Corso di Laboratorio 2 – Programmazione C++

Silvia Arcelli

14 Novembre 2014

ROOT Trees

- I Tree (TTree) sono delle strutture dati di root utilizzate per storare, leggere ed analizzare un gran numero di entità costituite da un insieme eterogeneo di dati di vario tipo: *n-tupla* di variabili, o oggetti

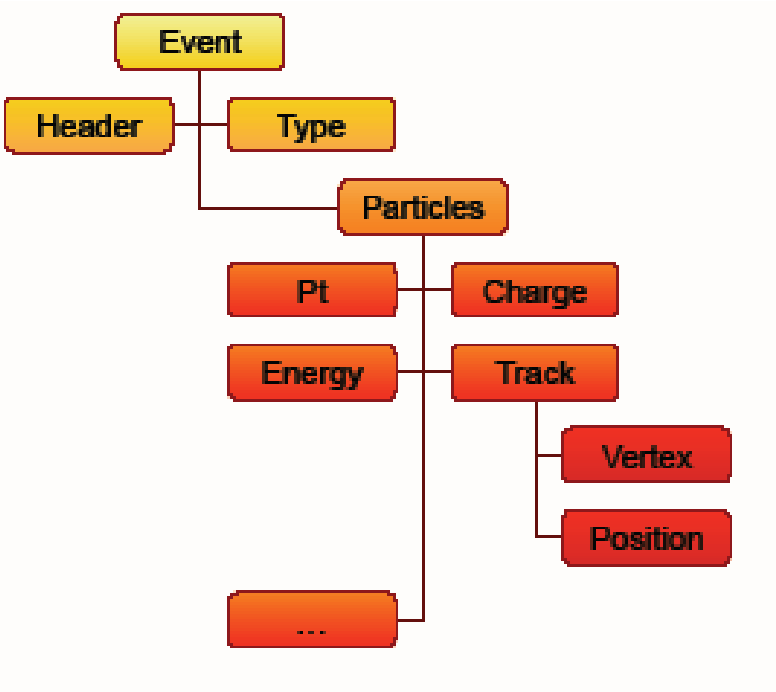
- Molto utili dal punto di vista della persistenza dei dati. Ottimizzati per ridurre lo spazio disco e la velocità di accesso in I/O. In particolare, molto efficienti in una situazione Write Once, Read Many (“WORM”)

- Grande flessibilità in fase di analisi (selezione sulle variabili del tree, calcolare espressioni complesse delle variabili , creare istogrammi, etc)

ROOT Trees

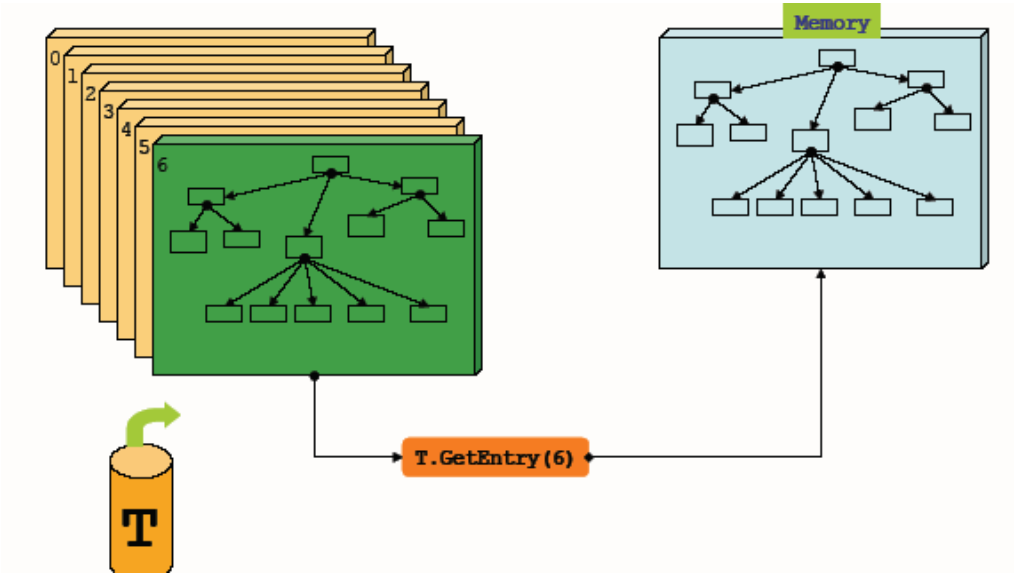
- Possono gestire ogni tipo di dato (una n-tupla di variabili "native", o strutture più complesse descritte da oggetti)

