Programmazione Avanzata e parallela

Lezione 02

Luca Manzoni

Cosa vedremo oggi

Organizzare progetti in C

- Compilazione separata (in C)
  - File header, object files
  - Librerie
- Opzioni del compilatore
  - Ottimizzazione
  - Debug

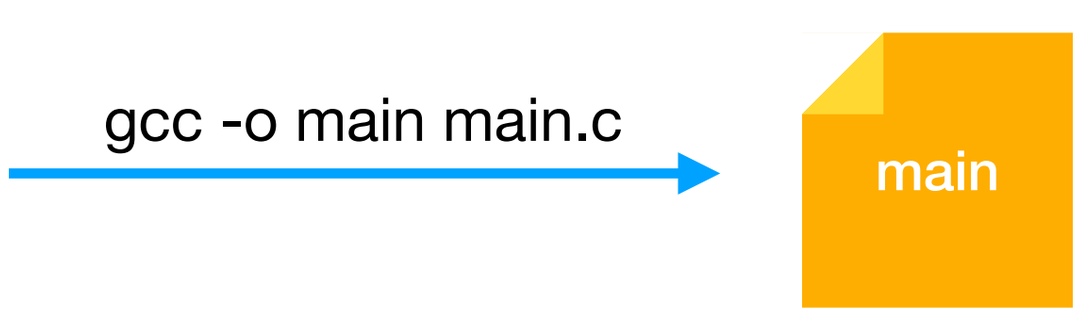
Compilazione semplice

Quello fatto fino a ora

```
int f(…)
{
  //…
}

int main(…)
{
  //…
  int x = f(…)
  //…
}
```

main.c



Compilazione semplice

Quello fatto fino a ora

- E se volessimo usare una funzione definita in main.c in un’altro programma?
- Dovremmo copiarla
- Solitamente non facciamo copia e incolla di “printf” o “malloc” da altri file, quindi qualcosa di meglio deve essere possibile
- Idea: dividere le funzioni e le definizioni di strutture su più file
- Serve un modo per dire “questa funzione esiste da qualche altra parte”

Spezzare in più file

E object file

- Possiamo spezzare in più file
- Dobbiamo dire al compilatore di fermarsi in un momento “intermedio” in cui:
  - Il codice macchina è stato generato…
  - …ma alcune funzioni non esistono ancora.
    Pensate a un “branch a [TODO]”
- Servirà poi un passo finale per mettere assieme tutto (“linking”)

