

# 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  

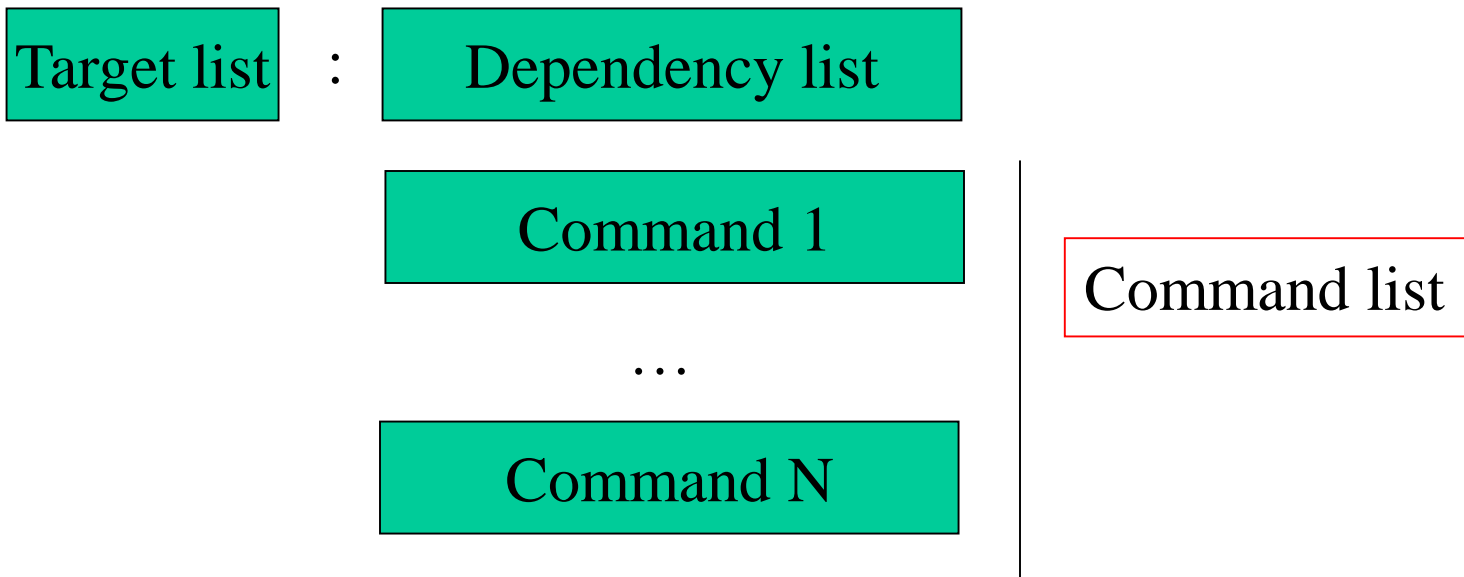
```
gcc -Wall -pedantic -c f.c
```
- In terminologia make :
  - la regola di aggiornamento di uno o più target viene detta make rule

