LOGICIELS SCIENTIFIQUES

S2 - Master PHEAPC

Pr. M. EL KACIMI[(1)]

[(1)] **Université Cadi Ayyad**
Faculté des Sciences Semlalia - Département de Physique

Année universitaire 2017/2018

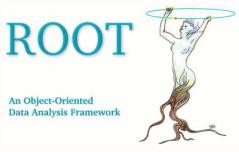
Chapitre 2 : SESSION INTERACTIVE

1. **Environnement de ROOT**

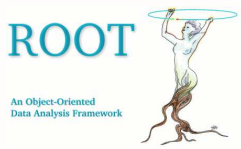
2. **Interpréteur C/C++ : CINT/CLING**
   - **Quelques commandes CINT /CLING**

3. **Revue de C++**
   - **Variable, Pointeurs, Tableaux**
   - **Structures de contrôle : Conditions et Boucles**
   - **Lire et écrire sur un fichier**
   - **Classe, Encapsulation, Héritage et polymorphisme**
     - Classe
     - **Encapsulation des données**
     - **Héritage : Masquage et Démasquage**

4. Histogrammes



Chapitre 2 : SESSION INTERACTIVE



Environnement de ROOT

Configuration et variables d'environnement

Lors du démarrage, ROOT lis les deux fichiers :

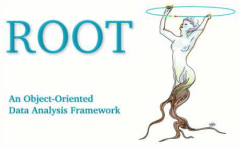
.rootrc : Configure les différents chemins utilisés par ROOT , entre autres, ROOTSYS et LD_LIBRARY_PATH ;

rootlogon.C : Définit les définitions de style, $\cdots$.

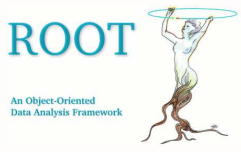
Taper à l'invite du SHELL :

```
[ elkacimi MacBook-Pro-de-Mohamed ~ ] locate rootrc
/usr/local/root/etc/system.rootrc
[ elkacimi MacBook-Pro-de-Mohamed ~ ] more /usr/local/root/etc/system.rootrc
# @(#)root/config:$Id$
# Author: Fons Rademakers  22/09/95

# ROOT Environment settings are handled via the class TEnv. To see
# which values are active do: gEnv->Print().
```

```
[ elkacimi MacBook-Pro-de-Mohamed ~ ] locate rootlogon.C
~/my_rootlogon.C
[ elkacimi MacBook-Pro-de-Mohamed ~ ] more ~/my_rootlogon.C
{
...
 TStyle *atlasStyle= new TStyle("ATLAS","Atlas style");

// use plain black on white colors
 Int_t icol=0;
 atlasStyle->SetFrameBorderMode(icol);
 atlasStyle->SetCanvasBorderMode(icol);
 atlasStyle->SetPadBorderMode(icol);
 atlasStyle->SetPadColor(icol);
```



Chapitre 2 : SESSION INTERACTIVE



Interpréteur C/C++ : CINT /CLING

ROOT est doté d'un interpréteur intégré appelé CLING (CINT dans les versions antérieures à 6.00) :

$\Longrightarrow$ Permet de lire des commandes C++ à l'invite de ROOT sans avoir besoin de les compiler, de même pour des lignes de code de Python ;

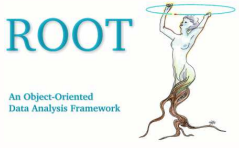
$\Longrightarrow$ Stocker les informations concernant les objets C++ saisis en ligne et constitue l'étage de sortie de ROOT ;

$\Longrightarrow$ Taper dans la ligne de commande du shell :

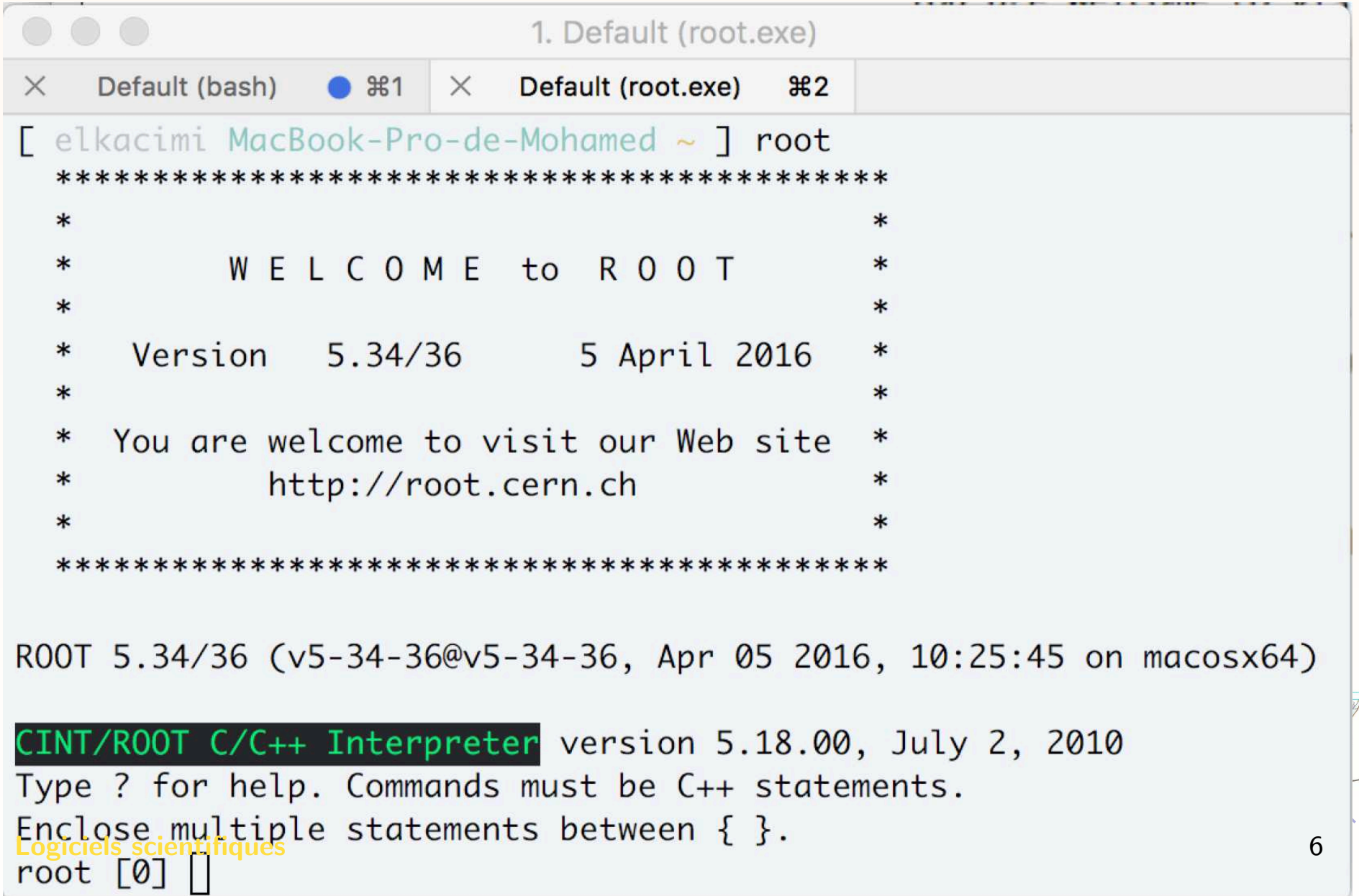
```
[Prompt>] root
```

et pour sauter le message de bienvenue, tapper la commande

```
[Prompt>] root -l
```