x	y	z
-1.10228	-1.79939	4.452822
1.867178	-0.59662	3.842313
-0.52418	1.868521	3.766139
-0.38061	0.969128	1.084074
0.552454	-0.21231	0.350281
-0.18495	1.187305	1.443902
0.205643	-0.77015	0.635417
1.079222	-0.32739	1.271904
-0.27492	-1.72143	3.038899
2.047779	-0.06268	4.197329
-0.45868	-1.44322	2.293266
0.304731	-0.88464	0.875442
-0.71234	-0.22239	0.556881
-0.27187	1.181767	1.470484
0.886202	-0.65411	1.213209
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
		3.562347



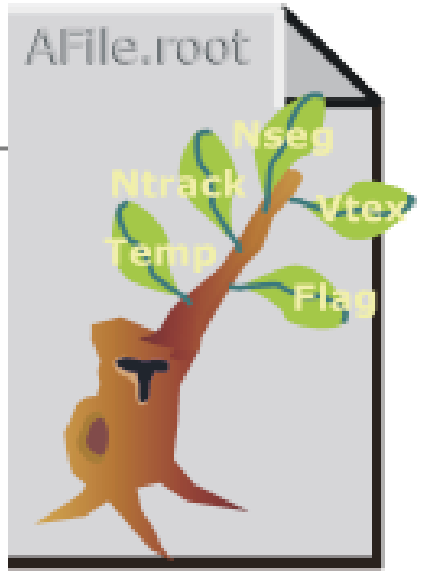
ROOT Trees

- Accesso diretto in qualunque punto del Tree
- Solo quell'elemento (o solo parte di esso) in memoria.



- Organizzazione dei dati ottimizzata per massimizzare la velocità di I/O e la compressione

ROOT Trees



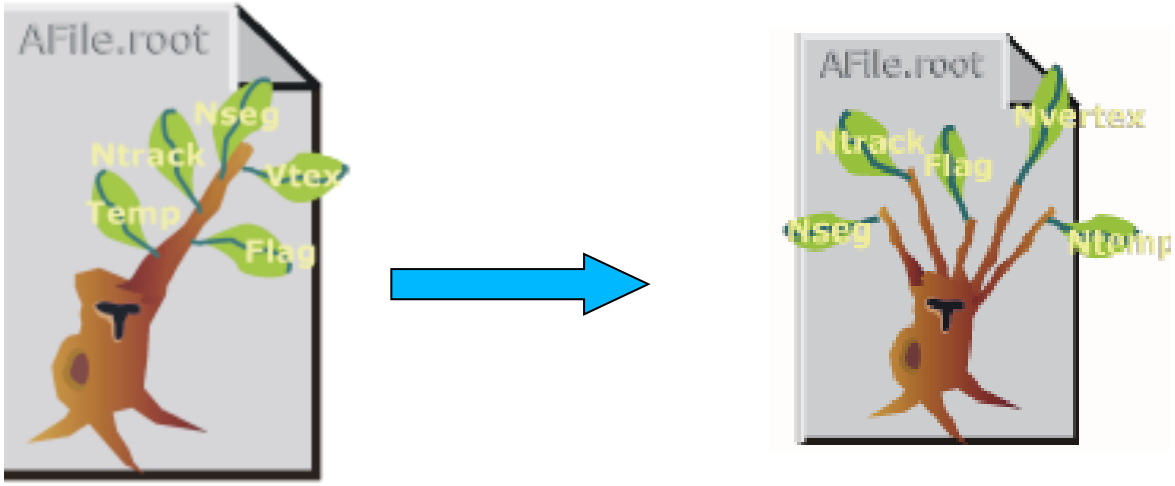
- Dati organizzati gerarchicamente in “Branches“ (TBranch), che possono essere viste come un equivalente di directories

- Ogni Branch contiene una sottostruttura, delle “Leaves“ (TLeaf), ciascuna delle quali corrispondonde a un singolo dato.

- I Branches possono essere letti selettivamente (con TTree::SetBranchStatus(...), performance ottimizzata in fase di analisi!). Variabili del Tree che raramente saranno usate insieme è quindi conveniente scriverle sempre in branch separati.