# makefile: idea di fondo (2)

- (3) L'idea fondamentale 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

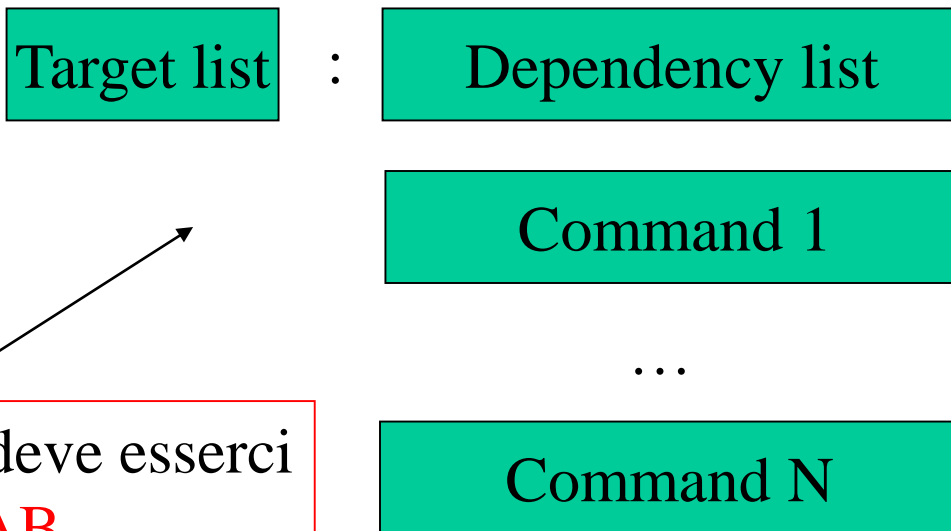


```
f.o : f.c t.h r.h
```

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

# Formato delle 'make rule' (2)

- ATTENZIONE!!!



Qua deve esserci  
un **TAB**

```
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
```

Fra due regole  
deve esserci almeno  
una **LINEA VUOTA**

```
f.o: f.c t.h r.h
```

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

```
r.o: r.c r.h
```

```
gcc -Wall -pedantic -c r.c
```

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 è/sono la radice dell'albero



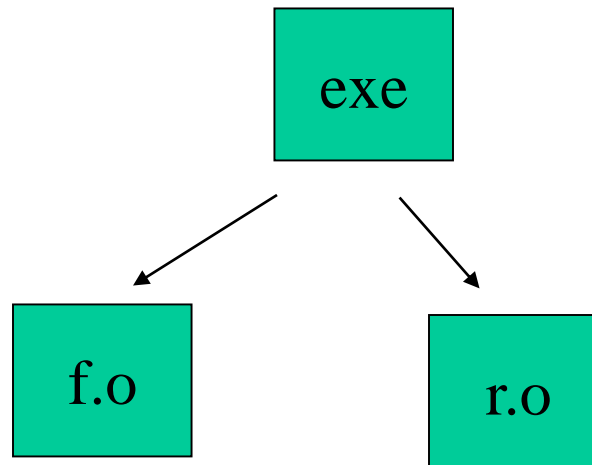
exe



# 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

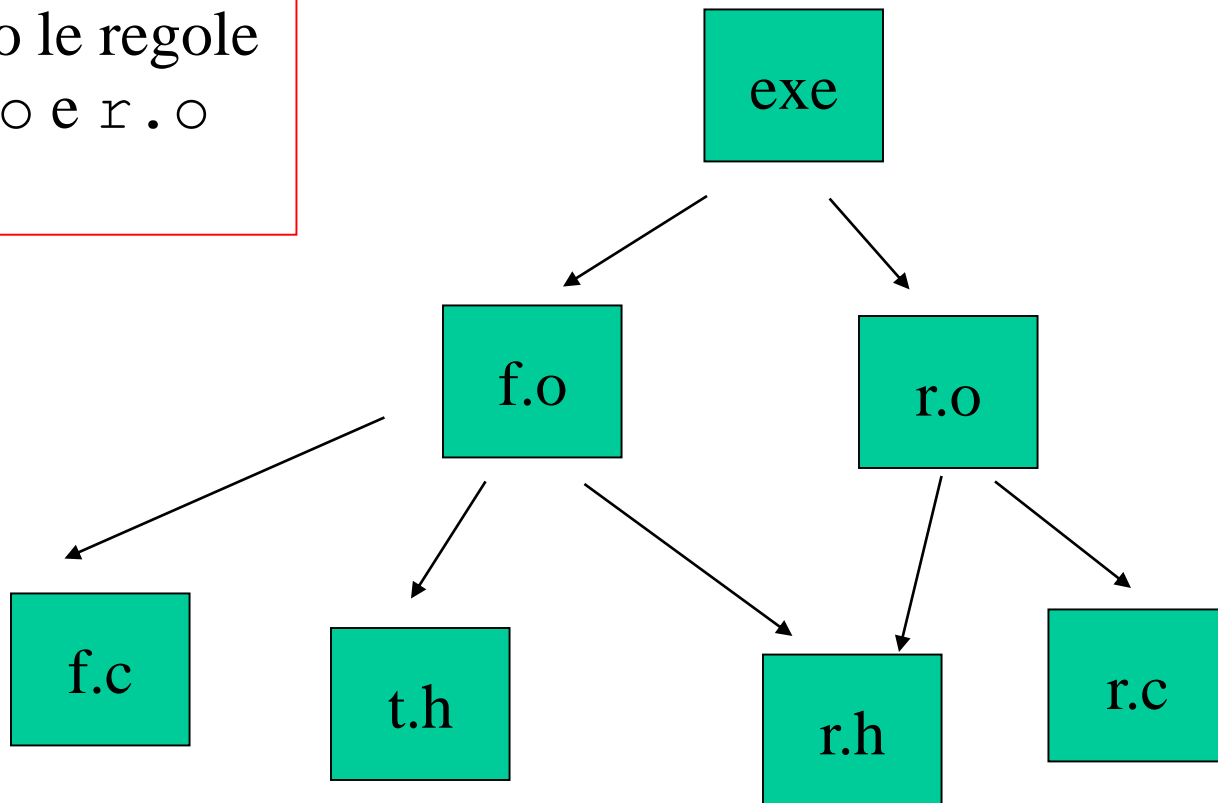
Ogni nodo nella dependency list della radice viene appeso come figlio



