Programmazione Avanzata e parallela

Lezione 03

Luca Manzoni

Opzioni extra

Ancora qualche opzione del compilatore

- **-Wall**
  abilita tutti i warning ragionevoli (non tutti!)

- **-pedantic**
  abilita tutti i warning relativi al non rispetto dello standard (e.g., utilizzo di estensioni specifiche del compilatore)

- -**Wextra**
  Ancora più warning di -Wall (non necessariamente da considerare tutti)

- **-std=c89** (o anche c90)
  -std=c99
  -std=c11
  -std=c17 (o anche c18)
  indica che versione del C si sta utilizzando (lo standard è stato aggiornato più volte negli anni)

Cosa vedremo oggi

Make e Makefile

- Gestire il build di progetti con make
  - Cosa sono i makefile
  - Regole dei makefile
  - Utilizzo di variabili
  - Regole template
  - A cosa prestare attenzione

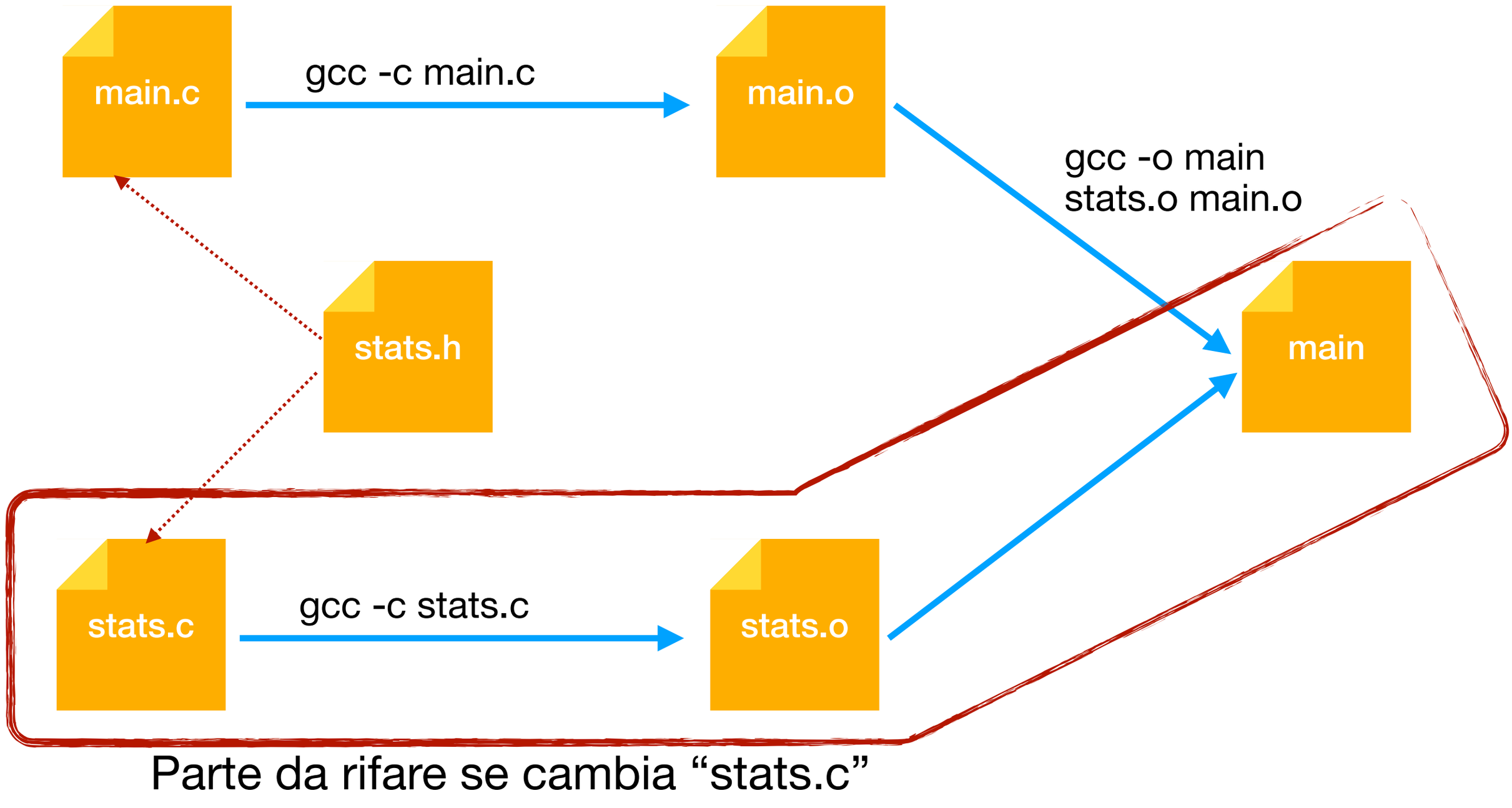
Make

Motivazioni per l'esistenza

- Se abbiamo decine o centinaia (o molte migliaia) di file .c e .h sarebbe utile gestire bene la compilazione separata
- Se modifico il file "test.c" devo sicuramente ricreare il file "test.o" e fare il linking
- Mi serve ricompilare tutto il resto?
- Se modifico il file "test.h"?
- Sarebbe utile avere un sistema che in automatico mi ricompila solo quello che serve

Passi di compilazione

E cosa servirebbe rifare



Make

Motivazioni per l'esistenza

- In generale vorremmo un sistema in cui possiamo specificare tre cose:
  - Che output vogliamo produrre (il file .o, l'eseguibile, etc)
  - Quali sono i prerequisiti per (e.g., avere i file .c e .h corretti, avere tutti i file .o)
  - Che comando eseguire per generare l'output dati i prerequisiti
- Il sistema dovrebbe poi essere in grado di trovare cosa serve per generare i prerequisiti se non li abbiamo (ma abbiamo una regola per generarli)

Make

Alcuni esempi

- Cosa vorremmo specificare:
  - *Obiettivo*: main
  - *Prerequisiti*: main.o, stats.o
  - *Comando*: gcc -o main main.o stats.o
  - *Obiettivo*: stats.o
  - *Prerequisiti*: stats.c, stats.h
  - *Comando*: gcc -c stats.c

