Git, Make, Shell scripting

Laboratorio di Metodi Computazionali e Statistici (2023/2024)

Fabrizio Parodi

Dipartimento di Fisica

Esercizio su classi astratte e polimorfismo run-time

- Supponiamo di voler creare un progetto OO per gestire le serie ad un indice (l'estensione a più indici non presenta difficoltà concettuali) che sia in grado di gestire, ad esempio, le seguenti serie:

$$\sum_{n=0}^{N} \alpha^n \qquad \text{Serie geometrica (convergente con } \alpha < 1)$$

$$\sum_{n=1}^{N} 1/n^{\alpha} \qquad \text{Serie armonica (convergente con } \alpha > 1)$$

$$\sum_{n=-N}^{N} (-1)^n/|n| \qquad \text{Serie di Madelung}$$

  (l'ultima si ritrova spesso nell'energia di legame di sali biatomici, nel caso 1D corrisponde alla catena di ioni)

- Richieste:
  - vogliamo definire un classe per ogni serie
  - studiare la convergenza in funzione di n, scegliendo run-time la serie da trattare

Analisi del problema

- L'oggetto serie è composto essenzialmente da un calcolo e quindi molto simile ad una funzione
- I dati sono l'opzione di somma (da 0 a n o da -n a n) e lo stato (ultima somma calcolata e ultimo valore di n)
- La serie può avere uno o più parametri o non averli

Lo stato (ultima somma calcolata insieme all'ultimo valore di n) permette di sommare in maniera efficiente evitando di ri-calcolare termini già sommati. L'implementazione di questa ottimizzazione non è però trattata (viene lasciata come esercizio opzionale).

Series.h

```
1  class Series{
2    public:
3     Series(bool neg=false):m_neg{neg},m_n{0},m_sum{0}{}
4     virtual double Sum(int n);
5     virtual double Term(int i)=0;
6    protected:
7     bool   m_neg;
8     int    m_n;
9     double m_sum;
10 };
11 class Madelung: public Series{
12  public:
13   Madelung():Series(true){}
14   double Term(int i);
15 };
16 class Armonic: public Series{
17  public:
18   Armonic(double par):m_par{par},Series(false){}
19   double Term(int i);
20  private:
21   double m_par;
22 };
23 class Geom: public Series{
24  public:
25   Geom(double par):m_par{par},Series(false){}
26   double Term(int i);
27  private:
28    double m_par;
29 };
```

Series.cpp

```
1  #include <cmath>
2  #include "Series.h"
3
4  double Series::Sum(int n){
5    m_sum=0;
6    if (m_neg){
7      for (int i=-n;i<=n;i++)
8        m_sum += Term(i);
9    } else {
10     for (int i=0;i<=n;i++)
11       m_sum += Term(i);
12   }
13   m_n = n;
14   return m_sum;
15 }
16
17 double Geom::Term(int i){
18   return pow(m_par,i);
19 }
20
21 double Madelung::Term(int i){
22   if (i!=0){
23     return pow(-1,i)/abs(i);
24   } else {
25     return 0;
26   }
27 }
28
29 double Armonic::Term(int i){
30   if (i!=0){
31     return 1/pow(i,m_par);
32   } else {
33     return 0;
34   }
35 }
```

