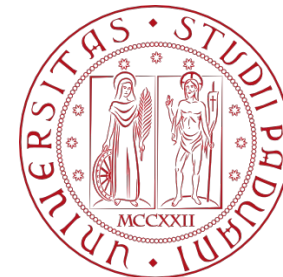


Compilazione



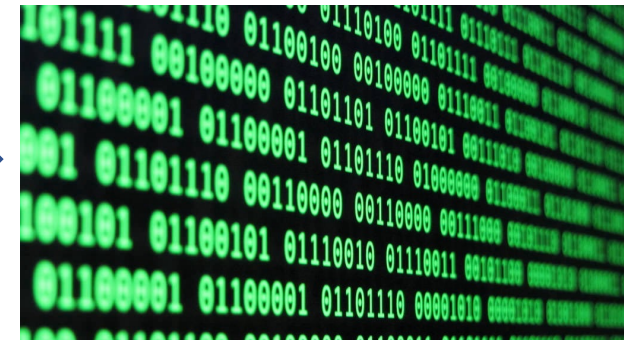
UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- In ultima istanza il computer può eseguire solamente programmi nel linguaggio macchina
- Il linguaggio della macchina è un linguaggio a basso livello (dipende dall'architettura)
 - varie architetture: MIPS, x86, ARM



- Si può avere un linguaggio di più alto livello che ci permetta di
 - evitare di implementare più volte lo stesso programma per architetture diverse e
 - fornisca comandi più vicini al nostro modo di pensare?
- Assieme alla specifica del linguaggio, si fornisce uno strumento che traduca i nostri programmi nel linguaggio della macchina ospite: il traduttore

```
1 #include <stdio.h>
2
3 int main (void) {
4     printf ("Hello, World!\n");
5
6     return 0;
7 }
```



- Interprete: traduce una istruzione di alto livello e la esegue immediatamente (Perl, Matlab)
- Compilatore: traduce tutte le istruzioni assieme che vengono poi eseguite tutte assieme direttamente in linguaggio macchina (C, C++)
- Esistono soluzioni intermedie: compilazione in bytecode ed interpretazione (Java)

- **Interprete:**
 - più lenta l'esecuzione del programma.
 - necessita del traduttore per eseguire il programma
 - se si ha il traduttore ed il codice sorgente, può essere eseguito su ogni computer
- **Compilatore:**
 - più veloce l'esecuzione (di solito riesce anche a ottimizzare il codice)
 - non necessità del traduttore, ma ogni volta che cambio il programma devo ricompilarlo
 - il codice deve essere compilato per ogni diversa architettura
- Scelte del C: linguaggio compilato;

Traduzione: Interpretazione vs Compilazione



- Scelte del C: insieme ristretto di comandi di base (ci si affida a librerie di funzioni), il compilatore è “facile” da scrivere, quindi portabilità

funzione 1: comando 7; comando 2;
comando 7;...; comando 5

funzione 2: comando 6; comando 1;
comando 8;

Libreria



comando 1; comando 2; ... ; comando N

11011111100001
10111110000111
10001110111101

11110011011101
10011011111101
11110001111110

10110011000101
11111111001100
11010000101110



```
#include <stdio.h>

int main () {

    /*
     * Il nostro primo programma stampa semplicemente
     * sullo schermo la scritta Ciao Mondo!
     */

    printf("Ciao Mondo!\n"); // nella stringa \n indica di andare a capo

}
```

Programma in C



```
1  #include <stdio.h>
2
3  int main () {
4
5      /*
6       * Il nostro primo programma stampa semplicemente
7       * sullo schermo la scritta Ciao Mondo!
8       */
9
10     printf("Ciao Mondo!\n"); // nella stringa \n indica di andare a capo
11
12 }
```

il comando printf è
disponibile nella libreria
stdio.h

inizio corpo della
funzione: {

sequenze di caratteri
(stringhe) devono essere
contenute tra “ ”

commento per
noi umani:

l'indentazione del codice
non conta per il C

Funzione. Ogni
programma C deve avere
una funzione main

Fine del corpo della
funzione {

I comandi sono
separati da ;