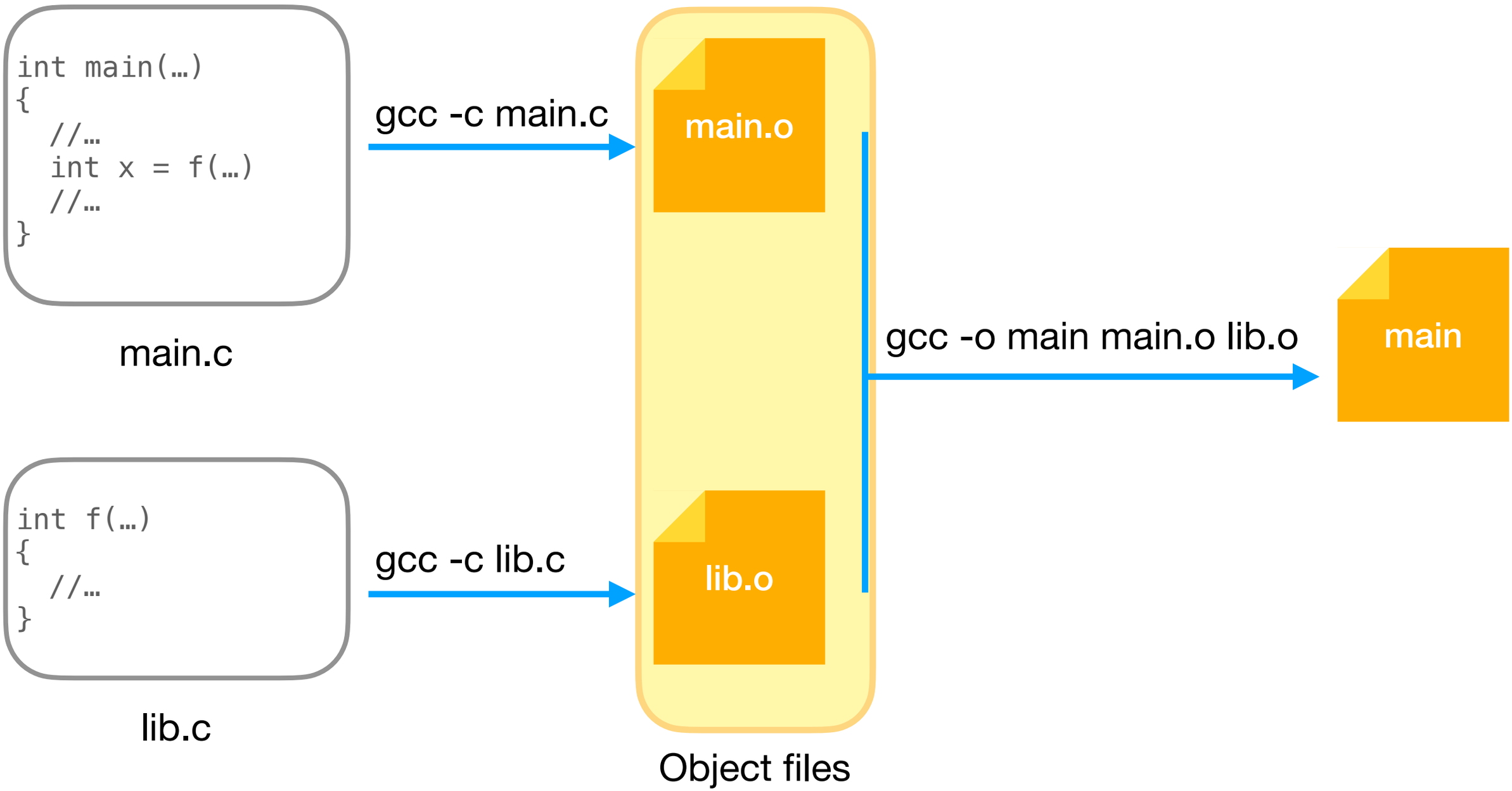
Spezzare in più file

Usare “extern”

- Dobbiamo poter dire che una funzione non definita nello stesso file “esiste”
- Per questo dichiariamo una funzione come “extern” e non la definiamo
- E.g., `extern int f(int x, int y);` in un file mentre un altro file conterrà l’effettiva definizione di f.
- Questo non può venir fatto con le strutture, però possiamo duplicare la definizione su più file (se è sempre la stessa non ci crea problemi)

