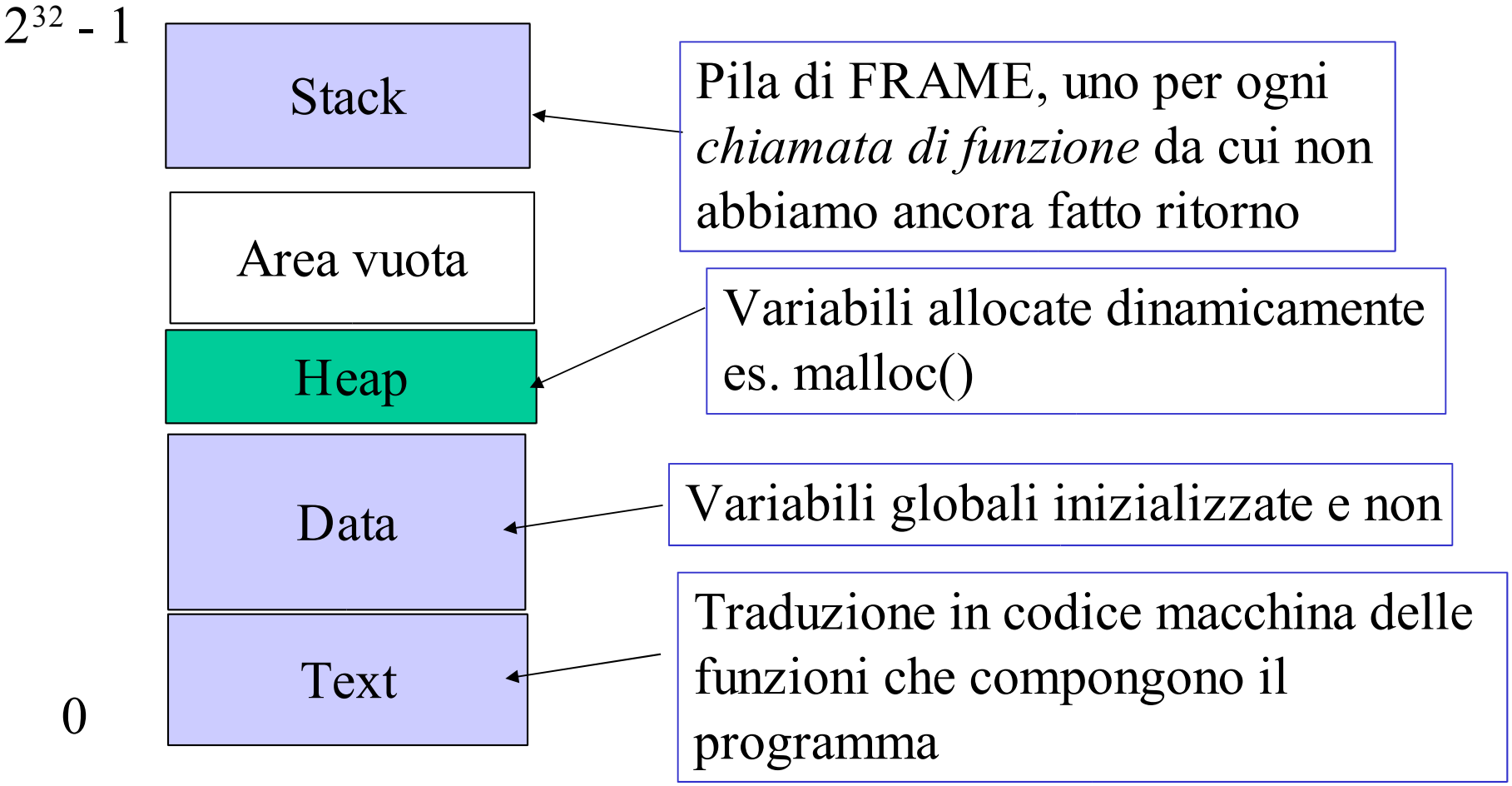
Preprocessing, compilazione ed esecuzione

Utilizzando strumenti GNU...

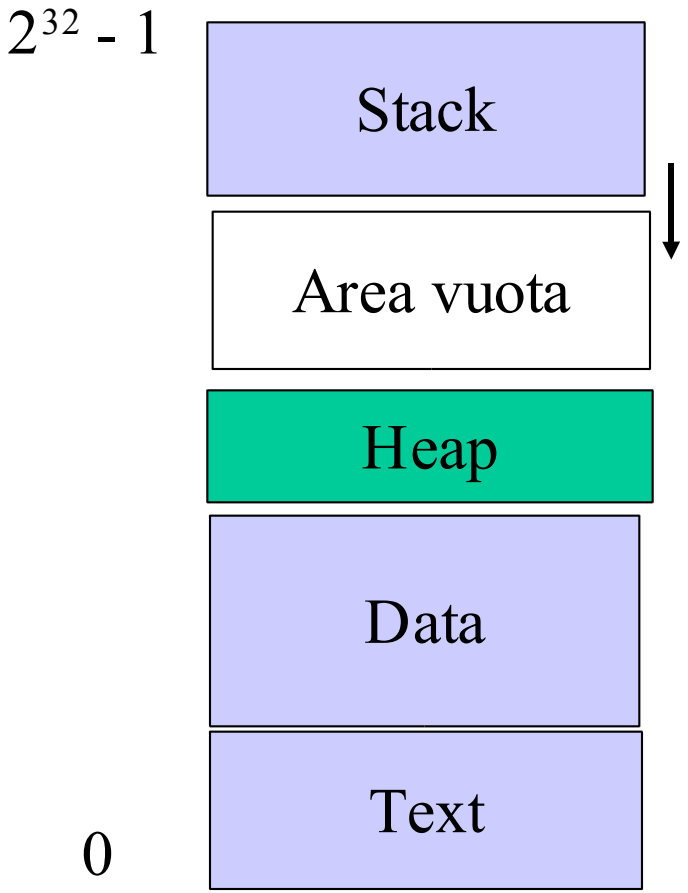
Spazio di indirizzamento

- Come vede la memoria un programma C in esecuzione



Spazio di indirizzamento (2)

- Lo stack

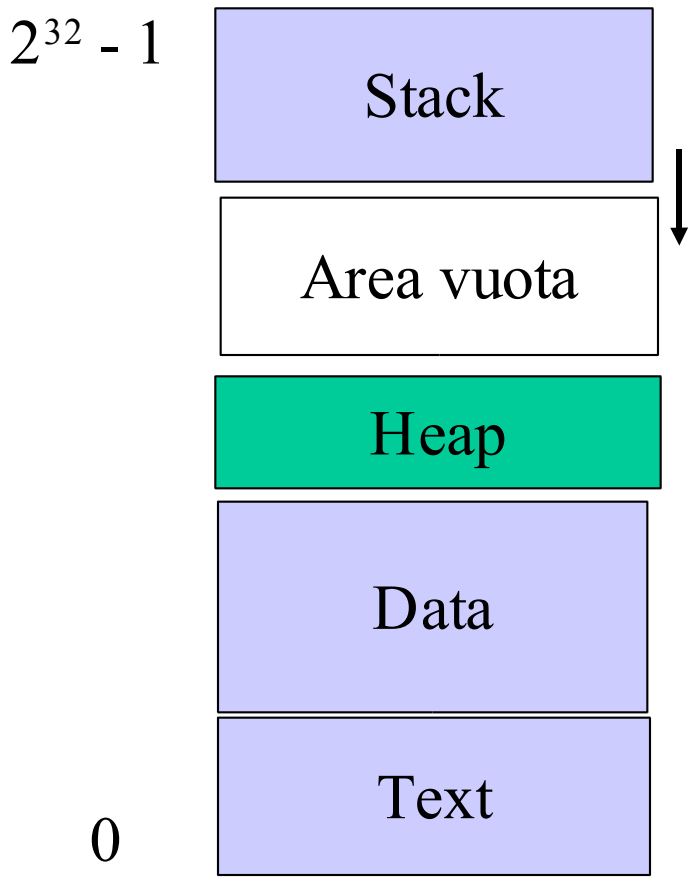


Direzione di
crescita dello stack

Contenuti tipici di un FRAME :
- variabili locali della funzione
- indirizzo di ritorno (indirizzo dell'istruzione successiva a quella che ha effettuato la chiamata alla funzione)
- copia del valore parametri attuali

Spazio di indirizzamento (3)

- Lo stack

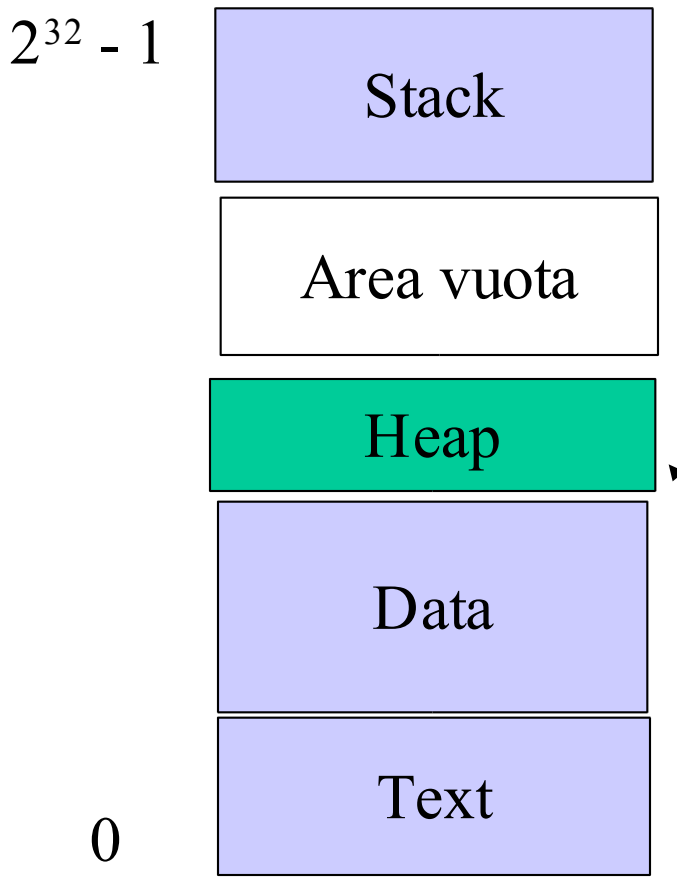


All'inizio dell'esecuzione lo Stack contiene solo il FRAME per la funzione `main`

