

PROGRAMMAZIONE 2: SPERIMENTAZIONI

Lezione 8 – Organizzazione di un programma e utilizzo del make



Agenda

- Introduzione
 - Il comando wc
- Organizzazione di un programma in C
 - File .h e .c
 - Direttiva per il preprocessore #include
 - Compilazione di file multipli
- Utilizzo di make
- Ciclo di compilazione



Video lezione disponibile su YouTube: https://youtu.be/R2nv_Nqc37s

Introduzione

- Per molti utenti un programma "grande" è formato da poche centinaia di righe di codice.
- A livello industriale, i programmi vengono scritti di solito da gruppi di programmatori; viene quindi considerato "grande" un programma che supera le 10000 righe di codice.

Introduzione

- Il codice sorgente di un sistema operativo come **Windows** è **dell'ordine di 5 milioni di righe di codice** (se si pensa che la pagina di un libro può contenere 50 righe, ci vorrebbero 100 libri da 1000 pagine) e ciò riguarda solo la parte **kernel**.
- **Si può quindi facilmente intuire che la scrupolosa organizzazione di un programma sia fondamentale.**

Il comando `wc`

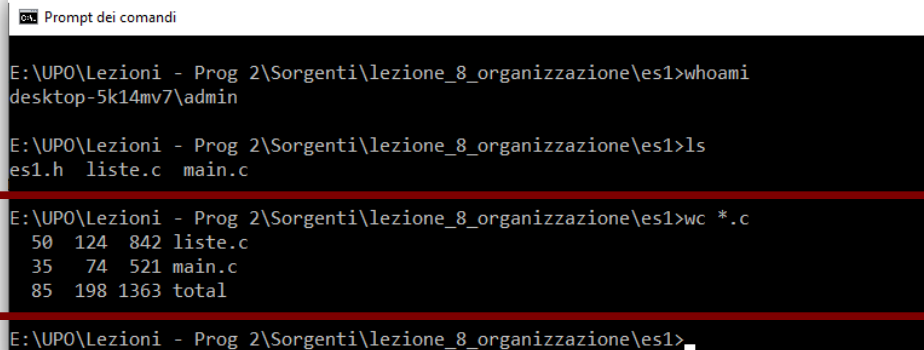
- In ambiente **Linux** è presente il comando `wc` (`w`ord `c`ount) che conta il numero di linee, parole e singoli caratteri presenti in un file.
- Es.:
 - `wc *.h *.c` (mostra righe, parole e caratteri di tutti i file `.h` e `.c` presenti nella directory in cui è eseguito);
 - `wc es1.c` (mostra righe, parole e caratteri del file `es1.c`);
 - `wc *.h *.c > README.txt` (salva nel file `README.txt` il numero di righe, parole e caratteri di tutti i file `.h` e `.c` presenti nella directory).

Il comando wc

- In ambiente **Windows** è possibile installare **GNU Core Utils**, un package contenente vari comandi Linux (tra cui `wc`) utilizzabili in ambiente MS-DOS.

 <http://gnuwin32.sourceforge.net/packages/coreutils.htm>

- Dopo aver installato il software, aggiungerlo al PATH di Windows per poterlo utilizzare più facilmente.



```
Prompt dei comandi

E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es1>whoami
desktop-5k14mv7\admin

E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es1>ls
es1.h  liste.c  main.c

E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es1>wc *.c
 50 124 842 liste.c
 35  74 521 main.c
 85 198 1363 total

E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es1>
```

Organizzazione di un programma

- Lo stile di scrittura di **ogni programma in C** in una directory apposita dovrebbe essere:
 - 1) collezione di file **d'intestazione** o **header** (estensione `.h`);
 - 2) collezione di file **d'implementazione** o **definizione** (estensione `.c`);
 - 3) un `makefile`.
- La regola di cui sopra si applica sia ai programmi "piccoli" che "grandi".

