Esercitazione: Strumenti di sviluppo (Makefile, librerie, Git)

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

Sommario

- Sviluppo modulare
- Makefiles
- Librerie
- Git

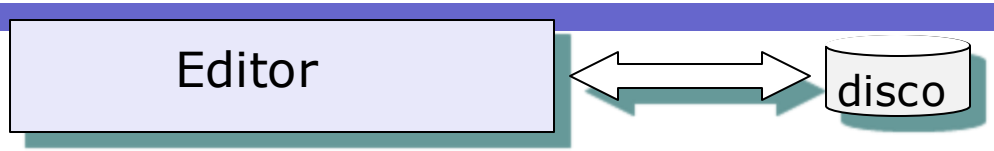
Riferimenti

- Dispense su Makefile e librerie

Esempi di codice

- https://github.com/rnatella/so_esempi

Ciclo di sviluppo dei programmi



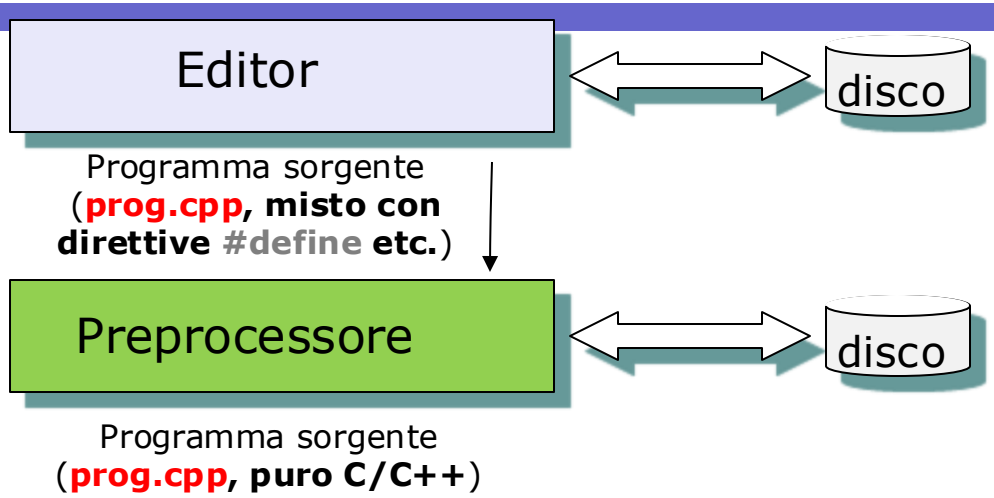
Programma sorgente
(**prog.cpp**, **misto con direttive #define etc.**)

Si inizia dai singoli file sorgente (unità di compilazione)

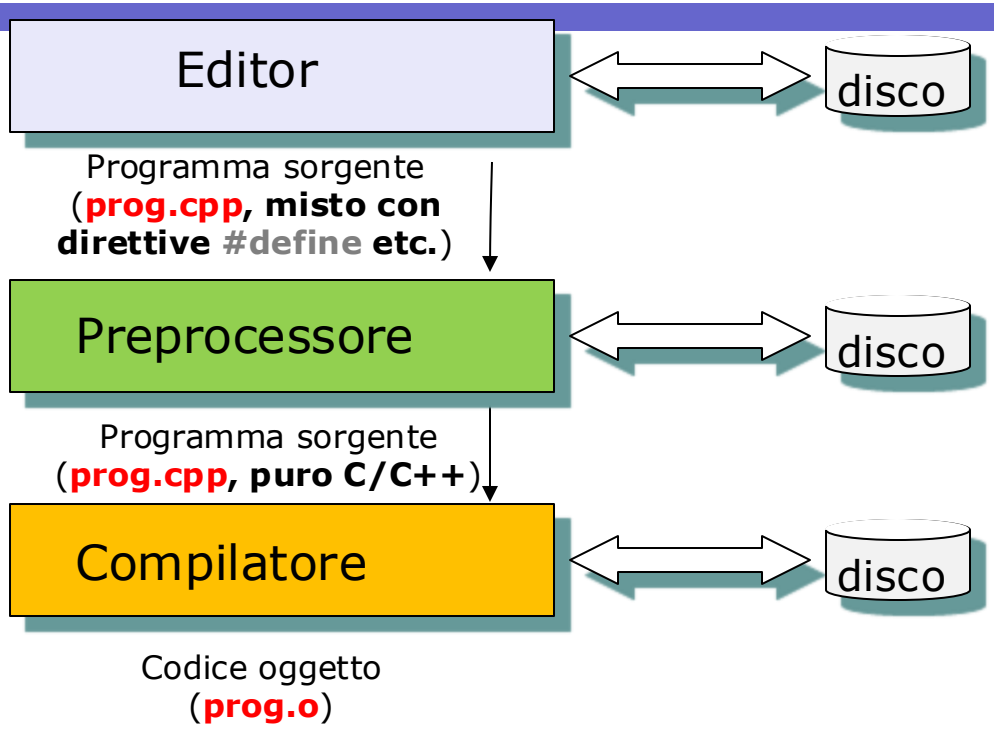
```
#include "header.h"
#define N 100

int main() {
    ...
}
```

Ciclo di sviluppo dei programmi



Ciclo di sviluppo dei programmi

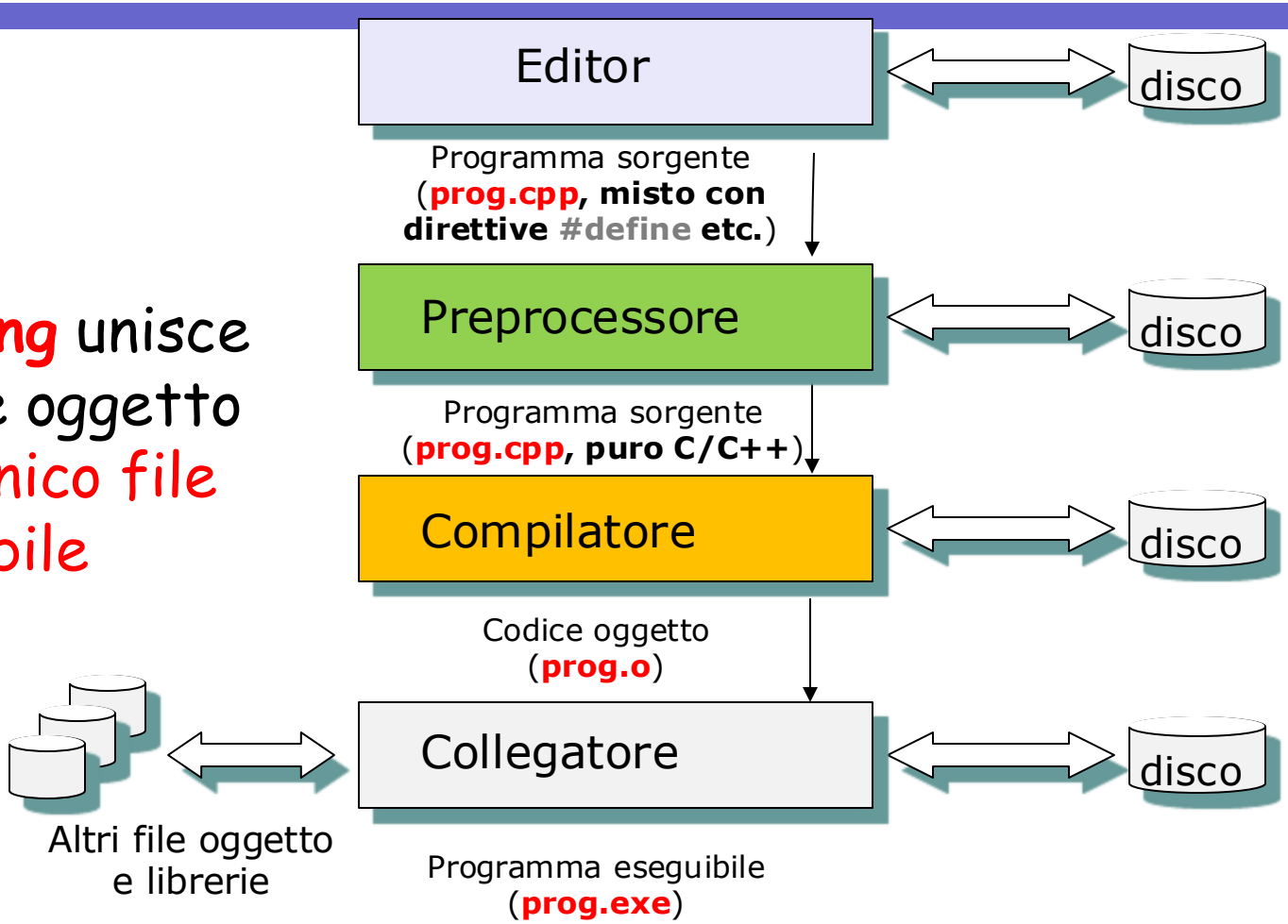


File oggetto: traduzione del sorgente in linguaggio macchina (**codice binario**)