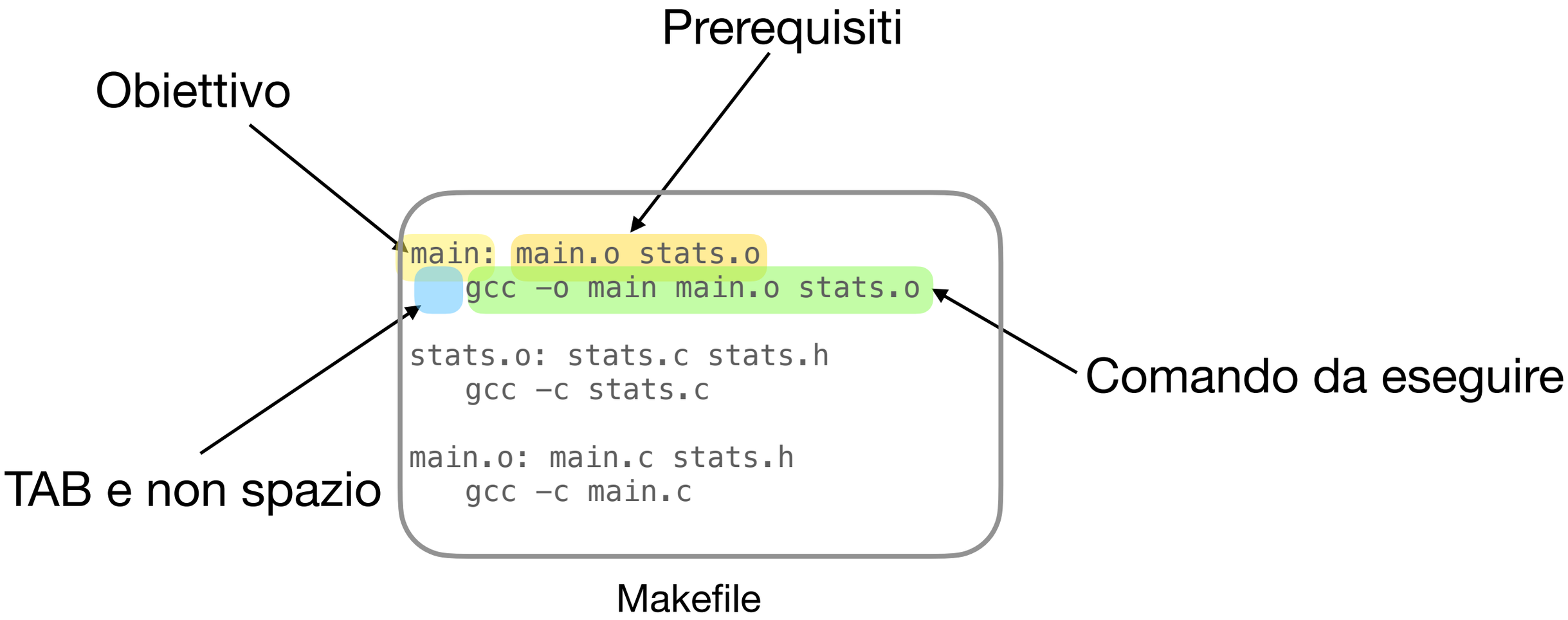
Makefile

Struttura di base

- Make è uno strumento che ci permette di specificare queste tre cose (obiettivi, prerequisiti e comandi da eseguire)
- La prima versione è del 1976, esistono molte implementazioni che, per le funzionalità essenziali, sono tutte compatibili
- Di default basta creare un file chiamato “Makefile” e invocando make quello sarà il file di comandi considerato
- L’idea è che invocando “make” posso fare tutti (e soli) i passi necessari a costruire la versione aggiornata del mio programma