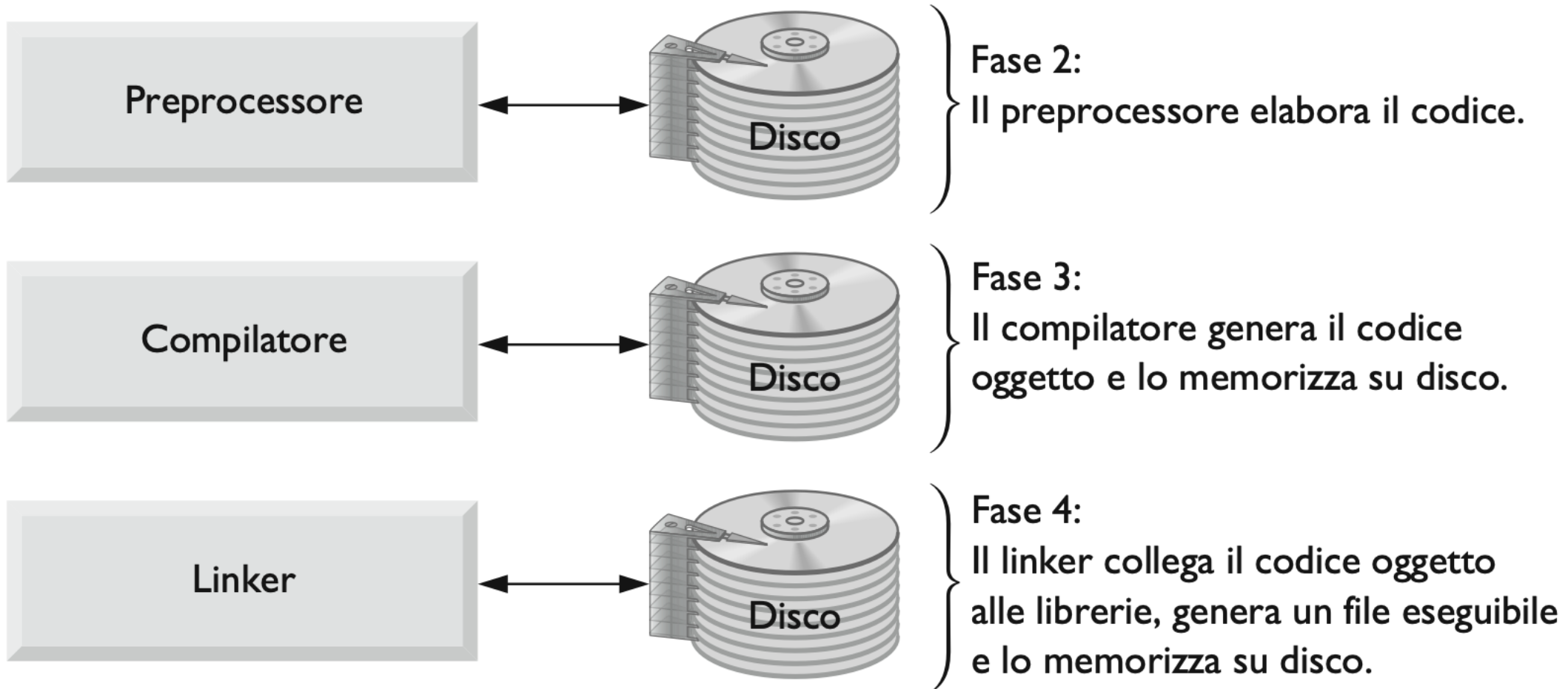
- Commenti: descrizione ad alto livello di cosa fa un frammento di codice o un intero programma

```
1  #include <stdio.h>
2
3  int main () {
4
5      /*
6       | Il nostro primo programma stampa semplicemente
7       | sullo schermo la scritta Ciao Mondo!
8       */
9
10     printf("Ciao Mondo!\n"); // nella stringa \n indica di andare a capo
11
12 }
```

- Obiettivo dei commenti: ridurre il tempo necessario per far comprendere il codice a chi lo leggerà
- “Un commento spiega del codice che non si spiega da solo”¹
- Nota: nel corso userò // per commenti che normalmente non metterei nel codice, ma che aggiungo per motivi didattici, /* */ per i commenti che metterei normalmente in un programma

1. <https://www.youtube.com/watch?v=g51HPBFCOwc>

- I commenti non devono essere banali (descrivere cose che si possono intuire senza sforzo leggendo il codice):
 - `3+2; //somma 3 e 2`
- o troppo prolissi
 - `/* Il nostro primo programma stampa semplicemente sullo schermo la scritta Ciao Mondo! */`
- I programmi e le funzioni dovrebbero indicare cosa fanno e come essere invocati (quando non sia ovvio).
 - `MCD(x,y) /* Calcola il Massimo comun divisore tra x e y */`
- Se usate un algoritmo inusuale per risolvere un problema, indicatelo
 - `/* calcolo massimo comun divisore usando l'algoritmo di euclide
(https://it.wikipedia.org/wiki/Algoritmo_di_Euclide) */`
- I commenti devono essere corretti!
 - `Somma(x,y) /* Restituisce il prodotto tra x e y */`





