**Laboratorio Sistemi Operativi: Make**

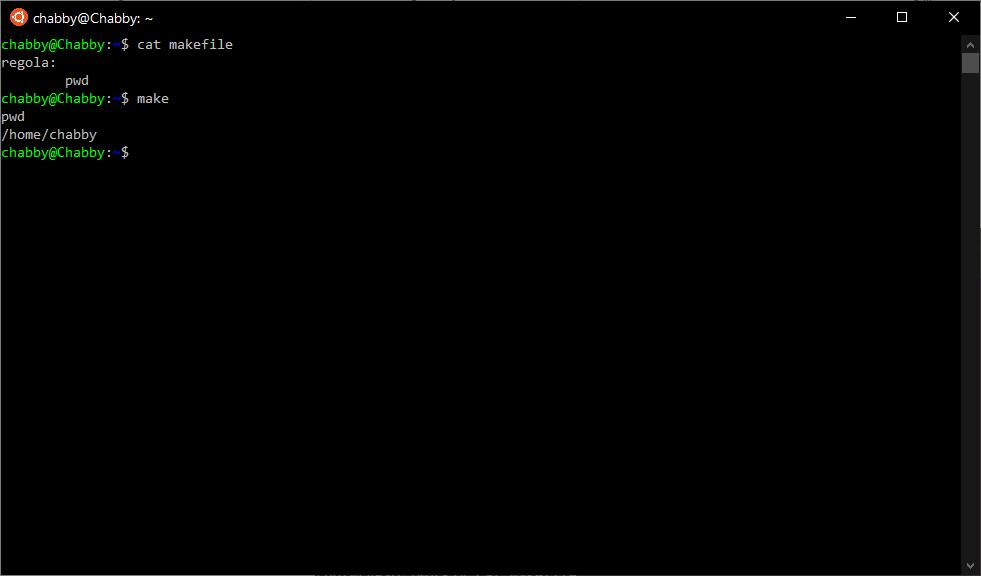
* Make è un programma usato per **organizzare il workflow** di programmazione.
* Viene usato per il **building** di un’applicazione (oltre che per lo sviluppo di Kernel o per automatizzare una serie di operazioni):
  + Nella programmazione viene usato per **automatizzare** ilprocesso che parte da un **file sorgente** (o lista di sorgenti) per poi arrivare ad un’**applicazione finale**.
* Si basa su omonimo comando evocabile da riga di comando:
  + **make [option]**
  + make --version: mostra la versione di make installata.
* **Makefile**:
  + **Componente fondamentale** di make è il file **makefile** (file di testo) che **contiene** un **insieme di regole** che devono essere eseguite.
  + Il makefile contiene una **lista di operazioni** suddivise in gruppi e ogni **regola** (operazione) ha come riferimento un **target specifico** (normalmente è il contenuto del file system, un **file**).
  + Ricapitolando, il makefile è un **file che contiene un insieme di regole** e ogni regola ha come riferimento finale un file (normalmente questo file deve essere generato):
    - Queste **regole contengono una serie di operazioni** che sono necessarie per arrivare al target (per costruire il file finale).
* Quello che ci interessa è capire come poter **automatizzare il workflow** classico (soprattutto se si ha più di un file sorgente):
  + Creazione un file (**example.c**) che conterrà codice.
  + Per compilarlo uso il comando: **gcc example.c -o example**.
  + Per eseguirlo devo eseguire il comando: **./example**.
* Il **processo di building** (il workflow che porta dai file sorgenti al file binario finale) può **richiedere una serie di passi complicati** ed essere molto **oneroso** (soprattutto se ho **tanti sorgenti** da gestire e spesso non serve ricompilare tutto il progetto, ma solo i file modificati).
* Con il tool make e quindi **tramite il makefile possiamo definire una serie di regole** (step) da **eseguire**, così da organizzare meglio il lavoro:
  + Queste **regole sono definite nel makefile** (è possibile definire una catena di makefile).
  + Ogni regola è definita come segue:

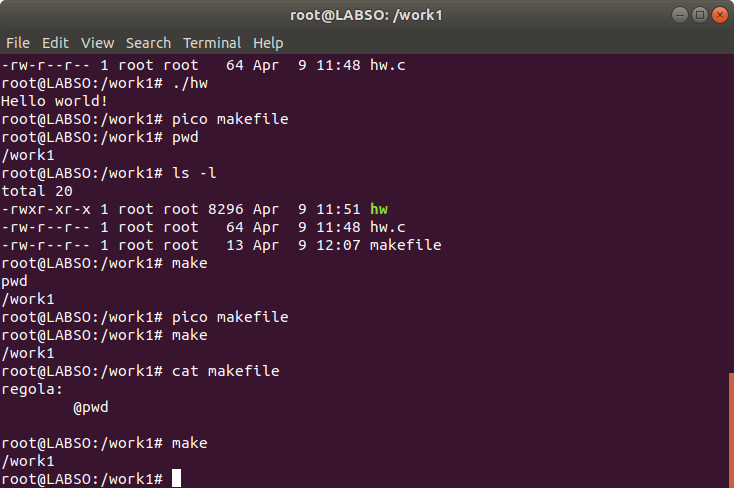
**rule [options]:**

**action 1**

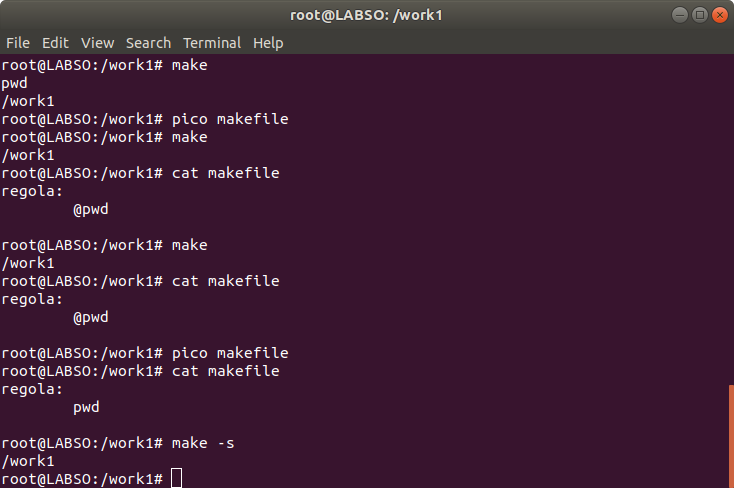
….

**action n**

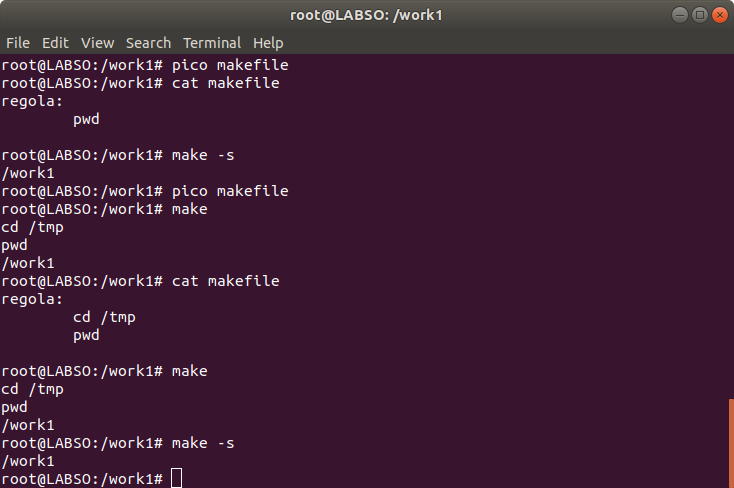
* + Il **nome della regola deve essere contenuto in un’unica riga** (posso aggiungere opzioni) e le azioni da eseguire devono essere elencate sotto al nome della regola e **l’indentazione deve essere esclusivamente mediante tabulazione** (no spazi).
  + Le azioni possono essere **singoli comandi** bash o anche **liste di comandi** da eseguire.
* All’interno del **makefile** si possono avere delle **direttive** (caratteristiche aggiuntive):
  + Utilizzeremo delle **variabili speciali** (simili a delle **macro**) in modo che a dei contenuti ripetitivi possa assegnargli un nome mnemonico e riutilizzarli più volte facilmente all’interno del file descrittivo
  + Posso avere **commenti**.
  + Come sappiamo, ogni regola ha una serie di azioni che la caratterizzano, ma **ogni azione è eseguita in un processo separato**.
  + Digitando il comando make (senza opzioni) il **tool cerca nella cartella corrente il makefile**:
    - Esistono **3 denominazioni standard** per il makefile:
      * **GNUmakefile** (riservato per codice di sistema)
      * **Makefile** e **makefile**
    - Se make trova un file con uno di questi nomi, lo utilizza in automatico.
    - Posso anche **specificare il nome del makefile** da usare tramite l’opzione **-f**:
      * **make -f [“nome makefile”]**
    - Se eseguo il comando **make senza opzioni**, vengono **eseguite tutte le regole** contenute nel makefile (viene **mostrato il comando eseguito** e il suo **output** su terminale).
    - Anteponendo **@** davanti ai comandi (nel makefile) si può **nascondere la stampa del comando** quando si esegue il comando make, quindi si vede solo l’output del comando eseguito:
      * **@ va messa davanti a tutti i comandi di cui non voglio mostrare il comando** ma solo il suo risultato:



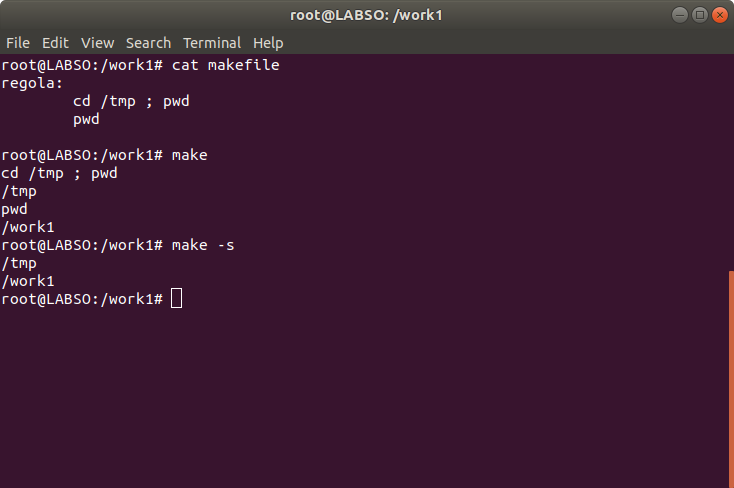
* + - Si può ottenere lo **stesso comportamento** usando l’opzione **-s**:
      * **make -s**
      * in questo modo tutti i comandi contenuti nel makefile mostrano solo l’output :



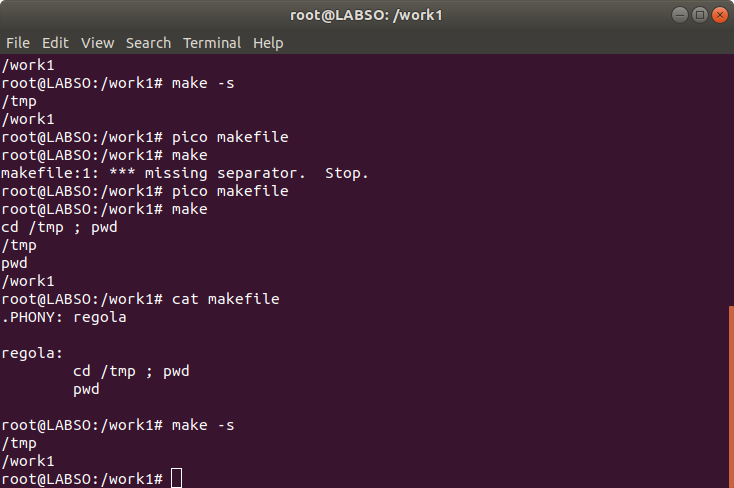
* + - I **comandi contenuti nel makefile vengono eseguiti su processi diversi**:



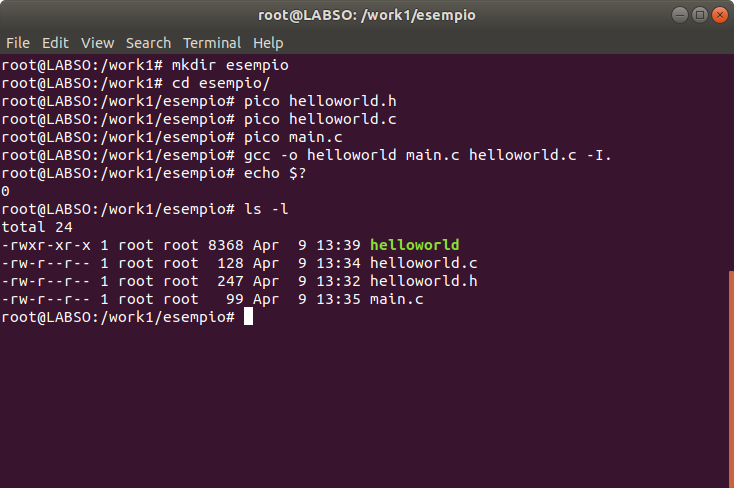
* + - * Viene eseguito il comando ‘*cd /tmp*’, ma quando si esegue il comando ‘*pwd*’ (stampa cartella corrente) non viene stampato ‘*/tmp*’ ma sempre ‘*/work1*’.
      * Questo avviene perché i **due comandi** (scritti su due righe diverse) sono **eseguiti su due processi diversi**, dunque **ogni processo è eseguito in maniera separata ed indipendente da tutti gli altri comandi**.
      * Se **concateno i comandi sulla stessa riga**, quello che ottengo è che i **comandi sulla stessa riga vengono eseguiti sullo stesso processo**:



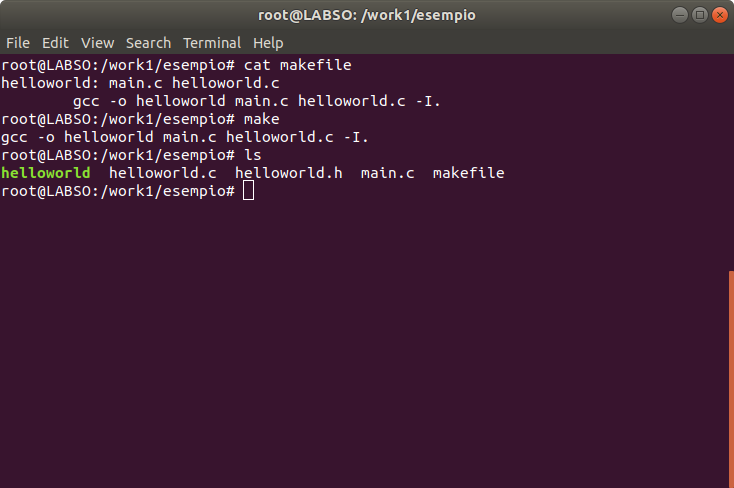
* + - * Comandi “sulla stessa riga” vengono eseguiti nello **stesso contesto** (stesso processo) e quindi sono collegati tra loro.
* **Regole**:
  + Sono composte da:
    - **Target** (solitamente è un **file**):
      * La **lista di azioni da eseguire permette di arrivare al target indicato dalle dipendenze** (ad esempio, il target potrebbe essere un binario da generare e le dipendenze possono essere il sorgente).
      * Il **nome della regola è il target** (file da generare).
      * Per generare una **regola** **con** **un** **target fasullo** (file che non voglio che venga creato), bisogna specificare che la regola che sto creando è un fake target **tramite l’indicazione** ‘**.PHONY’**:
        + **.PHONY: [regola1, … , regolaN**] è una **direttiva** (metaregola) che **indica** che tutte **le denominazioni indicate successivamente sono delle** **regole presenti nel makefile che non rappresentano dei file** (sono solo regole di denominazione di comodo):