Spezzare in più file

Processo di compilazione



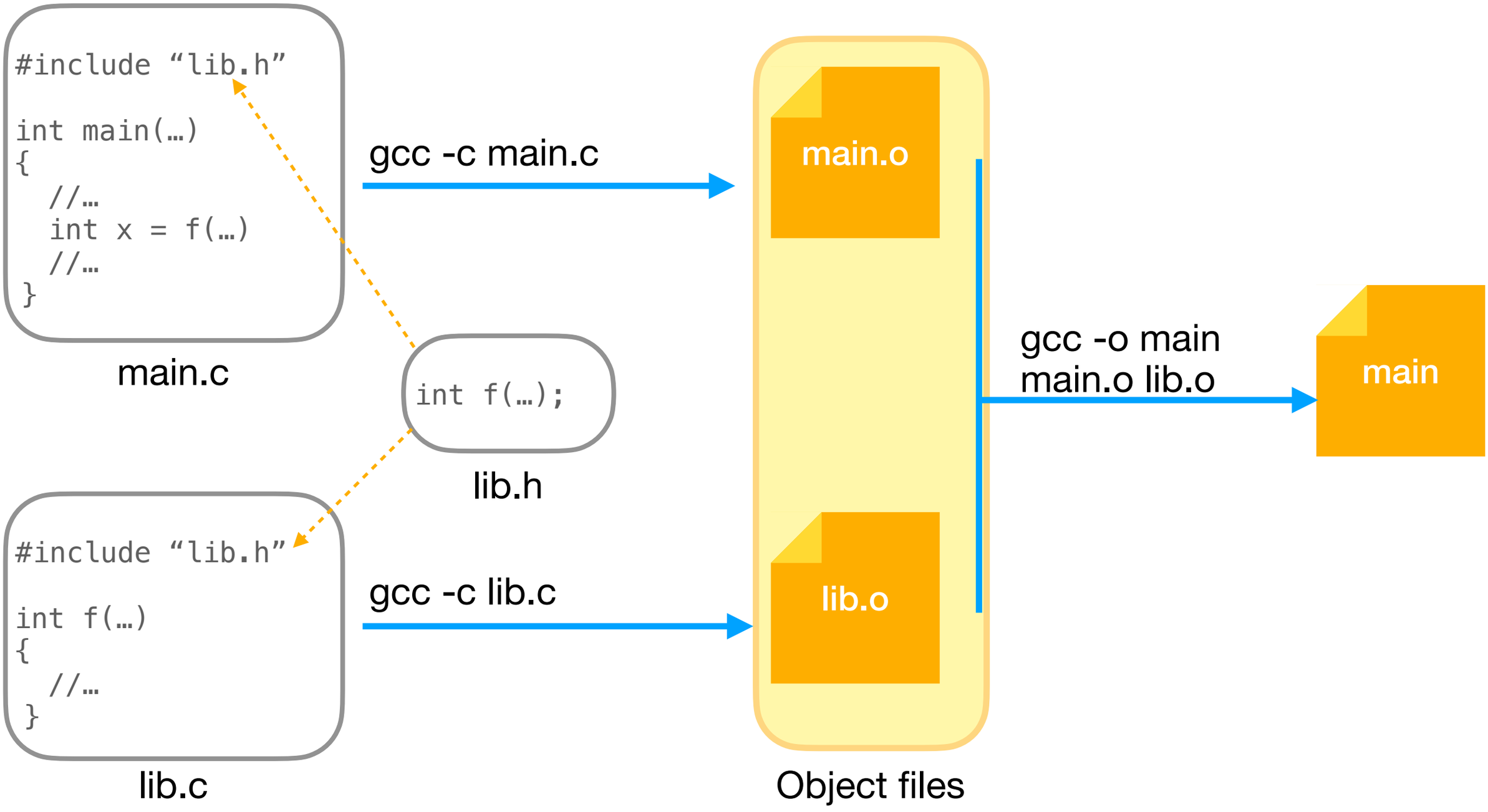
File Header

Gestire meglio la compilazione separata

- Per le funzioni l'uso di "extern" è scomodo
- E definire più volte le strutture può portare a problemi inattesi
- Sarebbe utile avere un file che dica "queste sono le strutture e le funzioni utilizzabili"
- Questo è il compito dei file header (solitamente con estensione .h)
- Questi sono inclusi con #include "file.h" o #include <file.h>
- La differenza è dove sono cercati i file header (directory locale o path definiti dal sistema)

Spezzare in più file

Processo di compilazione



File Header

Gestire meglio la compilazione separata

- Generalmente il file header include:
  - La dichiarazione delle funzioni
  - La definizione delle strutture da utilizzare
- È buona norma includere il file header anche nel file di implementazione (il corrispondente .c):
  - Per usare le definizioni di strutture
  - Per fare in modo che le dichiarazioni delle funzioni siano consistenti con la loro definizione

File Header

Inclusione multipla

- E se includiamo un file header più di una volta?
  - Per le dichiarazioni di funzioni nessun problema
  - Per le definizioni di strutture (e funzioni) invece c’è un problema di duplicazione della definizione!
- Verificare a mano di fare una sola inclusione è difficile
- Solitamente si risolve con delle macro del preprocessore C

File header

Evitare la doppia inclusione

```
#include “lib.h”

int main(…)
{
  //…
  int x = f(…)
  //…
}
```