Successivamente :

* ogni volta che viene chiamata una nuova funzione viene inserito un nuovo frame nello stack
* ogni volta che una funzione termina (`es. return 0`) viene eliminato il frame in cima dello stack e l'esecuzione viene continuata a partire dall'*indirizzo di ritorno*

Spazio di indirizzamento (4)



Direzione di
crescita dello *heap*

malloc() calloc() realloc() fanno crescere lo heap verso l'alto
la memoria viene deallocata solamente con chiamate alla funzione free() . Se la free() non viene invocata la memoria rimane occupata (*memory leak*)

Formato del file eseguibile

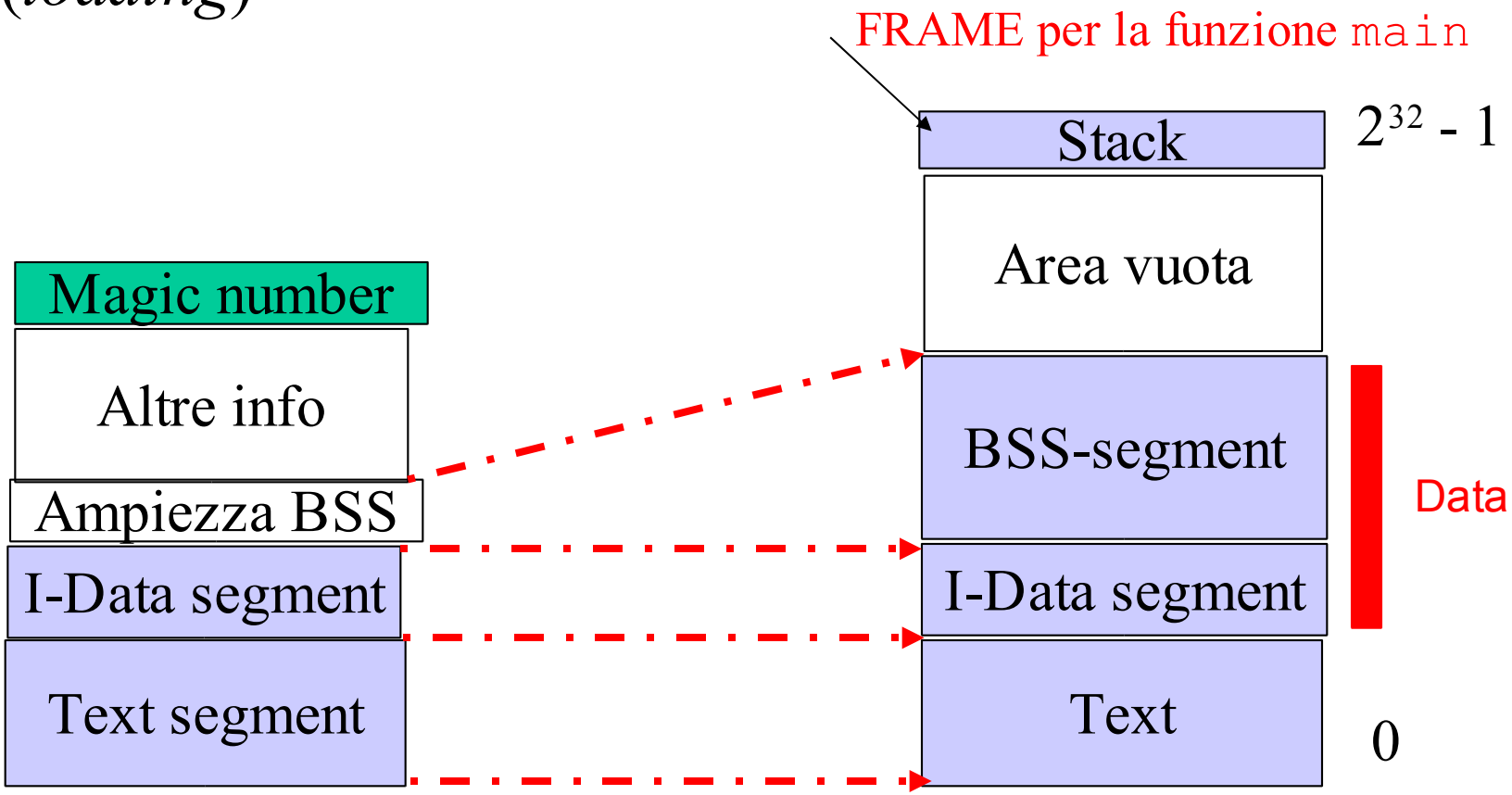
- La compilazione produce un file eseguibile
- Il formato di un eseguibile dipende dal sistema operativo
- In Linux un eseguibile ha il formato ELF (*Executable and Linking Format*)
  - eseguibili e moduli oggetto hanno lo stesso formato
  - tabelle eliminabili con il comando **`strip`**

Formato del file eseguibile (2)

- Formato di un eseguibile ELF
  - leggibile con **`readelf, objdump, nm`**

Formato del file eseguibile (3)

– L'eseguibile contiene tutte le informazioni per creare la configurazione iniziale dello spazio di indirizzamento (*loading*)

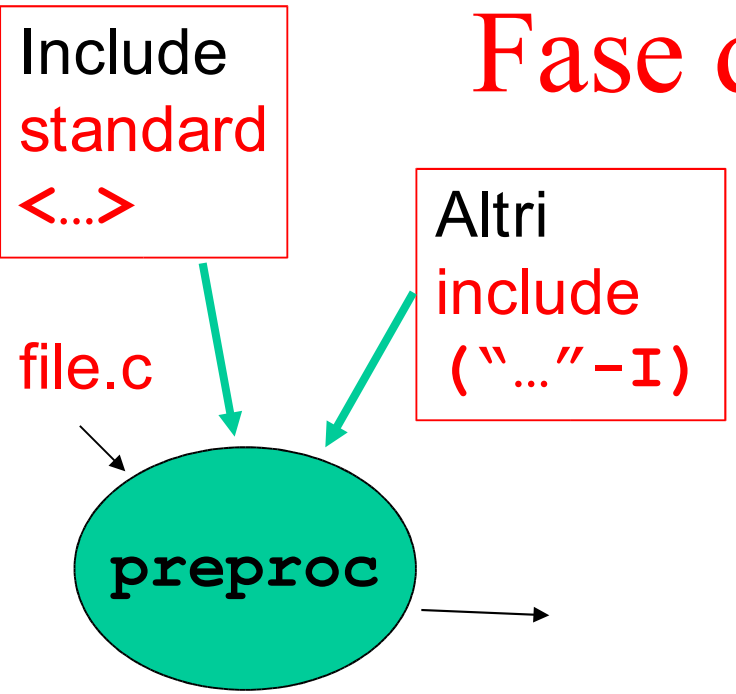


C: dal sorgente all'eseguibile

- Prima di essere eseguito dal processore il programma deve essere
  - 1. pre-processato
  - 2. compilato
  - 3. collegato (*linking*)

- Vediamo come funzionano le varie fasi

Fase di preprocessing




```
cpp prova.c
gcc -E prova.c
```

Preprocessing

- Espansione degli **`#include`**
- Sostituzione della macro (**`#define`**)
- Compilazione condizionale (**`#if #ifdef #endif`**)

Preprocessing: esempio

File **`prova.c`**

direttive per **`cpp`**

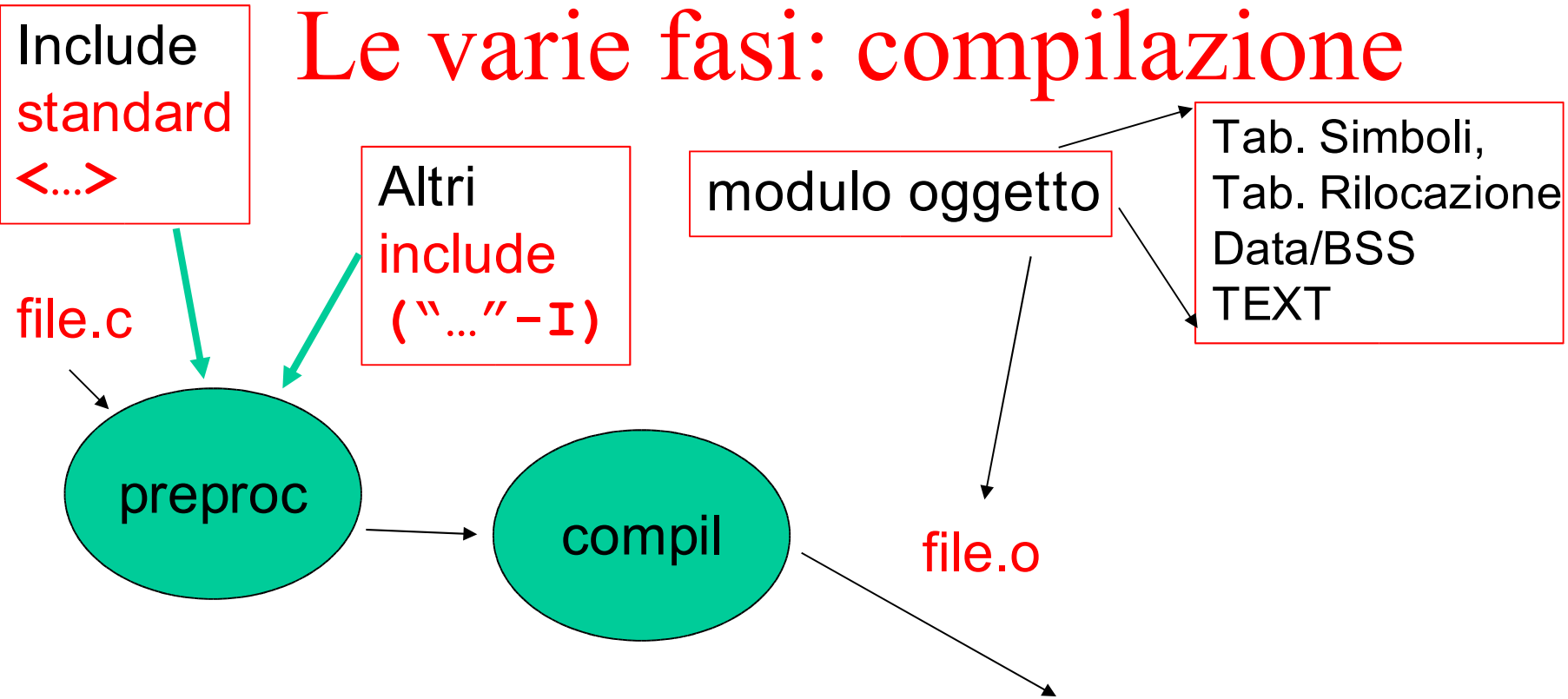
```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Preprocessing: esempio (2)

Dopo
```
cpp prova.c
gcc -E prova.c
```