File .h e .c

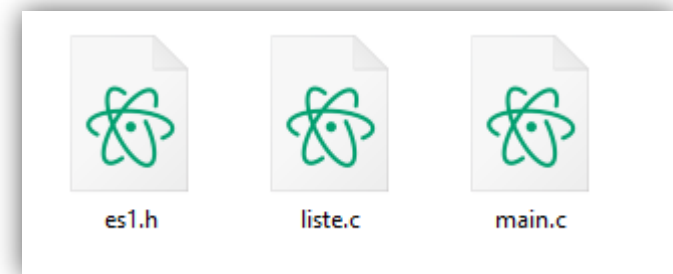
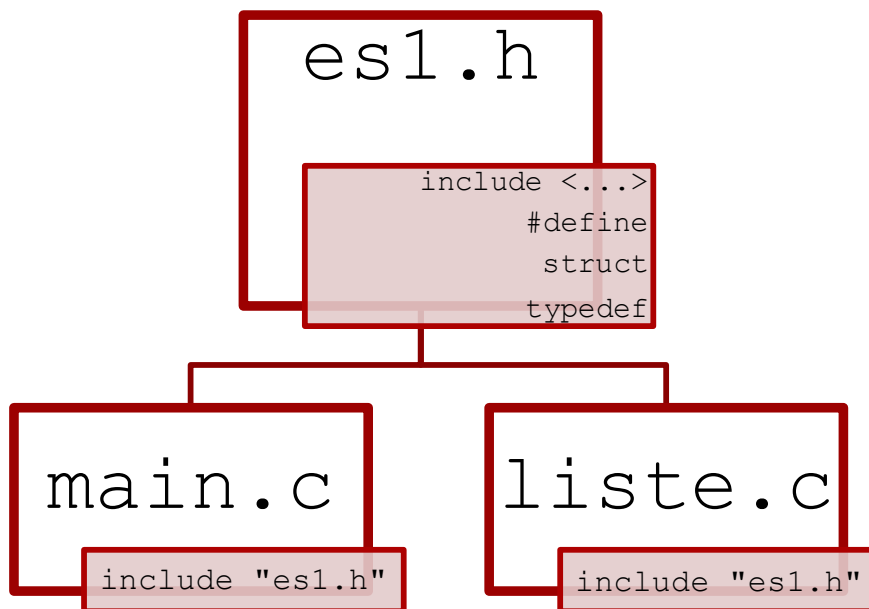
- Nei file **header** (estensione `.h`) vengono inseriti:
 - `#include` (librerie standard);
 - `#define` e costanti (macro);
 - definizioni di strutture (`struct`);
 - `typedef`;
 - prototipi delle funzioni.
- Solitamente il file `.h` contiene elementi che risultano utili in tutto il programma.

File .h e .c

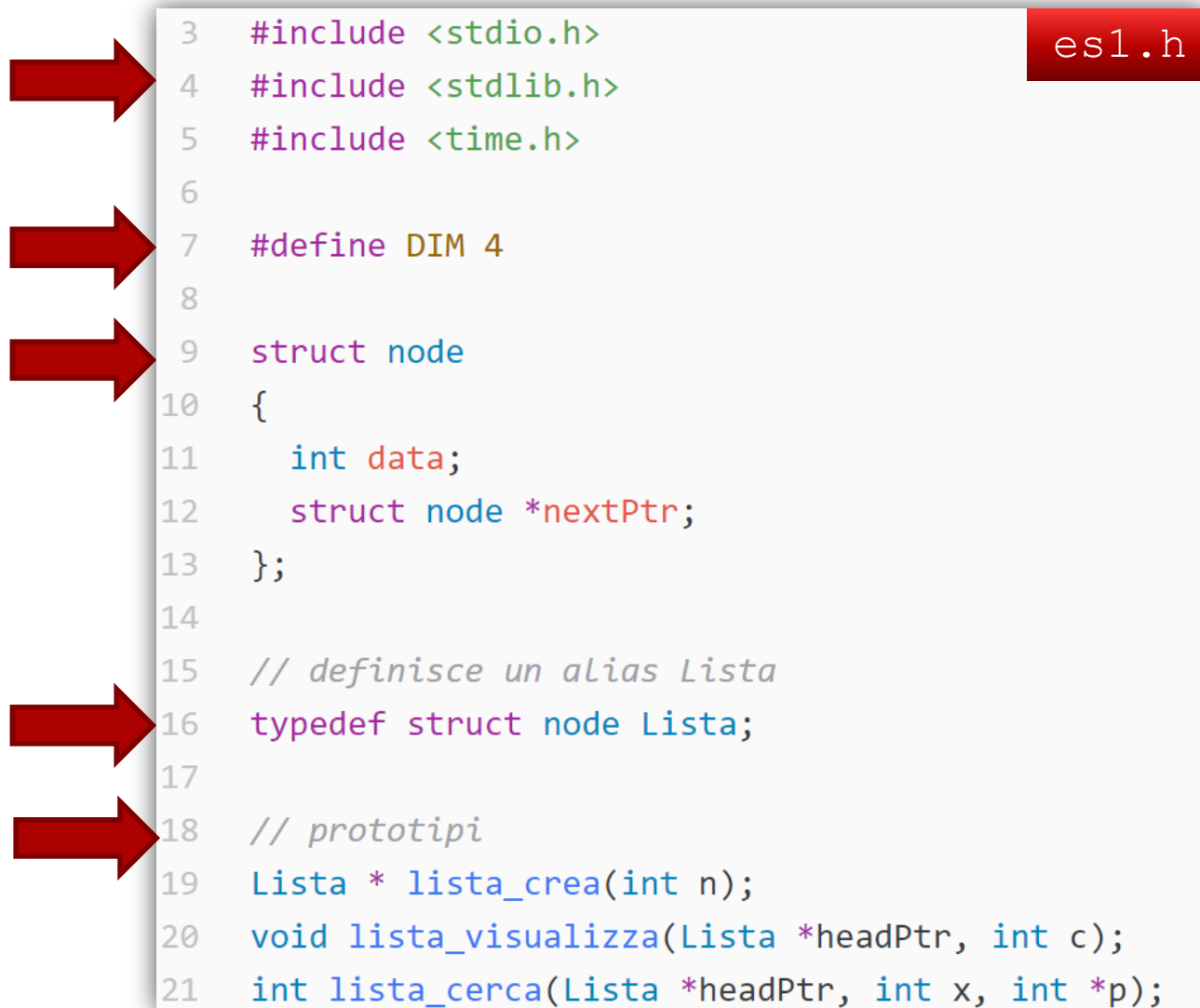
- Poiché il file d'intestazione `.h` agisce come "collante" tra le varie parti del programma, deve essere richiamato tramite `#include` in ogni file `.c` del progetto.
- Solitamente la funzione `main()` si trova in un file separato da quello in cui vengono definite altre funzioni; tale file si chiama generalmente `main.c`.