main.c

Se non è già definito…

Identificativo univoco

```
#ifndef _LIB_H
#define _LIB_H

int f(…);

#endif
```

lib.h

```
#include “lib.h”

int f(…)
{
  //…
 }
```

lib.c

Alla seconda inclusione _LIB_H è già definito e non viene incluso nulla

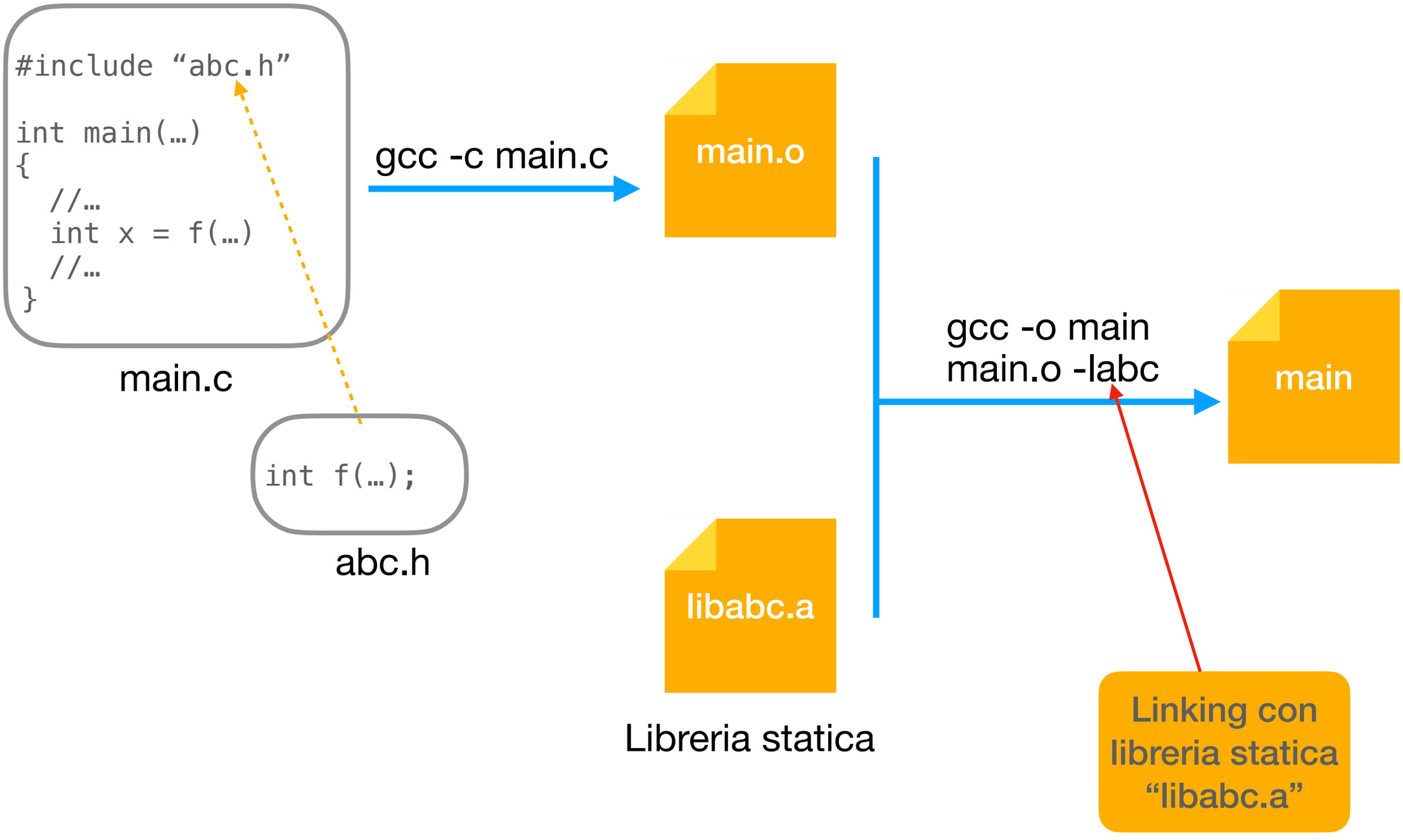
Librerie statiche

Rendere disponibile il vostro codice

- Una possibilità per rendere il codice (anche definito da più file) disponibile ad altri è tramite delle librerie
- Le librerie statiche sono incluse nel programma al momento del linking
- Le librerie dinamiche sono caricate quando il programma viene eseguito (ma devono essere presenti nel sistema)
- La creazione di librerie statiche è fatta tramite il comando “ar”
- È possibile esplorare il contenuto delle librerie statiche col comando “nm”

Librerie statiche

Processo di compilazione



Librerie statiche

Sequenza di creazione

- Innanzitutto compiliamo creando gli object file
  `gcc -c file1.c file2.c`

- Mettiamo assieme i file con “ar”:
  `ar r libabc.a file1.o file2.o`

- Ora serve creare un “indice” dei simboli della libreria (in pratica cosa la libreria fornisce):
  `ranlib libabc.a`