```
…… -- copia di stdio.h
# 2 "prova.c" 2
-- qua era la #define
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",10);
  for (i = 0; i < 10; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Le varie fasi: compilazione




```
gcc -c file.c
   (modulo oggetto)
gcc -S file.c
   (assembler simbolico text e data
     in file.s)
```

Compilazione : un esempio

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Globali a 0, DATA BSS

```
$ nm prova.o
0000000 B max
…
$ objdump -D prova.o
```

Compilazione : un esempio (2)

Var locali, STACK

tradotte in istruzioni che lavorano sullo stack (TEXT)

```
$ objdump -d prova.o
```

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Compilazione : un esempio (3)

codice, TEXT

assembler + 2 call a **printf()** e 1 **scanf()**

$ objdump -d prova.o

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Compilazione : un esempio (4)

Simboli esportati (g),
SYMBOL TABLE:
max, main

$ objdump -t prova.o
$ nm prova.o

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  max=0;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Compilazione : un esempio (5)

Simboli non-definiti (*UND*)
SYMBOL TABLE
printf, scanf

$ objdump -t prova.o

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

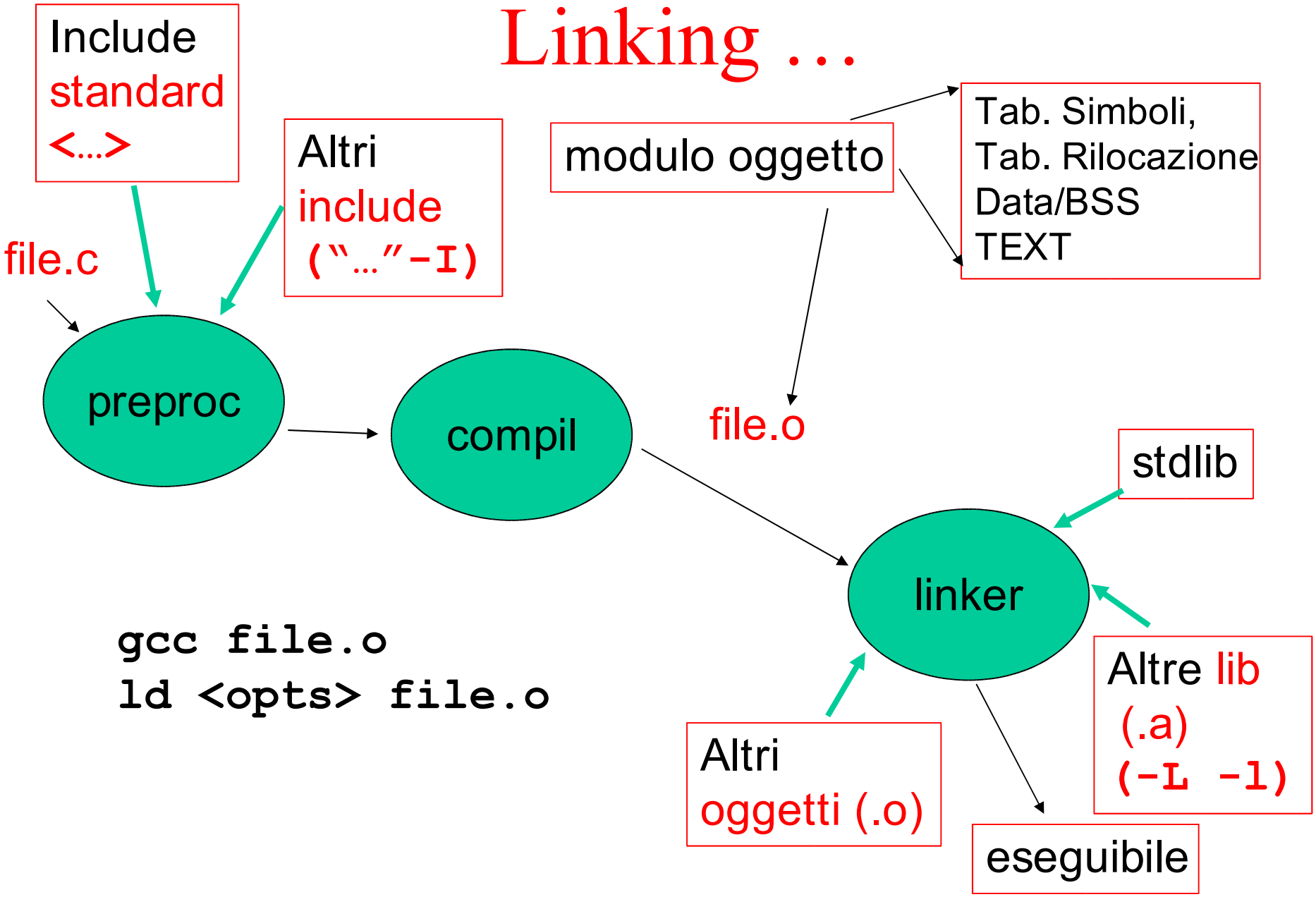
Compilazione : un esempio (6)

Indirizzi da rilocare
RELOCATION RECORDS
max, printf, scanf

$ objdump -r prova.o

```
#include <stdio.h>
#define N 10
int max = 0;
int main (void){
  int i, tmp;
  printf("Inserisci %d interi positivi\n",N);
  for (i = 0; i < N; i++) {
     scanf("%d", &tmp);
     max = (max < tmp)? max : tmp ;
  }
  printf("Il massimo è %d \n",max);
  return 0;
}
```

Linking …



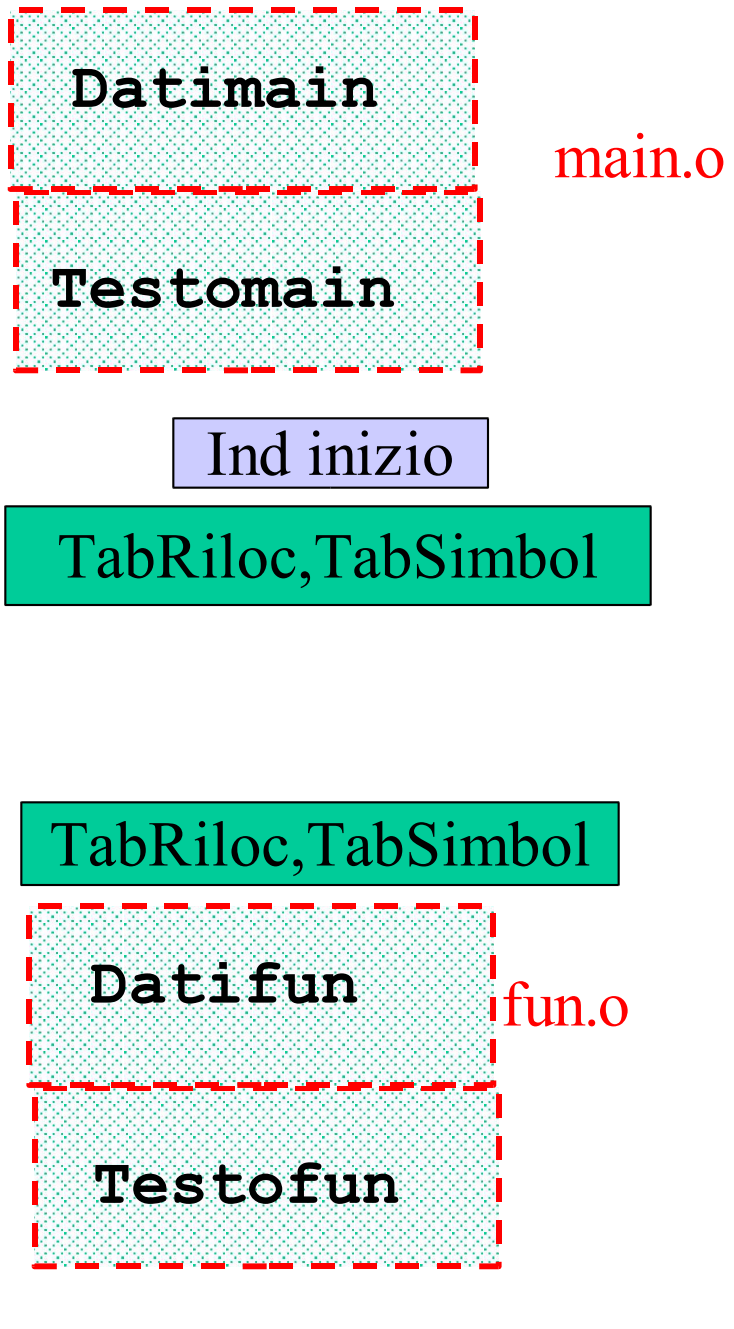

```
gcc file.o
ld <opts> file.o
```

Linking: esempio statico

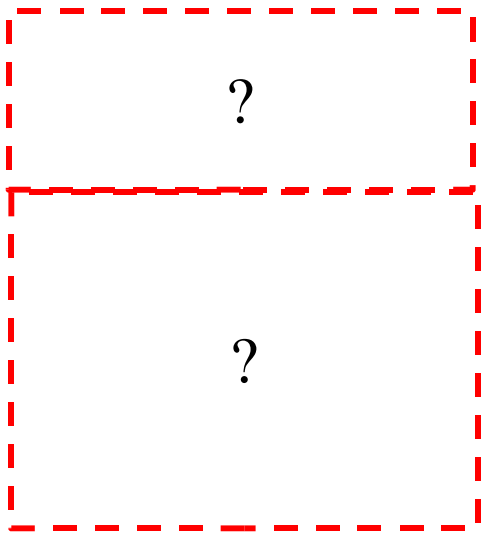
- Si collegano assieme più moduli oggetto
  - **`file.o`** oppure librerie di oggetti **`libfile.a`**
  - ...per creare un file eseguibile
- Ogni modulo oggetto contiene
  - L'assemblato del sorgente **testo e dati** (si assume di partire dall'indirizzo 0)
  - La **tabella di rilocazione**
  - La **tabella dei simboli** (esportati ed undefined)