# Formato delle 'make rule' (6)

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

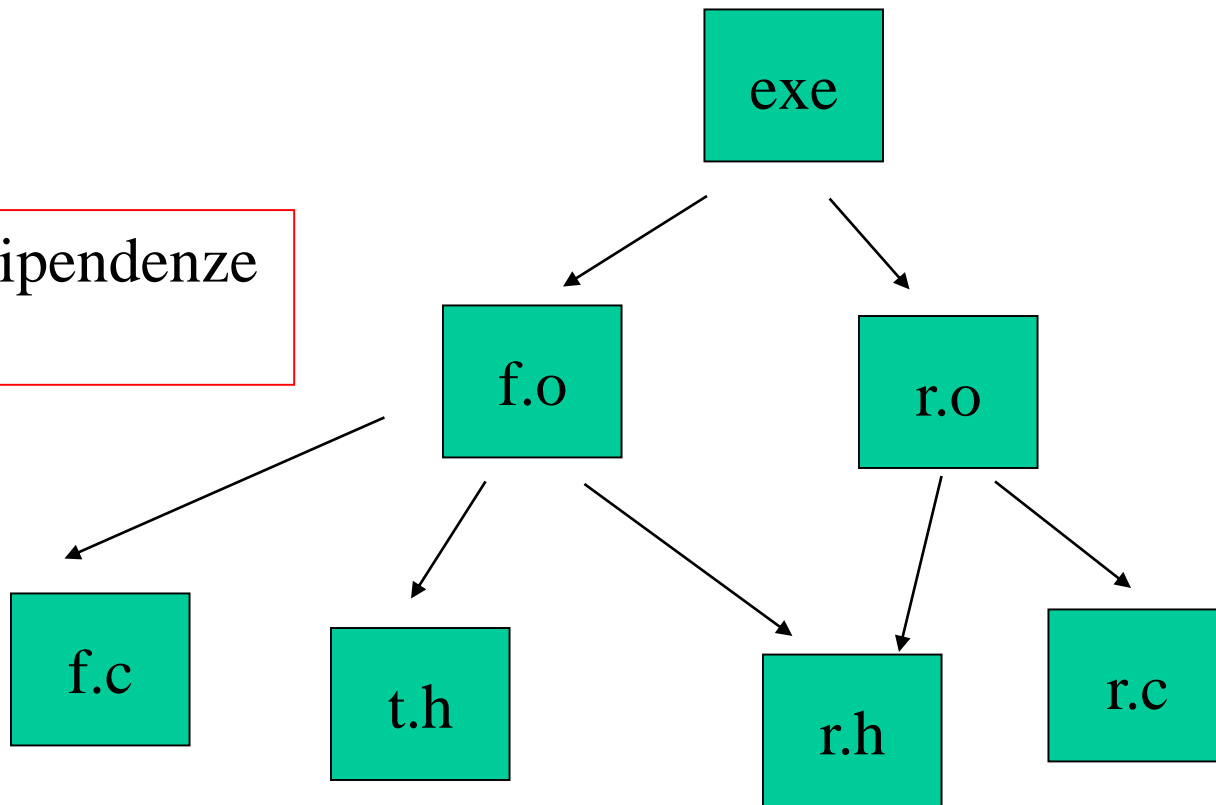
Si considerano le regole che hanno `f.o` e `r.o` come target