File .h e .c

- Supponendo di scrivere un programma che esegue varie funzioni su una lista e che tali funzioni vengano richiamate dal `main()`, una possibile organizzazione del progetto sarà la seguente.



File .h e .c



```

3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  #define DIM 4
8
9  struct node
10 {
11     int data;
12     struct node *nextPtr;
13 };
14
15 // definisce un alias Lista
16 typedef struct node Lista;
17
18 // prototipi
19 Lista * lista_crea(int n);
20 void lista_visualizza(Lista *headPtr, int c);
21 int lista_cerca(Lista *headPtr, int x, int *p);
  
```

es1.h

File .h e .c



```
#include "es1.h"
```

```
liste.c
```

```
2
```

```
3  Lista * lista_crea(int n)
```

```
4 > { }
```

```
20
```

```
21 void lista_visualizza(Lista *headPtr, int p)
```

```
22 > { }
```

```
34
```

```
35 int lista_cerca(Lista *headPtr, int x, int *p)
```

```
36 > { }
```

File .h e .c



```
#include "es1.h"
```

main.c

```
2
```

```
3 int main(void)
```

```
4 > { ... }
```

```
36
```

Direttiva per il preprocessore `#include`

- Come si è potuto notare, nei file `liste.c` e `main.c` è stato fatto un uso leggermente diverso della direttiva `#include`.
- La differenza tra `#include<...h>` e `#include "...h"` sta nella posizione in cui il **preprocessore** inizia a cercare i file da includere:
 - se il nome è racchiuso tra le parentesi angolari `<...>`, si indicano file di intestazione della libreria standard; in tal caso, il preprocessore cercherà i file nelle directory (cartelle) preconfigurate dal compilatore e dal sistema;
 - **se il nome è racchiuso tra virgolette** (doppi apici), **si indicano file da ricercare nella stessa directory** (cartella) **dove viene compilato il programma**. Tale metodo è solitamente utilizzato per includere intestazioni personalizzate dal programmatore.

Compilazione file multipli

- Una volta organizzato il file `.h` e i relativi file `.c`, è possibile procedere alla compilazione.
- Programmi con file di **definizione multipli devono essere compilati indicando tutti (e soli) i file .c.**

```
gcc -o es1.exe main.c liste.c
```

oppure (in ambiente Linux)

```
gcc -o es1.out main.c liste.c
```

Utilizzo di make

- Come si è potuto capire, mantenere in un unico file un programma di dimensioni medio-grandi che debba essere ricompilato ripetutamente, risulta inefficiente e costoso sia per il programmatore che per la macchina su cui è sviluppato.
- Una strategia migliore, come si è visto, consiste nel suddividere il programma in più file `.c`, da compilarsi separatamente quando necessario.

Utilizzo di make

- Se però si dovessero mantenere molti file sorgenti, sarebbe necessario ricompilare ogni volta tutti i file .c.

- Es.:

```
main.c modulo1.c modulo2.c modulo3.c
```

comporta

```
gcc main.c modulo1.c modulo2.c modulo3.c
```

- Si può facilmente intuire che ricompilare ogni volta tutti i moduli (file .c) comporta non solo perdite di tempo, ma anche dei rischi, in quanto è possibile commettere errori eseguendo queste compilazioni in modo manuale.

Utilizzo di make

- L'utility `make` viene in aiuto in questi casi, garantendo una compilazione esente da errori e solo di quei moduli di cui non esiste già il file oggetto aggiornato.
- L'utility `make` è un programma che di per sé non fa parte del `gcc`, perché non è altro che un **program manager**, utile a gestire un gruppo di moduli (file `.c`) di un programma o una raccolta di programmi.
- Sviluppata originariamente per UNIX, questa utility per la programmazione in C ha ricevuto il *porting* su altre piattaforme, tra cui Linux e Windows.