Interpréteur C/C++ : CINT /CLING

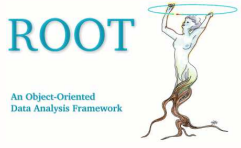


Interpréteur C/C++ : CINT /CLING

Quelques commandes CINT /CLING

Toutes les commandes en ligne commencent par un point "." à l'exception de {} et ? ;

```
Help:          ?             : help
Shell:         ![shell]      : execute shell command (pwd, ls, wich)
Source:        v <[line]>:   : view source code <around [line]>
Evaluation:    x [file]      : load [file] and execute function [file]
Load/Unload:   L [file]      : load [file]
Monitor:       g <[var]>     : list global variable
               l <[var]>     : list local variable
               proto <[scope]::>[func] : show function prototype
               class <[name]> : show class definition (one level)
               Class <[name]> : show class definition (all level)
               include       : show include paths
               file          : show loaded files
....
Quit:          q             : quit cint
```

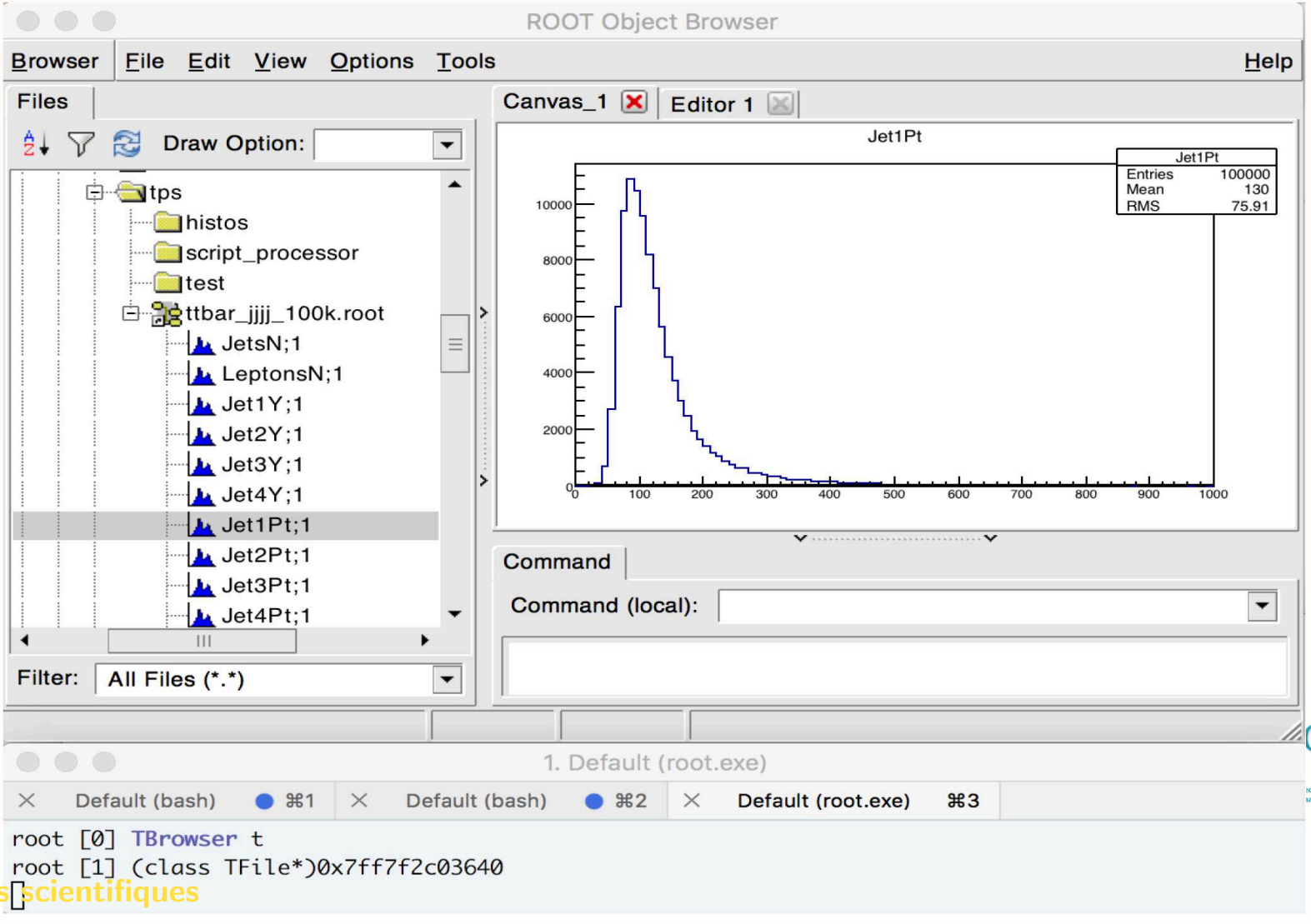


Interpréteur C/C++ : CINT /CLING

Browser

A l'invite de ROOT , taper

```
root [.] TBRowser
```



Interpréteur C/C++ : CINT /CLING

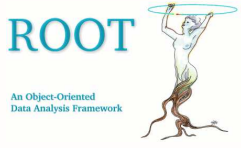
ROOT comme calculatrice de poche ...