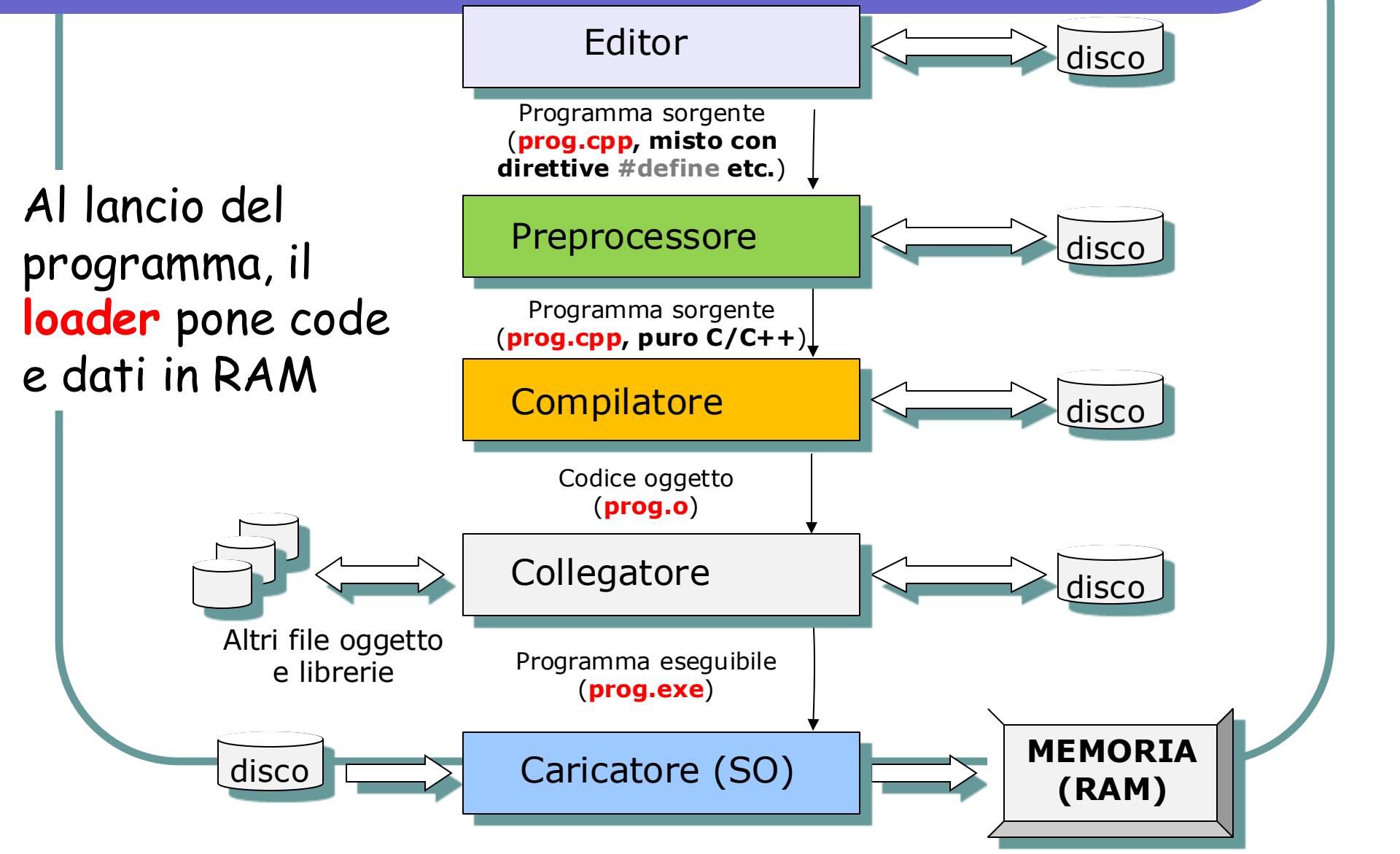
```
f3 0f 1e fa 55 48 89 e5
48 83 ec 10 8b 05 00 00
```

Ciclo di sviluppo dei programmi

Il **linking** unisce più file oggetto in un unico file eseguibile



Ciclo di sviluppo dei programmi



Al lancio del programma, il **loader** pone code e dati in RAM

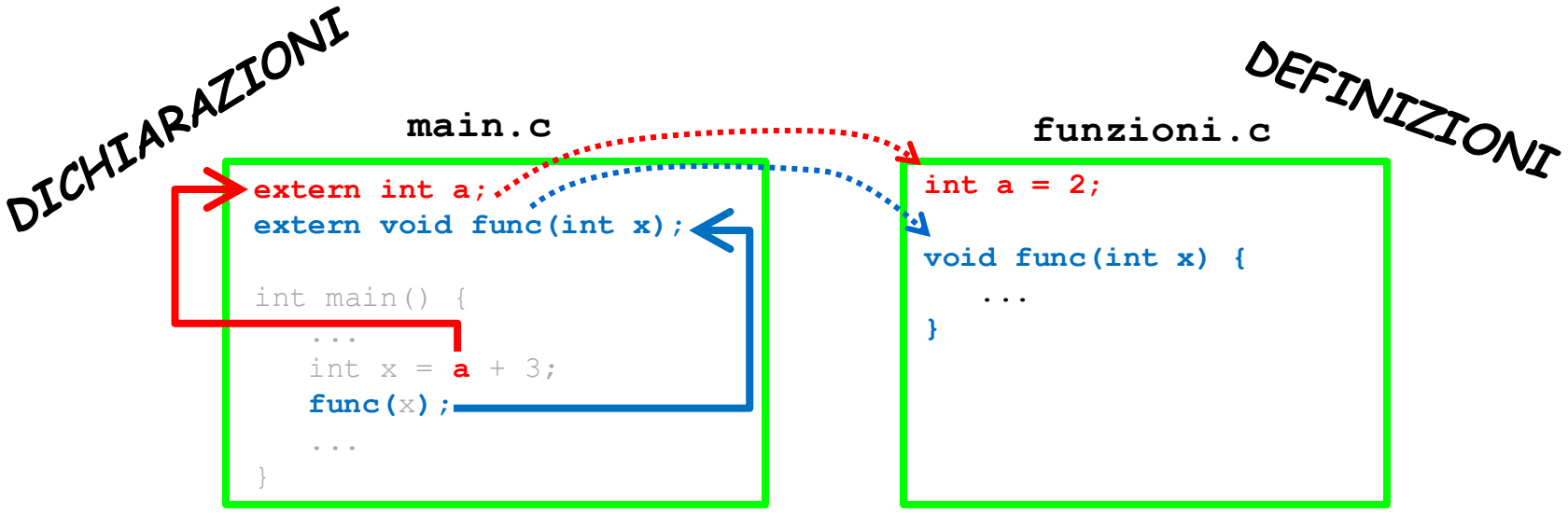
Sviluppo modulare

- I linguaggi *C* e *C++* hanno vari meccanismi per lo sviluppo modulare dei programmi:
  - la compilazione separata
  - l'inclusione testuale
  - l'uso dei prototipi di funzioni