Utilizzo di make

- L'utility `make` utilizza un semplice file di testo di nome `makefile` (**senza estensione**), con all'interno **regole di dipendenza** e **regole di comando** (o d'azione).
- Considerando un programma contenuto in due file di nome `main.c` e `liste.c`, ciascuno dei quali include un file d'intestazione di nome `es2.h`, è possibile scrivere un `makefile` che genera un eseguibile `es2` (`es2.exe` in ambiente **Windows** o `es2.out` in ambiente **Linux**).

Utilizzo di make

makefile

```
1 es2: main.o liste.o
2     gcc -o es2 main.o liste.o
3 main.o: main.c es2.h
4     gcc -c main.c
5 liste.o: liste.c es2.h
6     gcc -c liste.c
7
```

Ricordando che:

- l'opzione `-o` sta per **output** e indica il nome da dare al file compilato;
- l'opzione `-c` serve a creare il file **oggetto** di un file sorgente;
- quindi `gcc -o es2 main.o liste.o` indica di generare un eseguibile chiamato `es2.exe` (o `es2.out`) utilizzando i file oggetto `main.o` e `liste.o`.

Utilizzo di make

makefile

```

1 es2: main.o liste.o
2     gcc -o es2 main.o liste.o
3 main.o: main.c es2.h
4     gcc -c main.c
5 liste.o: liste.c es2.h
6     gcc -c liste.c
7

```

- La **riga 1** indica che il file eseguibile `es2` dipende da due file oggetto che sono `main.o` e `liste.o`.
- La **riga 2** indica come il programma debba essere compilato nel caso uno dei due file oggetto sia stato modificato.

Utilizzo di make

makefile

```

1 es2: main.o liste.o
2     gcc -o es2 main.o liste.o
3 main.o: main.c es2.h
4     gcc -c main.c
5 liste.o: liste.c es2.h
6     gcc -c liste.c
7

```

- La **riga 3** indica che il file oggetto `main.o` dipende dal file di definizione `main.c` e `es2.h`.
- La **riga 4** indica come il file oggetto `main.o` debba essere ottenuto (ovvero come compilare il file di definizione).

Utilizzo di make

makefile

```

1 es2: main.o liste.o
2     gcc -o es2 main.o liste.o
3 main.o: main.c es2.h
4     gcc -c main.c
5 liste.o: liste.c es2.h
6     gcc -c liste.c
7

```

- La **riga 5** indica che il file oggetto `liste.o` dipende dal file di definizione `liste.c` e `es2.h`.
- La **riga 6** indica come il file oggetto `liste.o` debba essere ottenuto (ovvero come compilare il file di definizione).

Utilizzo di make

makefile

```

1 es2: main.o liste.o
2   → gcc -o es2 main.o liste.o
3 main.o: main.c es2.h
4   → gcc -c main.c
5 liste.o: liste.c es2.h
6   → gcc -c liste.c
7

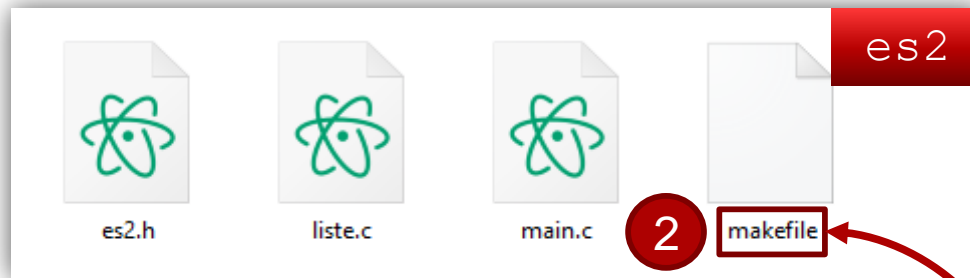
```

- Le righe 1, 3 e 5 sono le **regole di dipendenza**.
- Le righe 2, 4 e 6 sono le **regole di comando** (o **d'azione**).
 - Possono esistere più regole di comando per ogni regola di dipendenza.
 - Ogni riga della regola di comando **deve** iniziare con un carattere di tabulazione (tasto TAB della tastiera).

Utilizzo di make

- Una volta che il `makefile` è stato creato, è possibile compilare (o ricompilare) il progetto impartendo il comando `make` (in ambienti Windows con GNU GCC `mingw32-make.exe`).
- Il comando `make` cerca il file `makefile` nella directory (cartella) in cui è richiamato, crea un albero (grafo) delle dipendenze ed esegue per ogni dipendenza i comandi (o azioni) necessari definiti nel `makefile` per ottenere il file eseguibile finale.

