Appunti Lezione 03

Lorenzo Visca

1 Interpretazione e compilazione di una macro

L'istruzione presentata per utilizzare una macro in ROOT finora è stata:

```
root[ ]  .L macro.C
```

In questo modo la macro viene interpretata utilizzando l'interprete di ROOT (Cling).
L'interpretazione delle macro è utile per programmi molto leggeri, perché non prevede una fase di compilazione; tuttavia per programmi più complessi l'interpretazione diventa lenta: è quindi consigliato compilare la macro, per cui la compilazione è inizialmente più lenta ma in compenso l'esecuzione è estremamente rapida.
NOTA: il compilatore è più "severo" dell'interprete, è quindi comunque consigliato compilare le macro in modo da individuare eventuali errori "soft" che l'interprete non segnala.
Per compilare una macro, le istruzioni sono:

```
root[ ]  .L macro.C+
```

Oppure:

```
root[ ]  .L macro.C++
```

La differenza tra `+` e `++` è che la prima opzione compila la macro solo se è stata modificata dopo l'ultima compilazione, mentre la seconda forza la ricompilazione della macro anche se non è stata modificata. Questa seconda opzione è utile ad esempio quando si fa un upgrade di ROOT, del compilatore o del sistema operativo.
Quando una macro viene compilata vengono generati alcuni file:

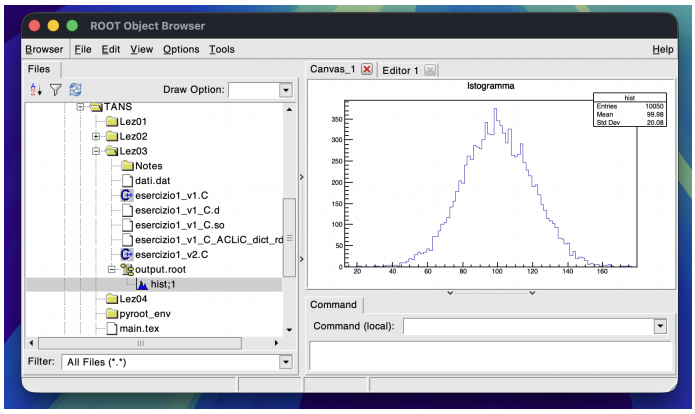
- un file `.d` che contiene le dipendenze della macro (ad esempio, se la macro include altre librerie);
- un file `.so` che contiene il codice compilato;
- un file `.pcm` che contiene altre informazioni di compilazione utilizzate dal sistema per velocizzare l'esecuzione.

2 Apertura di un file ROOT

Una volta creato un file ROOT (ad esempio contenente un istogramma come quello creato in `ioexample2.C`), è possibile accedervi in due modi:

- tramite il browser di ROOT, che fornisce un'interfaccia grafica per la navigazione dei file e la possibilità di effettuare varie azioni sul loro contenuto (per accedere all'istogramma fare doppio click sul nome del TFile che lo contiene);

  ```
  root[ ]  new TBrowser()
  ```



- richiamando il file all'interno di un'altra macro:

  ```
  TFile *myfile = new TFile("output.root");
  ```

  ```
  TH1D *myhist = (TH1D*) myfile->Get("hist");
  ```

  ```
  myhist->Draw();
  ```

  Il comando `Get` prende l'oggetto il cui nome ricordato da ROOT è `"hist"`, ovvero il primo argomento dato al costruttore di `TH1D` nella macro che l'ha creato.

  Normalmente `Get` restituisce un puntatore di tipo `TObject`, che è la classe base di tutte le classi in ROOT. Per interpretare l'oggetto come istogramma è quindi necessario effettuare un cast esplicito con `(TH1D*)`.

  Strutturando il codice in questo modo l'istogramma rimane linkato a `myfile`, quindi quando esso viene chiuso non è più possibile accedere all'istogramma. Per slegare l'istogramma dal file, si può usare il comando:

  ```
  hh->SetDirectory(NULL);
  ```

3 Esercizio

Realizzare un programma analogo a `ioexample2.C` che però non richieda di sapere a priori l'intervallo in cui sono distribuiti i dati. Il programma non deve richiedere a priori il numero di dati presenti nel file di input.

4 Soluzione: descrizione

Codici di riferimento: `esercizio1_v1.C` `esercizio1_v2.C`

Il problema è affrontabile in due modi:

1. (`esercizio1_v1.C`) Il file di input viene letto due volte: la prima volta si conta il numero di dati e si tiene traccia del minimo e del massimo, la seconda volta si riempie l'istogramma. Questo metodo è vantaggioso dal punto di vista della memoria, in quanto non è necessario allocare un vettore per contenere i dati. Tuttavia, è svantaggioso in termini di tempo, in quanto le operazioni di lettura da file sono piuttosto lente e in questo caso tale operazione viene eseguita due volte.

