



Corso di Laboratorio 2 – Programmazione C++

Silvia Arcelli

11 Novembre 2015

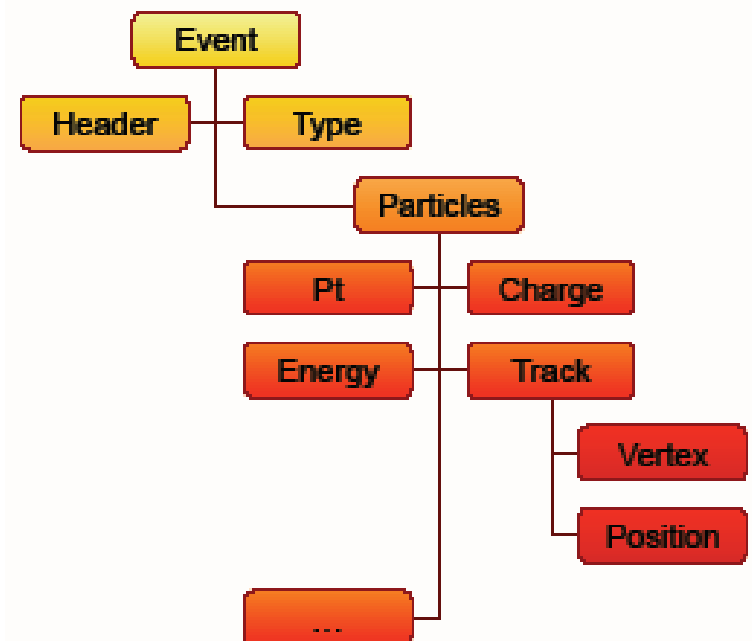
ROOT Trees

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

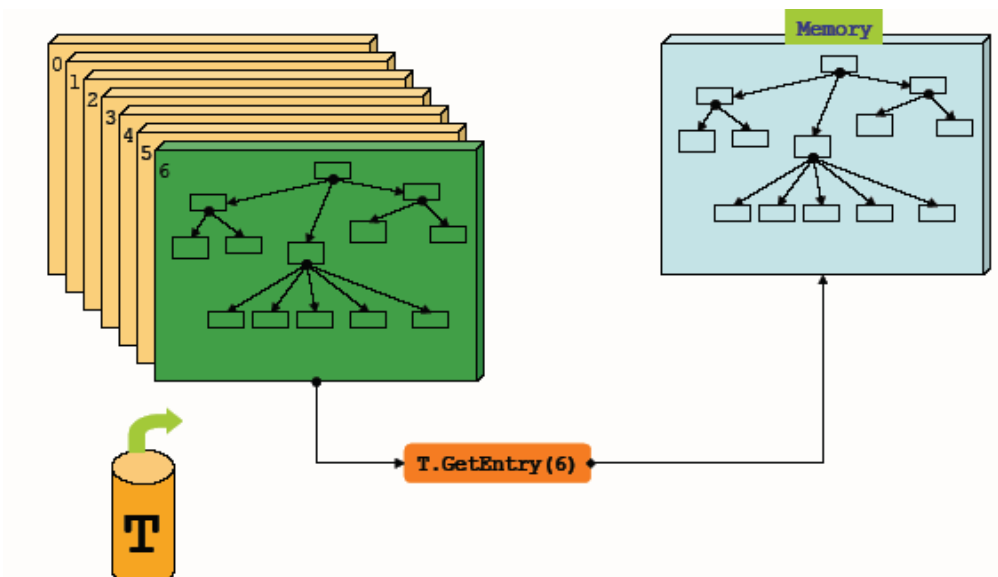
- Possono gestire **ogni tipo di dato** (una n-tupla di variabili “native”, o strutture più complesse descritte da oggetti)