Main

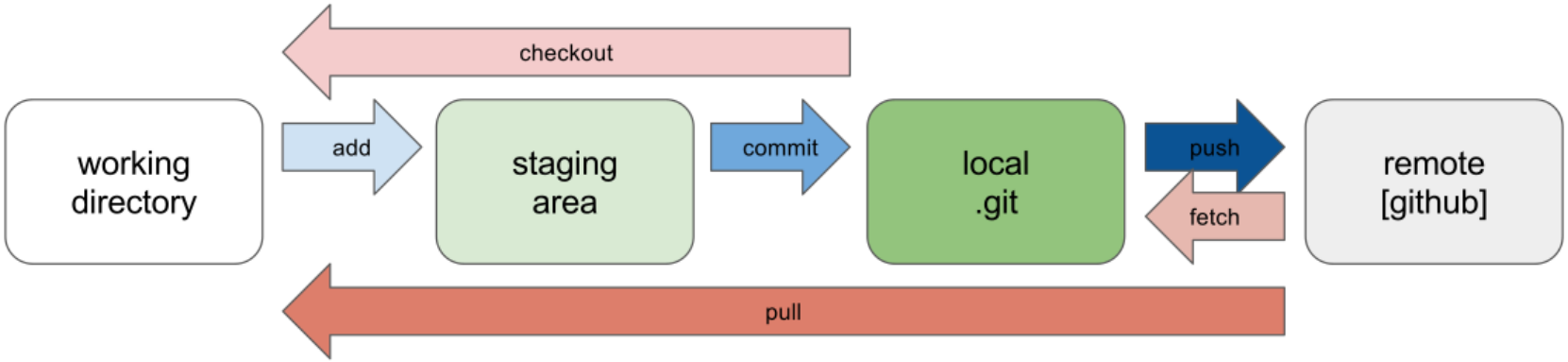
```
1  #include <cmath>
2  #include <iostream>
3  #include "TGraph.h"
4  #include "TApplication.h"
5  #include "Series.h"
6  int main(){
7    std::string nome;
8    std::cout << "Inserisci il tipo di serie" << std::endl;
9    std::cin >> nome;
10   Series *p;
11   if (nome=="geo"){
12     p = new Geom(0.5);
13   } else if (nome=="mad") {
14     p = new Madelung();
15   } else if (nome=="arm") {
16     p = new Armonic(2);
17   }
18   TApplication app("app",NULL,0);
19   TGraph gr;
20   for (int i=1;i<100;i++)
21     gr.SetPoint(i,i,p->Sum(i));
22   gr.SetMarkerStyle(20);
23   gr.Draw("AC");
24   app.Run(true);
25   return 0;
26 }
```

Gestione progetto OO

- La creazione di un progetto OO prevede la creazione di diversi files, con possibilità di editing da diversi autori. Questo richiede:
  - Strumenti per la condivisione di progetti/files.
  - Strumenti per compilazione di un grosso numero di files.

Git

Git è un software di controllo di versione distribuito utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005.
Tipico workflow di Git:



Github: creazione di un progetto

Github How-to:

- Iscriversi a GitHub
- Creare un progetto
- Inserire il codice iniziale nel progetto con i comandi:

```
git init
git add *
git commit -m "first commit"
git remote add origin https://github.com/MyAccount/NomeProgetto.git
git push origin master
```

Uso minimale di git

In realtà nel corso lo useremo soprtatutto per scaricare i files di partenza per le varie esercitazioni o i files mostrati a lezione.

Per copiare i files della prima esercitazione, ad esempio:

```
git clone  https://github.com/fabrizio-parodi/LabMCS-EsI.git  EsI
```

Ovviamente potete farne un utilizzo più massiccio per creare e archiviare vostri progetti.

Comando make

- Immaginate un progetto molto complesso, formato da molti moduli, e di voler cambiare solo una piccola parte di codice e di voler poi testare il programma. Sarebbe conveniente compilare solo i moduli appena modificati e poi effettuare il link con la parte di codice rimasta.
- Operazione impossibile (o molto noiosa) da fare a mano. Il comando make fa questo per voi !

Makefile e target

- Per funzionare make ha bisogno di un file chiamato Makefile in cui siano descritte le relazioni fra i vostri files ed i comandi per aggiornarli.
- Quando il make viene invocato esegue le istruzioni contenute nel Makefile.
- Una idea base del make è il concetto di target. Il primo target in assoluto e' il Makefile stesso. Se si lancia il make senza aver preparato un Makefile si ottiene il seguente risultato

```
make:*** No targets specified and no makefile found. Stop
```

- target in generale

```
target: dependencies
[tab] system command
```

Makefile e target

- Esempio:

```
all: hello
hello: hello.cpp lib.o
    g++ -o hello hello.cpp lib.o
lib.o: lib.cpp
    g++ -c lib.cpp
clean:
    rm -rf *.o
    rm hello
```

- Target specifici:
  - `install`, viene utilizzato per installare i file di un progetto e può comprendere la creazione di nuove directory e la assegnazione di diritti di accesso ai file.
  - `clean`, viene utilizzato per rimuovere dal sistema i file oggetto (`*.o`), i file core, e altri file tempornei creati in fase di compilazione
  - `all`, di solito utilizzato per richiamare altri target (o quelli default) con lo scopo di costruire l'intero progetto.

