare spazio nei file eseguibili e poter modificare le librerie senza ricompilare, se si usa il collegamento *dinamico*: il codice non fa parte dell’eseguibile, il collegamento avviene, con meccanismi dipendenti dal sistema, al momento del caricamento o della prima chiamata

Compilazione separata

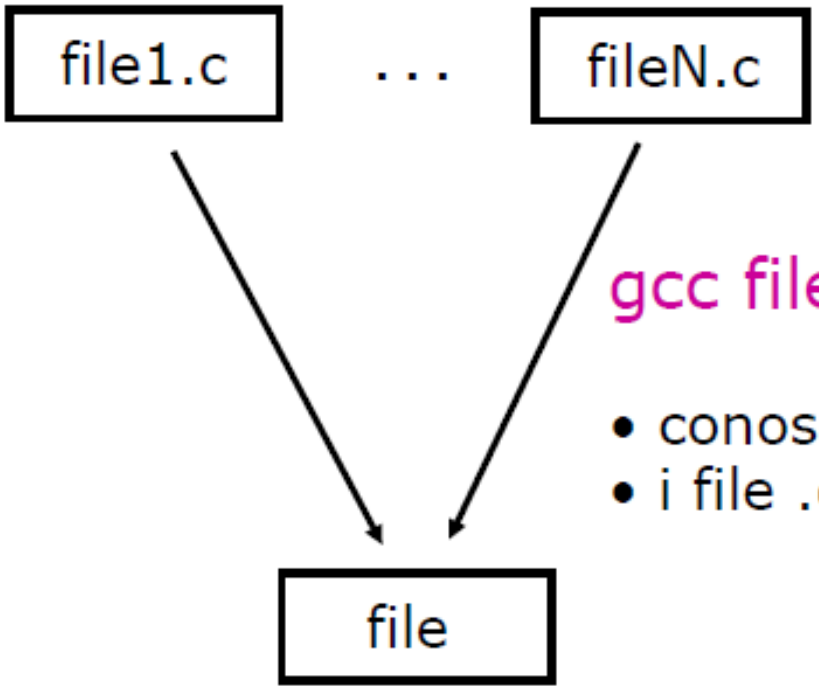
- Richiamiamo alcuni concetti visti ad Architettura degli Elaboratori e aspetti pratici sulla compilazione/collegamento del C su Unix

- Per produrre un file eseguibile dal C in generale si ha:



Compilazione separata

- I compilatori ci permettono di fare tutto in un colpo:

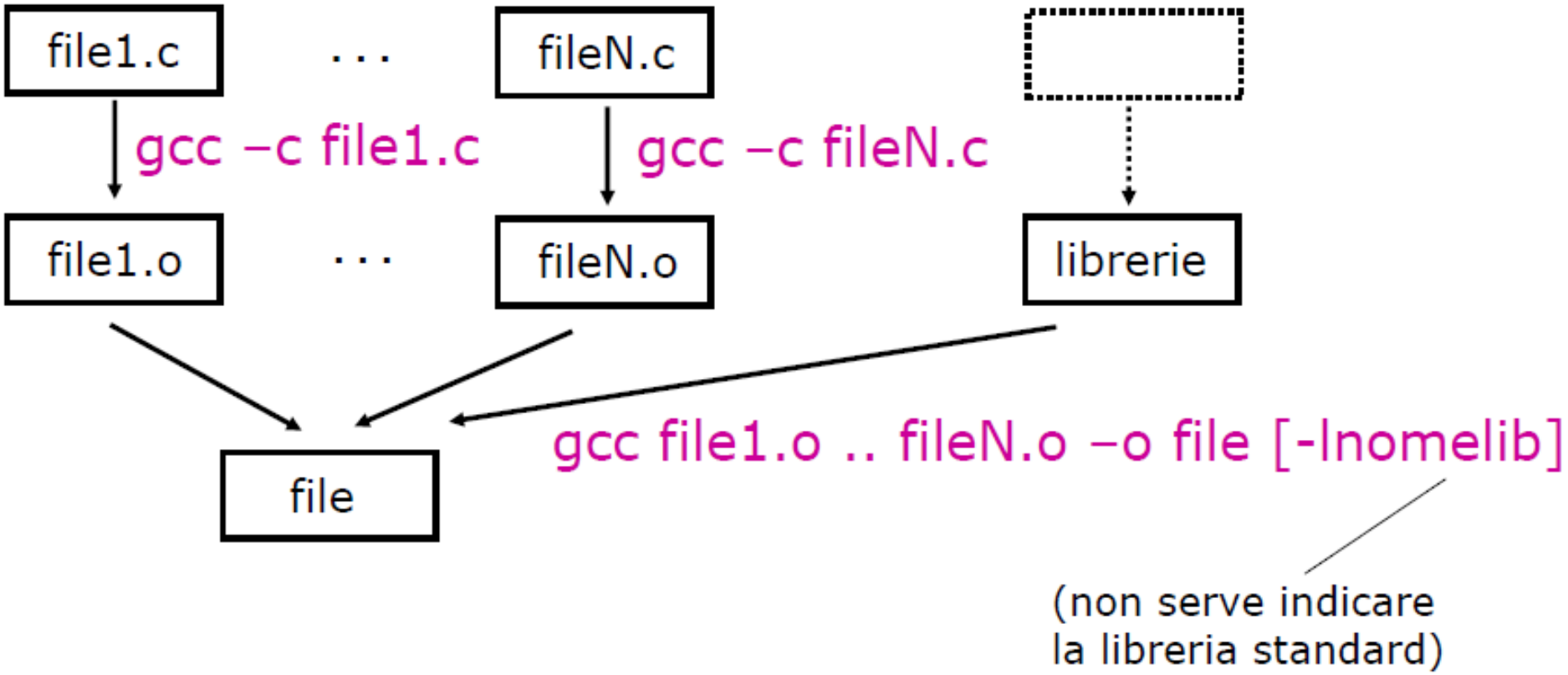


gcc file1.c .. fileN.c –o file

- conoscete il caso particolare N=1
- i file .o vengono creati e rimossi

Compilazione separata

- … o in singoli passi:



Programmazione strutturata

- Per maggiori informazioni vedere:
  - Kernighan & Ritchie, Il linguaggio C Principi di Programmazione, cap 4.
- Cos'è:
  - Con programmazione strutturata si intende la separazione di un programma in più file sorgenti separati chiamati anche **moduli**
  - Un modulo contiene un insieme di funzioni logicamente legate fra loro
    - Es: le funzioni per inserire nodo, estrarre un nodo, trovare un nodo in una lista;
- Perchè si usa:
  - Per favorire il riutilizzo del codice
    - Scrivo l'insieme di funzioni per operare sulle liste una sola volta in un modulo
    - Riutilizzo lo stesso modulo per differenti programmi
  - Le funzioni del modulo possono essere chiamate senza conoscerne l’implementazione, che può cambiare
  - Chi le utilizza deve solo sapere come **interagire** con le funzioni

- Per interagire con le funzioni bisogna conoscere:
  - La descrizione di cosa fa la funzione (non come lo fa)
  - I loro parametri di input (numero, significato, tipo, ecc…)
  - Il valore restituito dalla funzione
  - Eventuali altri comportamenti (**side effects**)
- Nota bene: come utilizzatori delle funzioni della libreria C state già usando sfruttando I vantaggi della programmazione strutturata !
- Ma da sviluppatori come si fa ?

Come si fa

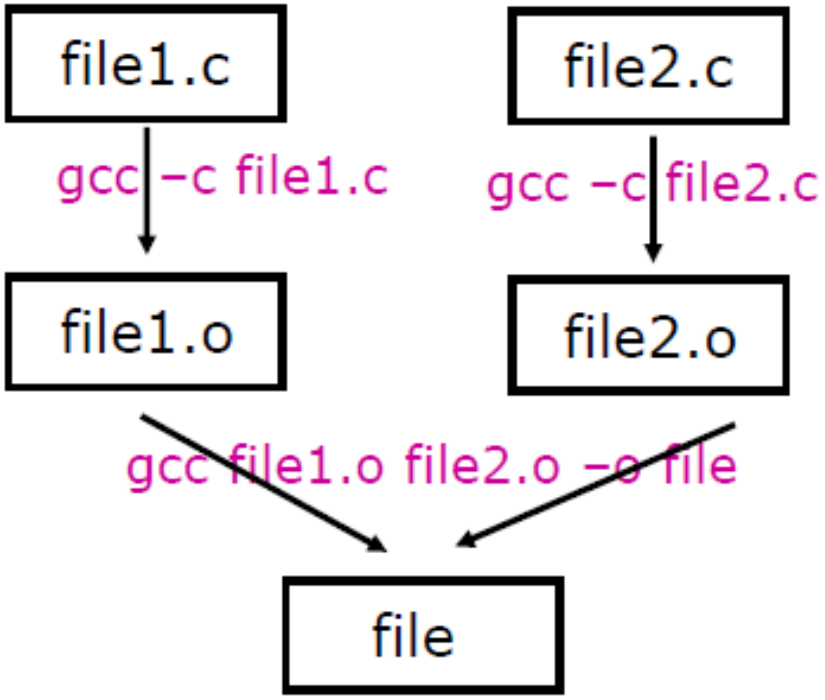
- Soluzione semplice: si preparano più file sorgente .c e si compilano
  - tutti assieme
  - separatamente e poi si linkano fra loro

  Problema:
  - Poichè si trovano in file separati, il compilatore come può assicurarsi che l'invocazione della funzione sia consistente con la definizione della funzione (in particolare con la dichiarazione degli argormenti e del tipo di ritorno)

- Serve un aiuto da parte dello sviluppatore aggiungendo i file .h tramite la direttiva #include `"file.h"`
  - È l'equivalente di un copia-incolla
  - Viene fatto dal pre-processore C

Il comando make

- Il comando make e i “makefile” da esso usati ci aiutano:
  - ad effettuare (solo) gli aggiornamenti necessari chiamando solo “make” quando uno dei file viene modificato – il che era più importante con le CPU lente di qualche annetto fa
  - a tenere traccia delle dipendenze – es. chi fornisce il sorgente di un pacchetto fornisce anche le dipendenze, chi lo riceve chiama solo “make”



Contenuto del “makefile”:

```
file: file1.o file2.o
<TAB>  gcc -o file file1.o file2.o
file1.o: file1.c
<TAB> gcc -c file2.c
file2.o: file2.c
<TAB> gcc -c file2.c
```

e si può fare molto altro...

Il comando make

`make -f nomefile` // prende i comandi da nomefile

`make -n` // elenca i comandi che dovrebbe eseguire ma non li esegue

`make TARGET` // esegue le azioni per creare TARGET (provare make clean nell'esempio fornito)

Se in un makefile inseriamo nella prima riga un target «fittizio» e lo dichiariamo dipendente da ogni altro eseguibile che vogliamo creare otteniamo un makefile che compila tutti gli eseguibili elencati.

NOTA:

non deve esitere nella cartella un file con lo stesso nome del target