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



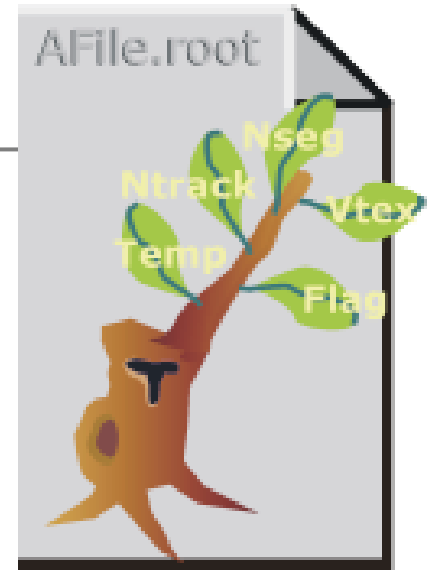
ROOT Trees

- 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

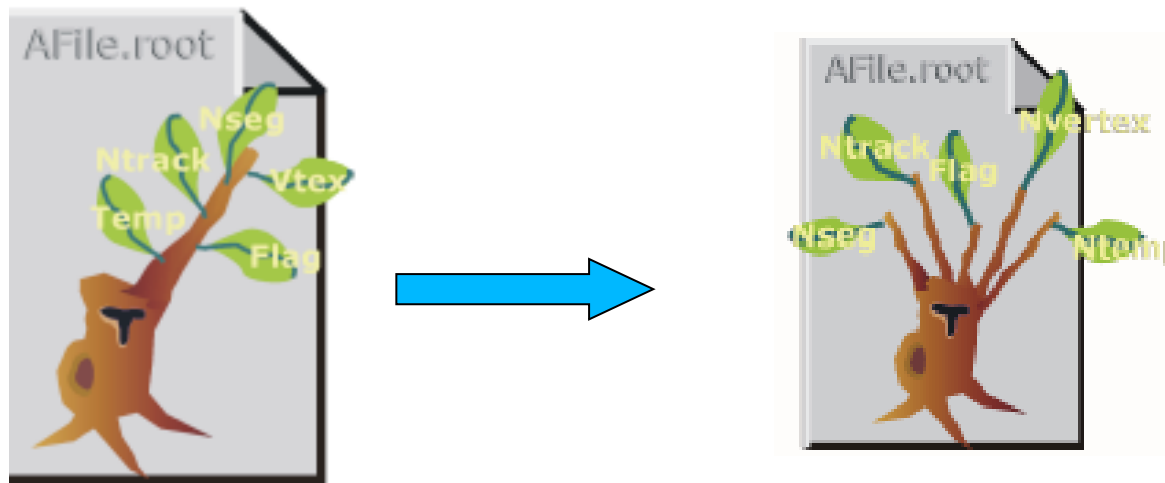
ROOT Trees



- 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 corrisponde 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 in fase di analisi 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 (dati scritti/generati una sola volta, analizzati tante volte e da molti utenti)