Utilizzo di make

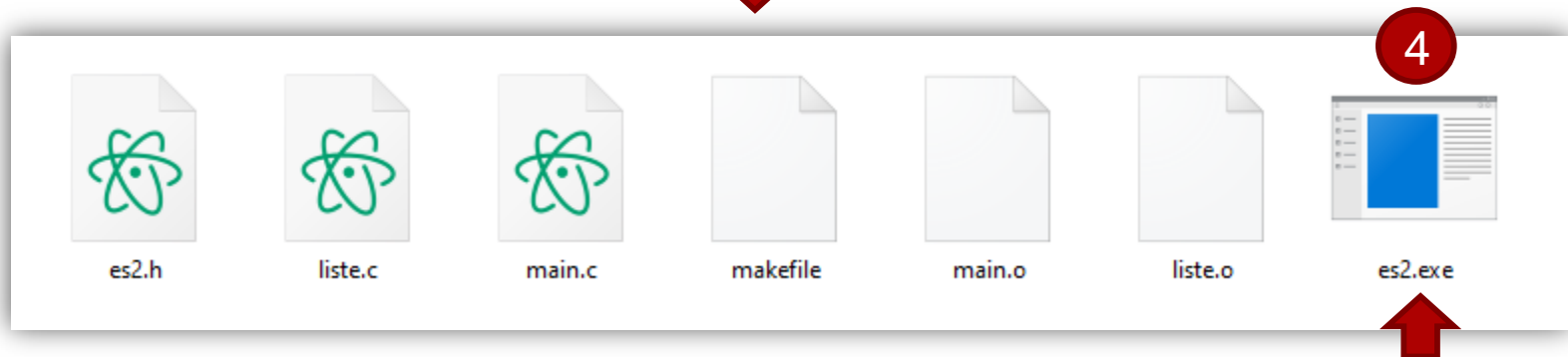


3

```
E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es2>make
gcc -c main.c
gcc -c liste.c
gcc -o es2 main.o liste.o

E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es2>
```

1



Ciclo di vita di un programma in C

- Prima di essere eseguiti, i programmi in C passano attraverso sei fasi:
 - 1) sviluppo del codice sorgente;
 - 2) preelaborazione;
 - 3) compilazione;
 - 4) linking (collegamento);
 - 5) loading;
 - 6) esecuzione.

Ciclo di vita di un programma in C

- Quando si compila un programma si è soliti scrivere:

```
gcc main.c
```

oppure (in ambiente Linux)

