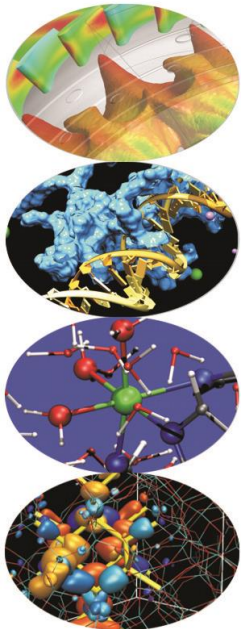
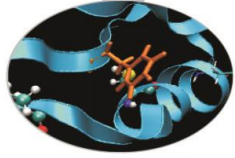


# Compilazione e Makefile



# Indice

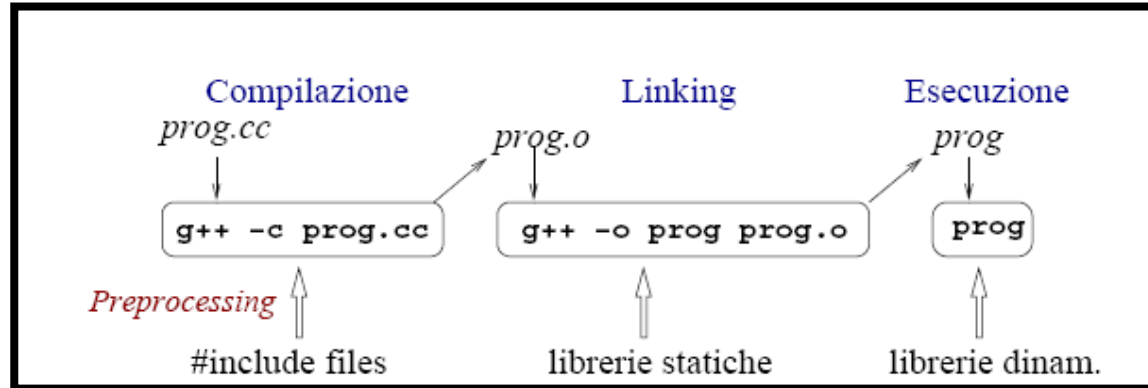


- **Il preprocessore**
- **Le istruzioni per il preprocessore**
- **Le MACRO**
- **L'ambiente linux: alcuni comandi**
- **Editor ed il compilatore gcc**
- **I makefile**
- **Il comando make**
- **I flag**



# Il preprocessore

Il processo di compilazione C/C++ comprende diverse fasi:



- La prima fase viene detta di istruzioni al pre-processor e prevede ad esempio l'inclusione di librerie che contengono le definizioni di funzioni usate nel programma.
- Le direttive al pre-processor sono numerose, e si distinguono nel codice per la presenza del simbolo # all'inizio della dichiarazione; noi considereremo solo le principali:
- **#include, #define, #if, #ifndef, #ifdef, #else, #elif, #endif, #undef ; insieme con gli operatori: #, ##**



# #include

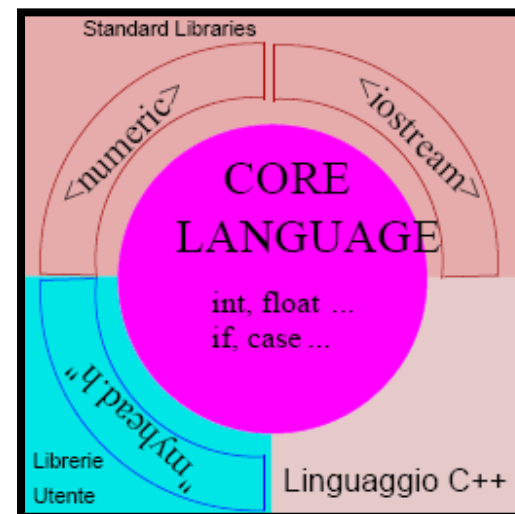
- Questa direttiva forza il compilatore a leggere e compilare un altro codice sorgente
- Ne esistono due forme a livello di sintassi:  
`#include"nome-file"`  
`#include<nome-file>`

Distinti convenzionalmente in base al fatto che il file che si vuole includere sia parte dell'installazione standard del compilatore (<>) oppure che sia stato creato dall'utente ("").