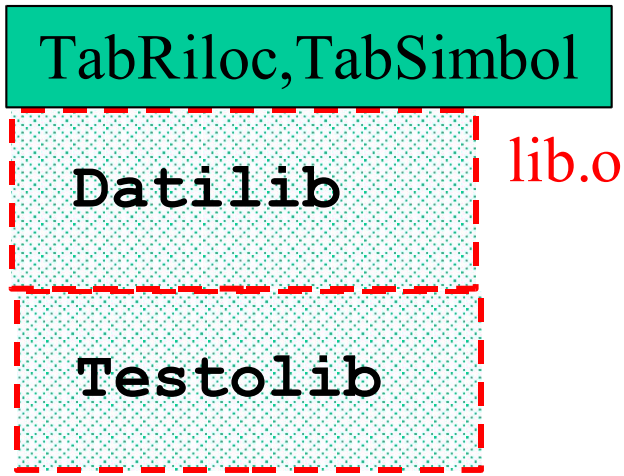
Linking: esempio statico (2)

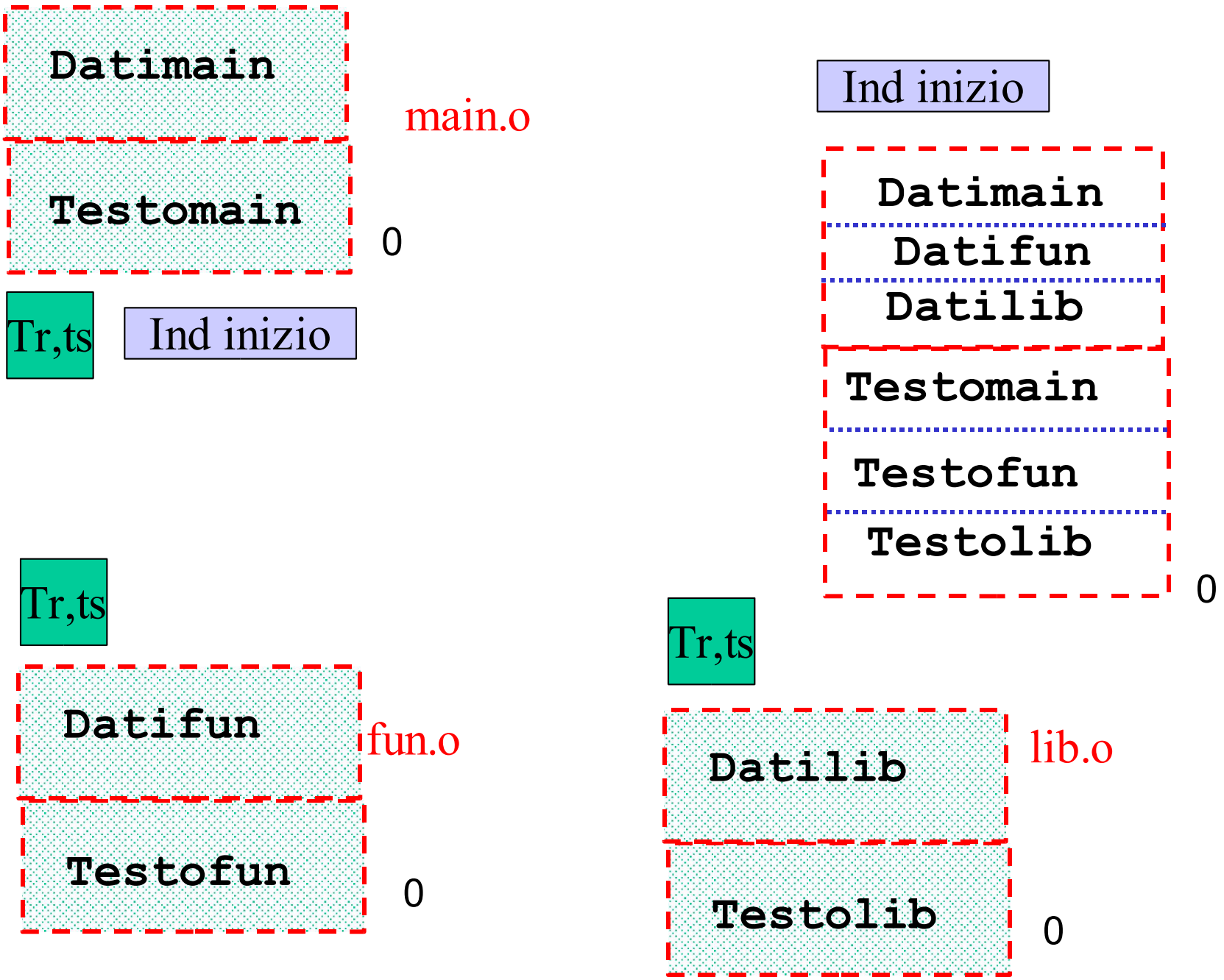
- Tabella di rilocazione
  - identifica le parti del testo che riferiscono indirizzi assoluti di memoria
    - es. JMP assoluti, riferimenti assoluti all'area dati globali (LOAD, STORE…)
  - questi indirizzi devono essere rilocati nell'eseguibile finale a seconda della posizione della prima istruzione del testo (**`offset`**)
  - all'indirizzo contenuto nell'istruzione ad ogni indirizzo rilocabile va aggiunto **`offset`**



Situazione iniziale eseguibile

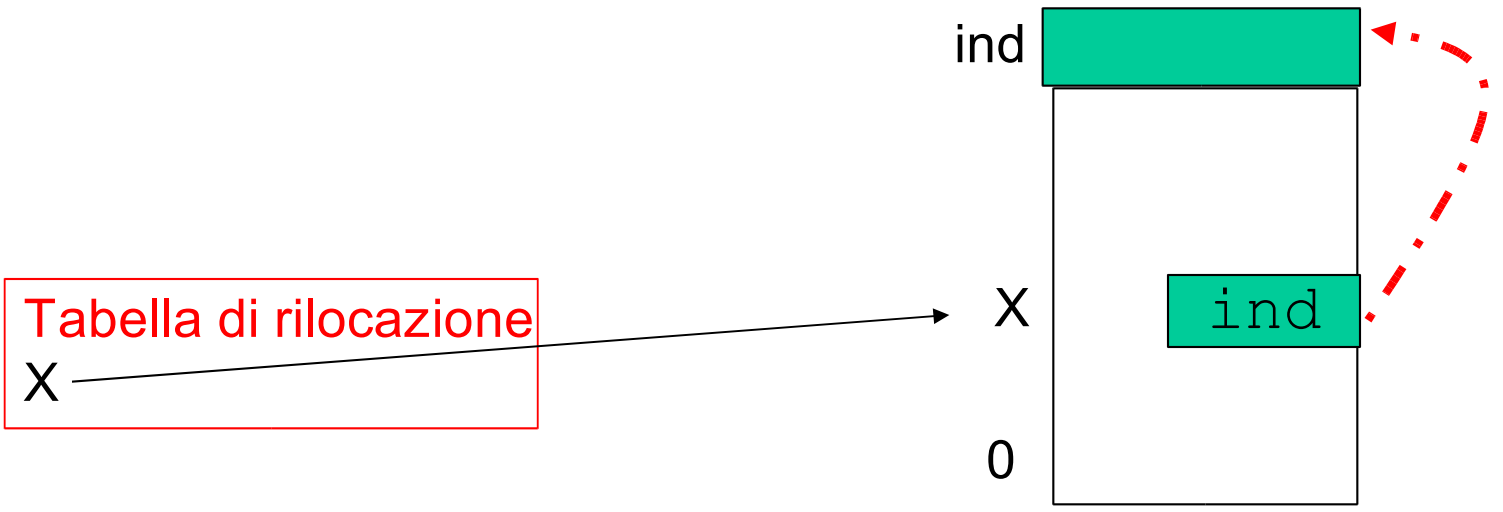





Datimain
Testomain
0
main.o
Tr,ts
Ind inizio
Tr,ts
Datifun
Testofun
0
fun.o
Ind inizio
Datimain
Datifun
Datilib
Testomain
Testofun
Testolib
0
Tr,ts
Datilib
Testolib
0
lib.o

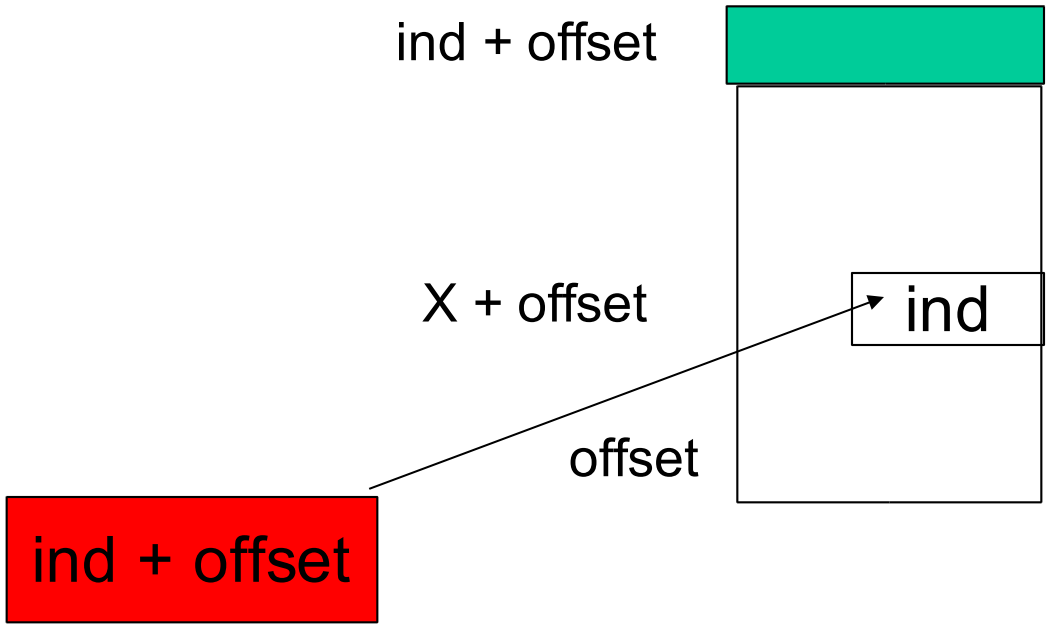
Linking: esempio statico (3)