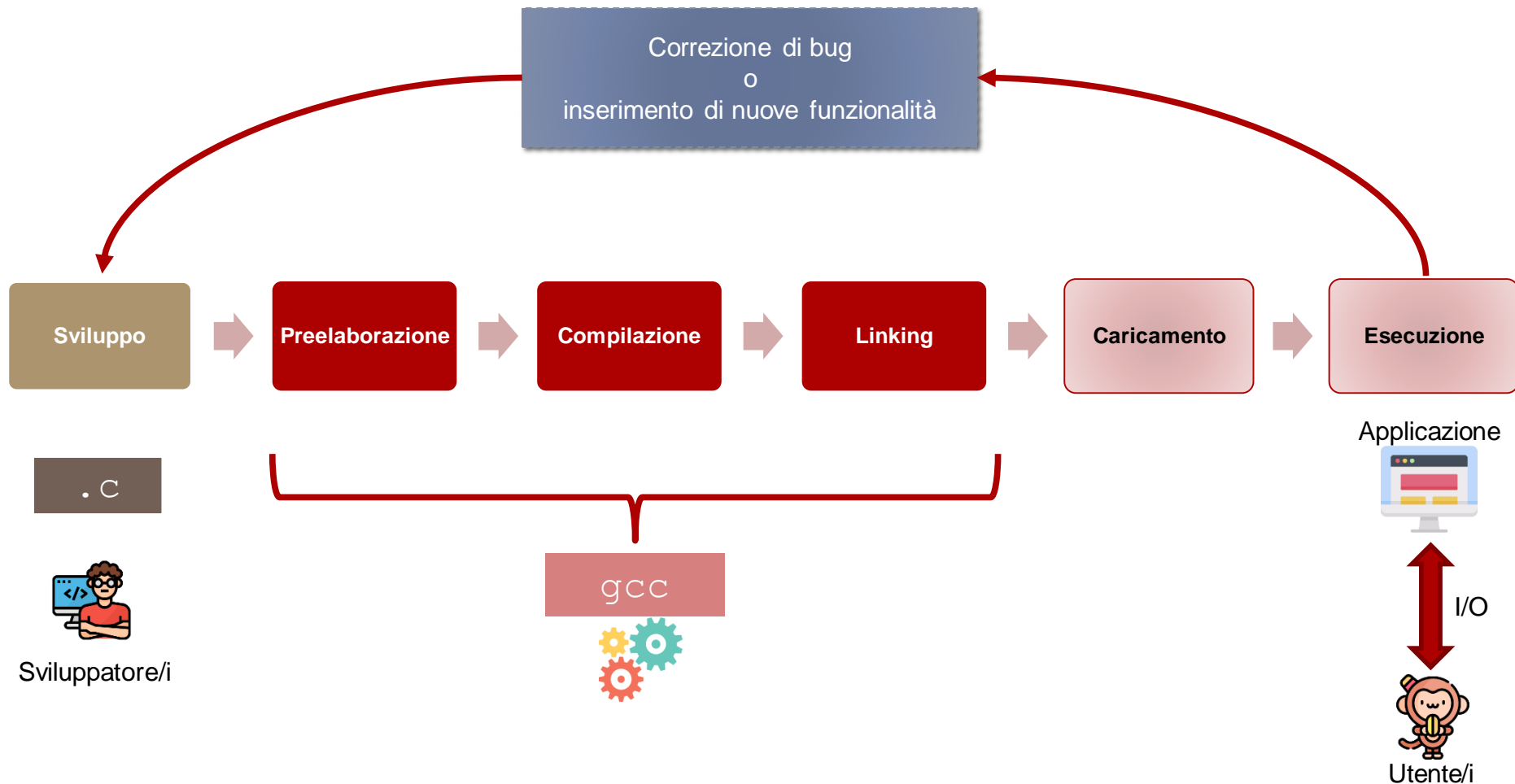
```
gcc main.c -o main.out
```

oppure (in ambiente Windows)

```
gcc main.c -o main.exe
```

- Il primo comando genererà in automatico il file `a.out` in ambiente Linux o `a.exe` in ambiente Windows.

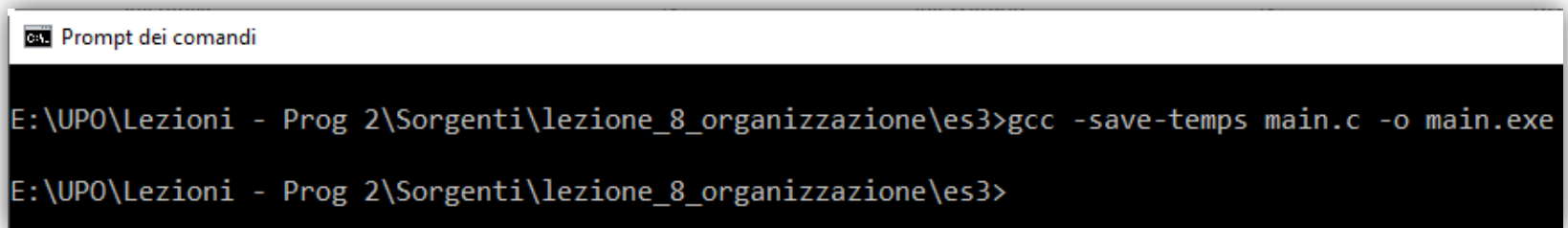
Ciclo di vita di un programma in C



Ciclo di vita di un programma in C

- Per comprendere al meglio tutte le fasi della compilazione è possibile compilare il programma con l'opzione `-save-temps`.
- Tale opzione indica al compilatore di salvare tutti i file temporanei generati in fase di compilazione sino al file eseguibile.

```
gcc -save-temps main.c -o main
```



```
Prompt dei comandi  
E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es3>gcc -save-temps main.c -o main.exe  
E:\UPO\Lezioni - Prog 2\Sorgenti\lezione_8_organizzazione\es3>
```

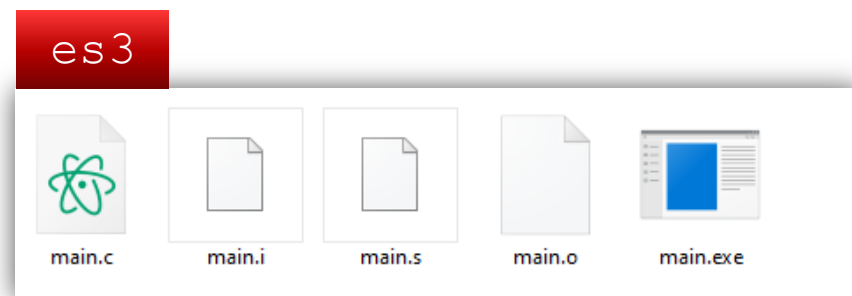
Ciclo di vita di un programma in C

- Si noteranno, oltre al file sorgente `.c` e a quello eseguibile `.out` (o `.exe` in ambiente Windows), altri tre file:

1) `.i`

2) `.s`

3) `.o`



Ciclo di vita di un programma in C

- Il file `.i` è il primo che si ottiene ed è il risultato del preprocessore.
- In questa fase vengono sostituite le macro, eliminati i commenti e importato all'interno del file da compilare tutto il codice dichiarato nei vari `#include`.
- Si noterà quindi che il numero di righe presenti nel file `.i` è di gran lunga più alto rispetto al file `.c` originale.