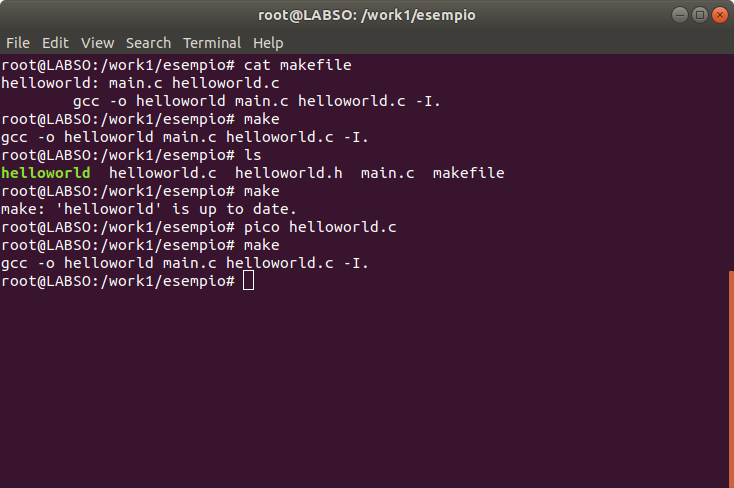
* + - **Lista di dipendenze**
    - **Lista di azioni da eseguire**
    - **Sintassi speciali**
* **Makefile** **non** **è un’alternativa allo scripting**, anzi **make utilizza la bash** (visto che le azioni che esegue sono comandi). La sua particolarità principale è il **controllo delle dipendenze**:
  + è una comodità che può essere utile soprattutto in progetti grossi per evitare di ripetere tutti gli step che portano alla generazione dell’applicazione in caso di piccole modifiche.
* **Esempio**:
  + Creo 3 file:
    - **helloworld.h**: contiene la dichiarazione della funzione.
    - **helloworld.c**: contiene il corpo della funzione.
    - **main.c**: richiama la funzione.
  + Per compilare devo eseguire il comando:
    - **gcc -o helloworld main.c helloworld.c -I**
      * **-o**: indica il **nome** che deve avere il **file binario** che si verrà a creare.
      * **-I**: indica che deve essere **inclusa** la **cartella corrente** nella ricerca dei file sorgenti.



* + - Partendo dai 3 file, il comando di compilazione genera il file binario finale.
  + Questo procedimento possiamo realizzarlo anche mediante make, dove nel makefile:
    - **Dichiariamo una nuova regola helloworld che andrà a generare un file chiamato helloworld** e **per generarlo ha bisogno di sorgenti**, che andranno **specificati sulla stessa riga**, dopo i due punti:
      * A **destra**:
        + Dopo i due punti ho le **dipendenze**, di **quali file ho bisogno per generare quanto indicato sulla sinistra**.
      * A **sinistra**:
        + Prima dei due punti ho il **nome del file che andrò a creare**.
    - La **regola** andrà a contenere il comando di compilazione:
      * **gcc -o helloword main.c helloworld.c -I**



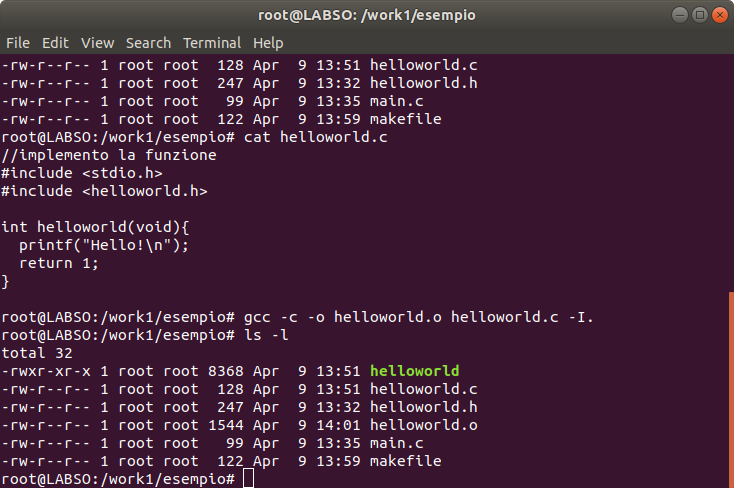
* + - Quando eseguo il comando make, questo andrà ad eseguire le regole contenute nel makefile:
      * **Se non trova il fine binario specificato**, **lo crea** eseguendo la regola.
      * Se trova il file binario, make se ne accorge e:
        + **esegue la regola**.
        + **non esegue la regola** (stampa il messaggio “*make: helloworld is up to date*”).
      * **Prima** di eseguire una regola, **make** **analizza le date di ultima modifica dei file** indicati come “**dipendenza**” del file binario (controlla helloworld.c e main.c):
        + Se **modifico il file sorgente** (helloworld.c o main.c), **make** analizzando le date di ultima modifica, **rileva che la data dei sorgenti è successiva alla data di creazione del file binario** (helloworld), capisce che la versione del file binario è obsoleta (qualcosa è cambiato nel file sorgente) e quindi **riesegue la regola per aggiornare l’eseguibile**.