- Tabella di rilocazione (cont.)
  - il codice pre-compilato è formato da testo e dati binari
  - l'assemblatore assume che l'indirizzo iniziale sia 0



Linking: esempio statico (4)

- Tabella di rilocazione (cont.)
  - ad ogni indirizzo rilocabile va aggiunto **`offset`**, l'indirizzo iniziale nell'eseguibile finale



Linking: esempio statico (5)

- Tabella dei simboli
  - identifica i simboli che il compilatore non è riuscito a 'risolvere', cioè quelli di cui non sa ancora il valore perché tale valore dipende dal resto dell'eseguibile finale
  - ci sono due tipi di simboli ...
    - definiti nel file ma usabili altrove (esportati)
      - es: i nomi delle funzioni definite nel file, i nomi delle variabili globali
    - usati nel file ma definiti altrove (esterni)
      - `es:` le funzioni usate nel file ma definite altrove (es. `printf()`)

Linking: esempio statico (6)

- Tabella dei simboli (cont.)
  - per i simboli esportati, la tabella contiene
    - nome, indirizzo locale
  - per i simboli esterni contiene
    - nome
    - indirizzo della/e istruzioni che le riferiscono

Linking: esempio statico (7)

- Il *linker* si occupa di risolvere i simboli.
  - Analizza tutte le tabelle dei simboli.
  - Per ogni simbolo non risolto (esterno) cerca
    - in tutte le altre tabelle dei simboli esportati degli oggetti da collegare (*linkare*) assieme
    - nelle librerie standard
    - nelle librerie esplicitamente collegate (opzione `-l`)

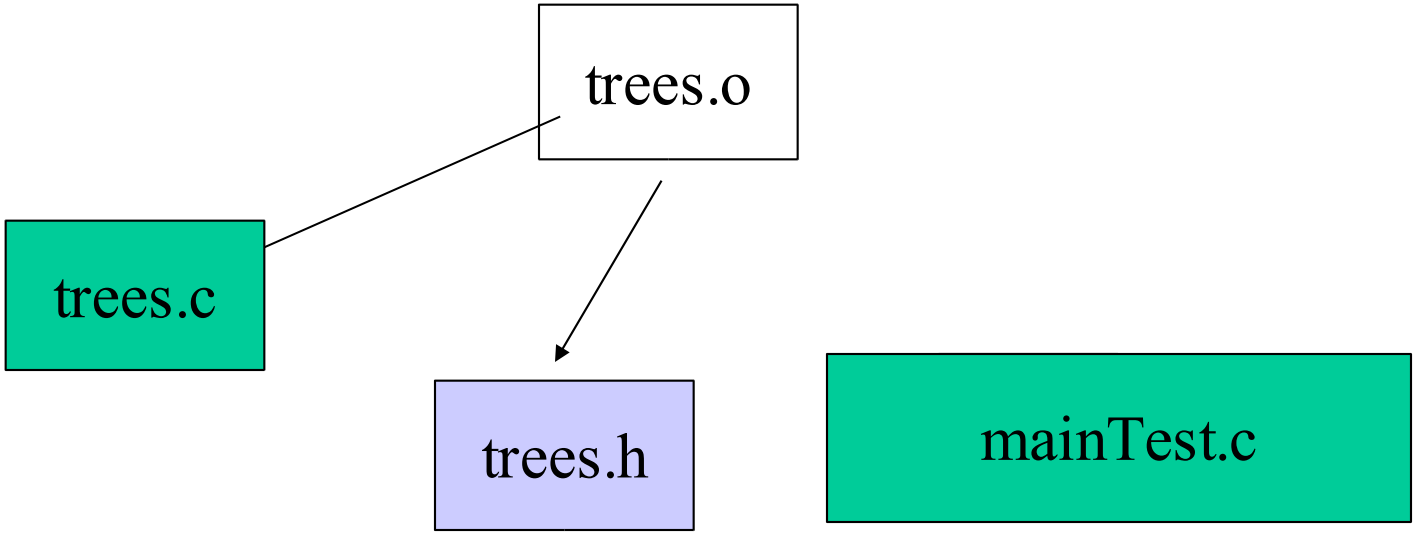
Linking: esempio statico (8)

- Il *linker* si occupa di risolvere i simboli (cont.)
  - Se il linker trova il simbolo esterno
    - ricopia il codice della funzione (linking *statico*) nell'eseguibile
    - usa l'indirizzo del simbolo per generare la CALL giusta o il giusto riferimento ai dati
  - Se non lo trova da errore ...
    - Provate a non linkare le librerie matematiche ...

Esempio

Compilare e linkare correttamente un piccolo progetto

Esempio: alberi binari

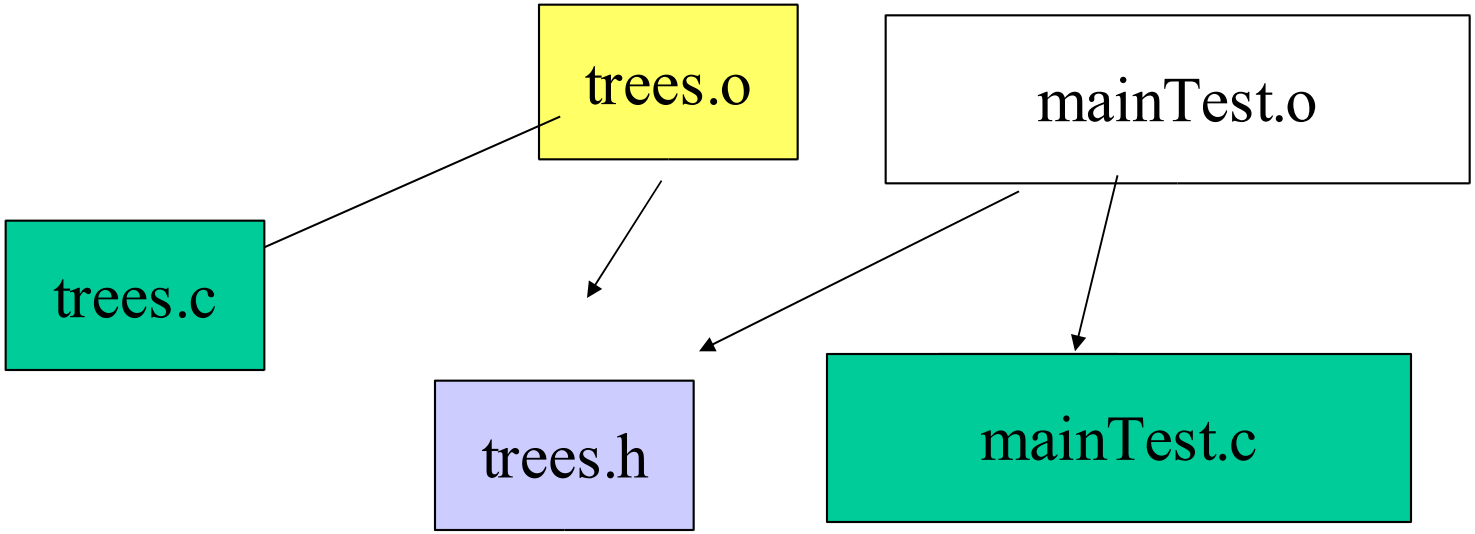


Passo (1):

```
bash:~$ gcc -Wall -pedantic -c trees.c
```

--crea trees.o

Esempio: trees … (2)

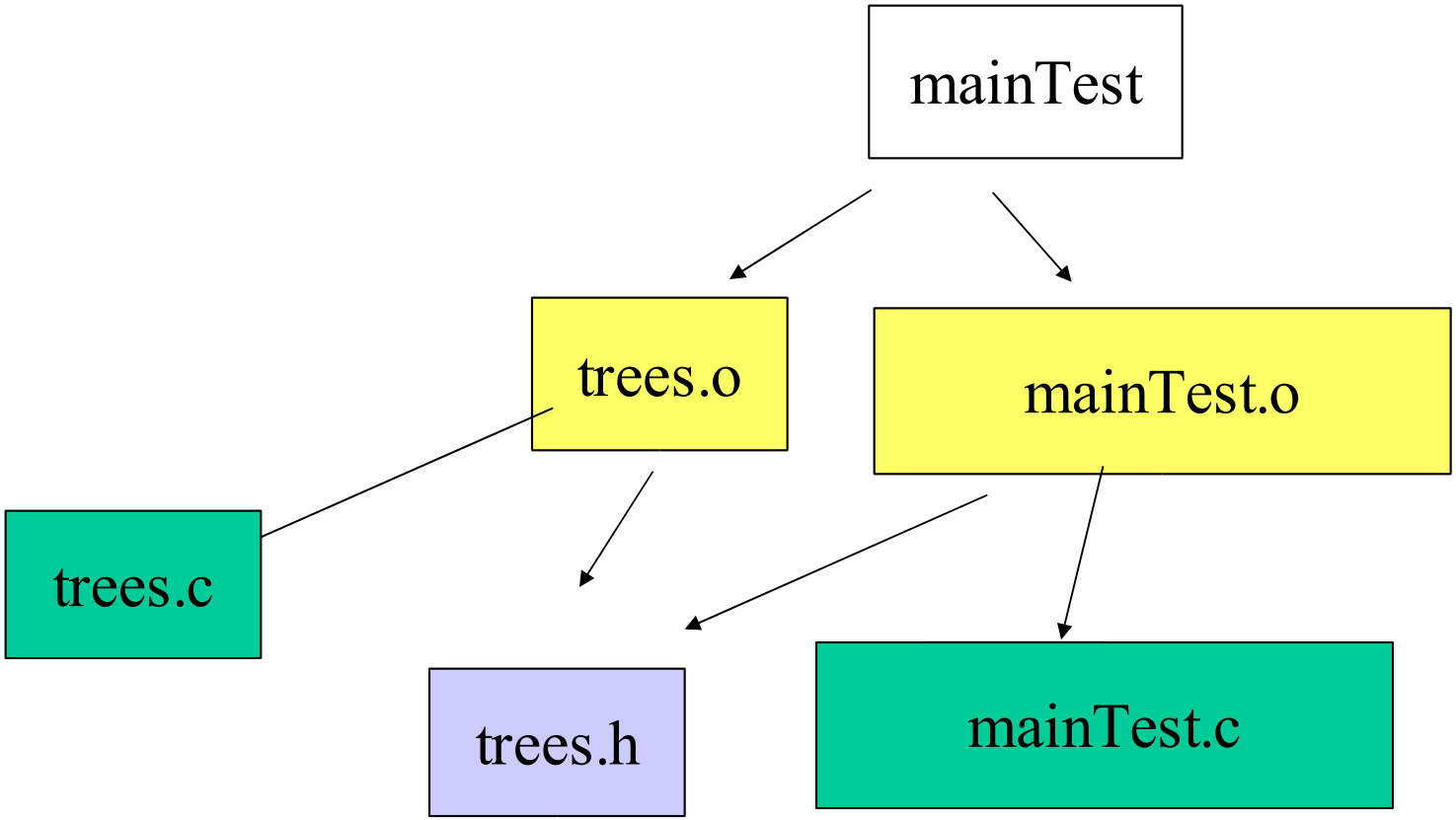


Come costruire l'eseguibile: passo (2):

```
$gcc -Wall -pedantic -c mainTest.c
```