ROOT Trees

- In generale, si tende a far corrispondere a ogni Leaf un Branch (accesso selettivo con massima granularità)



- Questa strategia comporta una elevata velocità in lettura (si accede solo ai branch attivati)

- Meno efficiente in scrittura; tuttavia, ottimale per la condizione Write Once-Read Many, situazione molto frequente in un esperimento di fisica

ROOT Trees

Esempio di tree con un set di variabili di tipo float (ad esempio, fate n misure di una quantità fisica x e di alcune condizioni "al contorno" y e z che potrebbero influenzare la misura):

```
TFile*F = new TFile("test.root",RECREATE); //open a file
TTree *T = new TTree("T","test"); // create the tree
Float_t x,y,z;
T->Branch("x",&x,"x/F");  // create branches
T->Branch("y",&y,"y/F");
T->Branch("z",&z,"z/F");
for(Int_t i=0;i<100;i++){
//
// Read/or calculate x,y and z in a loop
//.......
 T->Fill(); // fill the tree, for each entry.
}
T->Write(); //scrivo il tree sul file
F->Close(); // close the file
```

ROOT Trees

- Per analizzare il tree, avete a disposizione una serie di metodi.
  Se avete scritto il Tree su un file, per riaccedere all'informazione contenuta aprite il file .root su cui l'avete scritta:

```
TFile *file = new TFile("test.root");
```

- Recuperate il tree dal file con il metodo TFile::Get() facendo un cast (il metodo file->Get("...") vi ritorna un Tobject)

```
TTree * Tout= (TTree*)file->Get("T");
```

- Per sapere che variabili contiene, potete usare il metodo Print()

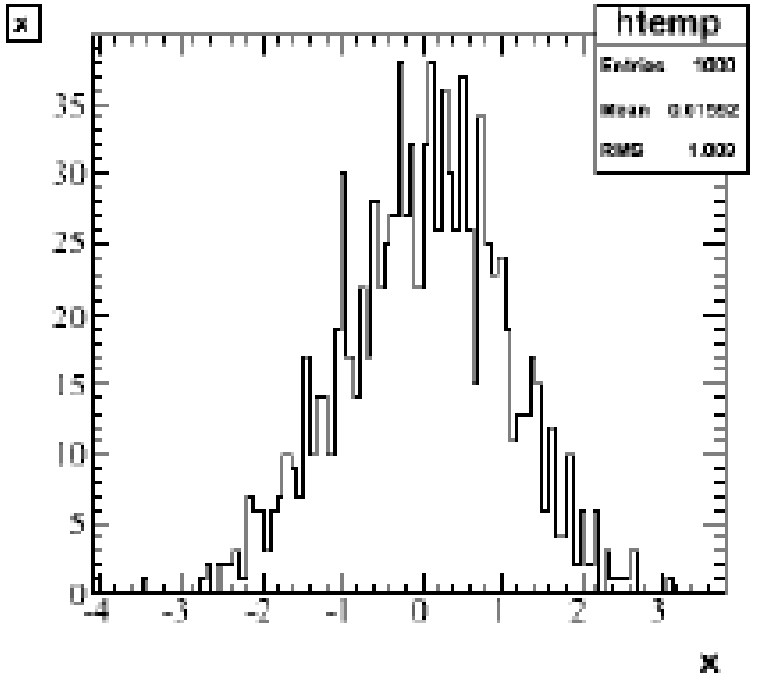
```
Tout->Print();
```

ROOT Trees

- Per graficare un branch, ad esempio x:

```
Tout->Draw("x");
```

(In questo caso il layout dell'istogramma –nome,binnaggio, range- è deciso in automatico da ROOT)



- Per riempire un istogramma da voi predefinito con una variabile di un tree usate >>:

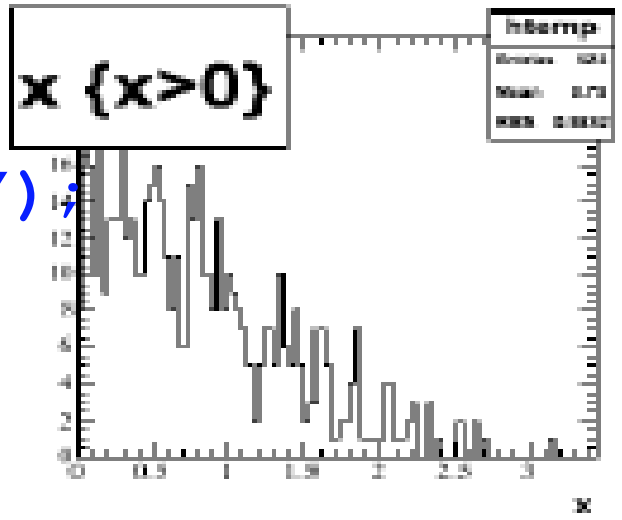
```
TH1F *h1=new TH1F("h1","hist from tree",50,-4, 4);
Tout->Draw("x>>h1");
```