Sviluppo modulare

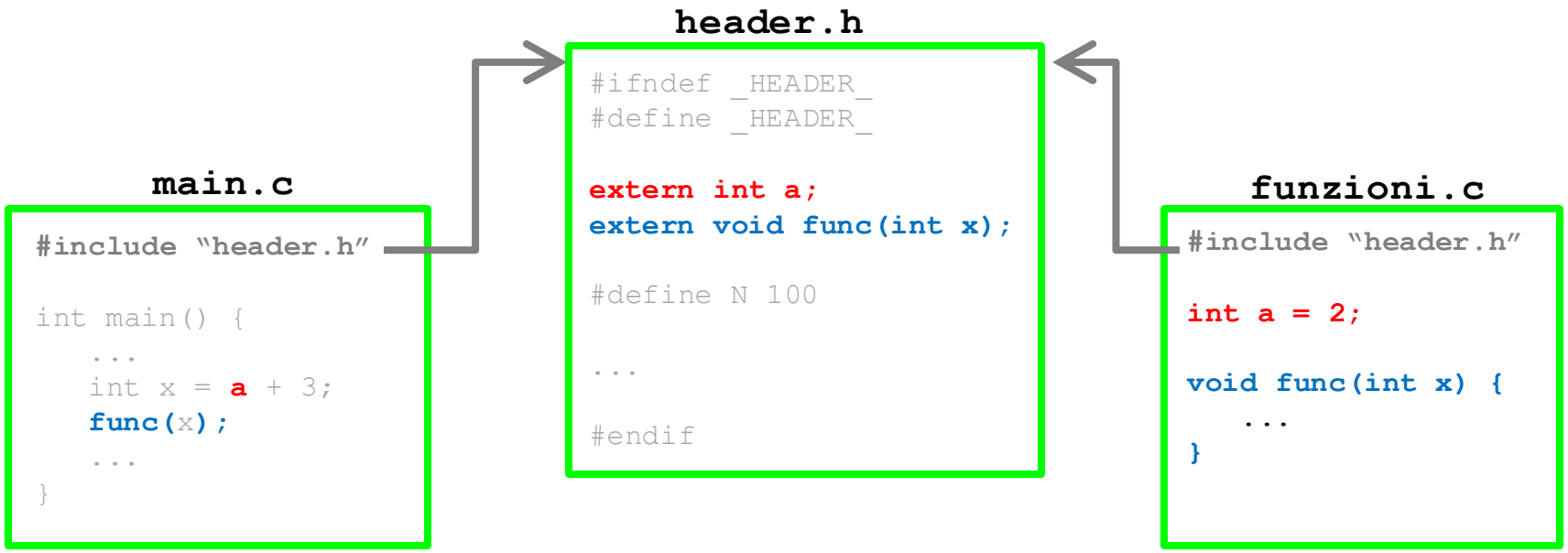
- Si separa la specifica di un modulo dalla sua implementazione
  - La specifica è fissa
  - L'implementazione è modificata nel tempo

Dichiarazioni e definizioni



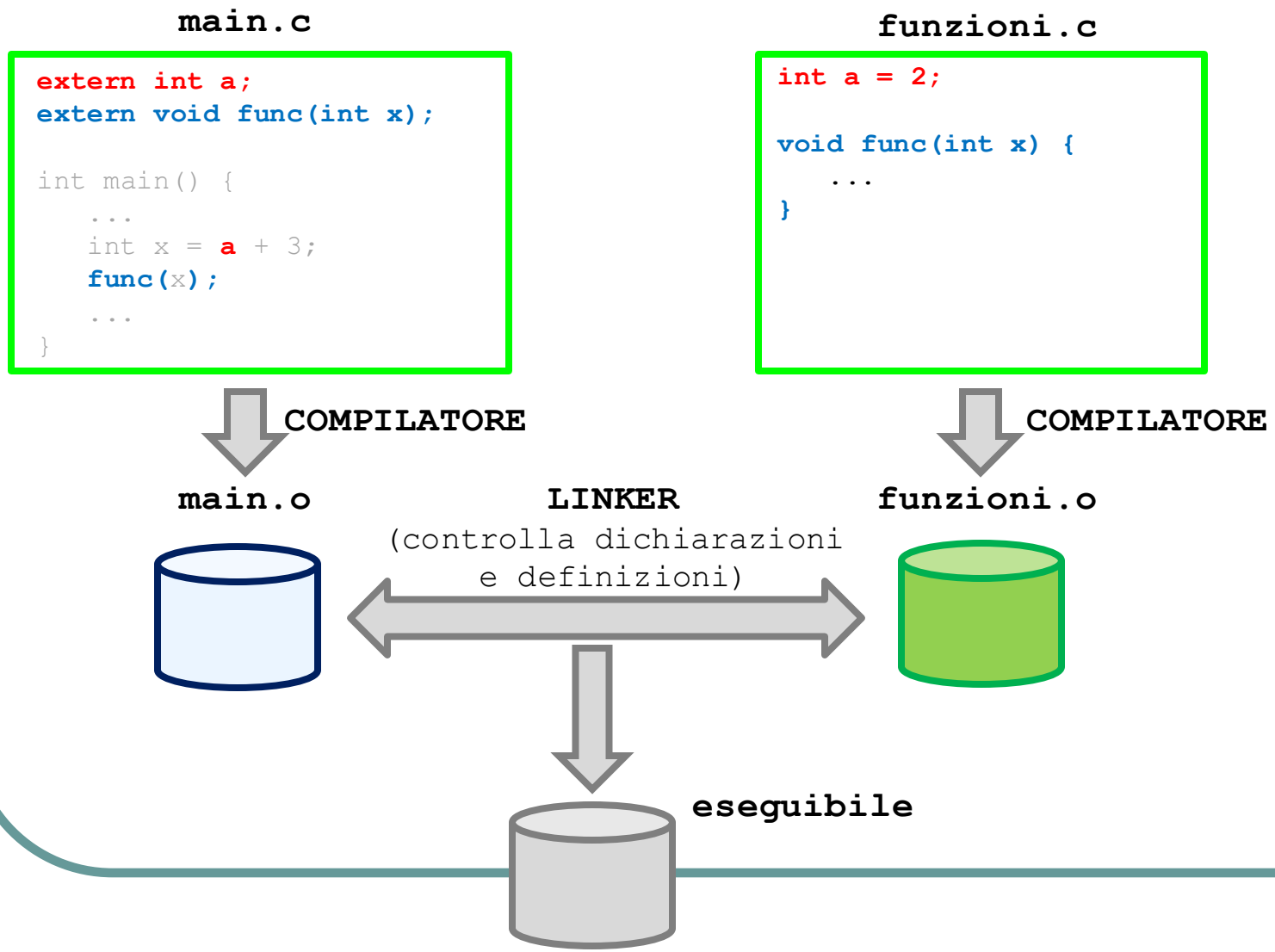
Un file può usare variabili e funzioni definite in un altro file

Pre-processore

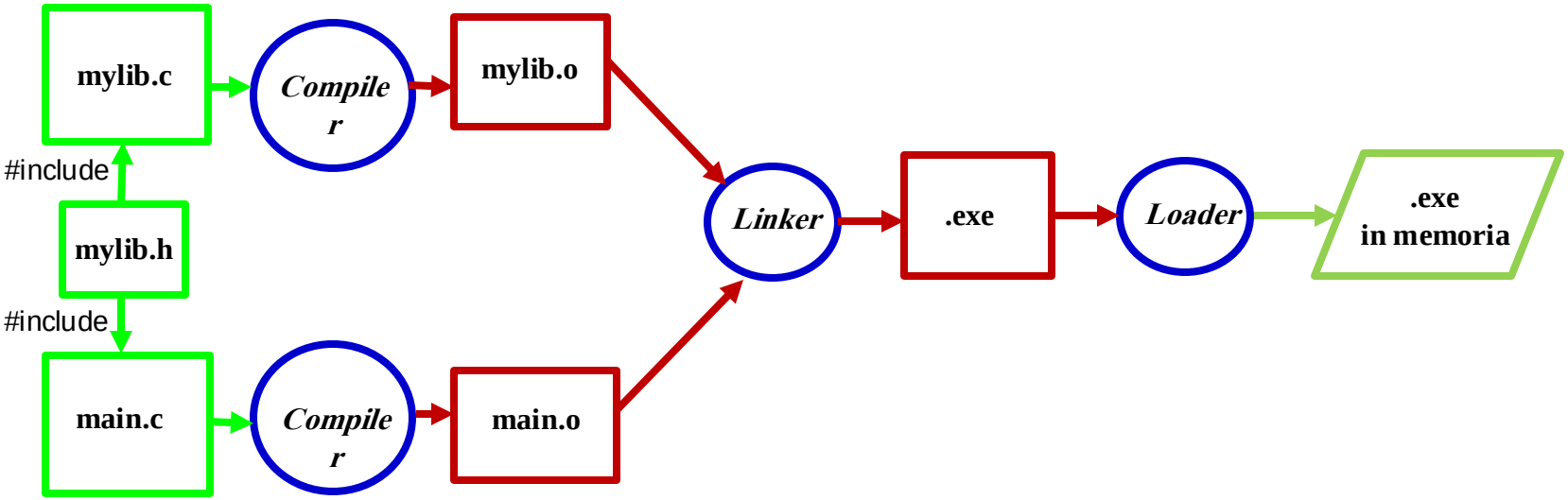


Le **dichiarazioni** sono inserite in un **file "header"**, da includere in tutti gli altri file