Types basiques de C++ sont translatés en types basiques de ROOT comme suit : Première lettre capitale et ajout du prefixe "_t" :

$$\texttt{int} \rightarrow \texttt{Int\_t} \qquad \texttt{float} \rightarrow \texttt{Float\_t} \qquad \texttt{double} \rightarrow \texttt{Double\_t} \quad \cdots$$

```
root [0] Double_t a(2); Double_t b(3); a+b
(double)5.0000000000000000e+00
root [2] Int_t i(1);a+i
(double)3.0000000000000000e+00
```

Types propres de ROOT (classes) : Les noms commencent avec "T" et les deux premières lettres du nom de la classe sont capitales : TDirectory, TFile, $\cdots$



Interpréteur C/C++ : CINT /CLING

Commandes en ligne : 2 manières

Première méthode :

Une seule instruction par ligne, ou plusieurs séparées par ";", à l'invite : Pas de ";" à la fin de la ligne

```
root [.] std::cout<<"Hello, World!\n"<<std::endl
Hello, World!
root [.] std::cout<<"What's up?\n"<<std::endl
What's up?
```

Plusieurs lignes de commande entourées par { }, les lignes doivent finir par ";" :

```
root [.] {
end with '}', '@':abort > std::cout<<"Hello, World!\n"<<std::endl;
end with '}', '@':abort > std::cout<<"What's up?\n"<<std::endl;
end with '}', '@':abort > }
Hello, World!
What's up?
```

Deuxième méthode :

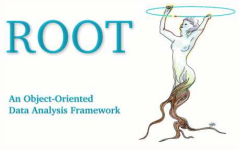
Script : fichier regroupant les commandes passées en ligne (Il est souhaitable de le code à la C++ → exemple01.C

```
#include <iostream.h>
int main()
{
std::cout<<"Hello, World!\n"<<std::endl;
std::cout<<"What's up?\n"<<std::endl;
}
```

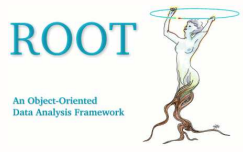
Pour l'executer à l'invite de ROOT :

```
root [0] .L example01.C
root [1] main()
Hello, World!
What's up?
```

Vous pouvez appeler un objet non déclaré, mais ROOT cherche



Chapitre 2 : SESSION INTERACTIVE



Revue de C++

Utilisation de la mémoire : variable, pointeurs, tableaux

Variable : stockage d'une information en vue de l'utiliser plus tard (donnée) . La taille de l'espace mémoire dépend du type de la variable : TYPE = NOM(INIT)

Type	Contenu
bool	vrai(true) ou faux(false)
char	Un caractère
int	Entier relatif
unsigned int	Entier positif ou nul
real	Réel
double	Réel en double précision
string	Chaine de caractères

Opérateurs	Actions
+,-,*,/	Opérations arithmétiques
==, !=, <, <=, >, >=	Opérations de comparaison
&&(ET), ||(OU), !(NON)	Opérations logiques
&, |, ^ , ∼	Opérations bit à bit
++, −	Incrém. et décrémentation
<<, >>, >>>	Décalage de bits
*,&, ., ->, [], (), new, delete	· · ·

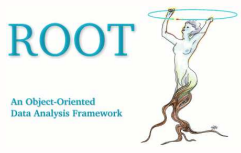
Pointeurs : TYPE *pt (&varType), c'est un pointeur vers varType. pt contient l'adresse de varType et *pt sa valeur.

Tableaux : succession de variables en mémoire.

$\Longrightarrow$ Statique : TYPE NOM [TAILLE] ={ · · · }, TYPE fonction(TYPE tableau[], int tailleTableau);

$\Longrightarrow$ Dynamique : std : :vector<TYPE> NOM(TAILLE)
std : :vector<TYPE>tableau : tableau.push_back(val), tableau.size(),

TYPE fonction(std : :vector<TYPE> tableau).



Revue de C++

Structures de contrôle : Conditions et Boucles

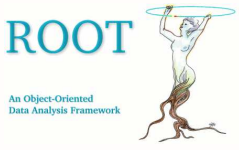
Conditions :

Prendre une décision : $\Longrightarrow$ **Conditions ;**

```
if(conditions)
{ Instructions ;}
else if(conditions)
{Instructions ;}
else{ Instructions ;}

switch (valeur)
{
    case val1 :
       Instructions ;
       break ;
:
:
    default :
       Instructions ;
       break ;
```

Boucles :

```
for(ideb ;ifin ;incrementation) { Instructions ;}

do
{
Instructions ; }while(conditions) ;

while(conditions)
{
Instructions ; }
```



Revu de C++

Flux

On parle de flux (stream) pour désigner les outils de communication d'un programme avec l'extérieur : flux vers les fichiers.

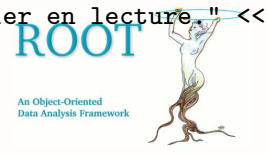
Le fichier entête qui contient les fonctions prototypes : fstream

```
#include <iostream> // flux vers écran
#include <fstream>  // flux vers fichier
using namespace std;

int main()
{
   // Nom du fichier
   string nomFichier("data.txt");

   //Déclaration du flux
   ofstream monFlux(nomFichier.c_str());

    if(monFlux)
    {
        monFlux << "Bonjour, j'écris dans un fichier." << endl;
    }
    else
    {
        cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
    }
    monFlux.close();
    return 0;
}
```

Exemple d'ecriture

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
   ifstream fichier("data.txt");

   if(fichier)
   {
      //L'ouverture s'est bien passée, on peut donc lire

      string ligne; //Une variable pour stocker les lignes lues

      while(getline(fichier, ligne)) //Tant qu'on n'est pas à la fin,
      {
         cout << ligne << endl;
         //Et on l'affiche dans la console
         //Ou alors on fait quelque chose avec cette ligne
         //À vous de voir
      }
   }
   else
   {
      cout << "ERREUR: Impossible d'ouvrir le fichier en lecture." <<
   }

   return 0;
}
```

Exemple de lecture



Revu de C++

Classe : Définition

Programmation orientée Objet $\Longrightarrow$ Notion de Classe :

- **Classe : Collection de variables -Attributs- et de procédures -Méthodes ;**
- **Objet : Instanciation de la classe $\longrightarrow$ Classe$\equiv$Type ;**

Définissons une classe "Particule" :

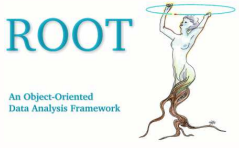
Particle.cxx

```
#include "Particule.h"
using namespace std;
// Constructeurs
Particule::Particule() : m_masse(0.), m_nom("")
{
}
Particule::Particule(double masse, string nom) : m_masse(masse),
m_nom(nom)
{
}
// Destructeur
Particule::~Particule()
{
}
// Afficher le nom et la charge de la particule
void Particule::afficher()
{
  cout<<"La particule "<<m_nom<<" a la masse "<<m_masse<<" GeV"<<endl;
}
// Modifier la masse de la particule
void Particule::modMasse(double masse)
{
m_masse = masse;
}
// Modifier le nom de la particule
void Particule::modNom(string nom)
{
m_nom = nom;
}
```

Déclarations : Classe * classe1, Classe classe2 : classe1->membre, classe2.membre

Particle.h