Ciclo di vita di un programma in C

main.c

```

1  #include <stdio.h>
2  #define A 10
3
4  int main(void)
5  {
6      int x, y;
7
8      // visualizza un messaggio a video
9      printf("Inserisci x: ");
10
11     // legge un dato da tastiera e lo salva in x
12     scanf("%d", &x);
13
14     y = x + A;
15
16     printf("Risultato: %d", y);
17
18     return 1;
19 }
```

main.i

```

539 int main(void)
540 {
541     int x, y;
542
543
544     printf("Inserisci x: ");
545
546
547     scanf("%d", &x);
548
549     y = x + 10;
550
551     printf("Risultato: %d", y);
552
553     return 1;
554 }
```

Ciclo di vita di un programma in C

- Il file `.i` è quindi compilato per ottenere il file `.s` contenente il codice **assembly**.
- Il codice assembly è un linguaggio *intermedio* contenente istruzioni che dovranno poi essere convertite in linguaggio macchina (una serie di istruzioni binarie composte di 0 e 1 comprensibili all'hardware del dispositivo sui cui viene eseguito il programma).
- *Non confondere il linguaggio assembly con l'assembler (che trasforma il linguaggio assembly in linguaggio macchina).*

Ciclo di vita di un programma in C

main.c

```

1  #include <stdio.h>
2  #define A 10
3
4  int main(void)
5  {
6      int x, y;
7
8      // visualizza un messaggio a video
9      printf("Inserisci x: ");
10
11     // legge un dato da tastiera e lo salva in x
12     scanf("%d", &x);
13
14     y = x + A;
15
16     printf("Risultato: %d", y);
17
18     return 1;
19 }

```

main.s

```

1  .section __TEXT,__text,regular,pure_instructions
2  .macosx_version_min 10, 13
3  .globl _main          ## -- Begin function main
4  .p2align 4, 0x90
5  _main:                ## @main
6  .cfi_startproc
7  ## %bb.0:
8  pushq %rbp
9  .cfi_def_cfa_offset 16
10 .cfi_offset %rbp, -16
11 movq %rsp, %rbp
12 .cfi_def_cfa_register %rbp
13 subq $32, %rsp
14 leaq L_.str(%rip), %rdi
15 movl $0, -4(%rbp)
16 movb $0, %al
17 callq _printf
18 leaq L_.str.1(%rip), %rdi
19 leaq -8(%rbp), %rsi
20 movl %eax, -16(%rbp)    ## 4-byte Spill
21 movb $0, %al
22 callq _scanf
23 leaq L_.str.2(%rip), %rdi
24 movl -8(%rbp), %ecx
25 addl $10, %ecx
26 movl %ecx, -12(%rbp)
27 movl -12(%rbp), %esi
28 movl %eax, -20(%rbp)   ## 4-byte Spill
29 movb $0, %al
30 callq _printf

```

Codice assembly

Ciclo di vita di un programma in C

- Il file `.s` è quindi **trasformato dall'assembler** in un file `.o` contenente il codice **oggetto** (o **codice macchina**).
- Essendo tale codice a livello macchina, il contenuto è incomprensibile.



Ciclo di vita di un programma in C

main.s

```

1  .section __TEXT,__text,regular,pure_instructions
2  .macosx_version_min 10, 13
3  .globl _main                ## -- Begin function main
4  .p2align 4, 0x90
5  _main:                      ## @main
6  .cfi_startproc
7  ## %bb.0:
8  pushq %rbp
9  .cfi_def_cfa_offset 16
10 .cfi_offset %rbp, -16
11 movq %rsp, %rbp
12 .cfi_def_cfa_register %rbp
13 subq $32, %rsp
14 leaq L_.str(%rip), %rdi
15 movl $0, -4(%rbp)
16 movb $0, %al
17 callq _printf
18 leaq L_.str.1(%rip), %rdi
19 leaq -8(%rbp), %rsi
20 movl %eax, -16(%rbp)        ## 4-byte Spill
21 movb $0, %al
22 callq scanf
23 leaq L_.str.2(%rip), %rdi
24 movl -8(%rbp), %ecx
25 addl $10, %ecx
26 movl %ecx, -12(%rbp)
27 movl -12(%rbp), %esi
28 movl %eax, -20(%rbp)        ## 4-byte Spill
29 movb $0, %al
30 callq _printf

```

main.o

```

1  0000 00 00 __text__TEXT_0__cstring__TEXT__compact_unwind__LD0 00 __eh_frame__TEXT0@0h$
2  8hPUH00H00 H0=P0E000H0=IH0u00E000H0=70M000
3  0M00u00E00000E000H00 ]0Inserisci x: %dRisultato: %d_zRx0$@0000000_A0CK-5.- _main_printf_scanf