# 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



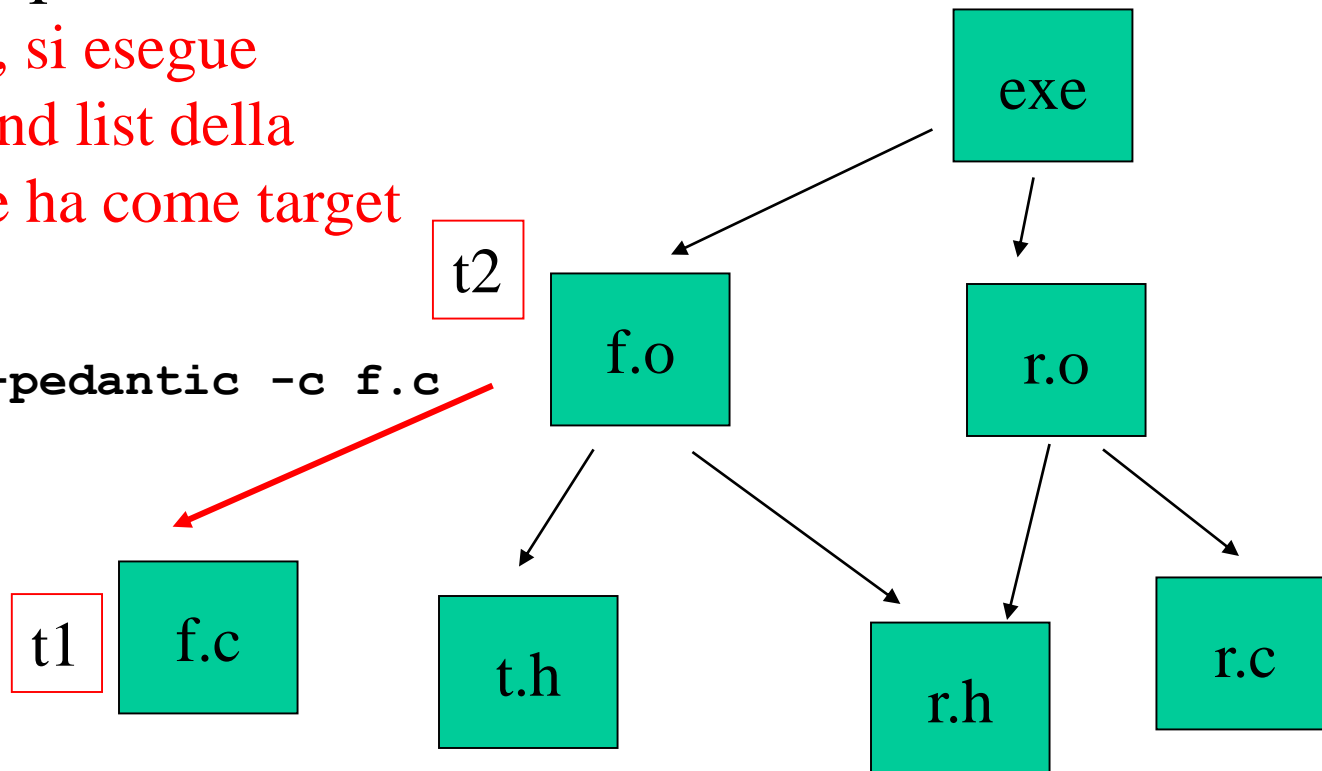
# Come viene usato l'albero ...