```
/***********
*   On décrit une particule par
*   m_masse    : masse
*   m_nom      : nom de la particule
************/
#ifndef DEF_PARTICULE  // Eviter plusieurs
#define DEF_PARTICULE  // inclusions
#include <iostream>    // input/ouput flux lib
#include <string>      // chaine de caract. lib

class Particule
{
    public:
// contructeur par défaut
        Particule();
// constructeur avec arguments
        Particule(double masse, double charge_e);
// Destructeur
        ~Particule();
// Afficher les propriétés de la particule
        void afficher();
// Modifier la masse de la particule
        void modMasse(double masse);
// Modifier la charge  de la particule
        void modNom(std::string nom);

    private:
        std::string m_nom;
        double m_masse;
};

#endif
```



Revue de C++

Classe : Constructeur et Destructeur

Notons deux méthodes particulières :

Constructeur : Méthode appelée automatiquement lors de la création d'un objet de la classe $\Longrightarrow$ Pas de type de retour, le même nom que la classe, et peut surchargée ;
$\Longrightarrow$ Initialiser les attributs de type élémentaire
Constructeur de copie : créer une copie d'un objet.

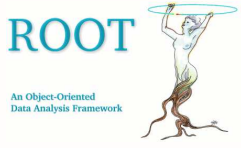
Destructeur : Méthode appelée automatiquement lorsqu'un objet est détruit ; Il ne peut pas être surchargé.

Particule.h

```
                    .
                    .
                    .
// Constructeurs
Particule();

//Surcharge
Particule(double masse, string nom);

// Constructeur de copie
Particule(Particule copie);

// Destructeur
Particule::~Particule()
{
}
```

Particule.cxx

```
                    .
// Constructeur       .
                    .
Particule::Particule() : m_masse(0.), m_nom("")
{
}
// surcharge du constructeur
Particule::Particule(double masse, string nom) : m
m_nom(nom)
{
}
// constructeur de copie
Particule::Particule(Particule copie) : m_masse(c
m_nom(copie.m_nom)
{
}
// Destructeur
Particule::~Particule()
{
}
```



Revue de C++
Encapsulation des données

Notons dans l'exemple précedent l'usage des mots clès : public , private.

Ces mots clès définissent l'autorisation d'accès aux membres de la classe :

- **public :** les membres sont accessibles à l'exterieur de la classe ;
- **private :** les membres ne sont accessibles qu'à l'interieur de la classe ;

Consdiérons le code

```
/***************
 * Encapsulation : les attributs sont déclarés "private" et ne
 *                 sont accessibles qu'à l'interieur de la classe
 *****************/
// Flux vers l'écran
#include <iostream>

// Chaine de caractères
#include <string>

// Prototype Particule
#include "Particule.h"

int main()
{
  std::string nom("muon");
  double masse_muon(102.4);
  Particule* meson = new Particule(masse_muon,nom);
  //meson->afficher();
  std::cout<<" Nom "<<meson->m_masse<<std::endl;
  return 0;
}
```

Le compilateur $\Longrightarrow$

```
error: 'm_masse' is a private member of 'Particule'
  std::cout<<" Nom "<<meson->m_masse<<std::endl;
                                ^
./include/Particule.h:47:10: note: declared private here
  double m_masse;
         ^
1 error generated.
```

Une troisième mot clé : protected : les membres sont accessibles aux classes dérivées "filles" (Voir la suite)

.

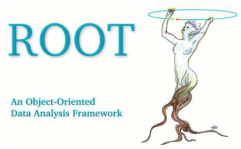
Revue de C++
Héritage

C'est l'une des propriétés importantes de la programmation orientée objet $\Longrightarrow$ Une technique qui permet de créer une classe (Fille/Dérivée) plus spécialisée à partir d'une classe existante (Mère/Base) : modèle de départ $\Longrightarrow$ éviter de réecrire le même code plusieurs fois.

Définissons une classe Quark, décrivant un quark par son nom, sa masse, sa saveur, sa charge de couleur, sa charge électrique fractionnaire : Notons que les deux premières propriétés sont définies dans la classe Particule. D'où

Prototype : Quark.h

```
//-----------------------------------
// Exemple d'Héritage
//-----------------------------------
/*******
 * Attributs :
 *  m_masse    : masse de la particule
 *  m_nom      : nom de la particule
 *  m_saveur   : saveur du quark (u, d, c, s, t, b)
 *  m_charge_elec   : 1/3, 2/3
 *  m_charge_coul   : bleu, jaune et rouge
 *
 * Méthodes :
 *  afficher()          : affiche les attributs de la particule
 *  modNom(nomNew)      : modifie le nom de la particule à nomNew
 *  modMasse(masseNew)  : modofie la masse de la particule à masseNew
 *  modChargeElec(charge_e) : modifie la charge électrique
 *  modChargeCouleur(charge_c) : modifie la charge de couleur
 *  modSaveur(saveur) : modifie la saveur
 *************/
// Prototype de la classe
#ifndef DEF_QUARK  // Eviter plusieurs
#define DEF_QUARK  // inclusions
#include <iostream>     // input/ouput flux lib
#include <string>       // chaine de caract. lib
// Inclure le prototype de la classe Particule :
#include "Particule.h"

// Créer une classe Quark héritant la classe Particule
class Quark : public Particule
{
 public:
  // contructeur par défaut
  Quark();

  // surcharge du constructeur
  Quark(double masse, double charge_elec, \\
        std::string charge_coul, std::string nom, std::string saveur);

  // Constructeur par copie
  Quark(Quark const& copie);

  // Destructeur
  ~Quark();

  // Masquage de la méthode afficher
  void afficher();

  // Modifier la charge électrique
  void modChargeElectrique(double charge_elec);

  // Modifier la charge  de couleur
  void modChargeCouleur(std::string couleur);

 protected:
  std::string m_saveur;
  double m_charge_elec;
  std::string m_charge_coul;

};
```

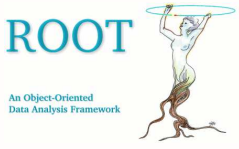


Revue de C++
Héritage

Implémentation : Quark.cc

```
      // Prototype de la classe
#include "Quark.h"

// utiliser le nom d'espace std
using namespace std;

// Constructeur
Quark::Quark() : Particule(), m_charge_elec(0.),
m_charge_coul(""), m_saveur("")
{
}
// surcharge du constructeur
Quark::Quark(std::string nom, double masse,  std::string saveur,
double charge_elec, std::string charge_coul) :
Particule(masse, nom), m_charge_elec(charge_elec),
m_charge_coul(charge_coul), m_saveur(saveur)
{
}

// Constructeur de copie
Quark::Quark(Quark const& quark) :
Particule(quark.m_masse,quark.m_nom)
{
  m_saveur      = quark.m_saveur;
  m_charge_elec = quark.m_charge_elec;
  m_charge_coul = quark.m_charge_coul;
}