* + - * + Se le **date di ultima modifica delle dipendenze sono minori della data di ultima modifica del file binario**, invocando **make**, questo **non esegue la regola**.
        + Se la **data di ultima modifica di una delle dipendenze è più recente rispetto al file binario**, allora in automatico **make** **riesegue la regola** e aggiorna il file binario.
    - È possibile definire delle **macro** nel makefile:
      * **CC=gcc**:
        + In questo modo se dovessi cambiare compilatore, basta modificare una volta la macro e non devo cercare e modificare nel codice il comando gcc.
      * È possibile **creare file oggetto**, file di **estensione .o,** dunque il comando di compilazione diventa:
        + **gcc -c -o helloworld.o helloworld.c -I**

**-c**: indica che voglio generare un file oggetto.

* + - * + Questo comando genera un file oggetto ma questo file non è ancora linkato all’eseguibile finale.



* + - Per essere più generici, nel makefile posso definire, usando la wildcard %:
      * **%.o : %.c $(DEPS):**
        + **Tutti i file .o dipendendo da corrispondenti file .c**
        + Questa **regola** viene **applicata a tutti i file .o** andando a **cercare** un **corrispondente file .c**
      * Se questa **regola** viene **applicata perché viene trovato un file di nome helloworld.o**, **verrà cercato un corrispondente file helloworld.c**
      * Visto che si ha un problema in fase di compilazione, in quanto la riga di compilazione varia a seconda del file, anche il comando di compilazione cambia:
        + **$(CC) -c -o @$ $< $(CCFLAGS)** corrisponde **gcc -c -o helloworld.o helloworld.c -I**, dove:

**$(CC)**:

macro che contiene il valore gcc.

**$(CCFLAGS)**:

macro che contiene il valore -I.

**@$**:

Questa variabile contiene il nome del file che si sta analizzando in quel momento.

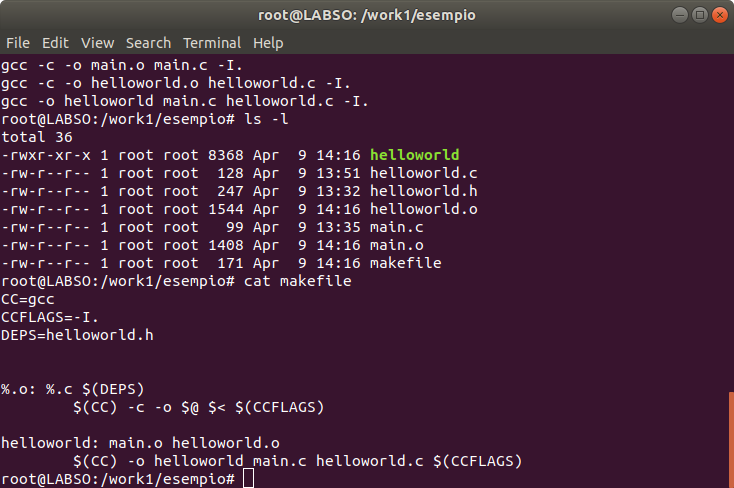
Contiene quello che è a destra dei due punti, nel nostro caso helloworld.o