Linking


main.c
extern int a;
extern void func(int x);
int main() {
...
int x = a + 3;
func(x);
...
}
funzioni.c
int a = 2;
void func(int x) {
...
}
COMPILATORE
COMPILATORE
main.o
LINKER
(controlla dichiarazioni
e definizioni)
funzioni.o
eseguibile

Ricapitolando



Compilazione in Linux

Linux fornisce i compilatori open-source **gcc** e **g++** (rispettivamente per C e C++)

```
$ gcc -c  main.c  -o main.o

$ gcc -c  mylib.c  -o mylib.o

$ gcc  main.o  mylib.o  -o programma

$ ./programma
x = 5
```

Compilazione dei file oggetto (opzione "-c")

Linking dell'eseguibile (senza "-c")

Caricamento ed esecuzione

ELF (Executable and Linkable Format)

- Il codice oggetto in Linux è in un formato standard
- ELF (Executable and Linkable Format)
- Tool di analisi dei file oggetto: objdump ed nm

objdump

```
extern int a;
void func(int x);

int main() {
    ...
    int x = a + 3;
    func(x);
    ...
}
```

$ objdump -d main.o

```
main.o:     formato del file elf64-x86-64

Disassemblamento della sezione .text:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 10             sub    $0x10,%rsp
   c:   8b 05 00 00 00 00       mov    0x0(%rip),%eax
  12:   83 c0 03                add    $0x3,%eax
  15:   89 45 fc                mov    %eax,-0x4(%rbp)
  18:   8b 45 fc                mov    -0x4(%rbp),%eax
  1b:   89 c7                   mov    %eax,%edi
  1d:   e8 00 00 00 00          callq  22 <main+0x22>
  22:   b8 00 00 00 00          mov    $0x0,%eax
  27:   c9                      leaveq
  28:   c3                      retq
```

Posizione nel file

Istruzioni in linguaggio macchina (vettore di byte)

Puntatori vuoti alla variabile "a" e alla funzione "func()"

Rappresentazione in linguaggio assembler (sintassi AT&T)

objdump

$ objdump -d eseguibile

```
...
0000000000001149 <func>:
    1149:   f3 0f 1e fa             endbr64
    114d:   55                      push   %rbp
    114e:   48 89 e5                mov    %rsp,%rbp
    1151:   48 83 ec 10             sub    $0x10,%rsp
    1155:   89 7d fc                mov    %edi,-0x4(%rbp)
    1158:   8b 45 fc                mov    -0x4(%rbp),%eax
    115b:   89 c6                   mov    %eax,%esi
    115d:   48 8d 3d a0 0e 00 00    lea    0xea0(%rip),%rdi
    1164:   b8 00 00 00 00          mov    $0x0,%eax
    1169:   e8 e2 fe ff ff          callq  1050 <printf@plt>
    116e:   90                      nop
    116f:   c9                      leaveq
    1170:   c3                      retq

0000000000001171 <main>:
    1171:   f3 0f 1e fa             endbr64
    1175:   55                      push   %rbp
    1176:   48 89 e5                mov    %rsp,%rbp
    1179:   48 83 ec 10             sub    $0x10,%rsp
    117d:   8b 05 8d 2e 00 00       mov    0x2e8d(%rip),%eax
    1183:   83 c0 03                add    $0x3,%eax
    1186:   89 45 fc                mov    %eax,-0x4(%rbp)
    1189:   8b 45 fc                mov    -0x4(%rbp),%eax
    118c:   89 c7                   mov    %eax,%edi
    118e:   e8 b6 ff ff ff          callq  1149 <func>
    1193:   b8 00 00 00 00          mov    $0x0,%eax
    1198:   c9                      leaveq
    1199:   c3                      retq
    119a:   66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
...
```

Il programma finale (main.o + mylib.o) contiene l'immagine di memoria di **tutto il codice e dati**

Il **main()** è posizionato all'indirizzo di memoria **1171** (virtuale)

Gli indirizzi precedentemente vuoti sono riempiti dal linker

Make e Makefile

- Il comando **make** è una utility usata specialmente per **semplificare la compilazione separata** dei programmi

```
$ ls
Makefile  mylib.c  main.c

$ make
gcc -c main.c -o main.o
gcc -c mylib.c -o mylib.o
gcc main.o mylib.o -o programma

$ ls
Makefile  mylib.c  mylib.o
main.c    main.o   programma
```

legge Makefile

Make e Makefile

- Il **Makefile** è un file di configurazione, con lista di **regole**

- Ogni regola contiene:
  1. nome della regola ("target")
  2. file necessari per l'esecuzione della regola ("dipendenze")
  3. comandi da eseguire

Tipicamente, un Makefile contiene più regole per varie operazioni

- compilazione
- linking
- cancellazione dei file oggetto
- ...

Esempio: compilazione di un file oggetto

```
file.o: file.cpp file.h
	g++ -c file.cpp -o file.o
```

- Target: `file.o`
- Dipendenze: `file.cpp` `file.h`

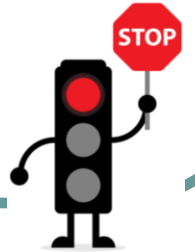
Il comando make controlla se le dipendenze (`file.cpp`, `file.h`) sono **più recenti** del target (`file.o`)

Sei i file .cpp/.h sono più recenti:

- significa che **i sorgenti sono stati modificati** dall'ultima volta che `file.o` è stato compilato
- il comando **viene eseguito**

Altrimenti:

- non ci sono state modifiche recenti
- il comando **non viene eseguito**



Esempio: linking di un eseguibile

```
eseguibile: obj1.o … objN.o
      g++ -o eseguibile  obj1.o … objN.o
```

- Target: `eseguibile`
- Dipendenze: `obj1.o`, ... `objN.o`

- Questa regola effettua il **linking** dei file oggetto .o
- Viene eseguita solo se i file sono stati **prima compilati**

Makefile: sintassi delle regole

```
target: dipendenze
[tab]    comando di sistema
```

Il carattere TAB è importante! make non accetta semplici spazi

Esempio completo

```
programma: mylib.o main.o
  gcc mylib.o main.o -o programma

mylib.o: mylib.c mylib.h
  gcc -c  mylib.c  -o mylib.o

main.o: main.c mylib.h
  gcc -c main.c  -o main.o
```

Regole:

1) compilazione mylib.o
2) compilazione main.o
3) linking (mylib.o + main.o)

Esempio completo

```
programma: mylib.o main.o
   gcc mylib.o main.o -o programma

mylib.o: mylib.c mylib.h
   gcc -c  mylib.c  -o mylib.o

main.o: main.c mylib.h
   gcc -c main.c  -o main.o
```

Le dipendenze hanno lo **stesso nome di altre due regole** nel Makefile

Le si esegue ricorsivamente

```
$ make
```

viene eseguita (di default) la **prima regola** nel Makefile

Esempio completo

```
programma: mylib.o main.o
  gcc mylib.o main.o -o programma

mylib.o: mylib.c mylib.h
  gcc -c  mylib.c  -o mylib.o

main.o: main.c mylib.h
  gcc -c main.c  -o main.o
```

```
$ make
```

