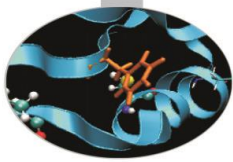
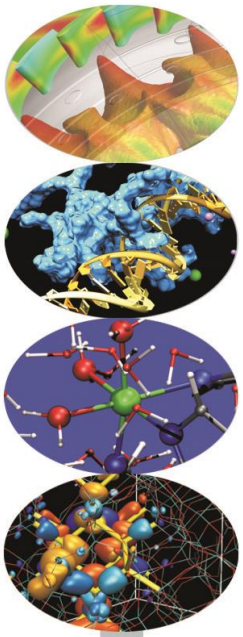
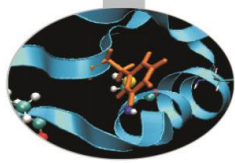


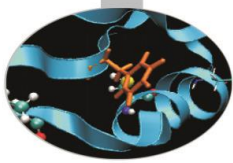
Compilazione e Makefile





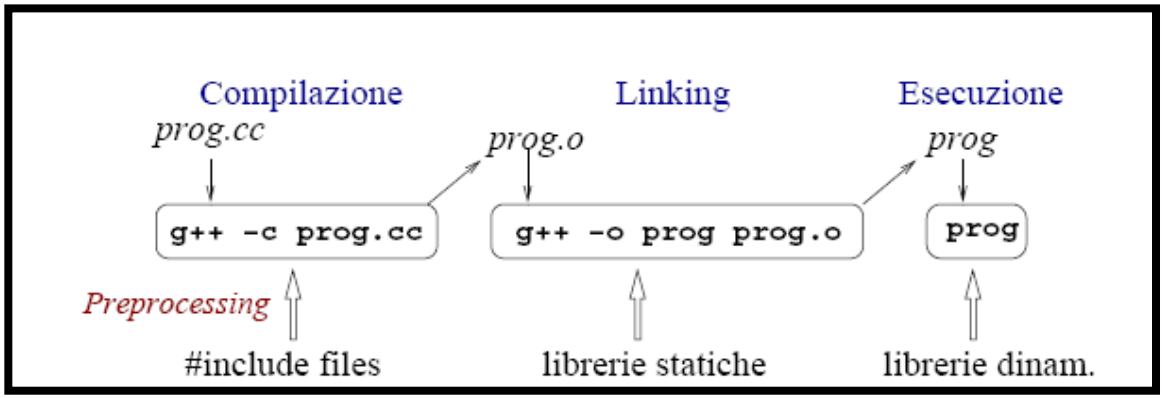
Indice

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

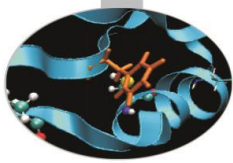


Il preprocessore

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



- La prima fase viene detta di istruzioni al pre-processore e prevede ad esempio l'inclusione di librerie che contengono le definizioni di funzioni usate nel programma.
- Le direttive al pre-processore sono numerose, e si distinguono nel codice per la presenza del simbolo # all'inizio della dichiarazione; noi consideremo 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:

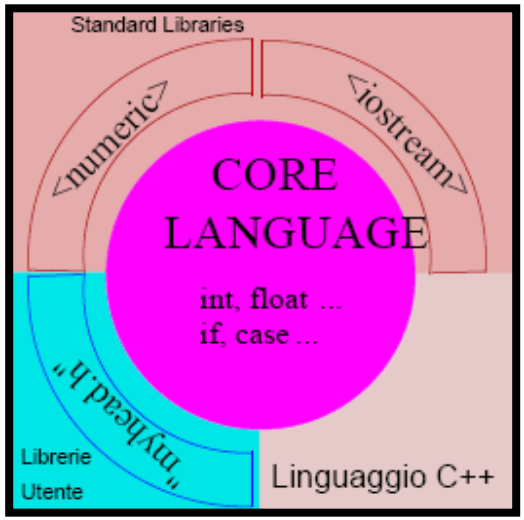
  #include”nome-file”

  #include<nome-file>

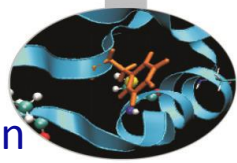
Distinti convenzionalmente in base la fatto che il file che si vuole includere sia parte dell’installazione standard del compilatore (<>) oppure che sia stato creato dall’utente (“”).

Esempio:

```
#include<iostream>

#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

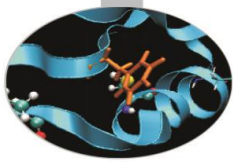
  /*ad ogni occorrenza nel codice del nome-macro viene sostituita la sequenza-di-caratteri*/
- La macro può avere argomenti e in questo caso la macro assomiglia molto ad una funzione.

Esempio:

```
#include<iostream>
#define MYMACRO(a) ((a)<0 ? -(a) : (a))
using namespace std;

int main(){
cout << "valore assoluto di -8: "<< MYMACRO(-8)<< endl;
return 0;
}
```

NOTA: in C++ l'uso delle funzioni inline e dei valori const ha di fatto permesso di soppiantare l'uso delle macro definite tramite la direttiva #define tipiche del C.

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

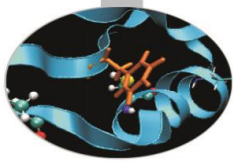


- Questo insieme di direttive è utilizzata per imporre al compilatore delle compilazioni selettive di varie porzioni di codice.
- L'idea generale è la seguente: se l'espressione a valle del controllo tramite le istruzioni **#if, #ifdef o #ifndef** è vera allora il codice compreso tra quel punto e la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

Esempio:

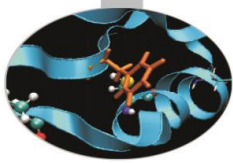
```
#ifndef nome-macro
#define nome-macro
//…
#else …
#endif
//
```

#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
```