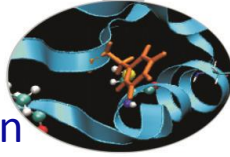
Esempio:

```
#include<stdio.h>
```

```
#include"mia_lib.h"
```



# #define



- Questa direttiva è utilizzata per effettuare macro sostituzioni di parti di codice con altre.
- La forma generale è data da:

***#define nome-macro sequenza-di-caratteri***

*/\*ad ogni occorrenza nel codice del nome-macro viene sostituita la sequenza-di-caratteri\*/*

- La macro può avere argomenti e in questo caso la macro assomiglia molto ad una funzione.

## Esempio:

```
#include<stdio.h>
```

```
#define MYMACRO(a) ((a)<0 ? -(a) : (a))
```

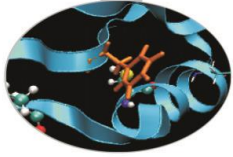
```
int main() {
```

```
printf("valore assoluto di -8: %d \n",MYMACRO(-8));
```

```
return 0;
```

```
}
```

**NOTA:** L'introduzione dell'istruzione "inline" e del qualificatore "const" a partire dal C99 ha di fatto permesso di soppiantare l'uso delle macro definite tramite la direttiva #define tipiche dello standard C89.



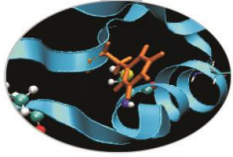
# #if #ifdef #ifndef #else #elif #endif

- Questo insieme di direttive è utilizzata per imporre al compilatore delle compilazioni selettive di varie porzioni di codice.
- L'idea generale è la seguente: se l'espressione a valle del controllo tramite le istruzioni **#if**, **#ifdef** o **#ifndef** è vera allora il codice compreso tra quel punto e la direttiva di termine (**#endif**) verrà compilato altrimenti verrà saltato.
- L'uso della direttiva **#else** è intuitivo.

Esempio:

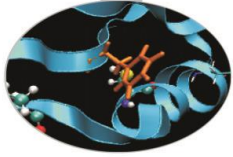
```
#ifndef nome-macro
#define nome-macro
//...
#else ...
#endif
//
```

# #if #ifdef #ifndef #else #elif #endif



- Tipico uso ne viene fatto nella scrittura di proprie librerie per evitare di ridefinire le stesse variabili qualora la libreria venisse richiamata da più file nello stesso programma.

```
//nel file header.h  
#ifndef header_h_  
#define header_h_  
    //codice  
#endif
```



# #undef

- Questa direttiva è utilizzata per effettuare la rimozione di macro definite precedentemente
- La forma generale è data da:

*#undef nome-macro*

## Esempio:

```
#define MYLEN 1000
```

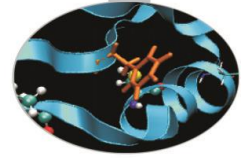
```
#define MYWIDTH 1000
```

```
char array[MYLEN][MYWIDTH];
```

```
#undef MYLEN
```

```
/* la macro MYLEN non esiste più, mentre MYWIDTH  
   continua ad essere definita */
```





# Gli operatori # e ##

- Questi operatori vengono utilizzati all'interno di una #define macro.
- L'operatore # posizionato in una #define macro impone che l'argomento che lo segue sia trasformato in una stringa.

```
#define WARN_IF(EXP) \  
do { \  
    if (EXP) printf ("Warning: " #EXP " \n"); \  
} while (0)
```

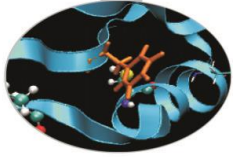
L'occorenza:

```
WARN_IF (x == 0);
```

Genera la seguente sostituzione nel sorgente :

```
do { \  
    if (x == 0) printf ("Warning: " "x == 0" "\n"); \  
} while (0);
```