// Destructeur
Quark::~Quark()
{
}
```

```
// Afficher le nom et la charge de la particule
void Quark::afficher()
{
  cout<<" Nom                          "<<m_nom<<endl;
  cout<<"Quark  : saveur               "<<m_saveur<<endl;
  cout<<"         masse                "<<m_masse<<" GeV"<<endl;
  cout<<"         charge électrique    "<<m_saveur<<endl;
  cout<<"         charge couleur       "<<m_charge_coul<<endl;
}
// Modifier la masse de la particule
void Quark::modChargeElectrique(double charge_elec)
{
m_charge_elec = charge_elec;
}
// Modifier la charge de couleur
void Quark::modChargeCouleur(std::string charge_coul)
{
m_charge_coul = charge_coul;
}
// Modifier la saveur
void Quark::modSaveur(std::string saveur)
{
m_saveur = saveur;
}
```



Revue de C++

Héritage : Exemple

Fichier principal : heritage.cxx

```
/***************
 * Héritage
 *
 *****************/
// Prototype Particule
#include "Quark.h"

int main()
{
  std::string nom_muon("muon");
  double masse_muon(102.4);
  Particule* meson = new Particule(masse_muon,\
      nom_muon);
  meson->afficher();
  //
  std::string nom_quark("quark");
  double masse_quark(1.5);
  std::string saveur("up");
  std::string charge_coul("yellow");
  double charge_elec(2./3.);
  Quark* quark = new Quark(nom_quark, masse_quark, \
      saveur, charge_elec, charge_coul);
  quark->afficher();
  return 0;
}
```

Pour l'exécuter :

$\Longrightarrow$ **Makefile :**

```
make mainP=heritage; ./heritage.exe
```

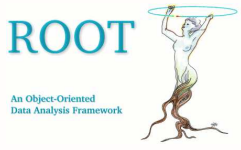
$\Longrightarrow$ **Commande en ligne : (src/*.cc, src/*.cxx et ./*.obj)]**

```
g++ -c src/heritage.cxx src/Quark.cc /src/Particule.cc
g++ -o heritage.exe heritage.o Quark.o Particule.o
./heritage.exe
```

Résultat :

```
[./examples ] ./example_classe.exe
La particule pion a la masse 0.144 GeV
La particule baryon a la masse 1 GeV
```

Notons que la méthode afficher() a été redéfinie : On parle de Masquage. Quark : :aficher() masque Particule : :afficher().



Revue de C++

Héritage : Masquage et Démasquage

Résumons :

Déclarations : • classe classeFille : public classeMere
$\Longrightarrow$ Déclaration de la classe fille ;
• classeFille() ;
$\Longrightarrow$Constructeur par défaut ;
• classeFille(type argMere, type argFille) ;
$\Longrightarrow$Constructeur surchargé ;
• ClasseFille(ClassFille const& copie) ;
$\Longrightarrow$Constructeur par copie ;
• void afficher() ;
$\Longrightarrow$Redéfinition de afficher() : Masquage ;

La classe Fille hérite, en accès public, tous les membres publics et protégés de la classe Mère.

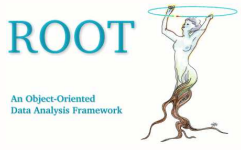
On peut affecter un objet classeMere à un objet classeFille et non l'inverse.

Implémentations :

```
•classeFille : :classeFille() : classeMere() ;
• classeFille : :classeFille(type argFille, type argFille) :
classeMere(argMere), m_argFille(argFille) ;
{
}
• classeFille : :classeFille(classeFille const& copie) :
classeMere(copie.argMere)
{
m_argFille = copie.argFille ;
}
• void classeFille : :afficher()
{
Suppléments ;
}
```

Demasquage : Si les redéfinitions consistent à compléter la méthode de la classeFille, on parle de demasquage.

```
• void classeFille : :afficher()
{
classeMere : :afficher() ;
Redéfinitions ;
}
```

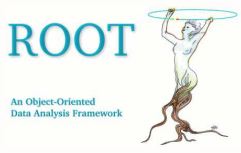


Revue de C++
Polymorphisme

Polymorphisme (grec) : qui peut prendre plusieurs formes $\Longrightarrow$ Concept essentiel de la programmation orientée objet utilisé sur les fonctions, méthodes et les opérateurs.

On distingue deux type de polymorphismes :

- **ad-hoc :** mécanisme de surcharge des méthodes et des opérateurs ;
- **universel :** on en distingue
  - **paramétrique :** un même code peut être appliqué à n'importe quel type $\Longrightarrow$ Code générique ($\equiv$ Template) ;
  - **d'inclusion :** un même code peut être appliqué à des données de types différents liées par une relation d'héritage.



Revue de C++
Polymorphisme : ad-hoc (Surcharge)

On a déjà utilisé ce concept :

Surchage de fonction ou de méthode
Prototype :

```
...
void swap(int& , int&);
void swap(float&, float&);
```

Signature d'une méthode : type retour, le nombre et les types d'argument.

Surchage d'opérateurs
Prototype :

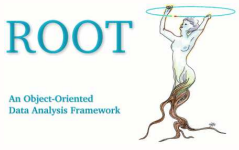
```
...
classe1 classe1::operatorOP(classe1& obj);
...
```

Surchage de fonction ou de méthode
Implémentation :

```
...
void swap(int& val1, int& val2)
{
int temp; temp=val1; val1=val2; val2=temp;
}
void swap(float& val1, float& val2);
{
float temp; temp=val1; val1=val2; val2=temp;
}
```

Surchage d'opérateurs
Implémentation :

```
classe1 classe1::operatorOP(classe1& obj)
{
 this->m_attribut1 OP obj.attribut1;
 this->m_attribut2 OP obj.attribut2;
        ... ;
 return this;
}
```



Revue de C++

Polymorphisme : universel-paramétrique

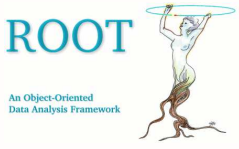
Dans le cas de la surcharge, on définit autant de méthodes que de cas différant par leurs signatures ;

$\Longrightarrow$ On peut définit une seule méthode mais dont les types restent flottants : On peut définir un patron ;

$\Longrightarrow$ mot-clé = template ;

```
/**********
 * Utilisation de template
 ************/
// Prototype

// On utilise typename
// On peut utiliser class aulieu de typename
// template<class Type> Type calculerSomme\
//          (Type operande1, Type operande2)
template<typename Type> Type calculerSomme\
          (Type operande1, Type operande2)
{
    Type resultat = operande1 + operande2;
    return resultat;
}
```