- Rimozione dei commenti
- Ogni linea che inizia per # indica una direttiva per il preprocessore
- `#include <x>`: il contenuto del file x viene ricopiato in questo punto del file
 - `#include <x>` permette di accedere ai comandi messi a disposizione dalla libreria x
 - Es. `stdio.h` permette di utilizzare il comando `printf`

```
#include <stdio.h>

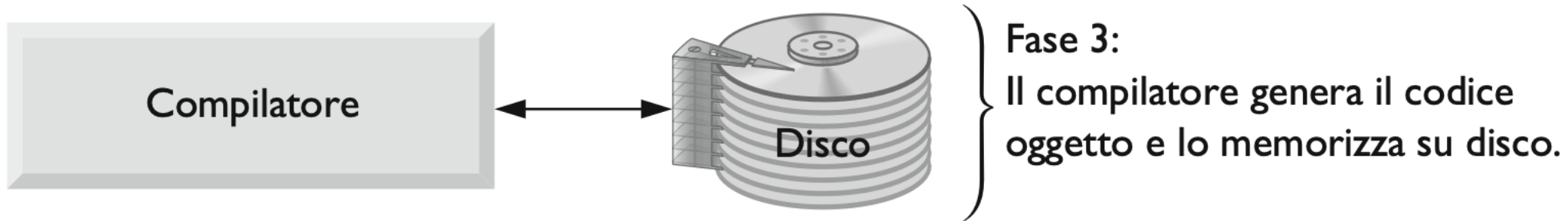
int main () {

    printf("Ciao Mondo!\n");

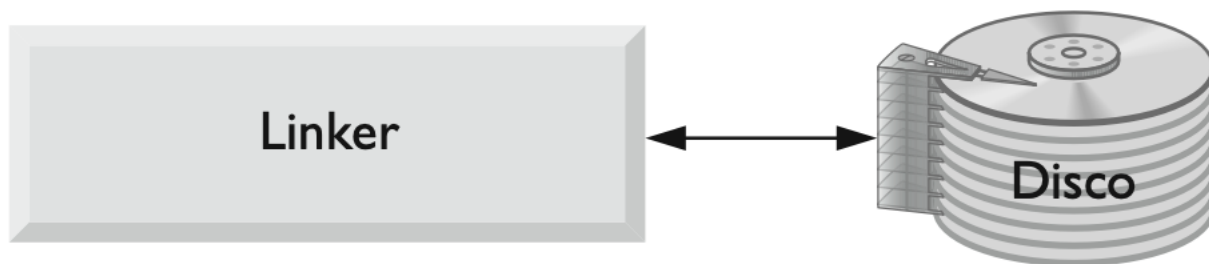
}
```



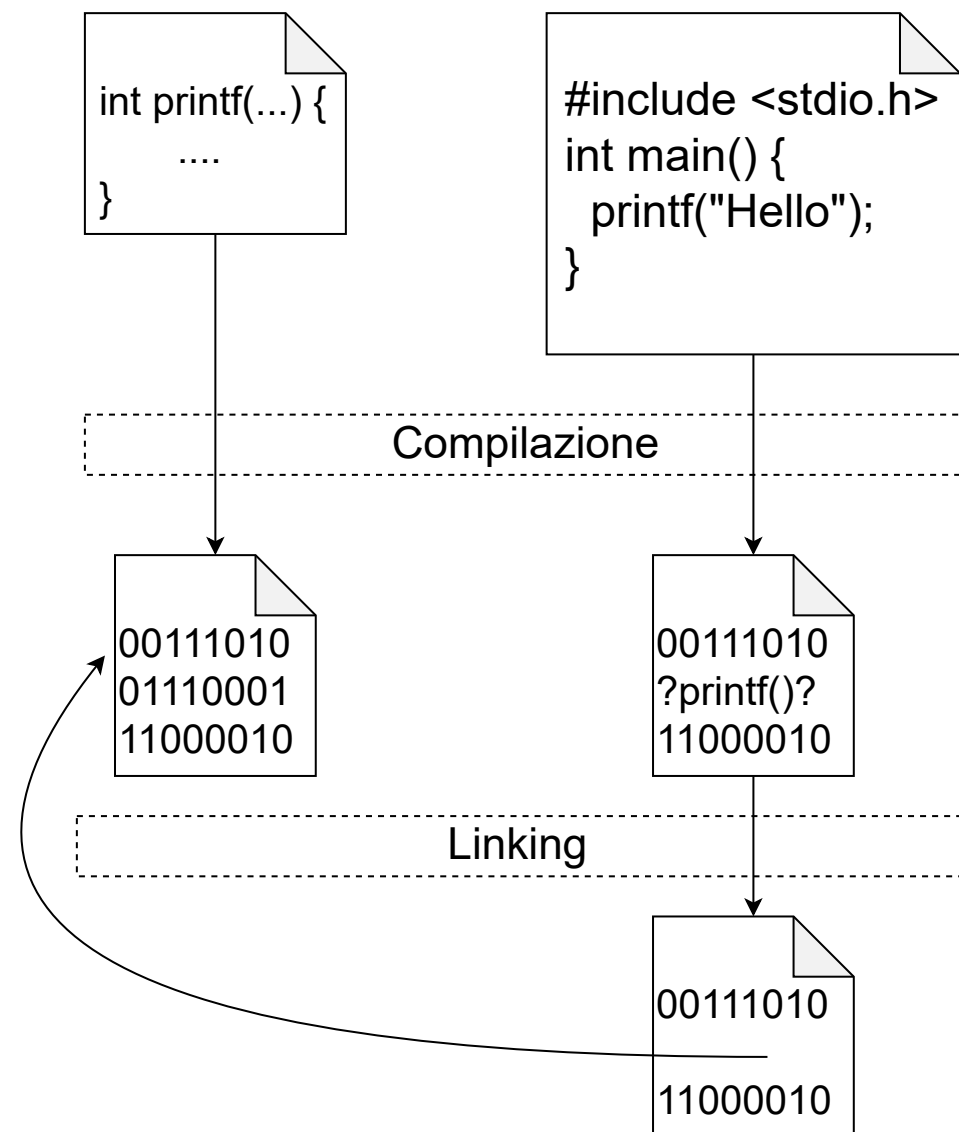
- Espansione delle macro (le vedremo a breve)
 - `#define X 3`, sostituisce ogni occorrenza di `X` nel file con `3`
- Compilazione condizionale (utile se alcune librerie hanno nomi diversi in diversi sistemi operativi)