--crea mainTest.o

Esempio: trees … (3)



Come costruire l'eseguibile: passo (3):

```
$gcc trees.o mainTest.o -o mainTest
```

--crea l'eseguibile 'mainTest'

Esempio: trees … (4)

```
$gcc -Wall -pedantic -c trees.c       --(1)
$gcc -Wall -pedantic -c manTest.c     --(2)
$gcc trees.o mainTest.o -o mainTest   --(3)
```

- se modifico **`trees.c`** devo rieseguire (1) e (3)
- se modifico **`trees.h`** devo rifare tutto

Esempio: trees … (5)

```
$ gcc -M trees.c
  --fa vedere le dipendenze da tutti i file
 anche dagli header standard delle librerie
  trees.o : trees.c /usr/include/stdio.h \
            /usr/include/sys/types.h \
            … … … … …
$ gcc -MM trees.c
  trees.o : trees.c trees.h
$
```

- perche' questo strano formato ?
  - per usarlo con il make ….

Makefile

Il file dependency system di Unix

(serve ad automatizzare il corretto aggiornamento di più file che hanno delle dipendenze)

makefile: idea di fondo

- (1) Permette di esprimere dipendenze fra file
  - es. **`f.o`** dipende da **`f.c`** e da **`t.h`** ed **`r.h`**
- in terminologia **`make`** :
  - **`f.o`** è detto *target*
  - **`f.c, t.h, r.h`** sono una *dependency list*

makefile: idea di fondo (2)

- (2) Permette di esprimere cosa deve fare il sistema per aggiornare il target se uno dei file nella *dependency list* è stato modificato
  - es. se qualcuno ha modificato **`f.c`**, **`t.h`** o **`r.h`**, per aggiornare **`f.o`** semplicemente ricompilare **`f.c`** usando il comando

    ```
    gcc -Wall -pedantic -c f.c
    ```

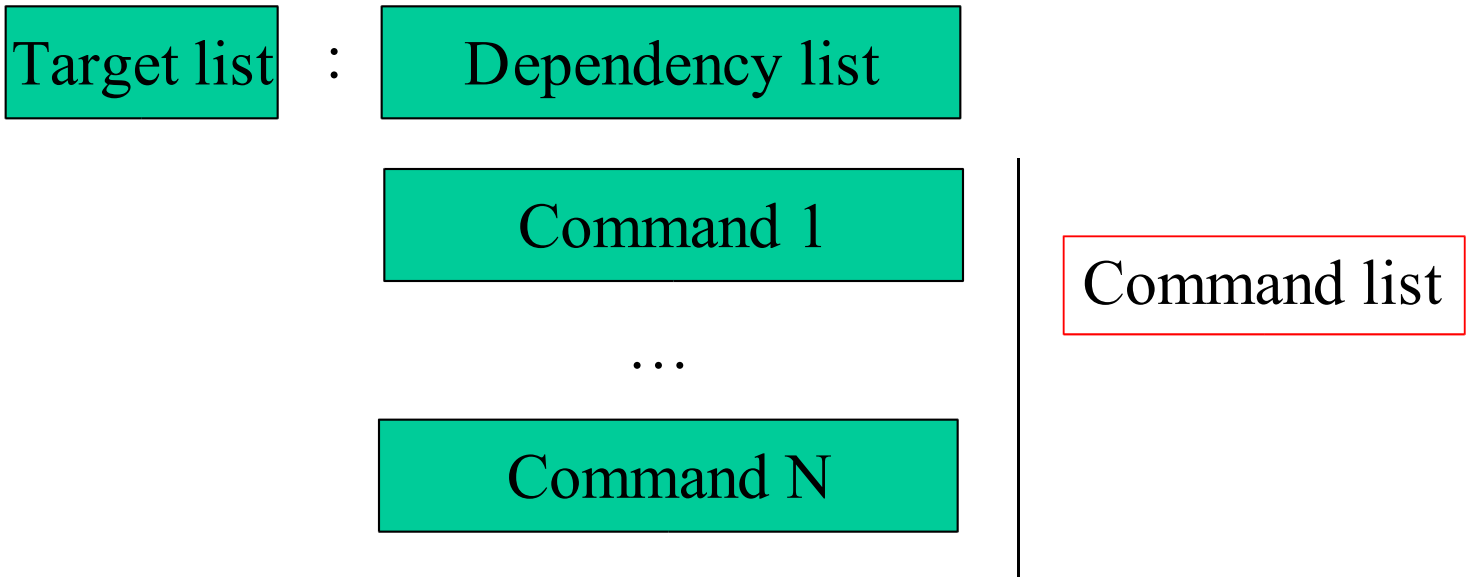
- In terminologia make :
  - la regola di aggiornamento di uno o più target viene detta *make rule*

makefile: idea di fondo (2)

- (3) L'idea fodamentale è:
  - descrivere tutte le azioni che devono essere compiute per mantenere il sistema consistente come make rule in un file (*Makefile*)
  - usare il comando **`make`** per fare in modo che tutte le regole descritte nel *Makefile* vengano applicate automaticamente dal sistema

Formato delle ‘make rule’

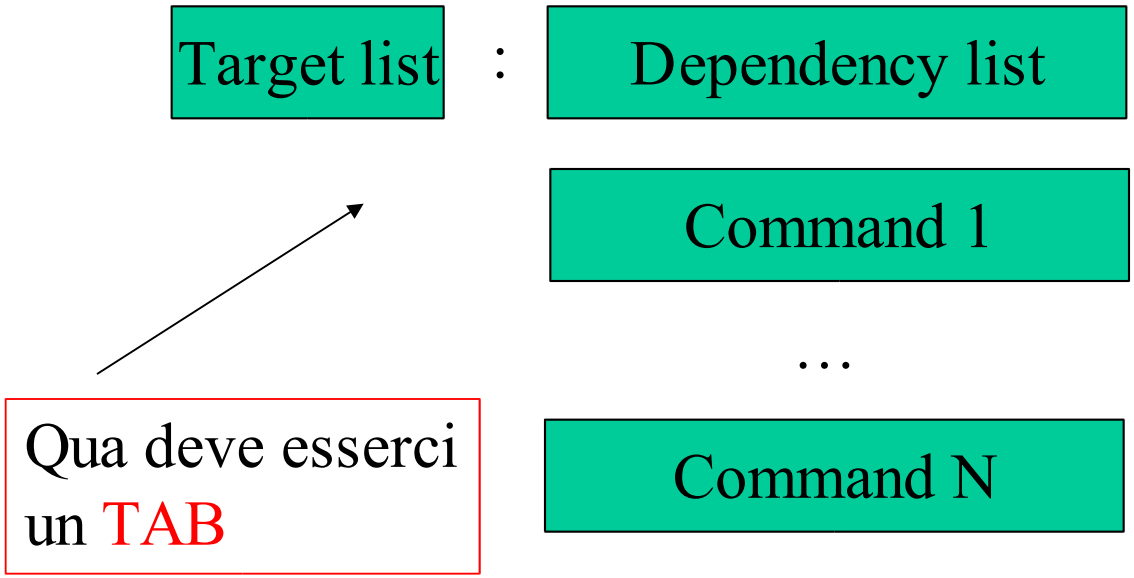
- Formato più semplice




```
f.o : f.c t.h r.h
      gcc -Wall -pedantic -c f.c
```

Formato delle ‘make rule’ (2)

- ATTENZIONE!!!




```
f.o : f.c t.h r.h
      gcc -Wall -pedantic -c f.c
```

Formato delle ‘make rule’ (3)

- Esempio con più regole

```
exe: f.o r.o
     gcc f.o r.o -o exe

f.o: f.c t.h r.h
     gcc -Wall -pedantic -c f.c

r.o: r.h r.c
     gcc -Wall -pedantic -c r.c
```

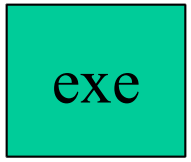
Fra due regole deve esserci almeno una LINEA VUOTA

Il file deve terminare con un NEWLINE

Formato delle ‘make rule’ (4)

- L’ordine delle regole è importante!
  - Il make si costruisce l’albero delle dipendenze a partire dalla prima regola del makefile

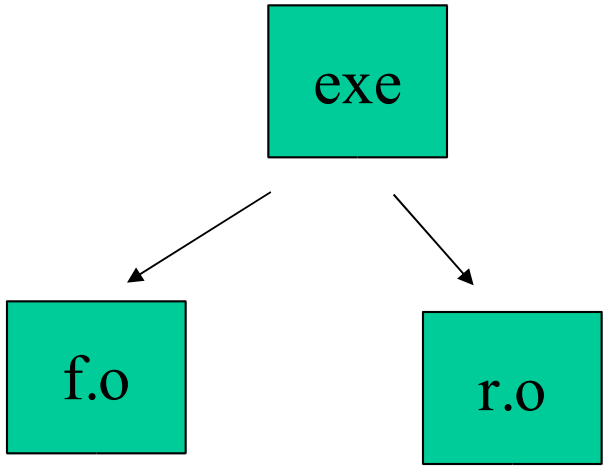
Il/I target della prima regola trovata sono la radice dell’albero



Formato delle ‘make rule’ (5)