Make verifica se le dipendenze sono state aggiornate.
Nel caso, vengono eseguiti i comandi di compilazione

Esempio completo

```
programma: mylib.o main.o
   gcc mylib.o main.o -o programma

mylib.o: mylib.c mylib.h
   gcc -c  mylib.c  -o mylib.o

main.o: main.c mylib.h
   gcc -c main.c  -o main.o
```

Make torna ad eseguire la prima regola, ed effettua il linking

```
$ make
```

Invocare regole singole

- È possibile anche far eseguire una sola regola del Makefile
- Ad esempio:

```
$ make mylib.o
```

- Compila solo mylib.c
- Non viene linkato l'eseguibile

Invocare regole singole

- Spesso il Makefile specifica una **regola di "clean"** (senza dipendenze)
- Cancella i file oggetto ed eseguibili

```
clean:
	rm -f  *.o
	rm -f  nomeEseguibile
```

- Per invocare la regola:

```
$ make clean
```

Librerie di moduli software

- I SO semplificano il riuso del codice mediante le **"librerie di codice"**
- Ad esempio, la **C Standard Library** (**libc**) fornisce le funzioni *printf()*, *strlen()*, etc.

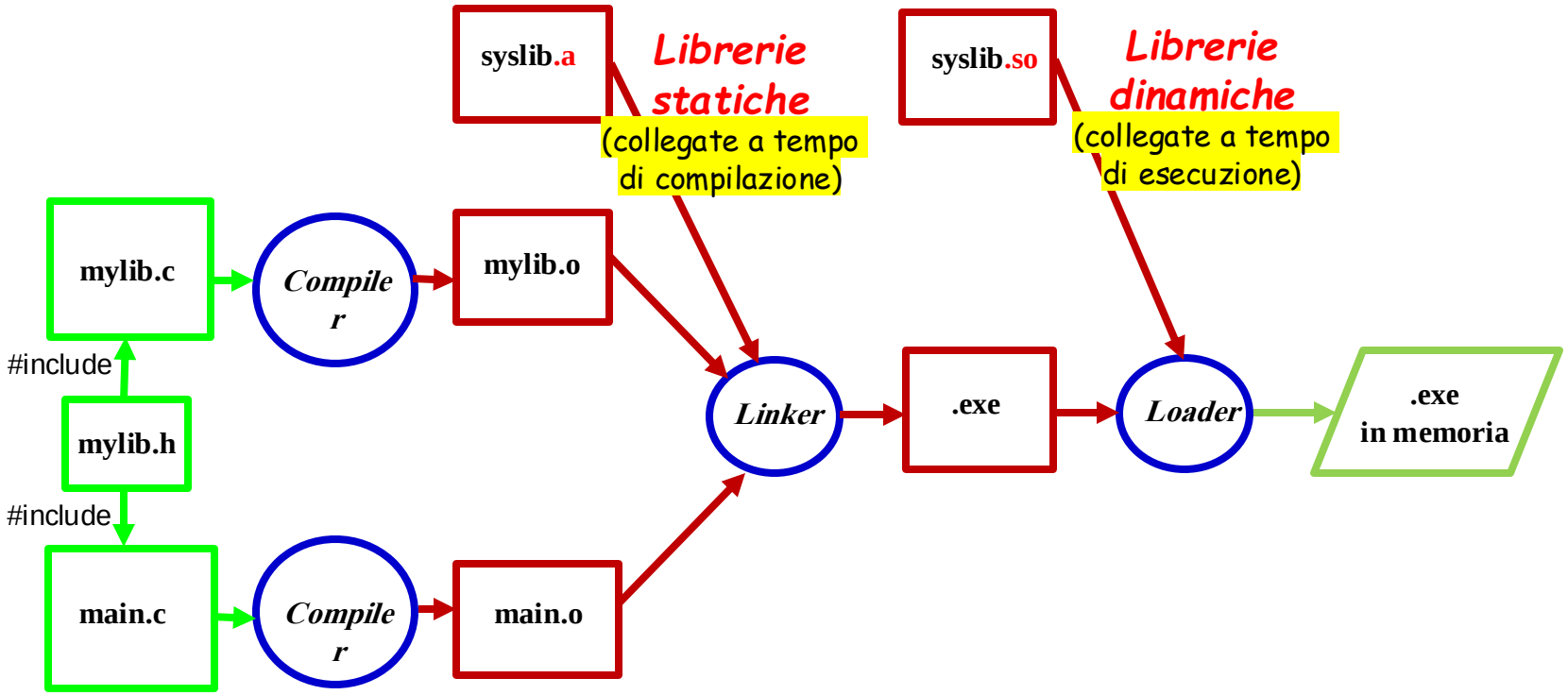
Librerie di moduli software

- Esistono **numerose librerie open-source** per lo sviluppo di applicazioni
  - grafiche (es. QT)
  - sicurezza (es. OpenSSH)
  - multimediali (es. FFMPEG)
  - database (es. SQLite)
  - ...