```
/*************
 * Utilisation du polymorphisme paramétrique
 ********/

#include <iostream>
#include "patron.h"

using namespace std;

int main()
{
  int n(4), p(12), q(0);
// On appelle notre fonction calculerSomme comme
  q = calculerSomme(n, p);
  cout << "Le resultat est : " << q << endl;

    return 0;
}
```



Revue de C++

Polymorphisme d'inclusion : résolution statique

Il s'agit d'une redéfinition d'une méthode d'une classe fille. Illustrons par l'exemple suivant :

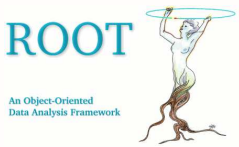
pm_statique.h

```
 class A
{
public:
   void print()
   {
      std::cout << "A" << std::endl;
   }
// ...
};
 class B : public A
{
public:
   void print()
   {
      std::cout << "B" << std::endl;
   }
// ...
};
```

On note que c'est la méthode de la classe A qui est invoquée même si un objet de la classe B est passé en argument. ⟶ Résolution des liens statique.

example_pmSt.cxx

```
#include <iostream>
#include "pm_statique.h"

void execution(A const& obj)
{
   std::cout << "On est dans la classe : ";
   obj.print();
}

int main( void )
{
  A *a1 = new A();
  execution(*a1);
  cout<<"--------------------"<<endl;

  B *b1 = new B(); // ...
  execution(*b1);

  delete a1;
  delete b1;
  //
}
```



Revue de C++

Polymorphisme d'inclusion : résolution dynamique

On aurait souhaiter dans l'exemple précédent que ce soit la méthode print de la classe B qui soit exécutée : que la résolution ait lieu au moment de l'appel $\Longrightarrow$ Résolution dynamique.

Pour ce faire, il suffit de déclarer la méthode concernée de la classe mère précédée du mot clé **virtual** :

la méthode est dite virtuelle.

Réécrivons le code. Notons qu'il suffit de déclarer vritual seule la méthode de la classe mère.

```
 class A
{
public:
   virtual void print()
   {
      std::cout << "A" << std::endl;
   }
// ...
};
```

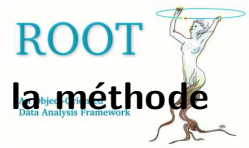
L'exécution de example_pm01 donne le résultat

```
./example_pm01.exe
On est dans la classe : A
--------------------
On est dans la classe : B
```

On note bien que cette fois-ci c'est la classe B qui est affichée et non A.

Quelques remarques :

- Le mot clé virtual peut être déclaré également devant la méthode en question de la classe fille ;
- La méthode de la classe fille doit avoir la même signature que la méthode virtuelle ;
- La virtualité impose l'utilisation de pointeur ou de référence ;
- Dans le cas où le type de retour de la classe virtuelle est la classe mère, l'implémentation de la méthode de la classe fille renvoie le type fille : on parle dans ce cas d'un type retour covariant.
- azure Le destructeur de la classe mère doit être déclaré virtual.



Revue de C++

Polymorphisme d'inclusion : Méthode virtuelle pure - Classe abstraite

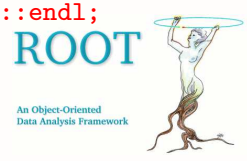
Rappelons l'intérêt de la virualité : l'ancien code peut appeler le nouveau code !

Prenons l'exemple suivant :

```
FormeGeometrique.h
class FormeGeometrique
{
   public :
     FormeGeometrique();
     FormeGeometrique(const int& );
     virtual ~FormeGeometrique();
     virtual float surface() const;
   protected :
     int m_nbCotes;
};
Rectangle.h
#include "FormeGeometrique"
class Rectangle : public FormeGeometrique
{
   public :
     Rectangle();
     Rectangle(float& , float&);
     float surface() const;
     void afficher() const;
   protected :
     float m_longueur, m_largeur;
};
```

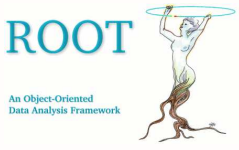
On note bien que l'on ne

```
FormeGeometrique.cc
#include <iostream>
#include "FormeGeometrique"
FormeGeometrique::FormeGeometrique() :m_nbCotes(4)
{}
FormeGeometrique::FormeGeometrique(const int& nbCotes) : \
   m_nbCotes(nbCotes)
{}
FormeGeometrique::~FormeGeometrique();
float FormeGeometrique::surface() const
{ ? }
Rectangle.h
#include "Rectangle.h"
Rectangle::Rectangle() : FormeGeometrique(), \
    m_longueur(10.), m_largeur(5.) {}
Rectangle::Rectangle(float& longueur, float& largeur) : \
    FormeGeometrique(), m_longueur(longueur), \
    m_largeur(largeur) {}
float Rectangle::surface()
{return m_longueur*m_largeur;}
void Rectangle::afficher()
{
  std::cout<<"Rectangle : longueur="<<m_longueur<<std::endl;
  std::cout<<"            largeur ="<<m_largeur<<std::endl;
  std::cout<<"            surface ="<<surface()<<std::endl;
}
```



Chapitre 2 : SESSION INTERACTIVE

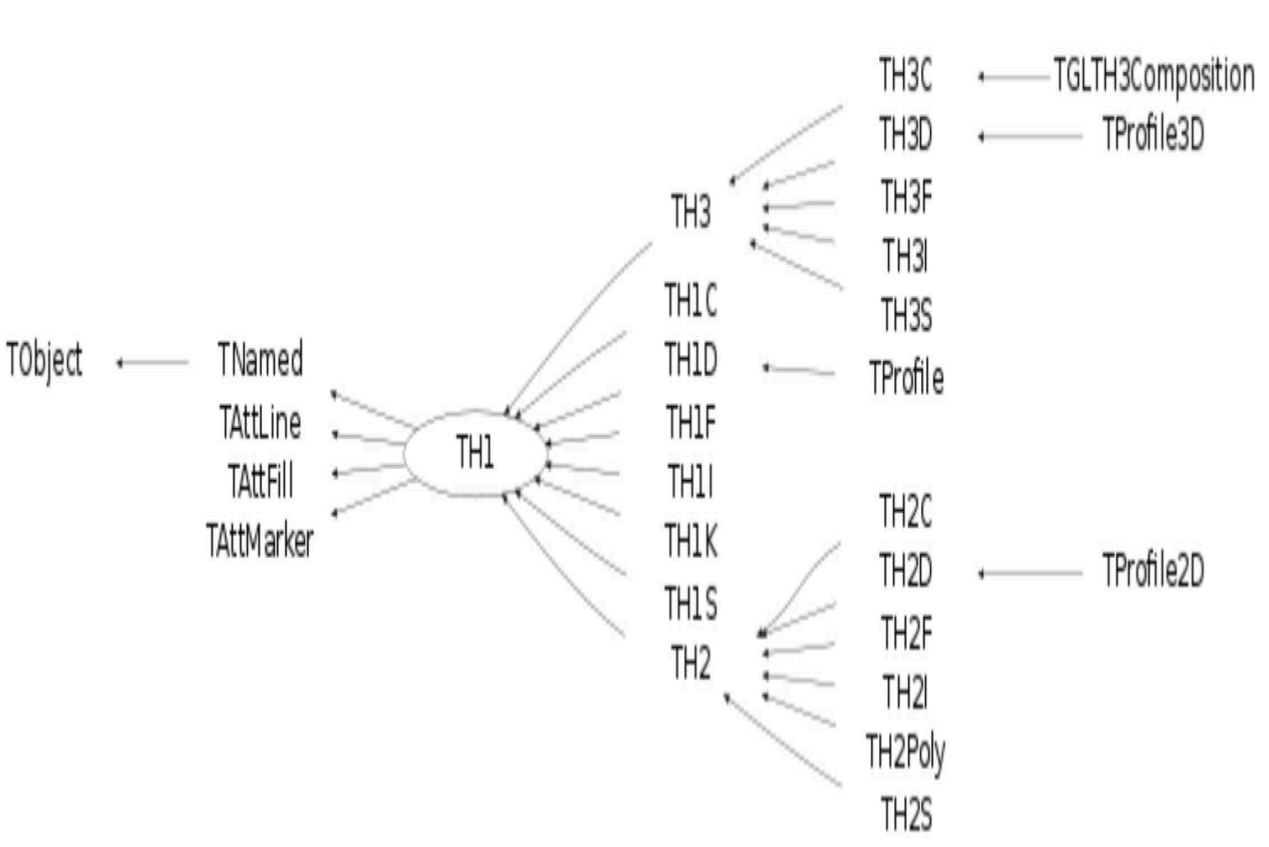
1. Environnement de ROOT
2. Interpréteur C/C++ : CINT/CLING
   - Quelques commandes CINT /CLING
3. Revue de C++
   - Variable, Pointeurs, Tableaux
   - Structures de contrôle : Conditions et Boucles
   - Lire et écrire sur un fichier
   - Classe, Encapsulation, Héritage et polymorphisme
     - Classe
     - Encapsulation des données
     - Héritage : Masquage et Démasquage
4. Histogrammes



Histogrammes

Hiérarchie des classes

Création $\Longrightarrow$ TH1F hist(nom, titre, Nb bins, Xmin, Xmax) ;
Remplissage $\Longrightarrow$ hist.Fill(variableX, poids) ;
Ecriture $\Longrightarrow$ hist.Write() ;



Opérations :

hist->Add(hist2), hist->Divide(hist2), hist->Multiply(hist2))

Histogrammes : Lissage

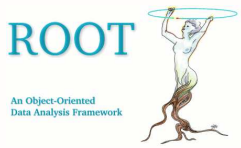
Expression des fonctions de Lissage : TFormula

L'expression analytique d'une fonction peut être construite en utilisant la classe TFormula dont les constructeurs sont donnés par :