# # e ##



- L'operatore ## è usato per concatenare, in una #define macro, due argomenti:

```
#include <stdio.h>
#define concatena(a,b) a ## b
int main() {
double hk=12.0;
printf(" %f\n",concatena(h,k));
int ab=200;
printf(" %d\n",concatena(a,b));
return 0; }
```

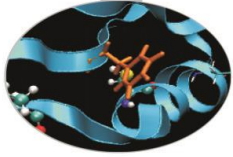
**/\*a livello del pre-processore si ha la sostituzione**

```
printf(" %f\n",concatena(h,k));
```

**in**

```
printf(" %f\n",hk);
```

```
*/
```



# Compilazione ed esecuzione

- Per compilare semplici programmi, costituiti da pochi files sorgenti, in generale si utilizza direttamente il compilatore nel modo seguente:

- Generazione eseguibile dalla riga di comando della shell di lavoro tramite i comandi:

```
user@linux>gcc file.c
```

oppure

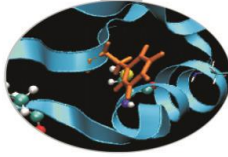
```
user@linux>gcc -o eseguibile.exe file.c
```

- Esecuzione codice compilato:

```
user@linux>./a.out
```

oppure

```
user@linux>./eseguibile.exe
```



# Compilazione ed esecuzione

Supponendo di disporre di un programma costituito da:

- mioprogram.h file dichiarativo
- mioprogram.c file di definizioni
- main.c file di utilizzo

le istruzioni per la compilazione del sorgente e la creazione dell'eseguibile sono:

## 1- Compilazione dei sorgenti:

```
gcc -c mioprogram.c
```

```
gcc -c main.c
```

che produce come output i file oggetto *mioprogram.o* e *main.o*

## 2- Linking:

```
gcc -o mioexe mioprogram.o main.o
```

Oppure unendo compilazione e linking:

```
gcc -o mioexe mioprogram.c main.c
```

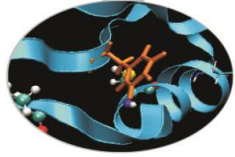
**Le dipendenze tra i file e le specifiche istruzioni di compilazioni vengono solitamente gestite attraverso un Makefile.**



# Makefile

- La compilazione di un codice sorgente C è una operazione facilmente automatizzabile perché è ripetitiva.
- All'interno di sistemi operativi UNIX/Linux e talvolta anche DOS, l'uso del comando *make* permette un controllo delle varie fasi di compilazione semplice ed automatico soprattutto nel caso di codici scritti su più files sorgenti.
- Un makefile contiene **linee di dipendenza** (ovvero linee che sintetizzano come un object file dipende da un file sorgente o da file header) e **linee di azione** (quali tipi di opzioni di compilazioni vanno usate per un certo file)
- Una linea di dipendenza e la linea d'azione relativa insieme costituiscono una **regola**

# Makefile - Esempio



Di seguito si ipotizza di avere un programma costituito da 3 files:

#-main.c

#-func.c

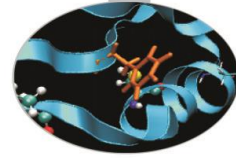
#-func.h (incluso da entrambi i file .c)

Tutti i 3 files sono presenti nella cartella di lavoro corrente.

```
mioexe: main.o func.o #linea di dipendenza
        gcc -O3 -o mioexe main.o func.o #linea di azione

main.o: main.c func.h
        gcc -O3 -c main.c

func.o: func.c func.h
        gcc -O3 -c func.c
```