OpenSSH

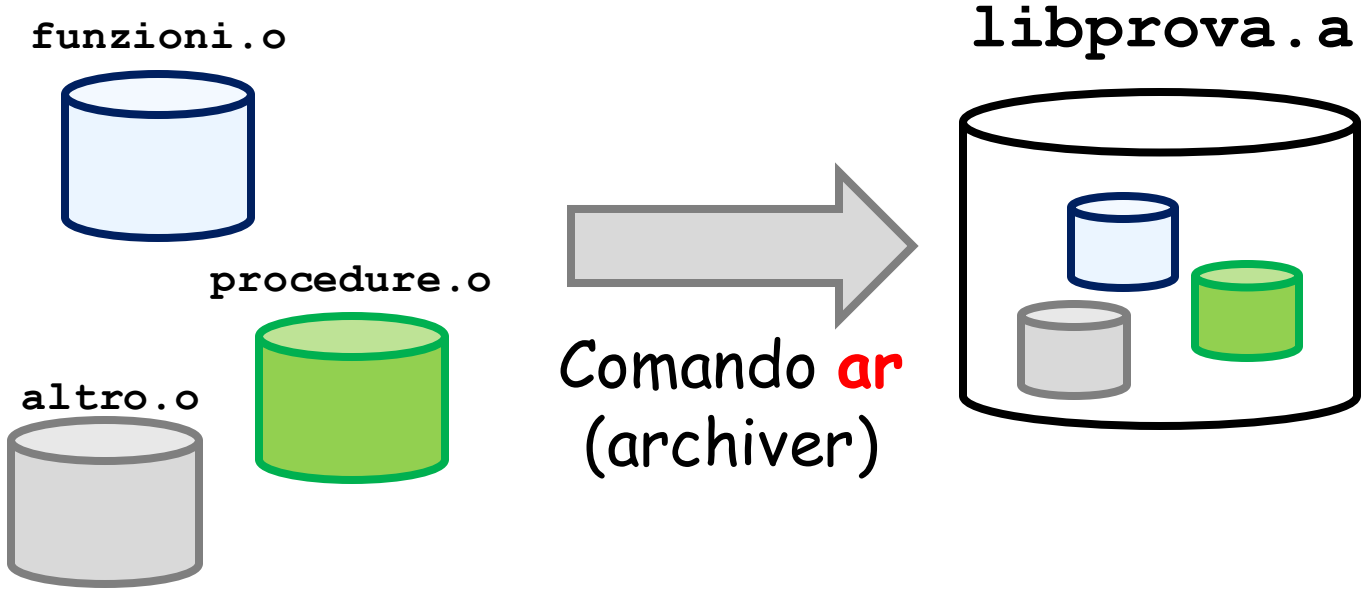
FFmpeg

Librerie in UNIX/Linux

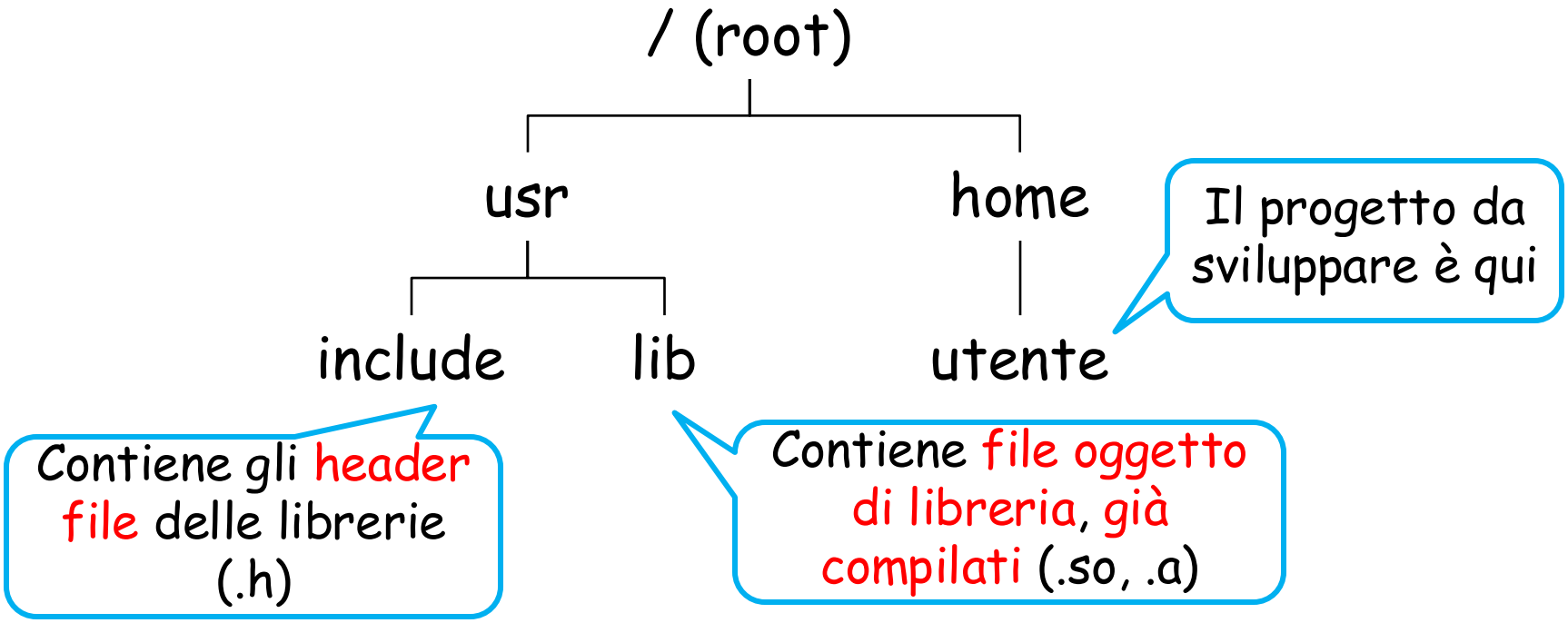


Librerie statiche

- Una libreria statica è un archivio di più file oggetto
- Prefisso "lib", estensione ".a"



Collegamento di una libreria



Collegamento di una libreria

Per collegare un eseguibile ad una libreria:

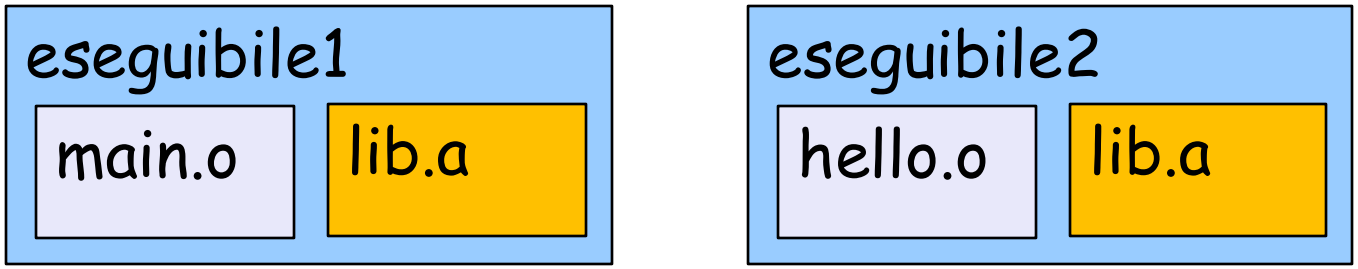
```
$ gcc -o prog obj1.o … objN.o -L/path/ -lprova
```

`-L`: percorso dove prelevare la libreria
(usare "." se è nella directory corrente)

`-l`: nome del file della libreria da collegare
(senza prefisso "lib" e senza estensione ".a")

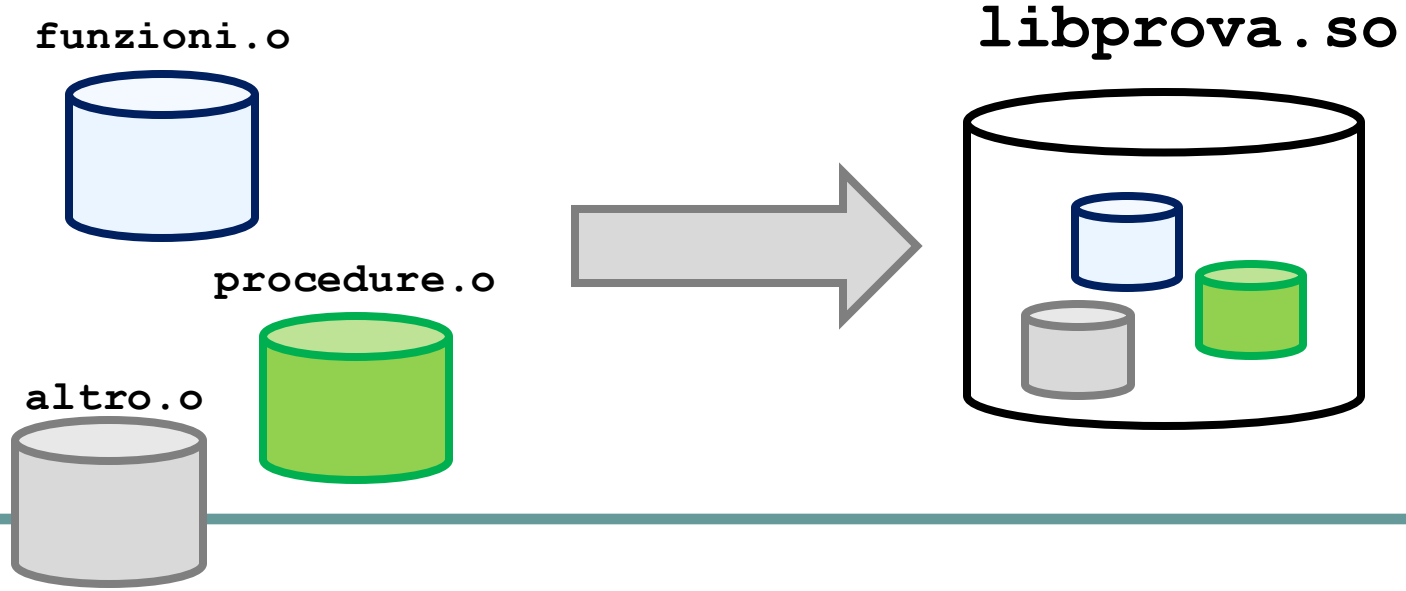
Librerie statiche

- Maggior consumo di spazio su disco degli eseguibili
- Maggior consumo di spazio in memoria RAM
  - Se due programmi usano la stessa libreria, sarà copiata in entrambi i programmi
- Se una libreria viene aggiornata, occorre ricompilare anche le applicazioni