ROOT Trees

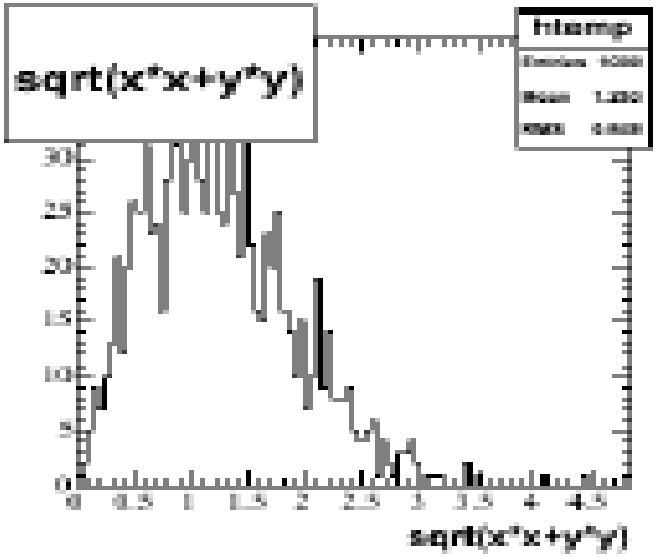
- Per graficare una variabile applicando **selezioni** su se stessa o sulle altre variabili (il tree è una n-tupla,mantiene le "correlazioni"!):

```
Tout->Draw("x","x>0");

Tout->Draw("x","y>0 && x<10");
```



- Potete fare operazioni sulle variabili:

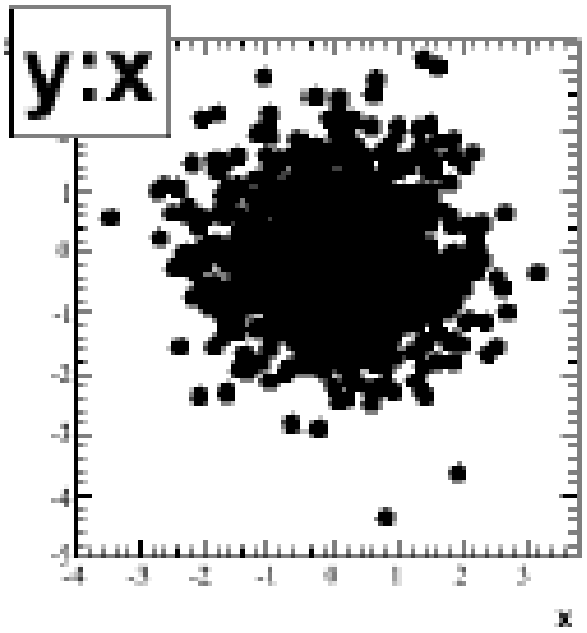


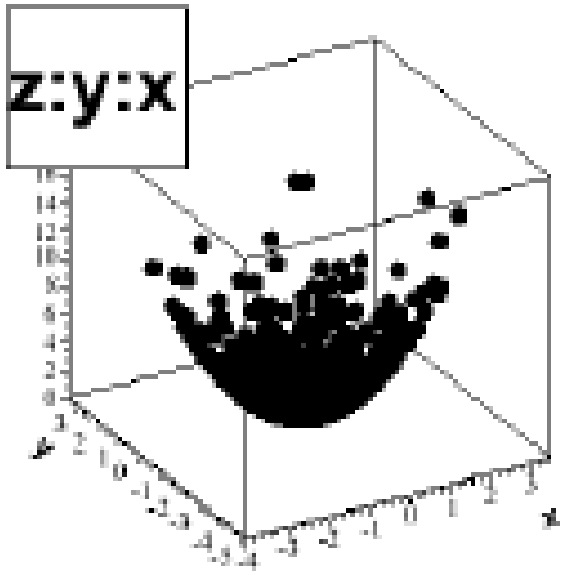

```
Tout->Draw("sqrt(x*x+y*y)");
```

flessibilità durante l'analisi dati

ROOT Trees

•Plot di correlazione 2 e 3-D:






```
T->Draw("x:y");
```

```
T->Draw("x:y:z");
```

Esempi

Macro che genera un tree di variabili di tipo nativo e lo scrive su file:

writeTree.C

Macro che legge il tree generato dalla macro precedente
da un file .root e lo analizza:

readTree.C

Nel prossimo turno vi si richiederà di finire di implementare metodi aggiuntivi della classe particle, e di iniziare a scrivere il programma di generazione MC. Prima di scrivere il programma di generazione, testate tutto riempendo l'array di Particletype, stampandone il contenuto, Istanziando delle particle per vedere se i metodi che avete scritto fanno quello che devono....