- L’ordine delle regole è importante!
  - Il make si costruisce l’albero delle dipendenze a partire dalla prima regola del makefile

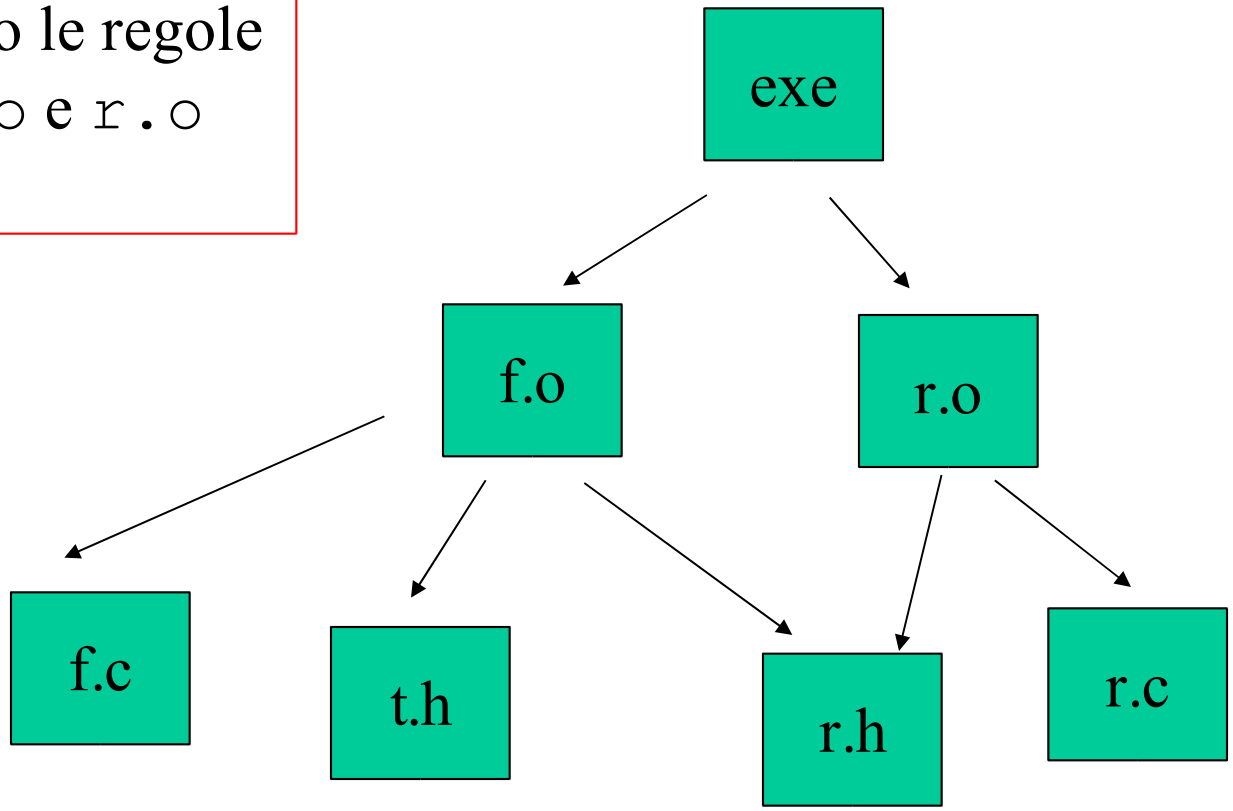
Ogni nodo nella dependency list della radice viene appeso come figlio



Formato delle ‘make rule’ (6)

- L’ordine delle regole è importante!
  - Poi si visitano le foglie e si aggiungono le dipendenze allo stesso modo

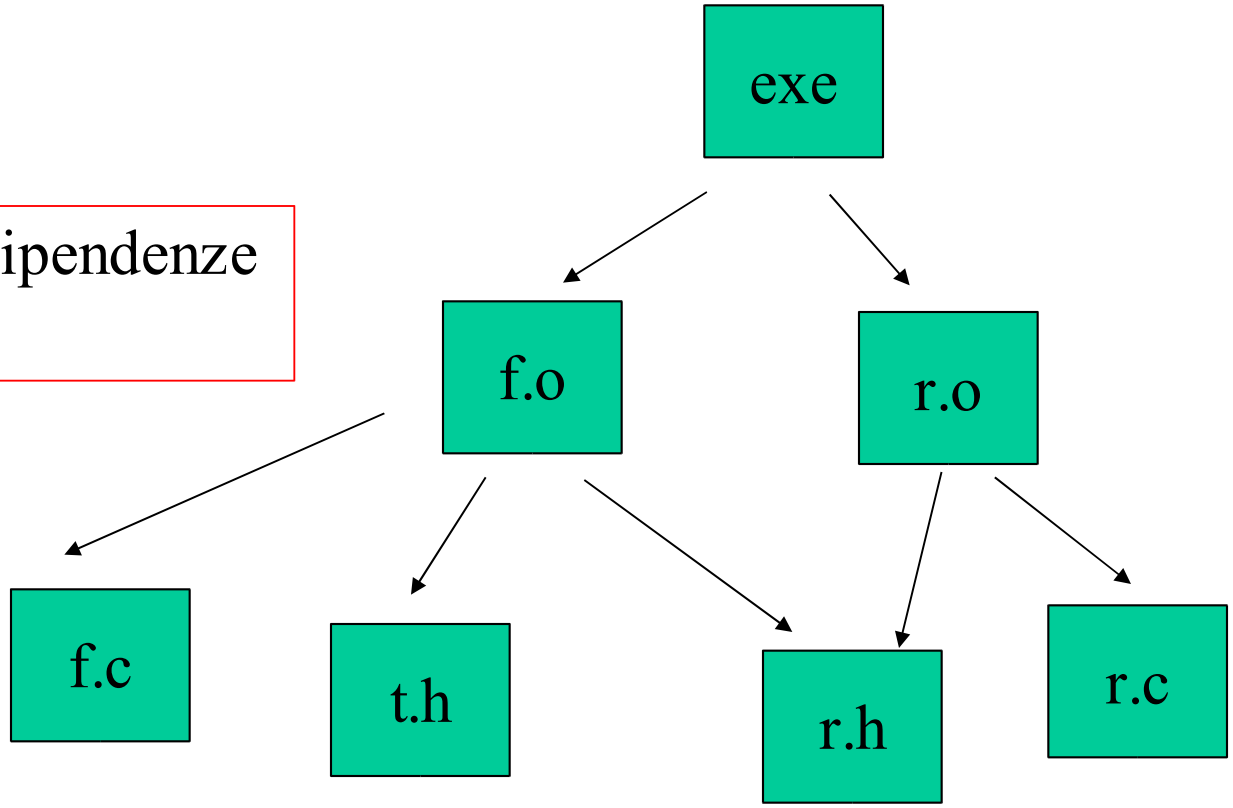
Si considerano le regole che hanno `f.o` e `r.o` come target



Formato delle ‘make rule’ (7)

- L’ordine delle regole è importante!
  - La generazione dell’albero termina quando non ci sono più regole che hanno come target una foglia

Albero delle dipendenze complessivo

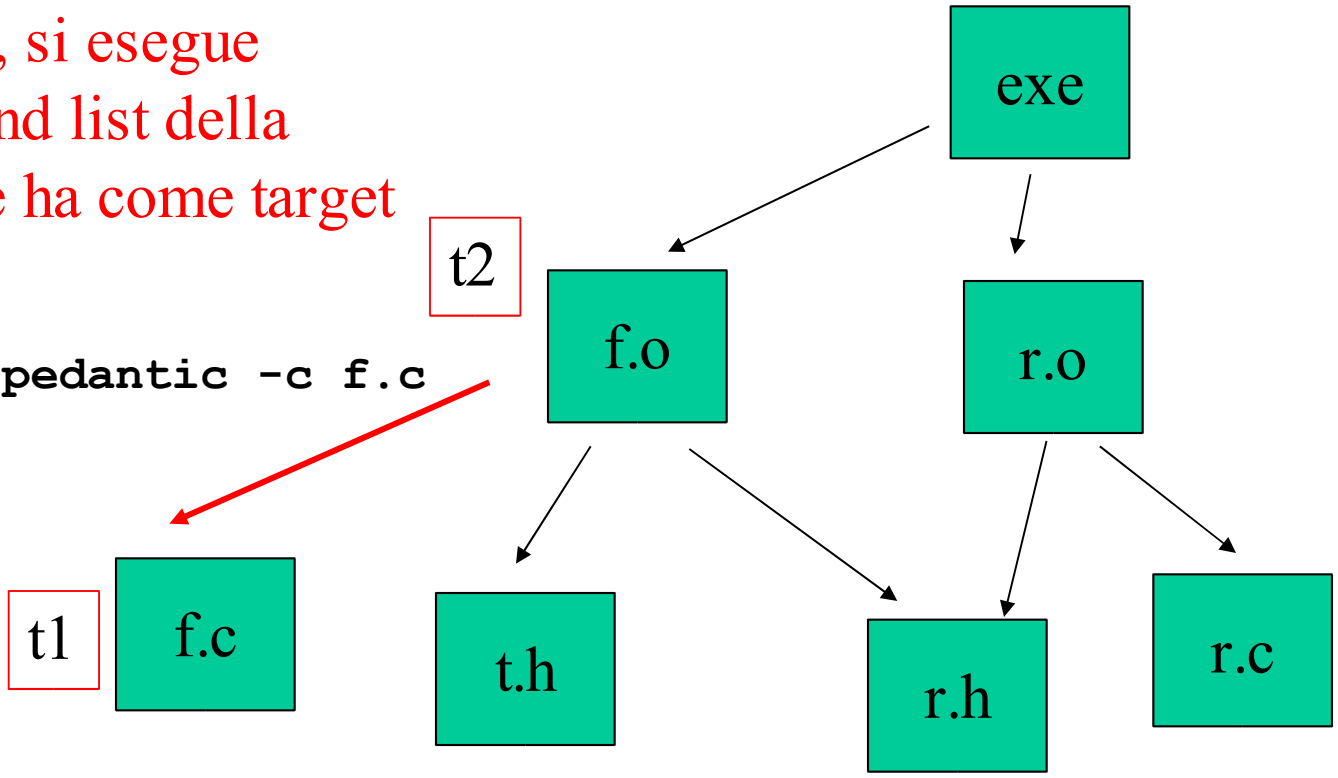


Come viene usato l'albero ...