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
//(Dati Reali o MC, esito di un calcolo), operazione molto
//spesso "costosa"
//o addirittura praticamente "irripetibile"
//.....
T->Fill(); // fill the tree, for each entry.
}
T->Write(); //scrivo il tree sul file
F->Close(); // close the file
```

Da qui in poi, passate alla fase di “analisi dati”.

ROOT Trees

- Per analizzare il tree, avete a disposizione una serie di metodi. Se avete scritto il Tree su un file, per accedere nuovamente 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()

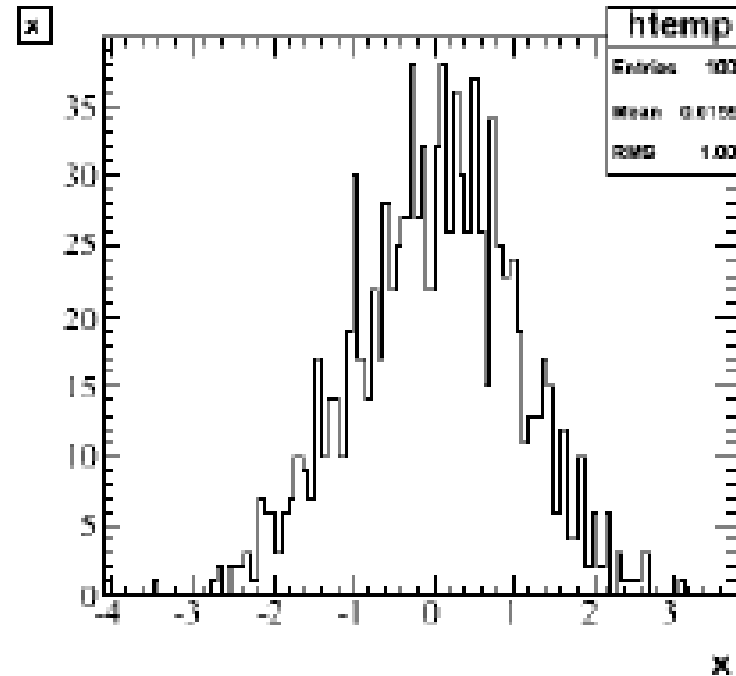
```
Tout->Print();
```

ROOT Trees

- Per graficare una variabile, 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** (binnaggio e range ad -hoc) con una variabile di un tree usate >>:

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

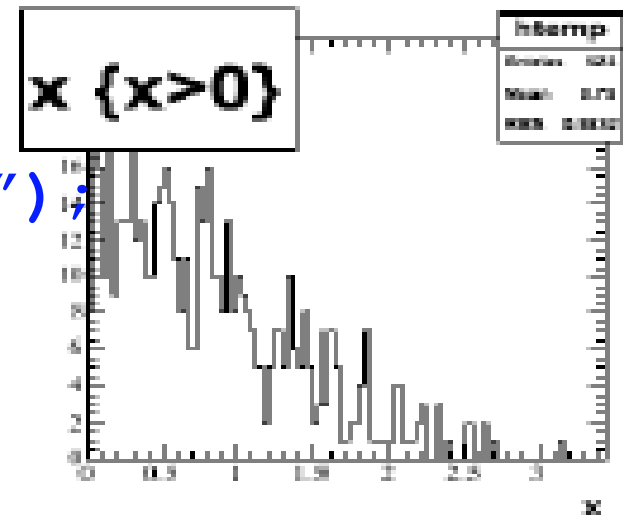
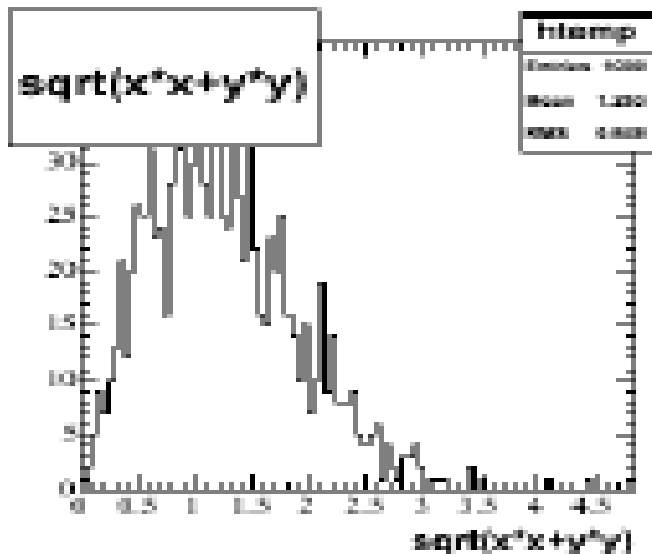
ROOT Trees

- Per graficare una variabile applicando **selezioni**, o 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:

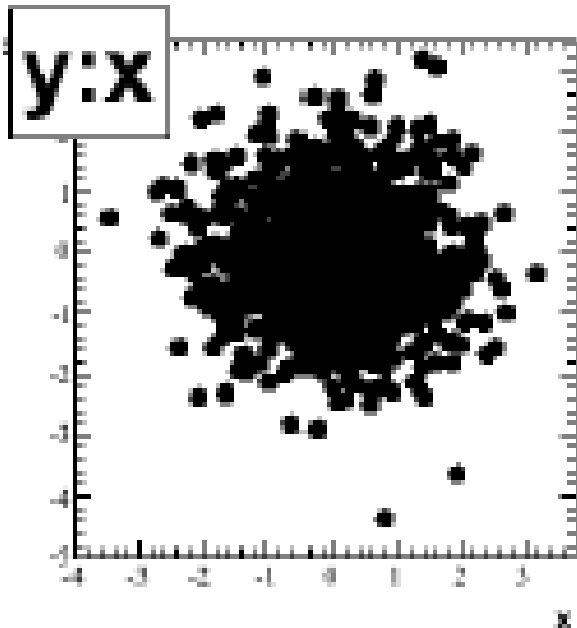