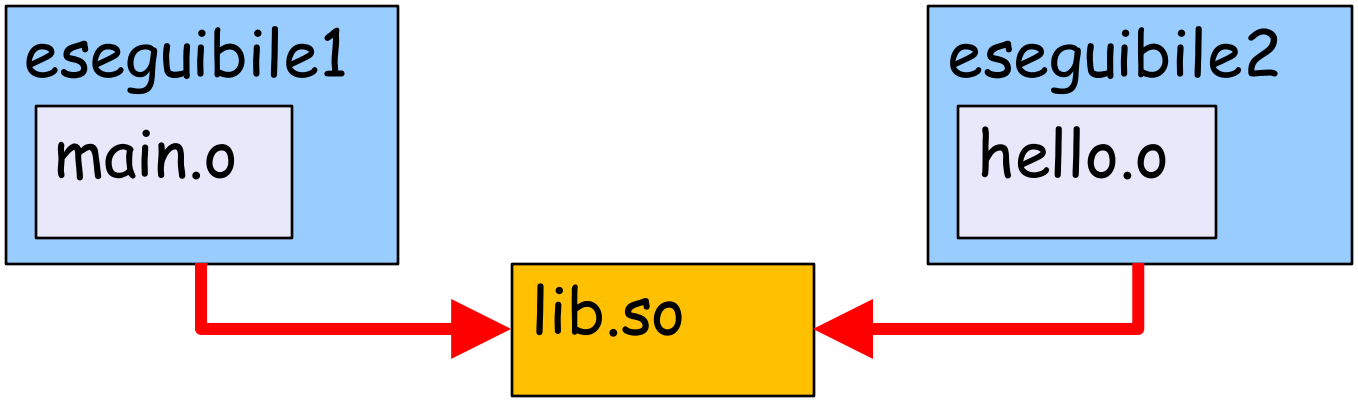
Librerie dinamiche

- A differenza delle librerie statiche, sono collegate all'eseguibile in fase di esecuzione
- Prefisso "lib", estensione ".so"



Librerie dinamiche

- Minore consumo di spazio su disco degli eseguibili
- Minore consumo di spazio in memoria RAM
  - la libreria è caricata **una sola volta** per tutti i programmi
- È possibile aggiornare una libreria senza ricompilare



Uso di librerie dinamiche

La libreria dinamica viene indicata in fase di linking

```
$ gcc -o prog obj1.o … objN.o -L/path/ -lprova
```

La libreria è caricata in memoria solo al momento della esecuzione del programma

```
$ ./prog
...esecuzione...
```

il SO carica anche libprova.so, dalle cartelle di sistema

ldd

- Il comando **ldd** mostra le librerie dinamiche usate da un programma

```
$ ldd provaOrdDynLib

  linux-vdso.so.1 (0x00007ffd0af24000)
  libalgord.so => ./libalgord.so (0x00007fce73079000)
  libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007fce72e86000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fce72c94000)
  libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fce72b45000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fce73085000)
  libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fce72b2a000)
```

Uso di librerie dinamiche

- Se la libreria non è in una cartella di sistema, occorre **indicare il percorso** tramite la shell
- **LD_LIBRARY_PATH**: variabile di sistema, contiene una lista di percorsi (separati da ":")

```
$ export LD_LIBRARY_PATH+=":/path"

$ ./prog

...esecuzione...
```

Git

- **Git** è un sistema di gestione del codice sorgente
  - Salva in remoto i sorgenti e la loro cronologia
  - Condivisione fra sviluppatori

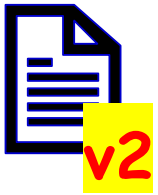
- **Github.com** è la piattaforma più famosa che adotta questo sistema

Flusso di lavoro

`git clone`
(scarica una copia dei file del progetto)

`git add, git commit`
(salva modifiche in cronologia)

`git push`
(invia modifiche)



progetto.c

progetto.c

Prima di iniziare

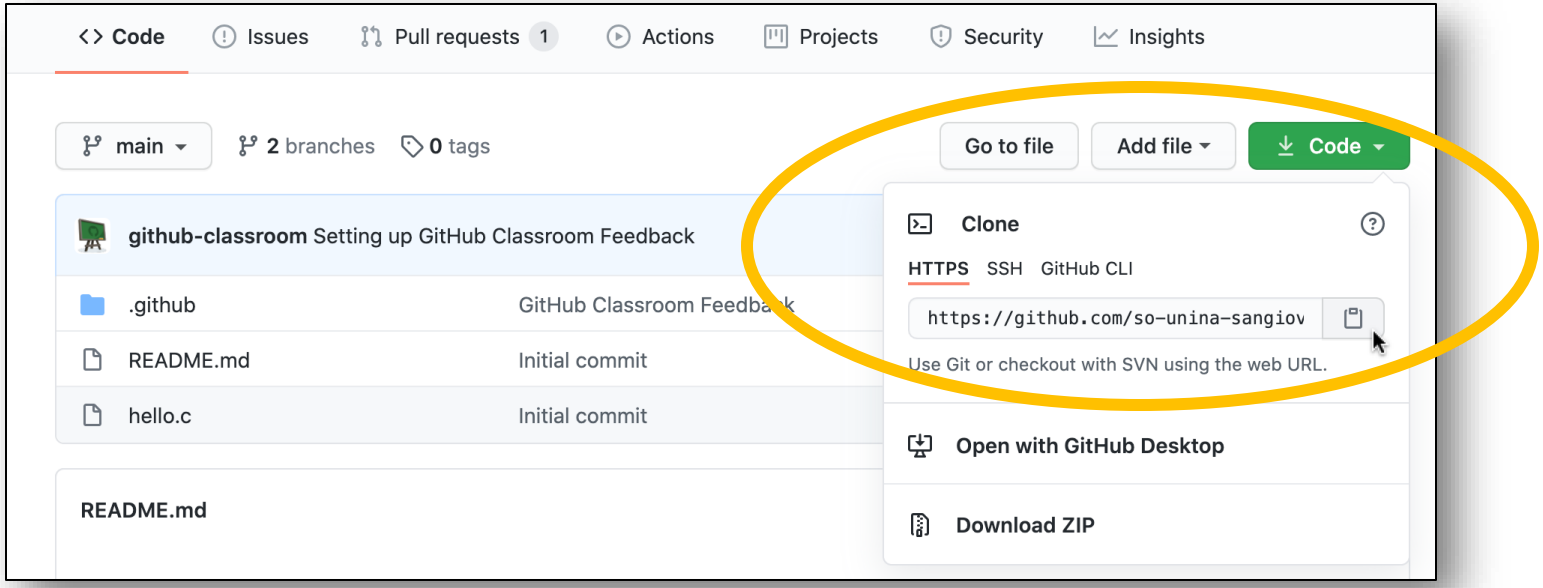
- Prima di iniziare, utilizzare i seguenti **comandi nel terminale (solo una volta!)**
- Indicate il vostro nome, cognome, email
- Appariranno nella cronologia delle modifiche

```
$ git config --global user.email "nome.cognome@studenti.unina.it"
$ git config --global user.name "Nome Cognome"
```

Nota sulla privacy

- Le nostre esercitazioni sono su repository privati (personali)
- I vostri dati inseriti **non saranno visibili in pubblico**