2. (`esercizio1_v2.C`) Il file di input viene letto una sola volta, durante la quale oltre a contare il numero di dati e tenere traccia del minimo e del massimo, si memorizzano i dati in un vettore. Questo metodo è vantaggioso in termini di tempo, in quanto il file viene letto una sola volta, tuttavia è svantaggioso in termini di memoria, in quanto è necessario allocare un vettore per contenere i dati. Inoltre, non è possibile sapere a priori quanto grande deve essere il vettore, quindi si utilizza un vettore dinamico della classe (`std::vector`).

5 Soluzione: extra

5.1 Argomenti opzionali nelle funzioni

Codice di riferimento: `esercizio1_v2.C`

In questo esempio viene data la possibilità di specificare il numero massimo di dati da leggere come argomento opzionale della funzione. Per definire un argomento opzionale, è sufficiente assegnargli un valore di default nella dichiarazione della funzione:

```
void esercizio1_v2(..., const unsigned int limit = 100000)
```

In questo modo, se la funzione viene chiamata con solo due argomenti, l'argomento `limit` assumerà il valore di default `100000`. Se invece la funzione viene chiamata con tre argomenti, `limit` assumerà il valore passato nella chiamata.

5.2 Metodo alternativo di riempimento

Codice di riferimento: `esercizio1_v2.C`

Finora la sintassi per riempire un istogramma è stata:

```
for(int i=0;i<data.size();i++) hist->Fill(data[i]);
```

Un metodo alternativo è utilizzare un *for-each* loop:

```
for(auto y:data) hist->Fill(y);
```

In questo modo, per ogni elemento `y` del vettore `data`, viene eseguita l'istruzione `Fill(y)`.
Il tipo `auto` permette al compilatore di dedurre automaticamente il tipo di `y` in base al tipo degli elementi del vettore `data`.

6 Soluzione: codici

6.1 esercizio1_v1.C

```
#include <Riostream.h>
#include <TH1D.h>
#include <TFile.h>
#include <TCanvas.h>
using namespace std;

void esercizio1_v1(const string& fimpName, const string& histName)
{
    ifstream in(fimpName);
    if (!in)
    {
        cout << "Il file " << fimpName << " non esiste." << endl;
        return;
    }
    double x, min, max;
    int count = 0;

    if(in >> x)
    {
        count++;
        min = x;
        max = x;
    }
    else
    {
        cout << "Il file " << fimpName << " e' vuoto." << endl;
        return;
    }

    while(in >> x)
    {
        count++;
        if(x < min) min = x;
        if(x > max) max = x;
    }

    in.clear();
    in.seekg(0, ios::beg);

    cout << "\nDati letti: " << count << "\nEstremi dell'istogramma: (" << min << ", " <<
     max << ")" << "\n\n";

    TH1D* hist;

    double tol = (max-min)*0.01;
    hist = new TH1D("hist", "Istogramma", 100, min-tol, max+tol);

    while(in >> x)
        hist->Fill(x);

    in.close();
    hist->Draw();


    TFile file(histName.c_str(), "recreate");
    hist->Write();
    file.Close();
}
```

6.2 esercizio1_v2.C

```
#include <Riostream.h>
#include <TH1D.h>
#include <TFile.h>
#include <TCanvas.h>
using namespace std;

void esercizio1_v2(const string& fimpName, const string& histName, const unsigned int limit = 100000)
{
    ifstream in(fimpName);
    if (!in)
    {
        cout << "Il file " << fimpName << " non esiste." << endl;
        return;
    }

    double x, min, max;
    vector<double> data;

    if(in >> x)
    {
        data.push_back(x);
        min = x;
        max = x;
    }
    else
    {
        cout << "Il file " << fimpName << " e' vuoto." << endl;
        return;
    }
    while(in >> x && data.size() < limit)
    {
        data.push_back(x);
        if(x < min) min = x;
        if(x > max) max = x;
    }

    in.close();

    if(data.size() == limit)
        cout << "WARNING: e' stato raggiunto il limite massimo di " << limit << " dati letti." << endl;

    cout << "\nDati letti: " << data.size() << "\nEstremi dell'istogramma: (" << min << ", " << max << ")" << "\n\n";

    TH1D* hist;
    hist = new TH1D("hist", "Istogramma", 100, min-1, max+1);

    for(auto y:data) hist->Fill(y);

    hist->Draw();

    TFile file(histName.c_str(), "recreate");
    hist->Write();
    file.Close();
}
```