**$<**:

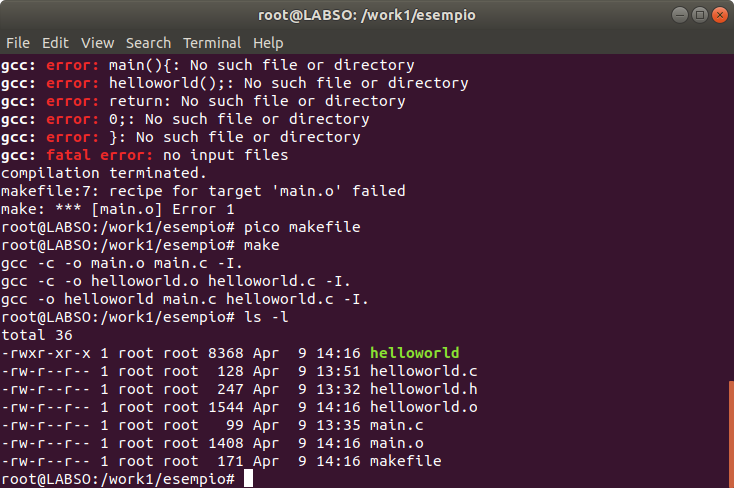
Di tutte le dipendenze a destra, questa variabile contiene la prima dipendenza valorizzata nel contesto in cui viene applicata.

Nel nostro caso helloworld.c

* + - * + Questo comando si può applicare anche nel caso ci siano altri file con estensione .c e volessimo generare un file .o per ognuno di loro.



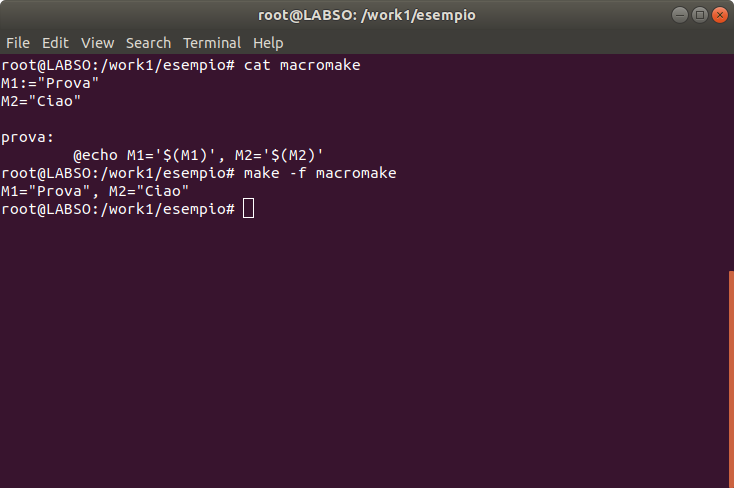
* + - * Nella regola che genera l’eseguibile, devo modificare il main.c in main.o (**l’istruzione sopra genera un file .o per ogni .c**, quindi anche main.c avrà un .o associato).
    - Richiamando make senza opzioni, **inizia l’esecuzione delle regole una dopo l’altra in ordine**:
      * Viene trovata ed eseguita la regola che per ogni file .c genera un corrispettivo file .o
      * L’ultima regola, prende i due file .o e genera l’eseguibile.