```
TFormula TFormula()
TFormula TFormula(const char* name, const char* formula)
TFormula TFormula(const TFormula& formula)
```

Exemple :

```
// Construire l'objet tf =a*exp(-b t)
   TFormula *myFormula = new TFormula("", "[0]*exp(-[1]*x)");
// Initialiser les paramètres a=[0], b[1]
   myFormula->SetParameter(0, 1);
   myFormula->SetParameter(1, 0.5);
// Evaluer la valeur en x=10
  myFormula->Eval(10.);
```



Histogrammes : Lissage

Expression des fonctions de Lissage : TF1, TF2, TF3

L'expression analytique d'une fonction peut être construite en utilisant la classe TF1 dont les constructeurs sont donnés par :

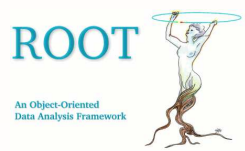
```
TF1 TF1()
TF1 TF1(const char* name, const char* formula, Double_t xmin = 0, Double_t xmax = 1)
TF1 TF1(const char* name, Double_t xmin, Double_t xmax, Int_t npar)
TF1 TF1(const char* name, void* fcn, Double_t xmin, Double_t xmax, Int_t npar)
TF1 TF1(const char* name, ROOT::Math::ParamFunctor f, Double_t xmin =0,
        Double_t xmax = 1, Int_t npar = 0)
TF1 TF1(const char* name, void* ptr, Double_t xmin, Double_t xmax,
        Int_t npar, const char* className)
TF1 TF1(const char* name, void* ptr, void*, Double_t xmin, Double_t xmax,
       Int_t npar, const char* className, const char* methodName = 0)
TF1 TF1(const TF1& f1)
```

Exemple

```
// Construire l'objet a+b*sin(x)/x
   TF1 *myFunction = new TF1("myFunction","[0]+[1]*sin(x)/x",0,10);
// Initialiser les paramètres de l'objet
   myFunction->SetParameter(0 , 1.); // On peut la même chose comme suit
   myFunction->SetParameter(1 , .5); //myFunction->SetParameters(1.,0.5)
// Une méthode de TH1
   myFunction->SetLineColor(kBlue);
   myFunction->Draw();
```

Quelque fonction prédéfinie :

```
"gaus"  : A gaussian with 3 parameters: f(x) = [0]*exp(-0.5*((x-[1])/[2])^2)).
"expo"  : An exponential with 2 parameters: f(x) = exp([0]+[1]*x).
"poiN"  :  A polynomial of degree N: f(x) = [0] + [1]*x + [2]*x^2 +...
```



Histogrammes : Lissage

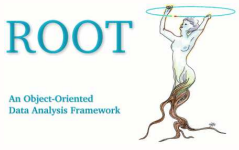
TH1 : :Fit

Par défaut, la méthode TH1 : :Fit permet de lisser un histogramme par une fonction définie par TF1 dont le prototype :

```
TH1::Fit(const char* formula, Option_t* option = "",
         Option_t* goption = "", Double_t xmin = 0,
         Double_t xmax = 0)  // *MENU*
TH1::Fit(TF1* f1, Option_t* option = "", Option_t* goption = "",
         Double_t xmin = 0, Double_t xmax = 0)
```

Accéder aux paramètres de la fonction de lissage :

```
TF1 *fit = hist->GetFunction(function_name);
Double_t chi2 = fit->GetChisquare();
Double_t p1 = fit->GetParameter(1);
Double_t e1 = fit->GetParError(1);
```



Tuples : TTree et TNtuple

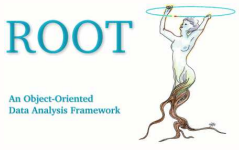
TTree : Création et écriture dans un fichier ROOT

Manipuler une large quantité de données relatives à des objets $\Longrightarrow$ Possibilité d'optimisation de gestion de stockage et d'accès de ces données : TTree et TNtuple.

TNtuple est une classe simplifiée de TTree : Limitée aux "floats".

Exemple : Supposons que l'on souhaite créer un tuple décrivant les composantes de l'impulsion d'une particule :