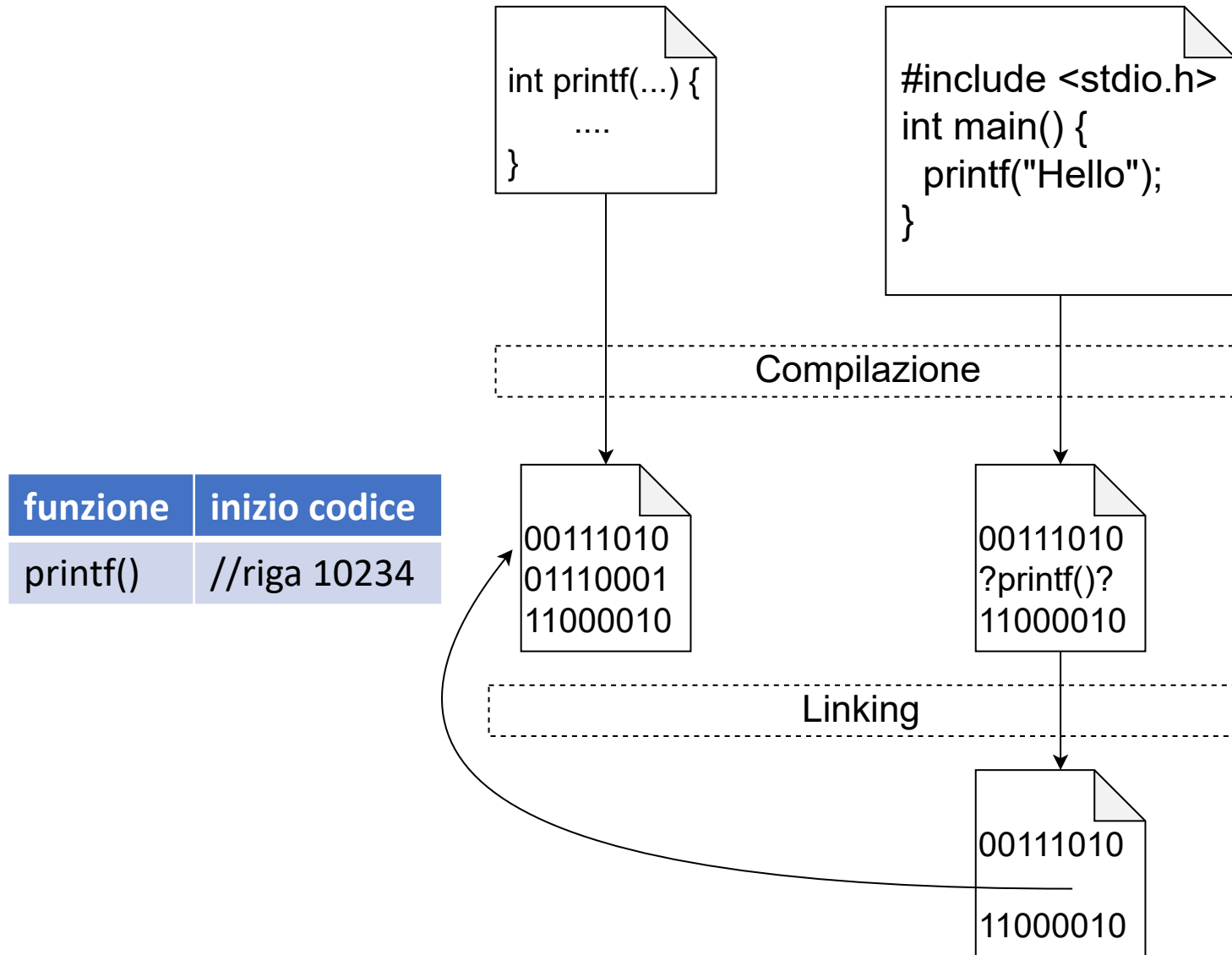


- Il compilatore analizza il file con il codice traducendolo in istruzioni del linguaggio a basso livello
- Le istruzioni devono seguire rigorosamente la sintassi definita dal linguaggio C.
- Un errore viene generato se il compilatore non riesce ad analizzare il nostro codice
- Se ci riesce, un file con le istruzioni nel linguaggio a basso livello viene generato



- Un programma è generalmente composto da molti file ed utilizza funzioni già scritte da altri (printf).
- Per evitare di duplicare il codice di tali funzioni, si caricano in memoria una volta e si collegano al nostro programma (linking)
- il linker viene invocato passandogli il file che usa printf ed il file dove printf è definita (entrambi compilati)





- Nella fase di compilazione si produce, oltre al codice macchina, una tabella con le funzioni che sono implementate nel file
- Il linker usa tali tabelle per risolvere i riferimenti alle funzioni

- Ma in pratica come si compila un programma? Dal terminale digitare:

```
gcc -o ciao hello_world.c
```

- Il comando esegue tutte le fasi della compilazione
- -o indica il nome del file eseguibile
 - se si omette “-o ciao” viene creato il file a.out
- Il codice tradotto è a questo punto eseguibile: [su linux] `./ciao`

```
#include <stdio.h>

int main () {

    printf("Ciao Mondo!\n");

}
```

file: hello_world.c

- Esiste un unico compilatore? NO
- Il C è nato negli anni 70, ha avuto molto successo, per cui molti compilatori sono stati creati indipendentemente