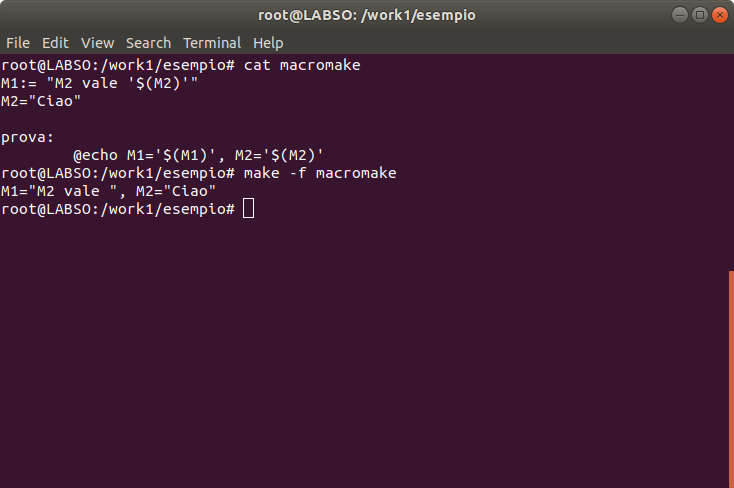


o

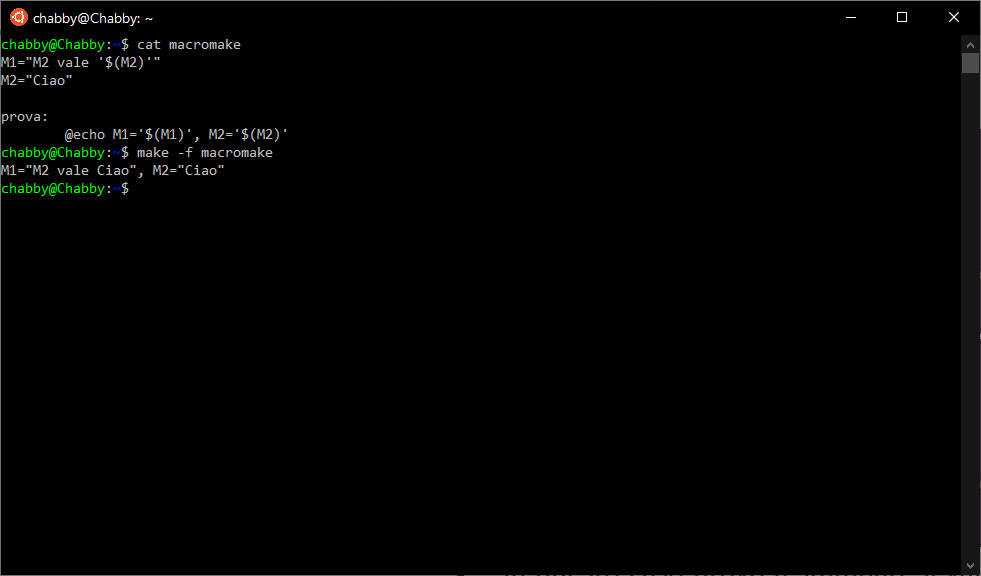
o

* + - La **prima regola** specifica che se è **presente un file con estensione .c**, bisogna **generare** il **corrispettivo file di estensione .o** eseguendo il comando specificato (valorizzato con i contenuti che ha trovato in quel momento).
    - La **seconda regola** specifica che **se vengono trovati 2 file di estensione .o** presenti nella cartella corrente, deve essere **generato** **un file eseguibile** (helloworld)
    - I **file che generano l’eseguibile** helloworld **all’inizio non sono presenti**, questi infatti vengono **generati dopo l’esecuzione della prima regola.**
    - Dunque la prima regola genera i file .o per ogni file .c mentre la seconda regola usa questi file .o per generare l’eseguibile helloworld.
* **Macro**:
  + Per settare una macro si hanno due modi:
    - **MACRO1 := Value**
    - **MACRO2 = $(MACRO1)**
    - Differenza:





* + - * In questo caso provo a stampare il valore di M2 prima che venga inizializzato il suo valore, dunque non viene stampato niente (si è usato la definizione MACRO:=VALUE).



* + - * In questo caso viene stampato valore di M2 in quanto l’inizializzazione avviene all’invocazione (si è usato la definizione con MACRO = VALUE):
        + Quando eseguo la regola, al posto di M1=’$(M1)’, il **valore della macro viene sostituito con la sua definizione**, quindi si ha che:

**M1 = ’$(M1)’ = ‘M2 vale $(M2)’**

* + - * + E a questo punto viene calcolato il valore di $(M2).
      * Se si definisce una macro usando:
        + **MACRO = VALUE**:

Il valore della macro viene definito ma è valorizzato nel momento dell’utilizzo.

Quando richiamo la macro, in quel momento, essa assuma il suo valore.

* + - * + **MACRO := VALUE**:

Si ha una definizione statica al momento della definizione.

Viene valutata l’espressione sulla destra nel momento della definizione.

* + Per utilizzare (avere il suo **valore**) una macro si usa **$()**, ad esempio: **$(MACRO1)**
* **Funzioni speciali**:
  + **Shell**:
    - Permette di catturare un output di un comando bash.
    - **CONT=$(shell cat dati.txt)**
      * Richiamo il comando cat per mostrare il contenuto del file dati.txt.
      * Catturo il suo output mediante il comando shell e il risultato lo assegno alla macro CONT.
      * Gli “*a capo*” sono convertiti in spazi singoli.
  + **Wildcard**:
    - Permette di elencare file
    - **SRCS=$(wildcard \*.c)**
      * Elenca tutti i file di estensione .c nella cartella di lavoro e la lista di questi file è assegnata alla macro SRCS.
  + Ecc…