```

Codice macchina

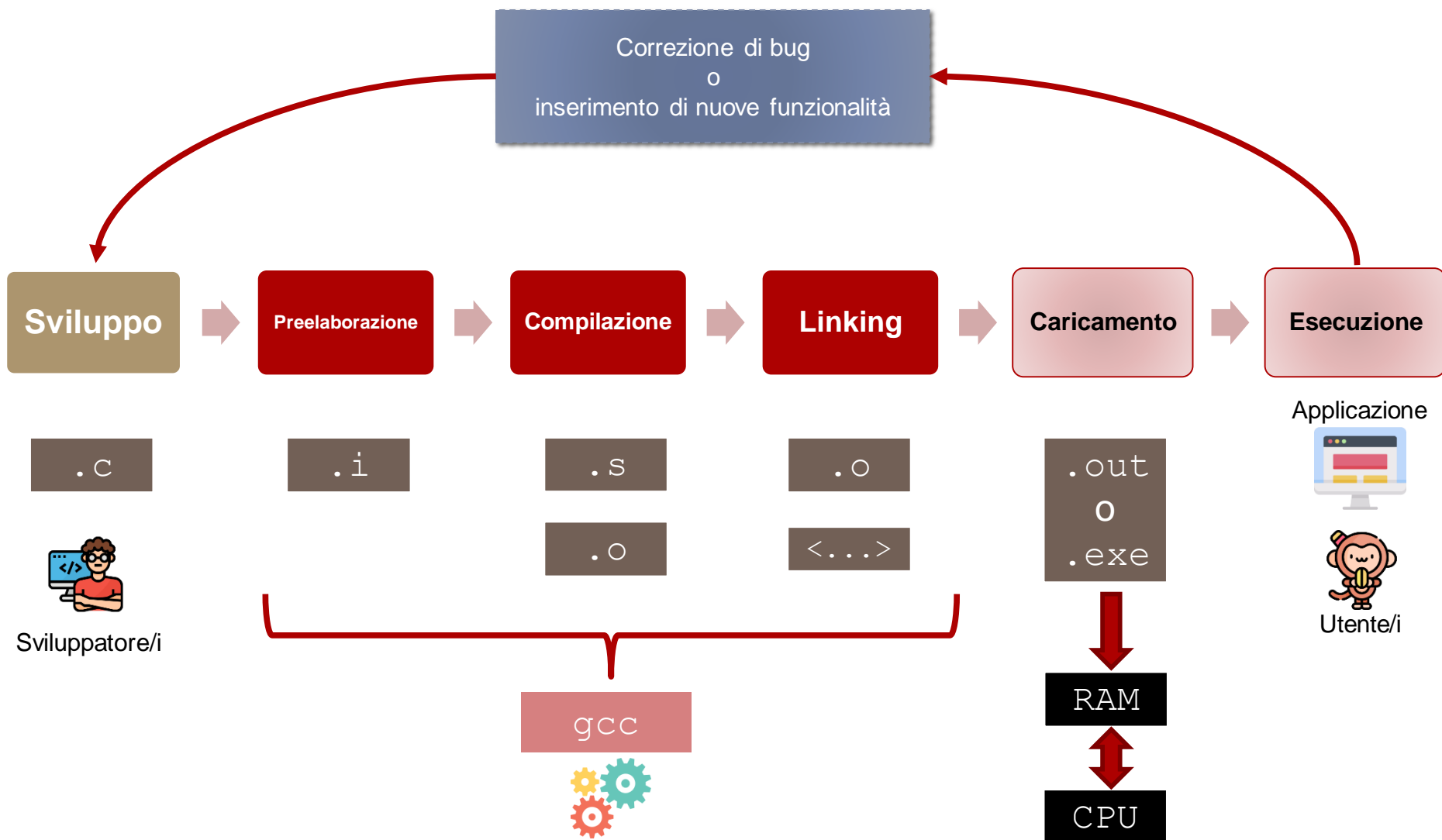
Ciclo di vita di un programma in C

- Dopo la generazione del codice oggetto (ovvero il codice macchina), la fase successiva è chiamata **linking** (collegamento).
- I programmi in C contengono spesso riferimenti a funzioni definite *altrove* (es.: la `printf()` è definita in `<stdio.h>`), quindi il codice oggetto `.o` contiene spesso "buchi" dovuti a questi riferimenti mancanti.
- Il **linker** collega il codice oggetto generato in fase di compilazione con il codice oggetto delle funzioni mancanti, così da produrre finalmente un'immagine **eseguibile** (senza parti mancanti).
- L'**eseguibile** sarà il programma finale utilizzato dagli utenti.

Ciclo di vita di un programma in C

- Il file eseguibile (.out o .exe) è quindi il programma (o applicazione) vero e proprio che sarà utilizzato da tutti gli utenti.
- Una volta eseguito, tutte le istruzioni del programma verranno caricate prima nella memoria centrale (RAM), quindi saranno eseguite dalla CPU.
- Eventuali correzioni di bug o migliorie del programma comporteranno la modifica del (o dei) file sorgente (.c) e un nuovo ciclo di compilazione per ottenere un nuovo file eseguibile.

Ciclo di vita di un programma in C



FINE PRESENTAZIONE