- ANSI C: una serie di specifiche che standardizzano il comportamento del compilatore
 - C89, C90 – ISO C 1990
 - C17 – ISO C 2017
 - C2x – in lavorazione
- In alcuni casi particolari non è specificato il comportamento atteso, e quindi ogni compilatore può fare quello che vuole!
 - se abbiamo un dubbio sul comportamento di un elemento del linguaggio, non dobbiamo solamente provarlo sulla nostra macchina, ma controllare lo standard!
- Noi useremo quello di default usato dal compilatore del laboratorio

- Potete vedere le opzioni disponibili per il vostro compilatore digitando il comando
`man gcc` su linux, `man clang` su MacOS
- `-std=c89` indica quale standard ISO seguire
 - Qual è il valore di default sul vostro sistema?
- `-c` esegue tutte le fasi della compilazione fino al linking escluso (genera file con estensione .o)

Compilazione: Esempio

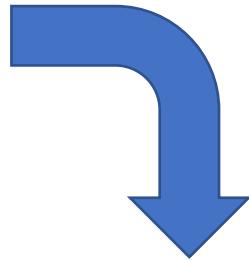


file: hello_world_stripped.c

```
#include <stdio.h>

int main () {

    printf("Ciao Mondo!\n")
}
```



- gcc hello_world_stripped.c
- genera un errore alla riga 5 colonna 26
- Non necessariamente l'errore è esattamente dove indicato, la posizione indica dove il compilatore si è "arreso"

```
dasan$ gcc hello_world_stripped.c
hello_world_stripped.c:5:26: error: expected ';' after expression
    printf("Ciao Mondo!\n")
                        ^
                        ;
1 error generated.
```

file: hello_world_stripped_w2.c

```
1  #include <stdio.h>
2
3  int main () {
4
5      printf("Ciao Mondo!\n");
6
7  }
```

- gcc hello_world_stripped_w2.c

?