git clone

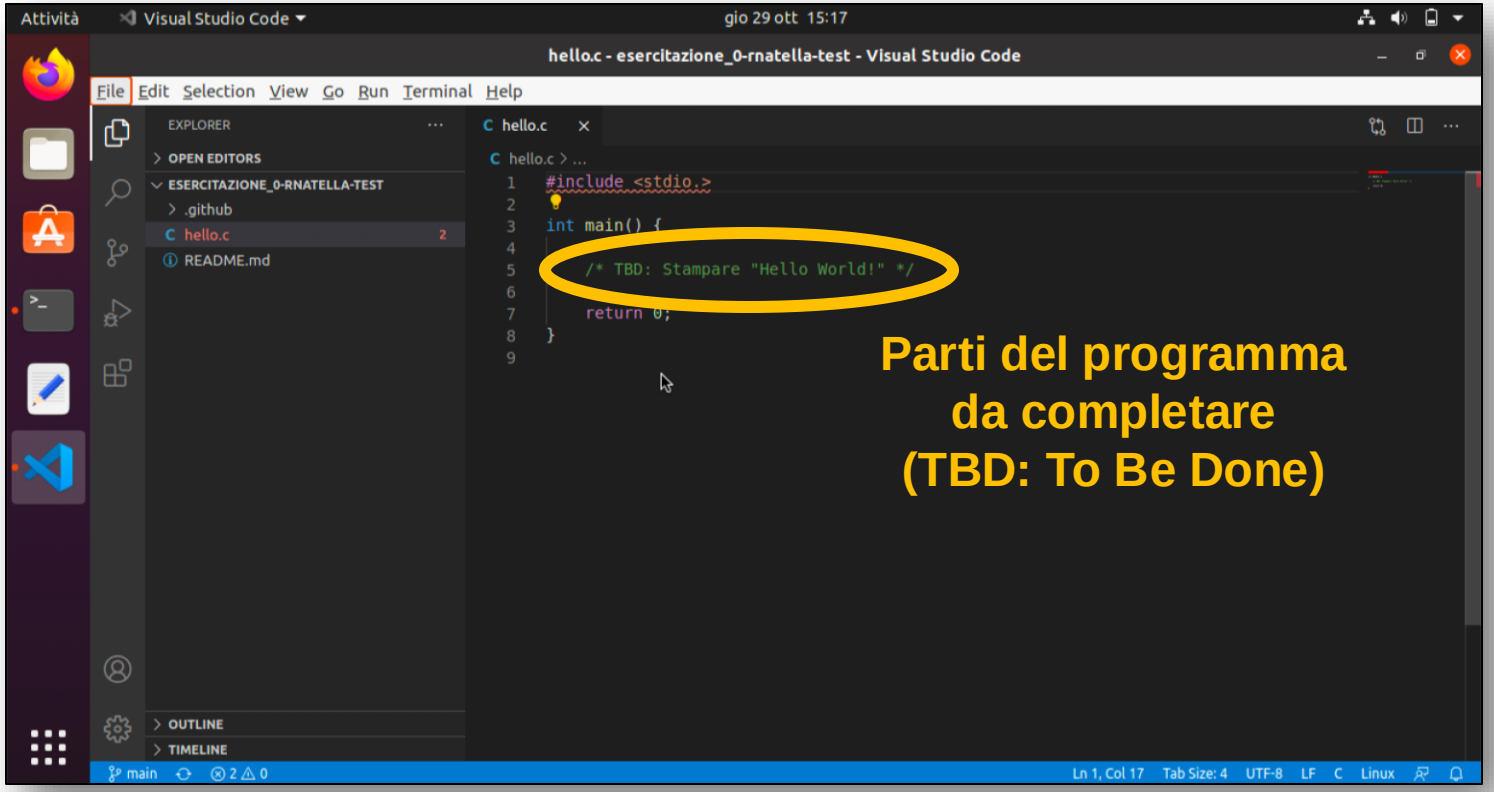



```
$ git clone <IL LINK DEL REPOSITORY PERSONALE, COPIATO DA github.com>

$ cd esercitazione_0-TUO-USERNAME

$ ... utilizzare make per compilare, e avviare da qui il programma ....
```

Svolgimento...



Per svolgere l'esercizio, è possibile aprire la cartella tramite Visual Studio Code

git add, commit, push

```
$ git add hello.c

$ git commit -m "Esercizio Hello World - Cognome Nome"

$ git push
```

1. Salvare i file
2. "**git add**": indica i file da includere nel commit
3. "**git commit**": salva **localmente** le modifiche, predisponendo per il push
4. "**git push**": copia le modifiche sul repository **online**

git add, commit, push

```
$ git add hello.c

$ git commit -m "Esercizio Hello World - Cognome Nome"

$ git push
```

Quali file includere nel commit?

- ❌ non includere i **file oggetto (".o")**
- ❌ non includere il **file eseguibile**
- ✅ includere i **file sorgente (".c" e ".h")**

Quanto spesso fare il commit?

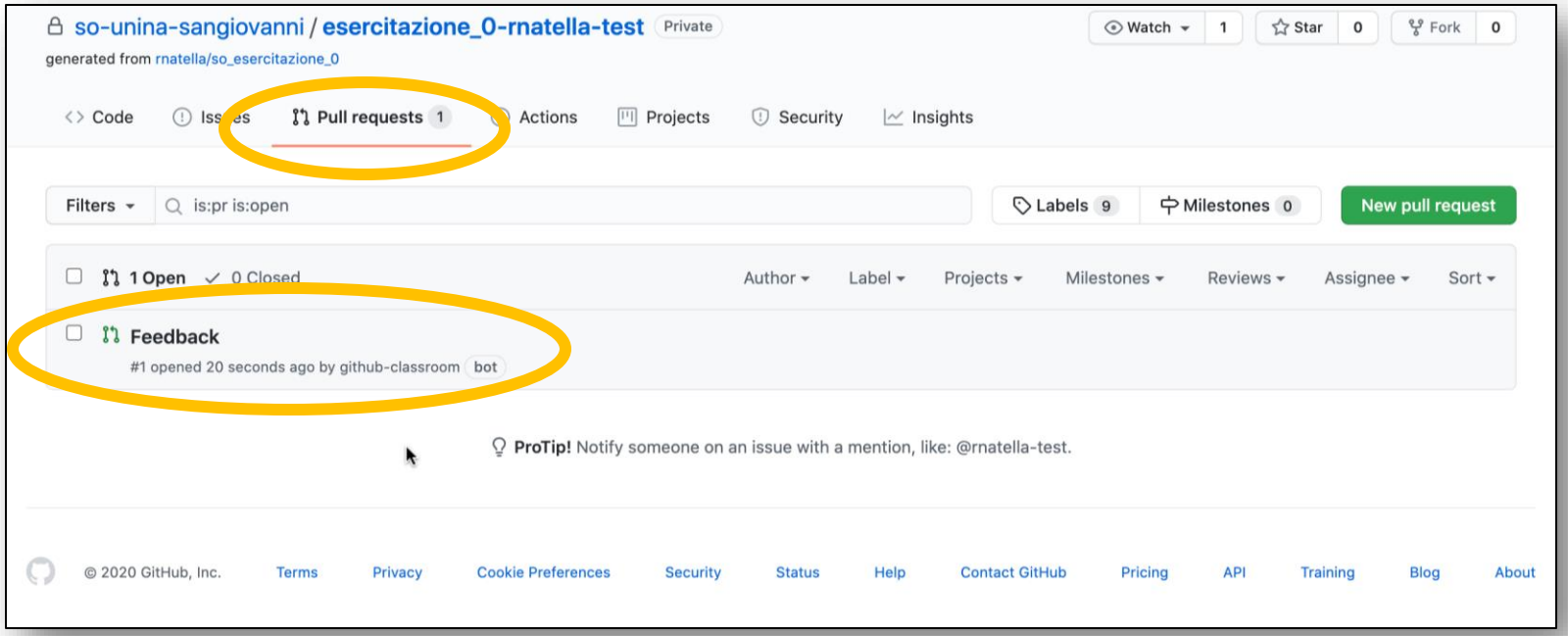
- Ogni volta che si aggiunge una nuova funzionalità
- Ogni volta che si corregge un bug
- ...

❌ **evitare i "micro-commit"**
(es. è inutile fare tanti piccoli commit in cui ripete la stessa modifica in parti diverse del programma, basta un solo commit con tutte le modifiche)

❌ **evitare commit "giganti"**
(es. non cumulare tante modifiche su più giorni/settimane)

https://softwareengineering.stackexchange.com/questions/74764/how-often-should-i-do-you-make-commits

Feedback



Cliccando su "**Pull requests**" e poi "**Feedback**", è possibile consultare eventuali **commenti e indicazioni** del docente sullo svolgimento che è stato consegnato

Feedback


rnatella reviewed 10 seconds ago
View changes
hello.c
2 2
3 3 int main() {
4 4
5 - /* TBD: Stampare "Hello World!" */
5 + printf("Hello World!");
rnatella 10 seconds ago
Ricorda sempre di includere "\n" alla fine di ogni stampa!
printf() accumula i caratteri in un buffer di memoria interno alla libreria C.
I caratteri vengono effettivamente mandati a video quando viene rilevato "\n".
Se manca "\n", la stampa verrà ritardata a quando si accumula un numero elevato di caratteri.
Reply...

Soluzioni

Esercizi di Programmazione Concorrente in Linux

Corso di Laurea Triennale in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Le **soluzioni** degli esercizi, e **altri esercizi** ancora, sono disponibili su questo repository pubblico:

https://github.com/rnatella/esercizi_linux