Macro e variabili d'ambiente

- È possibile definire macro che permettono di gestire una volta per tutte le opzioni di compilazione e link

```
1  CC           = g++
2  CFLAGS       = --std=c++11 -g -Wall
3  CFLAGSROOT = `root-config --cflags`
4  LIBSROOT     = `root-config --glibs`
5
6  all: Main
7
8  Vector3.o: Vector3.cpp
9      $(CC) $(CFLAGS) -c Vector3.cpp        $(CFLAGSROOT)
10 Main: Main.cpp Vector3.o
11     $(CC) $(CFLAGS) -o Main Main.cpp Vector3.o $(CFLAGSROOT) $(
           LIBSROOT)
12 clean:
13     rm *.o
```

Shell

- La shell o interprete dei comandi è un programma che serve ad impartire comandi al sistema operativo.
- Ci sono numerose scelte: sh, bash, ksh, zsh, csh, tcsh...
  - `bash` è diventata il default di Linux, di Mac OS X
  - anche Windows ha una sua shell (Windows PowerShell), ma qui (e nel resto del corso) preferiamo enfatizzare la portabilità. Inoltre recentemente (Windows 10) la bash è disponibile anche su Windows.
- I comandi sono righe di testo inserite al prompt, che vengono interpretate ed eseguite quando si preme il tasto di invio.
- L'immissione di un comando interpretabile fa partire un sotto-processo in cui quel comando viene eseguito.
- Numerose funzionalità:
  - utilizzo di variabili, wild-character, espressioni
  - auto-completion (tasto TAB) per i nomi di comandi, file, variabili
  - ridirezione di input e output
  - gestione dei processi e del flusso di esecuzione
  - history dei comandi (usando il cursore)

Shell scripting

- I comandi di shell sono estremamenti potenti e permettono di automatizzare molte operazioni per le quali, altrimenti sarebbe necessario, scrivere complessi programmi
- Unico svantaggio: sono disponibili sono su Linux, Unix e Mac OsX (che è di fatto Unix).
- Al primo anno abbiamo visto i comandi base (di cui faremo un breve riassunto) nel seguito analizziamo alcuni comandi più avanzati...
- L'opportunità di applicazione consiste nell'estrazione dei dati rilevanti (massa, posizione e velocità iniziali) sui corpi del sistema solare da un testo più complesso (effemeridi.h).

Comandi di shell

Comando	**Descrizione**
cd $< dir >$	cambia la directory di lavoro in $< dir >$
pwd $< dir >$	comunica quale sia la $< dir >$ corrente (print working directory)
mkdir $< dir >$	crea la directory di nome $< dir >$
du $< dir >$	mostra lo spazio su $< dir >$
du -h $< dir >$	mostra lo spazio su $< dir >$ in bytes
ls	mostra il contenuto della directory corrente
ls -a	mostra il contenuto della directory corrente compresi i files nascosti
ls -l	mostra il contenuto della directory corrente con altre informazioni
rm $< file >$	rimuove il file $< file >$
rmdir $< dir >$	rimuove la directory dir se è vuota)
rm -r $< dir >$	rimuove la directory $< dir >$ e tutto quanto in esso contenuta (comprese altre sotto $< dir >$)