Compilazione: Esempio (codice KCJLUD)

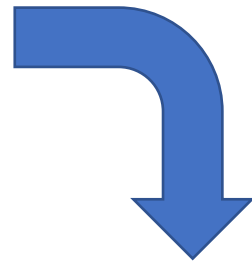


Compilazione: Esempio



file: hello_world_stripped_w2.c

```
1  #include <stdio.h>
2
3  int main () {
4
5      printf("Ciao Mondo!\n");
6
7  }
```



- **Warning:** non un errore (il codice macchina viene generato) ma qualcosa di insolito o "rischioso"
- Tanti errori in cascata, si inizia dal primo (che di solito genera anche gli altri)
- Ci siamo dimenticati i doppi apici alla fine di -Ciao Mondo!\n- !

```
$ gcc hello_world_stripped_w2.c
```

```
hello_world_stripped_w2.c:5:10: warning: missing terminating '"' character [-Winvalid-pp-token]
    printf("Ciao Mondo!\n");
           ^
```

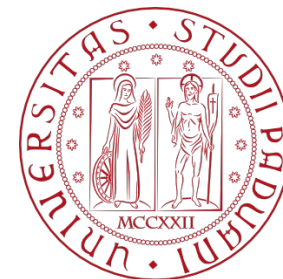
```
hello_world_stripped_w2.c:5:10: error: expected expression
```

```
hello_world_stripped_w2.c:8:1: error: expected '}'
    ^
```

```
hello_world_stripped_w2.c:3:13: note: to match this '{'
    int main () {
               ^
```

1 warning and 2 errors generated.

Progetti su più File



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

- Quando i nostri programmi diventano di grandi dimensioni, o per riutilizzare agevolmente funzioni già realizzate, è possibile implementare queste ultime in un file separato, che poi andremo a “collegare” al nostro file principale.
- Possiamo raggruppare funzioni simili in un file a parte, per esempio avere un file per funzioni matematiche o un file per funzioni su array
 - mantiene il codice ordinato
 - velocizza la ricerca delle funzioni
 - membri di una squadra possono lavorare su file diversi nello stesso momento

- Il compilatore deve verificare la correttezza sintattica del nostro programma
- Nella fase di compilazione per main.c, ho bisogno di sapere il tipo della funzione f() per poter controllare di utilizzarla correttamente, ovvero che:
 - i parametri sono del tipo giusto
 - utilizzo il tipo del valore restituito correttamente
- In pratica devo conoscere il prototipo della funzione

```
int f(int n);
```

```
int main() {  
    printf( "%d", 2 + f(3) );  
}
```

file: main.c



Risoluzione dei simboli



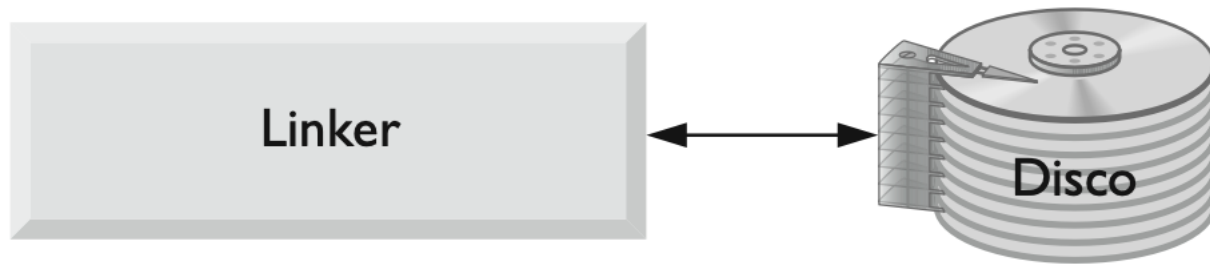
- Adesso il file main.c passa la fase di compilazione anche se f non è definita!
- Al posto dell'indirizzo di inizio della funzione f() si mette un segnaposto e si costruiscono due tabelle per ogni file
 - una che elenca le funzioni implementate nel file corrente
 - una che indica le funzioni per cui manca l'implementazione
- L'implementazione di f() viene trovata nella fase di linking

```
int f(int n);  
int doppio(int x) {return x*2;}  
int main() {  
    f(3);  
}
```