# Makefile – Regole sintattiche

- Il flag **-c** nella compilazione ha l'effetto di creare soltanto i file oggetto. Ottenuti i file oggetto, questi si collegano assieme per generare l'eseguibile "mioexe" (l'obiettivo principale).
- A parte le righe precedute da **#** che sono righe di commento, ogni riga del file può assumere due formati:
  - *linee di dipendenza*. La riga inizia specificando l'obiettivo, subito dopo a seguire il carattere **:**, si trovano i prerequisiti. Per esempio l'obiettivo finale (il primo specificato)
  - *linee di azione*. Ammesso che i prerequisiti siano soddisfatti, nelle righe sono specificati i comandi per il raggiungimento dell'obiettivo. Nel caso presentato ogni obiettivo è seguito da una sola riga di comando, ma potrebbero essere più di una. Tutte le righe di comando cominciano con il **Tab**. È la presenza di questo carattere che identifica la riga come riga di comando.



# Makefile – Regole sintattiche

- Fra due regole deve esserci almeno **una linea vuota**.
- Il file deve terminare con un **NEWLINE**.
- **Esecuzione del Makefile :**
  - Il comando per lanciare l'esecuzione del **Makefile** è: **make**
  - Il comando **make** di default si aspetta che nella posizione in cui viene lanciato vi sia un file chiamato **Makefile** o **makefile**.
  - Nel caso in cui il Makefile abbia un nome diverso la sintassi è la seguente:
    - **make -f nome\_makefile**
- **E' possibile eseguire una singola regola nel makefile con la sintassi:**
  - **make nome\_regola**
  - **make -f nome\_makefile nome\_regola**





# Makefile - Variabili

- Può essere comodo usare variabili nella scrittura di makefile qualora si voglia avere diverse opzioni di compilazione da utilizzare in istanti diversi
- Una volta dichiarate possono essere utilizzate tramite il simbolo: `$( )`

```
#uso due variabili per avere due compilatori diversi
```

```
CXX=g++
```

```
CC=gcc
```

```
#oppure per passare opzioni (flags) di compilazione
```

```
CFLG= -O3 -Wall -c
```

```
mioexe: main.o func.o #linea di dipendenza
```

```
$(CC) -O3 -o mioexe main.o func.o #linea di azione
```

```
main.o: main.c func.h
```

```
$(CC) $(CFLG) main.c
```

```
func.o: func.c func.h
```

```
$(CC) $(CFLG) func.c
```

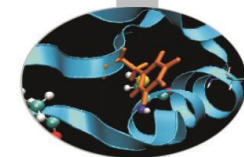
```
clean:
```

```
rm *.o
```



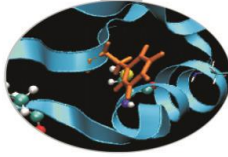
# Ambienti Linux e codici C

- Il pc a vostra disposizione fa uso di linux
- Aprendo una shell di lavoro si può creare ed accedere ad uno spazio di lavoro personale tramite i comandi:
- `user@linux> mkdir "nome della cartella"`
- `user@linux> cd "nome della cartella"`
- in questo spazio inizialmente non vi saranno file:
- `user@linux> ls -la [-tr -h]`



# Ambienti Linux e codici C

<u>Comando</u>	<u>Azione</u>
<code>ls -la [-tr -h]</code>	listato ordinato per data dei file contenuti nella cartella di lavoro
<code>mkdir "myworkdir"</code>	creazione di una cartella di lavoro
<code>cd /path/myworkdir</code>	cambio di cartella di lavoro dalla posizione corrente a "myworkdir"
<code>rm "nome_file"</code>	rimozione del file "nome_file"



# Editing

- Gli editor di testo disponibili in ambiente Linux sono:
  - **nedit** (più semplice ed intuitivo)
  - **emacs** (decisamente più potente e complesso)
  - **Code::Blocks** (ambiente di sviluppo comprensivo di GUI)
- In ambiente Windows
  - **Dev-C++** (<http://www.bloodshed.net/dev/>)
  - **Code::Blocks** (<http://www.codeblocks.org/>)
- Questi editor riconoscono la sintassi C/C++ e agevolano la lettura del codice grazie all'uso dei colori.