- 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

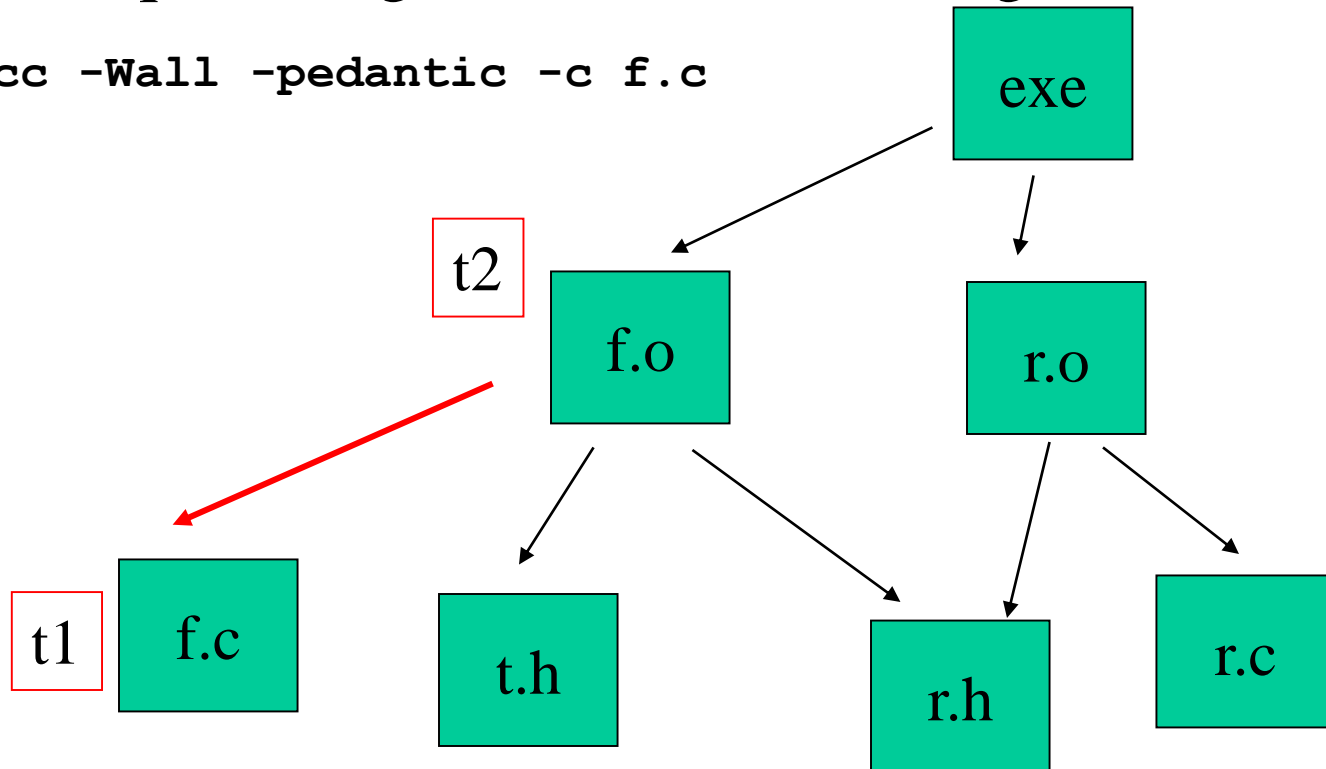
`gcc -Wall -pedantic -c f.c`



# 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 ....

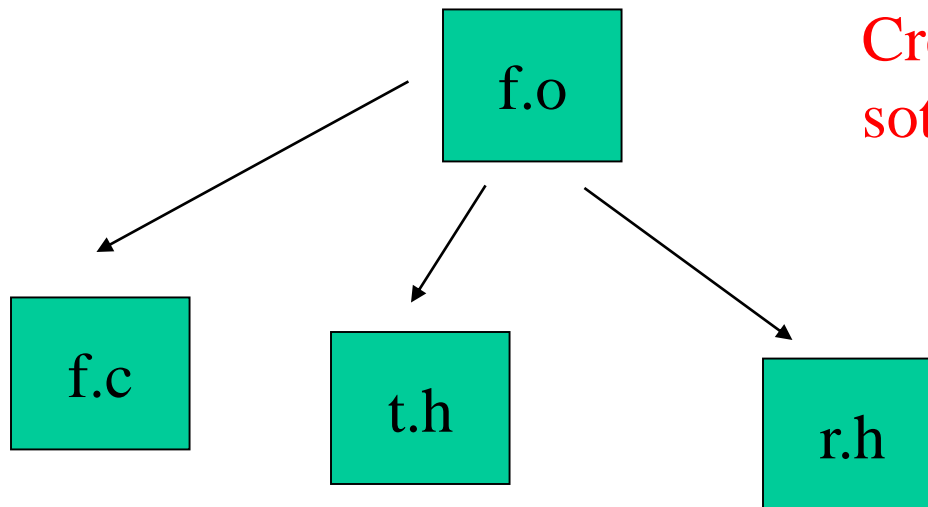
```
$ make -f nomefile  
gcc -Wall -pedantic -c f.c  
$
```

  - 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
- Vediamo un esempio ...



# Esempio di nuovo

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

# Variabili ...

- È possibile usare delle variabili per semplificare la scrittura del makefile
  - Sono *nomi* per 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 ?

# Esempio di nuovo

```
exe: $(objects)
      $(CC) $(objects) -o exe
```

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

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

# 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 regola 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) -c XXX.c
```

– e sfruttando le regole implicite del make può essere riscritta come (stile `gcc -MM`)

```
XXX.o: XXX.c 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: $(objects)
```

```
$(CC) $(objects) -o exe
```

```
f.o: f.c t.h r.h
```

```
r.o: r.c r.h
```

# Altre variabili

- Si possono usare altre variabili predefinite
  - `$@` : nome del target
  - `$$` : la dependency list
  - `$$` : il primo nome nella dependency list

```
exe: $(objects)  
      $(CC) $$ -o $@
```