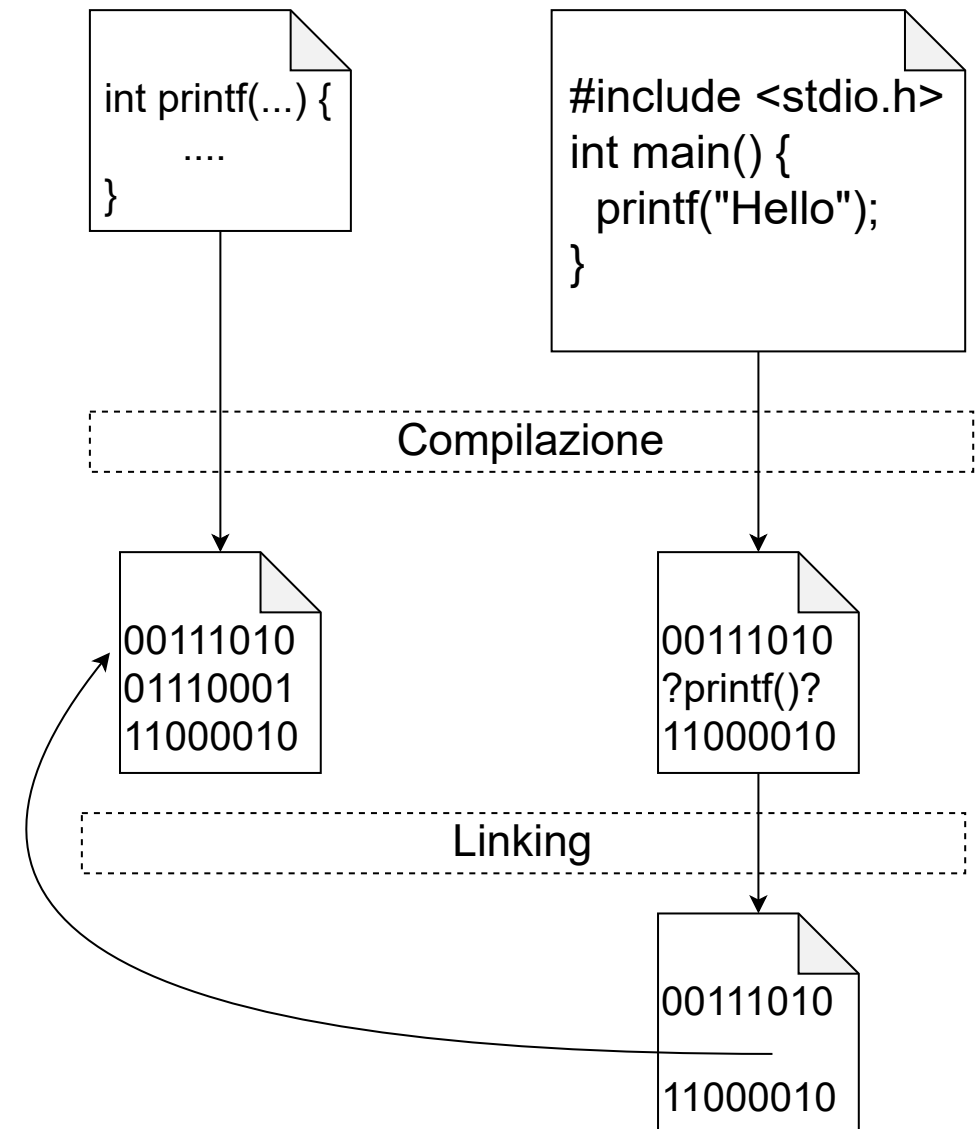
file: main.c

Simboli Implementati	Dove? (riga)
doppio()	2

Simboli da Risolvere	Dove? (riga)
f()	4



- il linker viene invocato passandogli il file che usa printf ed il file dove printf è definita (entrambi compilati)
- Il linker usa le tabelle di ogni file per risolvere i simboli (trovare dove inizia l'implementazione di f())




- Se a tempo di linking non forniamo una definizione della funzione `f()`, si ottiene un errore di linking e non si ottiene un file eseguibile
- `gcc -c f.c //compilazione per creare f.o`
- `gcc main.c f.o //linking`
- funziona anche `gcc main.c f.c // esegue automaticamente da sè`
`gcc -c f.c`

```
int f(int n);  
int main() {  
    f(3);  
}
```

file: main.c

```
int f(int n) {  
    return n+1;  
}
```

file: f.c

- Per automatizzare la gestione delle dichiarazioni, si introduce il concetto di header file (file di intestazione)
 - contenente tutte le dichiarazioni relative alle funzioni implementate nel file separato
- scopo: evitare ai clienti di dover trascrivere riga per riga le dichiarazioni necessarie
- basterà includere l'header file tramite una direttiva #include
- Es. `#include <stdio.h>` 
- Per le librerie fornite dal sistema, il C sa già dove andare a trovarle, per cui non serve che forniamo esplicitamente il file .o alla fase di linking

- In pratica si creano due file:
 - .c con l'implementazione delle funzioni
 - .h con le intestazioni (i prototipi) delle funzioni
 - Quindi c'è un file .h per ogni file .c dell'applicazione (escluso, eventualmente, il file che contiene il main)
- Abbiamo già visto alcuni esempi: stdio.h, assert.h
- Nel programma principale, per poter utilizzare le funzioni aggiuntive, basta utilizzare la direttiva #include
 - #include <stdio.h> (<> fanno sì che si cerca tra i file forniti dal sistema operativo)
 - #include "stringhe.h" ("") cerca stringhe.h prima nella cartella corrente)