```
Tout->Draw("`sqrt(x*x+y*y)`");
```

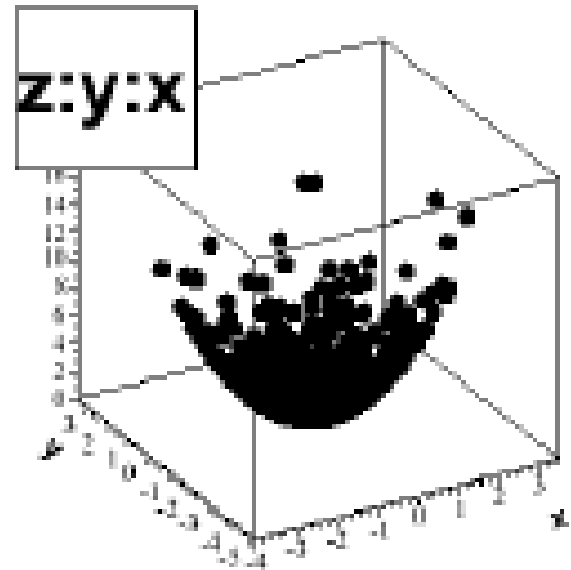
flessibilità durante l'analisi dati:
selezioni e operazioni sulle variabili

ROOT Trees

- Plot di correlazione 2 e 3-D:



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



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

Esempi

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

[writeTree.C](#)

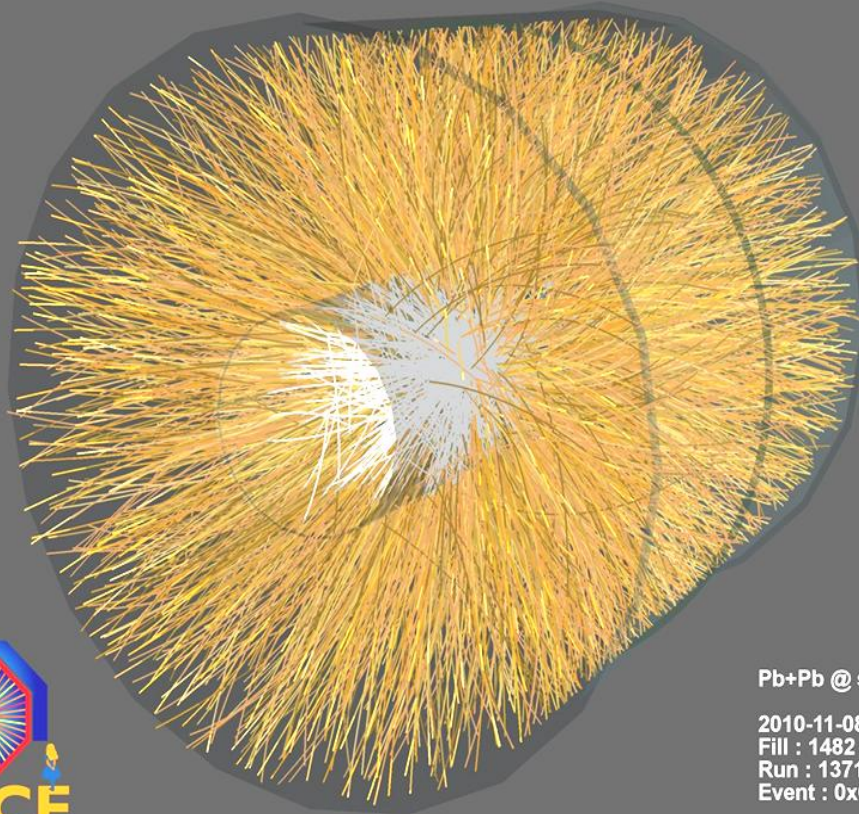
Macro che legge il tree generato dalla macro precedente da un file .root e lo analizza:

[readTree.C](#)

Esercitazione di laboratorio

- Scopo dell'esercizio:
 - implementare un **prototipo di codice** utilizzabile per rappresentare il contenuto di eventi fisici in collisioni di particelle
 - Fare pratica sulla programmazione ad oggetti: meccanismi di **ereditarietà e aggregazione**
 - **Generazione Monte Carlo**: applicare i metodi di generazione presentati a lezione.
 - Utilizzare root per **salvare le informazioni generate dalla simulazione Monte Carlo e analizzarle**.
-

Esercitazione di laboratorio



- N particelle/evento $\sim 10^4$
- N eventi $\sim 10^7$
- Tipi di particelle:
e, π , K, μ , p, n, γ ...
+ risonanze.



**Moltissime particelle,
ma di un numero limitato
_di tipi**

Esercitazione in laboratorio

Abbiamo quindi questo problema:

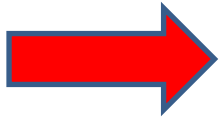
- generare molti eventi
 - In ciascun evento ci sono molte particelle, ma il loro tipo è limitato:
 - 40% π^+
 - 40% π^-
 - 5% K^+
 - 5% K^-
 - 4.5% P^+
 - 4.5% P^-
 - 1% K^0
- Proprietà delle particelle:
- nome,
 - massa,
 - carica,
 - eventualmente un altro parametro, la larghezza, legata alla vita media della particella.



due classi per rappresentare le proprietà di base delle particelle: **particleType** e **ResonanceType**

Esercitazione di laboratorio

A queste particelle poi dobbiamo conferire un impulso (3-D) che può variare da particella a particella, da evento a evento. Dobbiamo quindi aggiungere questa informazione:

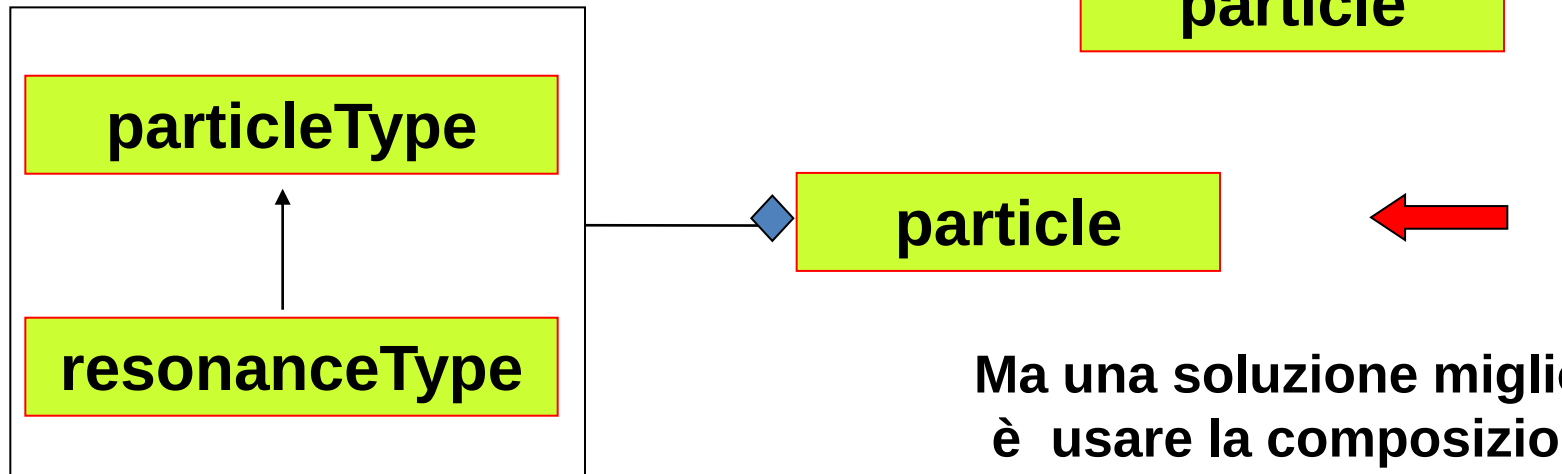


Classe **particle**

Per includere anche l'informazione delle proprietà di base, possiamo fare due scelte: un classe **particle** che eredita da **resonanceType**, o una strada diversa: **l'aggregazione (composizione)**

Esercitazione di Laboratorio

Si può strutturare scegliendo di utilizzare esclusivamente l'ereditarietà.....



Ma una soluzione migliore è usare la composizione (aggregazione).
Risparmio di memoria!

I Turno

SCRIVERE LE CLASSI NECESSARIE PER IL PROGRAMMA

Tre classi di supporto al programma:

- ❑particleType
 - ❑resonanceType