Schema del programma di generazione

Generazione "Monte Carlo" di eventi fisici contenenti particelle:
10^5 eventi, ognuno contenente 100 particelle di alcuni tipi predefiniti, fra cui uno stato risonante che può decadere. Generare:

- Secondo definite proporzioni dei tipi di particelle
- Distribuzione uniforme nelle direzioni
- Distribuzione esponenziale dell'impulso
- Istogrammi delle proprietà delle particelle con root

particle.h

```
#ifndef PARTICLE_H
#define PARTICLE_H
#include “particleType.h"
class particle {
public:
//membri pubblici
/*----*/
private:
//attributi privati
double fPx,fPy,fPz;
int fIparticle;
//attributi statici privati
static const int fMaxNumParticleType=10;
static int fNparticleType;
static particleType *fParticleType[fMaxNumParticleType];
//membri statici privati
/*----*/
};
 #endif
```

particle.h

```
#ifndef PARTICLE_H
#define PARTICLE_H
 #include “particleType.h"

class particle {
public:  //membri pubblici
particle(int iparticle,double px=0.0,double py=0.0,double pz=0.0);
particle(const char *name,double px=0.0,double py=0.0,double pz=0.0);
int GetParticleType() const {return fIparticle;}
static int AddParticleType(const char *name,double mass,int charge,double width=0.);
static void PrintParticleType(); // potrei dichiararla const??
void ChangeParticleType(int iparticle);
void ChangeParticleType(const char *name);
void Print() const;
particle& operator=(const particle &value);
particle operator+(const particle &value) const;
private:
/*----*/
//membri statici privati
static int FindParticle( const char *name);
};
#endif
```

particle.cxx

```
#include<iostream>
 using namespace std;
#include "particle.h“
 #include "particleType.h“
 #include "resonanceType.h"

// inizializzazione degli attributi statici
int particle::fNparticleType = 0;
particleType *particle::fParticleType[fMaxNumParticleType];

particle::particle(const char *name,double px=0,double py=0, double pz=0):
 fPx(px), fPy(py), fPz(pz) {
int ip=FindParticle(name);
if(ip != -1){
   fIparticle = ip;
 }
else{
  printf("Particle \"%s\" doesn't exist in the array\n",name);
  fIparticle = -1;
  }
}
```

Un costruttore

Il metodo FindParticle

particle.cxx

```
/*
int particle::FindParticle(const char *name){
for(int i=0;i<fNparticleType;i++){
const char *currentType = fParticleType[i]->GetParticleName();
if(name==currentType) return i;
}
return -1;
}
 */
int particle::FindParticle(const char *name){
for(int i=0;i<fNparticleType;i++){
  const char *currentType = fParticleType[i]->GetParticleName();
  int k=0;
  while((currentType[k] == name[k]) && currentType[k] != '\0' && name[k] != '\0'){
    k++;
  }
  if(currentType[k] == name[k]) return i;
}
return -1; // if no match
}
```

Funziona solo con char * inizializzati a stringhe costanti

Confronto carattere per carattere: questa implementazione funziona sempre (è quello che fa strcmp)

Il metodo AddParticleType(...)

particle.cxx

```
int particle::AddParticleType( const char *name,double mass,int charge,double
width){
 if(fNparticleType < fMaxNumParticleType){
    int ip = FindParticle(name);
    if(ip != -1){
       printf("A particle with this name (\"%s\") already exists in the array \n",name);
       return 2; }
    if(width > 0)
      fParticleType[fNparticleType] = new resonanceType(name,mass,charge,width);
    else
       fParticleType[fNparticleType] = new particleType(name,mass,charge);
    fNparticleType++;
  } else{
    printf("Array is full because you have already inserted %i particles (nothing
done)\n",fMaxNumParticleType);
    return 1;
   }
  return 0;
}
```

Inserzione di nuovi elementi con allocazione dinamica

Puntatori della classe base indirizzano anche classi derivate