- Se il file header.h contiene
 - `char *test ();`
- E nel nostro file principale

```
#include "header.h"
int main () {
    printf("%s\n", test());
}
```
- Allora il preprocessore trasformerà il file principale in

```
char *test (void);
int main (void) {
    printf("%s\n", test());
}
```

- Allora il preprocessore trasformerà il file principale in

```
char *test (void);  
int main (void) {  
    printf("%s\n", test());  
}
```
- Notate che non abbiamo ancora a disposizione l'implementazione di test(). Questo è accettabile nella fase di compilazione, ma l'implementazione della funzione test deve essere raggiungibile nella fase finale di linking.
- Questo permette la compilazione separata (ma non l'esecuzione!) dei vari file che costituiscono un progetto

- Per evitare di includere un file header più volte, che in alcuni casi può corrispondere ad un errore di compilazione (se si ha una variabile, viene dichiarata due volte)
- Il C mette a disposizione delle direttive del preprocessore per evitarlo

```
#ifndef HEADER_FILE
```

```
#define HEADER_FILE
```

```
//contenuto del file header.h: prototipi di funzioni, variabili e #define
```

```
#endif
```

- le istruzioni tra ifndef ed endif vengono copiate solo se la variabile HEADER_FILE non è definita
- se si vuole utilizzare una variabile definita in un altro file, la dichiarazione deve essere preceduta da extern. In

Evitare Inclusione Multipla



- se si vuole utilizzare una variabile definita in un altro file, la dichiarazione deve essere preceduta da `extern`.

```
#include "array.h"  
int X = 5;
```

file: array.c

```
#include "array.c"  
extern int X = 5;
```

file: array.h

```
#include "array.h"  
extern int X;  
int main() {  
    printf("%d", X);  
}
```

file: main.c

- Se si vuole creare un file con una serie di funzioni, per esempio che operano su stringhe, si deve
- creare il file .h con i prototipi delle funzioni e le #define (proteggere il file dall'inclusione multipla)
- creare un file .c (con lo stesso nome) che includa il file .h (così il compilatore controlla che le dichiarazioni di funzione sono coerenti)
- Nella fase di compilazione è sufficiente elencare i file .c che vogliamo compilare (non è necessario includere i .h)

`gcc -o programma main.c stringhe.c`

- Se i file da compilare sono molti, conviene utilizzare l'utility make per sveltire il processo di compilazione (richiede un file di configurazione, Makefile, nel quale in pratica si specifica il comando gcc)

- Compilazione di un progetto con più file:
 - `gcc -o hello file1.c file2.c file3.c file4.c file5.c file6.c`
 - scomodo riscrivere ogni volta il comando sopra
- Esistono programmi che ci permettono di salvare il comando di compilazione
 - `make`
- `make` si appoggia ad un file di testo di nome `Makefile` che contiene le istruzioni di compilazione
- `make` permette anche di
 - compilare versioni diverse del programma
 - cancellare i file `.o` ed eseguibili
 - eseguire test
 - installare software

hello.c

```
#include <stdio.h>
int main(void) {
    printf("hello world\n");
}
```

Makefile: Esempio



Makefile

```
hello: hello.c  
    gcc -o hello hello.c
```

hello.c

```
#include <stdio.h>  
  
int main(void) {  
    printf("hello world\n");  
}
```

Makefile: Esempio



Makefile

```
hello: hello.c  
    gcc -o hello hello.c
```

hello.c

```
#include <stdio.h>  
  
int main(void) {  
    printf("hello world\n");  
}
```

Terminale

```
$ make hello  
gcc -o hello hello.c  
$
```

Makefile

```
all: hello
```

```
hello: hello.c
```

```
    gcc -o hello hello.c
```

```
test: test.c
```

```
    gcc -o test test.c
```

Terminale

```
$ ls
```

```
hello.c test.c
```

```
$ make
```

```
gcc -o hello hello.c
```

Makefile: Esempio



Makefile

```
CC=gcc
all: hello

hello: hello.c
    $(CC) -o hello hello.c
test: test.c
    $(CC) -o test test.c
clean:
    rm hello test
```

Terminale

```
$ ls
hello.c test.c
$ make
gcc -o hello hello.c
```

Makefile: Esempio



Makefile

```
CC=gcc
CFLAGS=-g
all: hello

hello: hello.c
    $(CC) -o hello hello.c
test: test.c
    $(CC) $(CFLAGS) -o test test.c
clean:
    rm hello test
```

Terminale

```
$ ls
hello.c test.c
$ make
gcc -o hello hello.c
```