Comandi di shell

Comando	**Descrizione**
touch < *file* >	crea il file di nome < *file* > o ne modicata la data di aggiornamento
cat < *file* >	mostra il contenuto di un < *file* >
more < *file* >	mostra il contenuto di un < *file* >
less < *file* >	mostra il contenuto di un < *file* >
head -n < *n* > < *file* >	mostra le prime < *n* > righe (10 per default 10) del < *file* >
tail -n < *n* > < *file* >	mostra le ultime < *n* > righe (10 per default 10) del < *file* >
cp < *file*1 > < *file*2 >	copia file1 in file2
mv < *file*1 > < *file*2 >	ridenomina file1 in file2
find < *dir* > -name < *file* >	cerca il file di nome < *file* > a partire dalla directory < *dir* >

WildCards e redirezione

- Alcuni caratteri fungono da "jolly" (wildcard)
  - * sostituisce qualsiasi seguenza di caratteri
  - ? sostituisce un singolo carattere
  - [] duplicano la string con le opzioni contenute in [] (nome[1 – 4] equivale a nome1 nome2 nom3 nom4)

È possibile ridirigere l'output di un comando su un file:

- $>$ reindirizza l'output di un comando verso un file. Il file, se presente, viene sovrascritto
- $\gg$ reindirizza l'output ma il file non viene sovrascritto, i nuovi dati vengono aggiunti in coda.
- $>$ & redirige sia l'output che l'eventuale errore.

Pipe

- La pipe "|" permette di reindirizzare l'output di un comando verso l'input di un altro
  - È uno degli elementi chiave della shell perché permette di concatenare diverse operazioni.

```
$ ls -lt | head -5
total 5168
-rw-r--r-- 1 fabrizio staff 24059 6 Oct 22:15 lez2.tex
-rw-r--r-- 1 fabrizio staff 2113  6 Oct 22:12 lez2.aux
-rw-r--r-- 1 fabrizio staff 50713 6 Oct 22:12 lez2.log
-rw-r--r-- 1 fabrizio staff 962   6 Oct 22:12 lez2.nav
```

Stampa solo i 5 file più recenti (4 in realtà)

grep

- **grep** (**g**lobally search for a **r**egular **e**xpression and **p**rint matching lines) è una sorta di filtro: manda in output solo le linee che passano il filtro, es:
  - Contengono (o non contengono) una certa stringa
  - Iniziano/finiscono con un certo carattere
  - Sintassi:

    ```
    grep opzioni pattern file
    ```

  - (Alcune) opzioni:
    - `-A num` stampa `num` linee dopo il pattern trovato
    - `-B num` stampa `num` linee prima del pattern trovato
    - `-c` stampa solo il numero di righe che contengono il pattern
    - `-v` seleziona le linee che non contengono il pattern (selezione inversa)

grep

Supponiamo di avere ad esempio il file (originale) delle effemeridi del pianeta solare (contenute in un complesso header file C: effemeridi.h)

```
$ cat effemeridi.h
//
// Solar system barycentric velocity and position
// state of Mercury, Venus, EMB, ..., Pluto, MOON, Sun
// in the order dx/dt, x, dy/dt, y, dz/dt, z for each object.
//
//

// Sun
-2.8369446340813151639e-007
4.5144118714356666407e-003
5.1811944086463255444e-006
7.2282841152065867346e-004
2.2306588340621263489e-006
2.4659100492567986271e-004
...
```

s

grep

Esempio

- voglio copiare su un'altro file i soli dati del Sole

```
$ grep Sun effemeridi.h
// state of Mercury, Venus, EMB, ..., Pluto, MOON, Sun
// Sun
```

```
$ grep "// Sun" effemeridi.h
// Sun
```

Prendo le 6 righe che seguono e le redirigo in un file

```
$ grep -A6 "// Sun" effemeridi.h | grep -v "// Sun" > fileSun
$ cat fileSun
-2.8369446340813151639e-007
4.5144118714356666407e-003
5.1811944086463255444e-006
7.2282841152065867346e-004
2.2306588340621263489e-006
2.4659100492567986271e-004
```