- } descrittive delle proprietà di base - nome, massa, carica, (larghezza)
-
- ❑Particle Aggiunge proprietà cinematiche

Separare definizione di classe e sua implementazione in files .h e .cxx

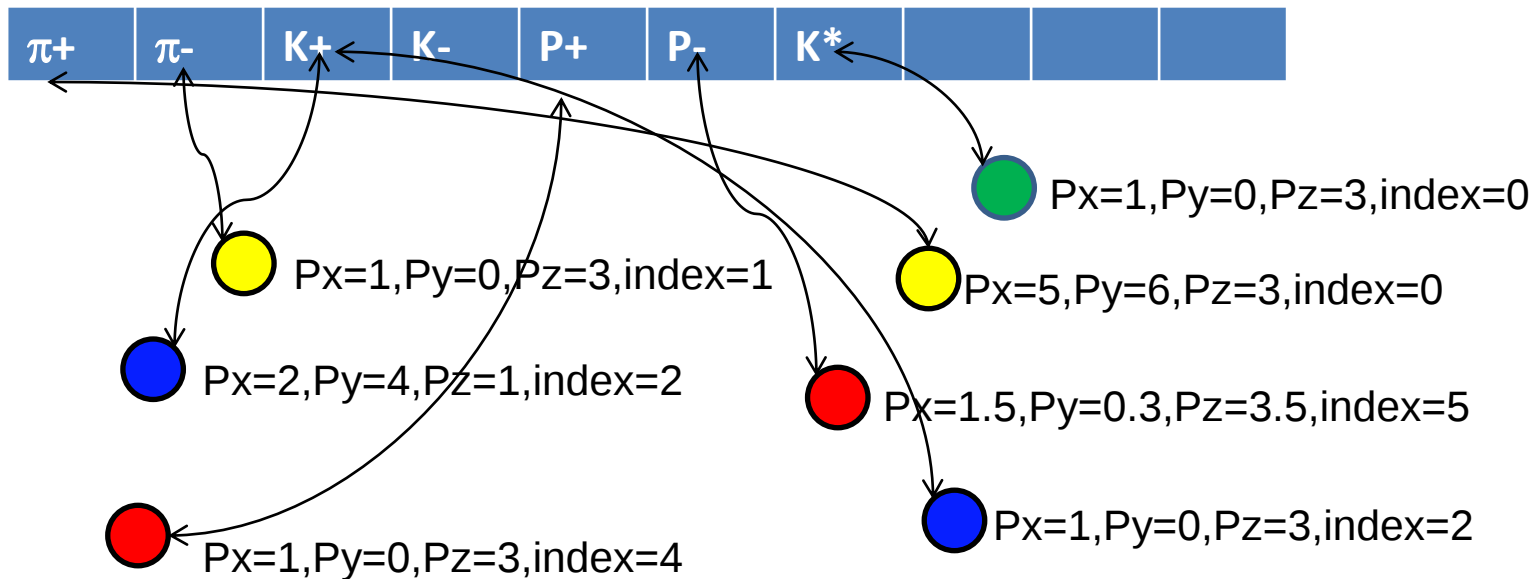
Metodi:

.Utilizzate ereditarietà virtuale

.dichiarate const quanto dichiarabile const

La classe particle

- Attributi: le tre componenti dell'impulso e un indice
- Un membro statico; un array di puntatori a particleType. Ha la funzione di “tabella” comune a tutte le istanze per descrivere le proprietà caratteristiche del tipo di particella



Il metodo AddParticleType

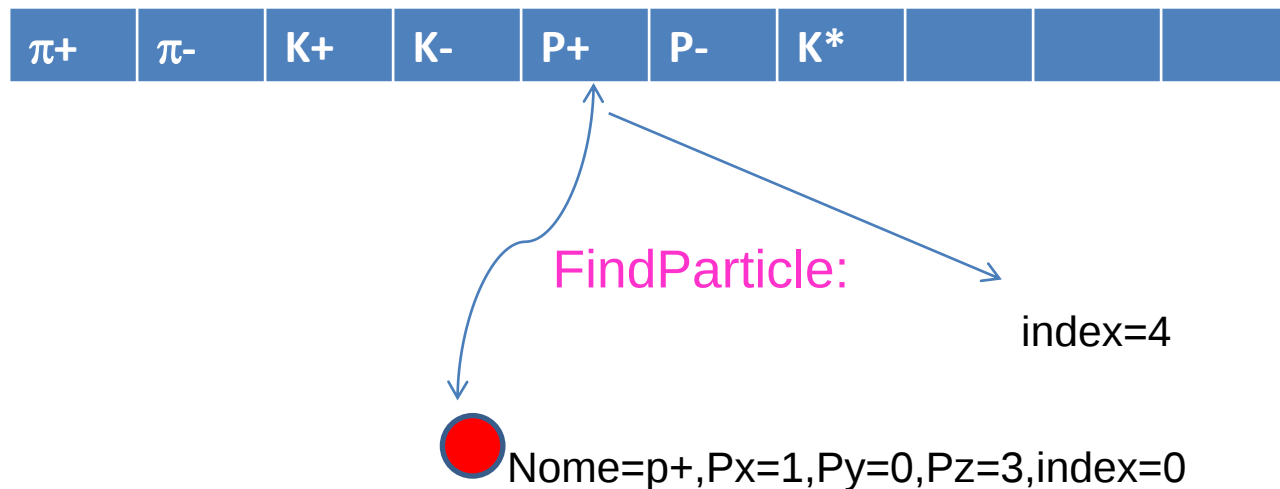
- Ha la funzione di riempire la tabella dei tipi di particelle: è un metodo statico, e può essere usato indipendentemente dalle istanze di particle.



Argomenti: Nome, massa, carica ed eventualmente larghezza

Il metodo FindParticle

- In base al nome dato dall'utente nel creare l'istanza di particle, deve cercare e verificare nella "tabella" la corrispondenza con l'indice dell'elemento appropriato. Tale indice deve poi essere utilizzato per assegnare un valore al membro "indice" della classe particle



Il Turno

GENERAZIONE MONTE CARLO

Implementare metodi aggiuntivi della classe particle, e scrivere il programma di generazione Monte Carlo.

Prima di scrivere il programma di generazione:

- fate un test del codice, inserendo gli elementi richiesti dalla traccia nell'array statico (membro della classe particle) di puntatori a Particletype
 - Stampatene il contenuto
 - Istanziate delle particle per vedere se i metodi che avete scritto fanno quello che devono....
-

Il Turno

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

- Secondo **definite proporzioni** (date dalla traccia) dei tipi di particelle
- **Distribuzione uniforme** nelle direzioni (angoli θ e φ)
- **Distribuzione esponenziale** dell'impulso
- Riempire istogrammi delle proprietà delle particelle con root

Fatelo “esplicitamente”,
usate i metodi MC illustrati
durante la scorsa lezione

III Turno

ANALISI

Analisi degli eventi generati attraverso gli istogrammi salvati su file root alla fine del ciclo di generazione:

- Verificare che le distribuzioni di impulso e angoli polari e azimutali siano coerenti con quanto simulato in fase di generazione attraverso un fit
- Dalla distribuzione di massa invariante, utilizzando i metodi degli istogrammi cercare di separare il segnale della risonanza K^*0 dal fondo di altre particelle
- Attraverso un fit del segnale (assumere una distribuzione gaussiana), estrarre i parametri della risonanza e confrontarli con quelli impostati in fase di generazione