- Visita bottom up
  - Per ogni nodo X si controlla che il tempo dell'ultima modifica del padre sia successivo al tempo dell'ultima modifica di X

Se t1 > t2, si esegue la command list della regola che ha come target il padre

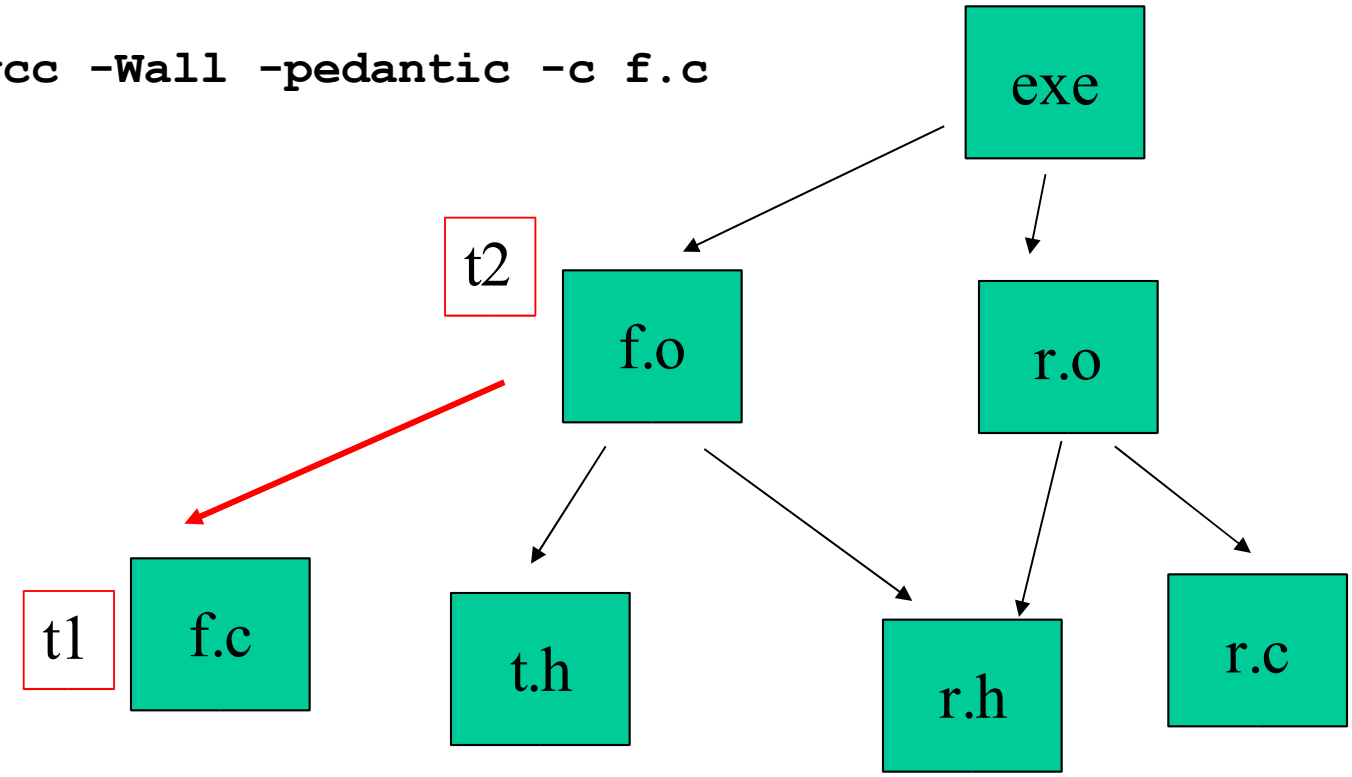
```
gcc -Wall -pedantic -c f.c
```



Come viene usato l'albero … (2)

- Visita bottom up
  - Se il file corrispondente ad un nodo X non esiste (es. è stato rimosso) … Si esegue comunque la regola che ha come target X

```
gcc -Wall -pedantic -c f.c
```



Come si esegue il make ...

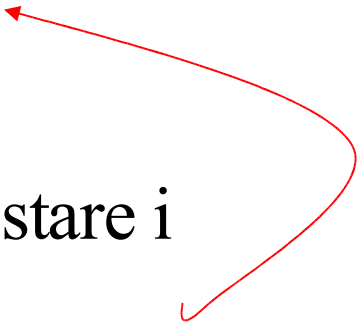
- Se il file delle regole si chiama 'Makefile'
  - basta eseguire

```
$ make
```

- altrimenti ….

```
$ make -f nomefile
gcc -Wall -pedantic -c f.c
$
```

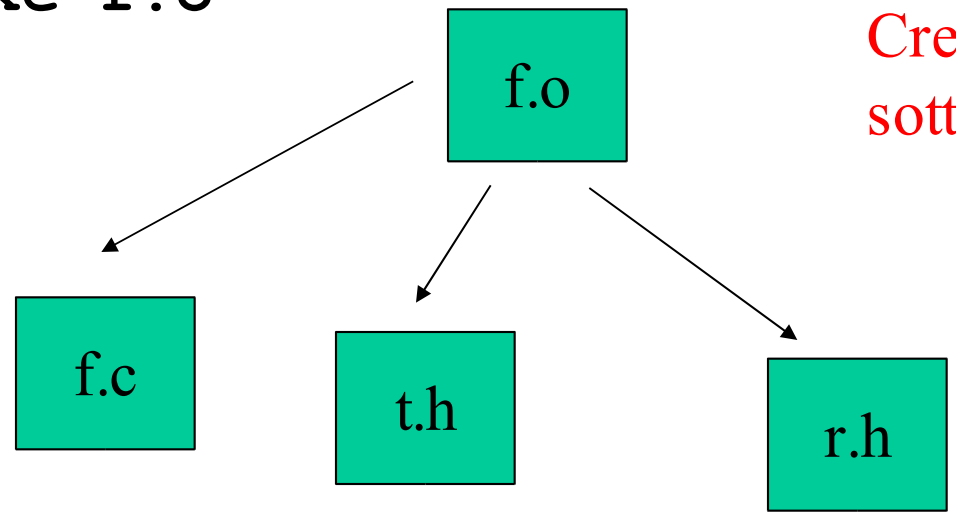
  - stampa dei comandi eseguiti per aggiustare i tempi sull'albero delle dipendenze
  - **`-n`** per stampare solo i comandi (senza eseguirli)



Come si esegue il make … (2)

- È possibile specificare una radice dell'albero diversa dal target nella prima regola del file
  - dobbiamo passare il nome del target come parametro al make. Es.

```
$ make f.o
```



Crea solo questo sottoalbero

Variabili ...

- È possibile usare delle variabili per semplificare la scrittura del makefile
  - stringhe di testo definite una volta ed usate in più punti

```
# nomi oggetti
objects = r.o f.o
# regole
exe: $(objects)
	gcc $(objects) -o exe
```

Variabili (2)

- Inoltre ci sono delle variabili predefinite che permettono di comunicare al make le nostre preferenze, ad esempio :
  - quale compilatore C utilizzare per la compilazione

    ```
    CC = gcc
    ```

  - le opzioni di compilazione preferite

    ```
    CFLAGS = -Wall -pedantic
    ```

- a che serve poterlo fare ?

Regole implicite ...

- Le regole che abbiamo visto finora sono più estese del necessario
  - Il make conosce già delle regole generali di dipendenza fra file, basate sulle estensioni dei nomi
  - es : *nel caso del C, sa già che per aggiornare un **XX.o** è necessario ricompilare il corrispondente **XX.c** usando $CC e $CFLAGS*
  - quindi una regole della forma

```
XXX.o: XXX.c t.h r.h
        gcc -Wall -pedantic -c XXX.c
```

Regole implicite … (2)

– È equivalente a

```
XXX.o: XXX.c t.h r.h
        $(CC) $(CFLAGS) XXX.c
```

– e sfruttando le regole implicite del make può essere riscritta come

```
XXX.o: t.h r.h
```

Regole implicite … (3)

- Riscriviamo il nostro esempio con le regole implicite e le variabili :

```
CC = gcc
CFLAGS = -Wall -pedantic
objects = f.o r.o

exe: f.o r.o
	$(CC) $(CFLAGS) $(objects) -o exe

f.o: t.h r.h

r.o: r.h
```

Phony targets ...

- È possibile specificare target che non sono file e che hanno come scopo solo l'esecuzione di una sequenza di azioni

```
clean:
        rm $(exe) $(objects) *~ core
```

  - siccome la regola non crea nessun file chiamato 'clean', il comando **`rm`** verrà eseguita ogni volta che invoco

```
$make clean
```

- 'clean' è un target fittizio (*phony*) inserito per provocare l'esecuzione del comando in ogni caso

Phony targets … (2)

- Questo stile di programmazione è tipico ma ha qualche controindicazione :
  - se casualmente nella directory viene creato un file chiamato ‘clean’ il gioco non funziona più
    - siccome la dependency list è vuota è sempre aggiornato!
  - È inefficiente!
    - Il make cerca prima in tutte le regole implicite per cercare di risolvere una cosa che è messa apposta per non essere risolta

Phony targets … (3)

- Soluzione :
  - prendere l'abitudine di dichiarare esplicitamente i target falsi

```
.PHONY : clean
clean:
        -rm $(exe) $(objects) *~ core
```

`'-rm'` significa che l'esecuzione del make può continuare anche in caso di errori nell'esecuzione del comando **`rm`** (es. uno dei file specificati non c'è)

Esempio

- un makefile per l'esercizio sugli alberi :

```
CC = gcc
CFLAGS = -Wall -pedantic

.PHONY: all test cleanall

all: mainTest
mainTest: mainTest.o trees.o
mainTest.o: mainTest.c trees.h
trees.o: trees.c trees.h
```

Esempio … (2)

- un makefile per l'esercizio sugli alberi (cont.):

```
test: mainTest
	@echo "Eseguo i test … "
	./mainTest 1> output
	diff output output.atteso
	@echo "Test superato!"

cleanall:
	@echo ”Removing garbage"
	-rm -f *.o core *.~
```

Documentazione su make

- Make può fare molte altre cose
- per una descrizione introduttiva Glass
  - pp 329 e seguenti
- per una descrizione davvero dettagliata *info* di *emacs*
  - ESC-X info
  - cercare (CTRL-S) "make"