Makefile

Esempio



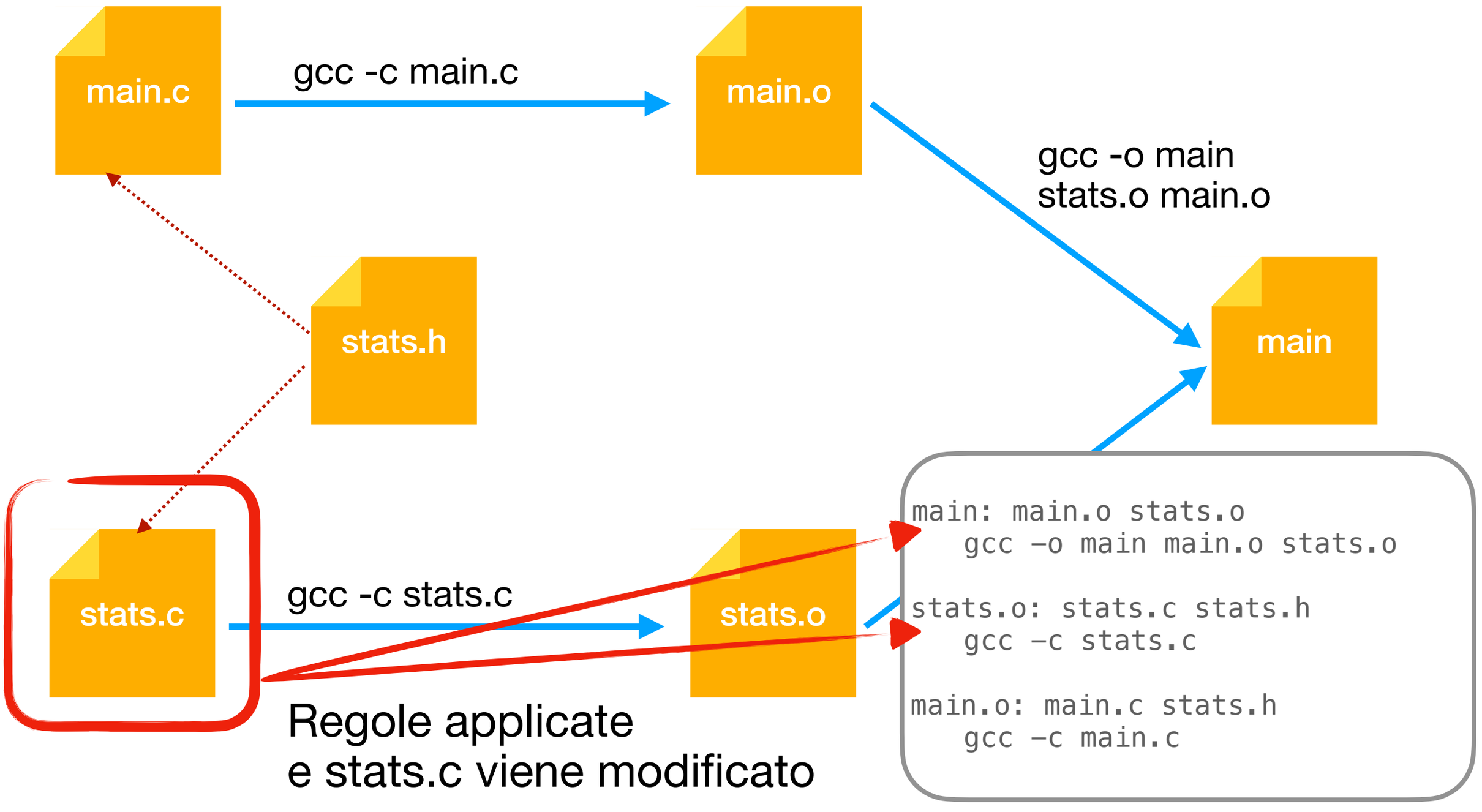
Regole del makefile

Quando sono invocate?

- Se i prerequisiti sono stati modificati…
- …allora l'obiettivo deve essere ricostruito
- Per fare questo sono eseguiti i comandi indicati nella regola (posso essere anche vuoti)
- La verifica dei prerequisiti richiede una applicazione ricorsiva:
  - Se il prerequisito non esiste, cerca una regola per crearlo
  - Se il prerequisito esiste, verifica se deve essere ricostruito (applicando la procedura indicata qui)

Passi di compilazione

E cosa servirebbe rifare



Make

Invocazione

- Adesso che abbiamo un makefile possiamo invocare make:
  - make main
    chiediamo di costruire il target “main”
  - make -f nome_makefile main
    se vogliamo un nome specifico per il makefile
  - make
    senza una indicazione del target viene richiamato il target “all”

Make

Invocazione

- Cosa succede se richiamiamo “make” una seconda volta?
  - “make: Nothing to be done for `all’.”
    - Se non modifichiamo nulla non ha senso rifare tutta la compilazione e il linking.
- E se modifichiamo un solo file .c?
  - Non vengono eseguiti tutti i comandi ma solo quelli in cui i prerequisiti sono stati modificati (quindi ricreazione del file .o e linking)

Make clean

Pulizia

- Un target comune è “clean”
- Non corrisponde a nessun file, ma semplicemente esegue una serie di comandi di pulizia dei file intermedi
- Esempio per eliminare i file oggetto:
    - clean:
        rm *.o
- Dato che il file “clean” non viene creato il comando viene sempre eseguito
- Per evitare malfunzionamenti in caso esista un file “clean” usiamo la direttiva “.PHONY”, per dire di non considerare mai aggiornato il target

Makefile

Versione attuale

```
all: main

main: main.o stats.o
   gcc -o main main.o stats.o

stats.o: stats.c stats.h
   gcc -c stats.c

main.o: main.c stats.h
   gcc -c main.c

clean:
   rm *.o
```

Makefile

Possiamo fare dei target per generare librerie statiche e dinamiche?

E se volessimo mettere le librerie in una directory specifica?

E se volessimo cambiare le opzioni del compilatore?