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



# 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 $(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!
    - La lista dei comandi non viene mai eseguita
  - 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 $(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)

altrimenti make si blocca al primo comando che ritorna errore ( $\neq 0$ )

# Esempio

- un makefile per l'esempio sulle liste :

```
CC = gcc
```

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

```
.PHONY: test cleanall
```

```
hTest: hTest.o liste.o
```

```
$(CC) $^ -o $@
```

```
hTest.o: hTest.c liste.h
```

```
liste.o: liste.c liste.h
```

# Esempio ... (2)

- (cont.):

```
test: hTest
```

```
@echo "Eseguo i test ... "
```

```
./hTest 1> output
```

```
diff output output.atteso
```

```
@echo "Test superato!"
```

```
cleanall:
```

```
@echo "Removing garbage"
```

```
-rm -f *.o core *.*
```

# Esempio ... (3)

- Una possibile esecuzione

```
bash$ make cleanall
```

```
Removing garbage
```

```
rm -fr *.o core *.*~
```

```
bash$ make hTest
```

```
gcc -Wall -pedantic -c liste.c
```

```
gcc -Wall -pedantic -c hTest.c
```

```
gcc liste.o hTest.o -o hTest
```

# Esempio ... (4)

- Una possibile esecuzione dei test

```
bash$ make test
```

```
Eseguo i test
```

```
./hTest 1> output
```

```
diff output output.atteso
```

```
Test superato!
```