- Alternativamente è possibile fare i due passi in una volta sola:
  `ar rs libabc.a file1.o file2.o`

Librerie statiche

Utilizzo

- Per usare una libreria statica dobbiamo dire al compilatore due cose:
- Dove trovare la libreria. Questo è fatto tramite l'opzione -L a cui si aggiunge il path dove si trovano i file .a
- Dire al compilatore di fare linking con la libreria. Per questo si usa l'opzione -l con il nome della libreria *senza il "lib"* iniziale
- Quindi per linking con libabc.a nella directory attuale:
  `gcc -L. -o main main.c -labc`

Librerie dinamiche

Rendere disponibile il vostro codice (2)

- Le librerie dinamiche sono caricate quando il programma esegue (sono i file .so in Linux, .dylib su macOS e .dll su Windows)
- Hanno come vantaggi che:
  - Potete aggiornarle senza ricompilare tutto
  - Più programmi possono usare la stessa libreria senza duplicare il codice nell'eseguibile (risparmiando spazio)
  - Potenzialmente potete caricare alcune librerie solo quando necessario
- La loro creazione è però un poco più complicata

Librerie dinamiche

Processo di compilazione

```
#include “abc.h”

int main(…)
{
  //…
  int x = f(…)
  //…
}
```

main.c

```
int f(…);
```

abc.h

gcc -o main.c -L. -labc

main

Processo in cui “main” è eseguito

libabc.so

Libreria dinamica

Librerie dinamiche

Difficoltà di creazione e uso

- Il codice generato deve essere “position independent”, dato che non è possibile stabilire al momento delle compilazione in che indirizzi di memoria sarà messo
  - Questo viene fatto passando l’opzione -fPIC al compilatore
- Al momento dell’esecuzione il loader deve sapere dove trovare la libreria dinamica da caricare
  - In Linux (e diversi UNIX) si utilizza la variabile d’ambiente LD_LIBRARY_PATH
  - In macOS si utilizza la variabile d’ambiente DYLD_LIBRARY_PATH

Librerie dinamiche

Creazione

- Generiamo codice “position independent”:
  `gcc -fPIC -c abc.c`

- Creiamo la libreria dinamica:
  `gcc -o libabc.so -shared abc.o`

- Possiamo ora compilare usando la libreria dinamica:
  `gcc -o main main.c -L. -labc`

- Possiamo vedere che il programma “main” compilato necessita di alcune librerie statiche usando i comandi:
  `ldd main` (Linux)
  `otool -L main` (macOS)

Librerie dinamiche

Utilizzo

- La posizione nel filesystem di una libreria dinamica può variare
- Alcuni path possono essere salvati nell'eseguibile
- Altri sono dati dalle variabili d'ambiente
  LD_LIBRARY_PATH (Linux) o DYLD_LIBRARY_PATH (macOS)
- Possiamo vedere cosa contengono con
  `echo $LD_LIBRARY_PATH`
- Possiamo aggiungere dei path (come "./lib") alle variabili con:
  `export LD_LIBRARY_PATH=./lib:$LD_LIBRARY_PATH`
- Cosa succede se la libreria non è nel path?

Opzioni comuni del compilatore

Ottimizzazione

- **-On** con $n \in \{0,1,2,3\}$:
  - **-O0** disattiva le ottimizzazioni
  - **-O3** è (solitamente) il massimo livello di ottimizzazione
- **-march=native** per dire di utilizzare tutte le estensioni (e ottimizzazioni) che hanno senso sulla macchina specifica (e non su una “generica” della stessa architettura)
- **-march=[nome]** di solito indica l’architettura target della compilazione
- Nota: su macchine più vecchie/diverse l’eseguibile potrebbe non funzionare!

Opzioni comuni del compilatore

Debugging

- **-g** compila coi simboli di debug
  - Rende più semplice utilizzare i debugged (e.g., gdb) mantenendo cose come i nomi delle variabili, etc.
  - Generalmente non consigliato con livelli di ottimizzazione elevati
- **-profile-generate** compila per permettere il profiling
  - Quando eseguito i programma genera un file con le informazioni su come il codice viene utilizzato
  - Che puoi aiutare le ottimizzazioni tramite **-fprofile-use**