Script

- In informatica uno script è un programma
  - è scritto in un linguaggio che può essere interpretato (invece che compilato)
  - automatizza l'esecuzione di determinate procedure
- Nel caso della shell
  - Il linguaggio è quello dei comandi di shell e dei programmi disponibili sul sistema
  - L'interprete è la shell stessa
- Lo script permette di automatizzare procedure di shell
  - Invece di eseguire passo passo una serie di comandi da terminale È possibile salvare la sequenza delle operazioni da svolgere in un file di testo ed eseguirne il contenuto al bisogno

Script

Ad esempio prendo il comando precedente e lo metto nel file `extract.sh`

```
1 #!/bin/bash
2 grep -A6 "// Sun" effemeridi.h | grep -v "// Sun" > fileSun
```

L'opzione `-v` fa si che grep funzioni come anti-filtro.

- `#!/bin/bash` indica il tipo di `shell` usata
- Per poterlo usare come comando devo poterlo eseguire

  `chmod u+x extract.sh`

- `./extract.sh`

- Oppure

  `source extract.sh`

Come passare i parametri allo script ?

```
./nomescript.sh str1 str2 ... strn
```

- $* ritorna una singola stringa con tutti i parametri separati da un spazio
- $@ ritorna la sequenza di stringhe corrispondenti ai parametri
- $1, $2, ...$*n* permette di accedere ai parametri uno alla volta
- $# ritorna il numero di argomenti

Script

Come posso passare dei parametri allo script?

- i parametri passati in linea dopo il nome dello script sono visti, all'interno, come 1,2..

```
1 #!/bin/bash
2 grep -A6 "// $1" effemeridi.h | grep -v "// $1" > file$1
```

  Così posso estrarre un file per ogni corpo celeste

- `./extract.sh Sun`

  Il file estratto si chiamerà (ancora) `fileSun`.

if

```
if [ <some test> ]
then
  <commands>
elif [ <some test> ]
then
  <different commands>
else
  <other commands>
fi
```

Esempio (num.sh):

```
if [ $1 -lt 100 ]
then
  if [[ $1%2 -eq 0 ]]   # doppie parentesi per espressioni
  then
    echo "Il numero (<100) e' pari"
  else
    echo "Il numero (<100) e' dispari"
  fi
else
  echo "Il numero  e' >100"
fi
```

```
./num.sh 10
Il numero (<100) e' pari
```

Cicli

```
for varname in list
do
 command1
 command2
 ..
done
```

```
for (( expr1; expr2; expr3 ))
do
 command1
 command2
 ..
done
```

Esempio:

```
#!/bin/bash
for planet in Sun Earth Mars
do
 grep -A6 "// $planet" effemeridi.h | grep -v "// $planet" > file$planet
done
```

Awk

- Awk è un comando per la manipolazione di testi
  - Vero proprio linguaggio (interpretato)
  - Solitamente utilizzati in "pipe" sull'output di altri comandi
  - Vedremo solo alcune delle loro potenzialità
- Awk è orientato alla manipolazione di testi formattati in colonne
  - Come tabelle
  - Awk permette di effettuare operazioni tra le colonne (come in un foglio di calcolo)
  - E' solitamente usato come filtro

Awk

Variabili predefinite:

- \$1, \$2, ... contengono il 1^o, 2^o,... campo della linea corrente;
- \$0 contiene l'intera linea corrente;
- *NF* contiene il numero dei campi della linea corrente;
- *NR* contiene il numero di linea corrente;

Il file viene maneggiato da awk viene visto come una tabella. Ovviamente al posto di un file può essere passato l'output di un altro comando.

Awk: esempio

```
ls -l | awk '{print $0}'
```

Stampa tutto l'output (senza filtro)

```
ls -l | awk '{print $1}'
```

Stampa solo la prima colonna (quella degli accessi)

```
ls -l | awk '{print $1" "$9}'
```

Stampa solo la prima colonna (accessi) e l'ultima (file)

Shell scripting

- Abbiamo fatto poco più che citare che esiste
- Estremamente potente e flessibile
- Molto materiale in rete
- Se vi capita di lavorare su Linux/Unix/Mac OsX sarebbe peccato privarsene.