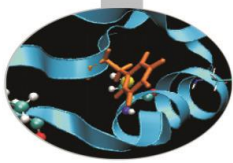
#undef

- 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 */
```

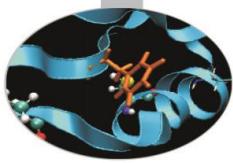


Gli operatori # e

- 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:

```
#include <iostream>
#define makestr(s)  # s
using namespace std;

int main(){
cout << makestr(questa è sintassi C++) << endl;
return 0; }
/*a livello del pre-processore si ha la sostituzione

cout << makestr(questa è sintassi C++); in
cout << "questa è sintassi C++"; */
```

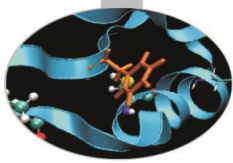
e



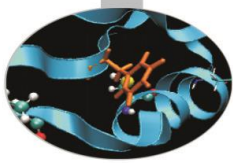
- L'operatore ## è usato per concatenare, in una #define macro, due argomenti:

```
#include <iostream>
#define concatena(a,b)  a ## b
using namespace std;

int main() {

double  hk=0.0;
cout << concatena(h,k) << endl;
return 0; }
/*a livello del pre-processore si ha la sostituzione

cout << concatena(h,k) << endl; in
cout << hk << endl;
*/
```

Ambienti Linux e codici C



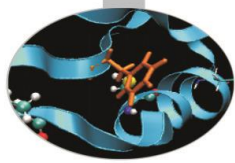
- Il pc a vostra disposizione fa uso di linux
- Aprendo una shell di lavoro si può creare ed accedere ad uno spazio di lavoro personale tramite i comandi:
- **user@linux> mkdir** *"nome della cartella"*
- **user@linux> cd** *"nome della cartella"*
- in questo spazio inizialmente non vi saranno file: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



Ambienti Linux e codici C

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

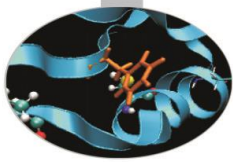
Editing



- Gli editor di testo disponibili sono:

  -**nedit** (più semplice ed intuitivo)

  -**emacs** (decisamente più potente e complesso)
- Questi editor riconoscono la sintassi C/C++ e agevolano la lettura del codice grazie all'uso dei colori.
- In ambiente Windows un ottimo compilatore free di tipo visual è il Dev-C++(http://www.bloodshed.net/dev/). Non verrà usato durante questo corso.



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>g++ “file.cpp”

    oppure

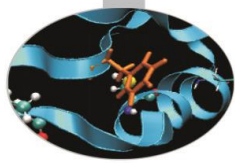
    user@linux>g++ “file.cpp” –o “eseguibile.exe”

  - Esecuzione codice compilato:

    user@linux>./a.out

    oppure

    user@linux>./eseguibile.exe



Compilazione ed esecuzione

Supponendo di disporre di di un programma costituito da:

- mioprog.h file dichiarativo
- mioprog.cpp file di definizioni
- main.cpp file di utilizzo

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

1- Compilazione dei sorgenti:

g++ -c mioprog.cpp

g++ -c main.cpp

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

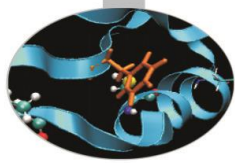
2- Linking:

g++ -o mioexe mioprog.o main.o

Oppure unendo compilazione e linking:

g++ -o mioexe mioprog.cpp main.cpp

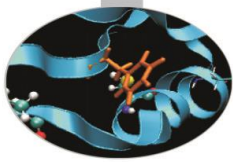
Le dipendenze tra i file e le specifiche istruzioni di compilazioni vengono solitamente gestite attraverso un Makefile.



Makefile

- La compilazione di un codice sorgente C/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**

Esempio



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

#-main.cpp

#-mioprog.cpp

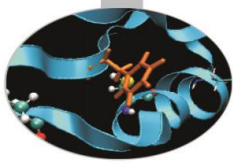
#-mioprog.h (incluso da entrambi i file .cpp)

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

```
mio: main.o mioprog.o  #linea di dipendenza
                g++ -o mio main.o mioprog.o #linea di azione
main.o: main.cc mioprog.h
                cc -c –O main.cpp
mioprog.o: mioprog.cpp mioprog.h
                cc -c –O mioprog.cpp
```

a questo punto tramite il comando make viene eseguito il nostro makefile mio

```
user@linux> make mio
```



Flag di compilazione basilari

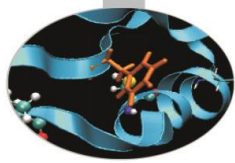
- Il flag **-c** nella compilazione ha l'effetto di creare soltanto i file oggetto. Ottenuti i file oggetto, questi si collegano assieme per generare l'oggetto mioprog (l'obiettivo principale).

- 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.

Uso di 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
CC=g++
CC1=gcc
#oppure per passare opzioni (flags) di compilazione
CFLG= -c -Wall

mio: main.o mioprog.o
   $(CC) $(CFLG) mio.cpp
main: main.o mioprog.o
   $(CC) -o esegui main.o mioprog.o
clean:
   rm -rf *.o
```