Variabili

Nei makefile

- A volte vogliamo poter parametrizzare i comandi in un makefile e usare una variabile comune per tutti comandi
- Esempi:
  - Il nome del compilatore (gcc, gcc-13, clang, etc)
  - I flag di compilazione (-O3, -g, etc)
  - La directory in cui spostare i file (e.g., “lib”, “.”, etc)
- Per fare questo abbiamo a disposizione le variabili nei makefile

Variabili

Nei makefile

- Una variabile è definita come
  NOME = stringa

- Da quel momento in poi possiamo usare **${NOME}** nel makefile e verrà sempre sostituito da “stringa”

- Notate che è sempre una sostituzione del testo, non vi sono operazioni che vengono compiute

- Esempio (dichiarazione):
  CFLAGS = -O3

- Esempio (uso):
  gcc ${CFLAGS} -o main main.o …

Variabili

Modifica da linea di comando

- Un vantaggio delle variabili è che possono essere modificate direttamente dalla linea di comando di make

- Esempio
  make CC=gcc-13
  utilizzerà un compilatore differente

- Questo permette di avere un default facilmente modificabile senza toccare il makefile

- Utile per le opzioni di compilazione (e.g., ottimizzazione, debug, profiling, etc).

Variabili automatiche

Nei makefile

- Alcune variabili esistono sempre e sono definite in automatico:
  - **$@**
    il target
  - **$^**
    la lista dei prerequisiti
  - **$<**
    Il primo prerequisito
- Questo ci permette di evitare di riscrivere alcuni dei nomi già presenti in target e prerequisiti

Regole template

One rule to rule them all

- In molti casi ci sono regole simili che cambiano solo per il nome del file considerato (e.g., generazione del file .o dal file .c)
- Per fare questo ci vengono incontro delle regole template in cui possiamo usare “**%**” per indicare “qualsiasi sequenza di caratteri” nel target e poi usare “**%**” per indicare la stessa sequenza nei prerequisiti
- Esempio:
  %.o: %.c
  gcc -c $<
  - Significato: per generare un file “nome.o” ti serve “nome.c”

Regole template

One rule to rule them all

- Quando usiamo le regole template abbiamo una variabile automatica aggiuntiva:
  - **$***
    Fa matching della parte template (ovvero %) per poterla usare nei comandi da eseguire
  - Questa ha senso solo quando abbiamo una regola template

Wildcards

Tutto ciò che fa matching di un pattern

- A volte ci serve una variabile come “tutti i file.c” o “tutti i file.h”
- Per questo possiamo usare le wildcard
- ${wildcard *.c}
  produrrà una lista di tutti i file .c presenti nella directory corrente
- Alcune versioni di make, come GNU make, hanno la possibilità di fare maggiori manipolazioni aggiungendo prefissi, suffissi, o sostituendo sottostringhe:
  https://www.gnu.org/software/make/manual/html_node/Functions.html

Esecuzione condizionale

Regole condizionali

- A volte è utile cambiare cosa è definito nel makefile in base a certe condizioni (e.g., l'ambiente di sviluppo usato)
- Abbiamo due possibilità:
  - Testare se due stringhe sono identiche
  - Testare se una variabile è definita
- La sintassi è lievemente differente

Esecuzione condizionale

Regole condizionali

ifeq “stringa1” “stringa2”

    # per verificare se una variabile è definita si utilizza ifdef

    # caso di uguaglianza

else

    # caso di non uguaglianza

endif

A cosa fare attenzione

Make è comunque degli anni ‘70

- Ogni riga dei comandi da eseguire all’interno di una regola viene eseguita in modo indipendente (i.e., non possiamo spezzare su più righe)
- È essenziale che ci siano dei tab per indentare, niente spazi.
- Utilizzando l’opzione “-p” è possibile stampare che comandi verranno eseguiti e che variabili e regole sono definite
- Aggiungendo @ prima di un comando non viene stampato il comando che viene eseguito

Altri sistemi di build

Cmake, bazel, etc

- Negli anni sono stati definiti molti sistemi di “build automation”, con più o meno successo. Alcuni che potete incontrare sono:
  - **CMake**. Non fa direttamente il build ma costruisce gli script/file (anche make) per fare il build con altri tool
  - **Autotools** (automake e autoconf). Tool per generare i makefile in maniera portabile
  - **Bazel**. Scritto in Java per effettuare build di progetti Java, C, C++, etc. Deriva da Blaze, il sistema di build interno di Google
  - **Gradle**. Della Apache foundation, più comune su progetti in Java (in quel caso trovate anche Ant e Maven)