```
// Creation de l'objet TTree
  TTree* myTree = new TTree("myTree","Composantes de l'impulsion px:py:pz");

// Définition des branches
  myTree->Branch("px",&px,"px/F");
  myTree->Branch("py",&py,"px/F");
  myTree->Branch("pz",&pz,"px/F");

// Remplissage
  for(..;..;..)
    { ...;
      myTree->Fill();
     ... }
// Print de la structure de TTree
   myTree->Print()

// Ecriture dans un fichier ROOT
   myTree->Write();
```

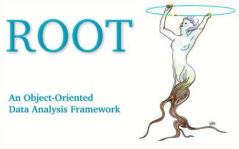


Tuples : Ntuple et Tree

Cas de TNtuple

Dans le cas de TNtuple :

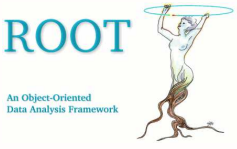
```
// Création de l'objet
   NTuple* myNtuple = new NTuple("myNtiple","Composantes de
                                          l'impulsion","px:py:pz")

// Remplissage
   for(...;...;...)
      {...;
       myNtuple->Fill(x,y,z);
       ...;}

//
```



Tuples : TTree et TNtuple

Types de données

- **C :** Caractère ;
- **B :** Entier signé 8 bits ;
- **I :** Entier signé 32 bits ;
- **L :** Entier long 64 bits ;
- **F :** Réel simple précision 32 bits ;
- **D :** Réel double précision 64 bits ;
- **O :** Variable booléenne.

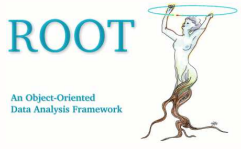


Tuples : TTree et TNtuple

Relecture d'un TTree

On se propose de relire le tuple précédent crée par TTree :

```
// Fichier contenant le tuple
   TFile* hfile = new TFile("fichier_ttree.root");

// Créer l'objet TTree : Donner le même nom que celui à lire
   TTree * myTree = (TTree*)hfile->Get("myTree");

// déclarer les variables
   float px,py,pz;

// Associer les variables avec les branches
   myTree->SetBranchAddress("px",&px);
   myTree->SetBranchAddress("py",&py);
   myTree->SetBranchAddress("pz",&pz);

// Npmbre d'entrée disponibles
   Long64_t nentries = myTree->GetEntries();

// Boucler sur les éléments du tuple
   for(Int_t i=0; i<nentries;i++)
     {// extraire le contenu
         myTree->GetEntry(i);
      // possibilité d'utiliser px, py et pz
         cout<<" px = "<<px<<" py = "<<py<<" pz = "<<pz<<endl;
      ...
     }
```

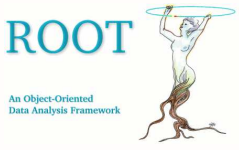


Tuples : TTree et TNtuple

Quelques méthodes de TTree : TTree : :Print()

Imprimer la structure du Ntuple :

```
// ouvrir le fichier contenant le Ntuple
   --> auto hfile = new TFile("dataTree.root")
(class TFile*)0x7fbc4357e7d0

// lister le contenu
   --> hfile->ls()
TFile** dataTree.root
 TFile* dataTree.root
  KEY: TTree myTree;1 Composantes de l'impulsion px, py et pz

// Imprimer la structure
   --> myTree->Print();
******************************************************************************
*Tree    :myTree    : Composantes de l'impulsion px, py et pz                *
*Entries :   100000 : Total =         1301047 bytes  File  Size =    1105255 *
*        :          : Tree compression factor =   1.09                       *
******************************************************************************
*Br    0 :px        : px/F                                                   *
*Entries :   100000 : Total  Size=     433551 bytes  File Size  =     372806 *
*Baskets :       13 : Basket Size=      32000 bytes  Compression=   1.08     *
*.............................................................................*
*Br    1 :py        : px/F                                                   *
*Entries :   100000 : Total  Size=     433551 bytes  File Size  =     372752 *
*Baskets :       13 : Basket Size=      32000 bytes  Compression=   1.08     *
*.............................................................................*
*Br    2 :pz        : px/F                                                   *
*Entries :   100000 : Total  Size=     433551 bytes  File Size  =     358886 *
*Baskets :       13 : Basket Size=      32000 bytes  Compression=   1.12     *
*.............................................................................*
```

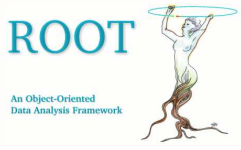


Tuples : TTree et TNtuple

Quelques méthodes de TTree : TTree : :Show(Int_t)

Lister les valeurs d'une entrée :

```
// ouvrir le fichier contenant le Ntuple
   --> auto hfile = new TFile("dataTree.root")
(class TFile*)0x7fbc4357e7d0

// lister le contenu
   --> hfile->ls()
TFile** dataTree.root
 TFile* dataTree.root
  KEY: TTree myTree;1 Composantes de l'impulsion px, py et pz

// Accéder au ntuple
   --> TTree* myTree = (TTree*)hfile->Get("myTree");
// Lister le contenu du 10 evenement
   --> myTree->Show(10)
======> EVENT:10
 px              = 0.411818
 px              = 0.868943
 px              = 0.96159
```

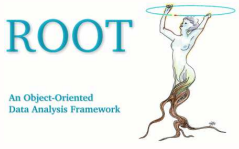


Tuples : TTree et TNtuple

Quelques méthodes de TTree : TTree : :Scan(“var1 :var2 :var...”)

Scaner les valeurs de certaines variables :

```
// ouvrir le fichier contenant le Ntuple
   --> auto hfile = new TFile("dataTree.root")
(class TFile*)0x7fbc4357e7d0

// lister le contenu
   --> hfile->ls()
TFile** dataTree.root
 TFile* dataTree.root
  KEY: TTree myTree;1 Composantes de l'impulsion px, py et pz

// Scaner
   --> myTree->Scan("px:py:pz");
************************************************
*    Row   *        px *        py *        pz *
************************************************
*        0 * -0.767548 * 0.6730216 * 1.0208275 *
*        1 * -3.062916 * 2.7595090 * 4.1226625 *
*        2 * 3.6199505 * -3.278217 * 4.8837232 *
*        3 * -15.45976 * 4.8703641 * 16.208784 *
*        4 * 0.4056560 * -5.879614 * 5.8935918 *
*        5 * 1.9033480 * -5.288165 * 5.6202693 *
*        6 * 5.0237937 * 1.9933451 * 5.4048061 *
*        7 * -4.917469 * -1.092992 * 5.0374736 *
```

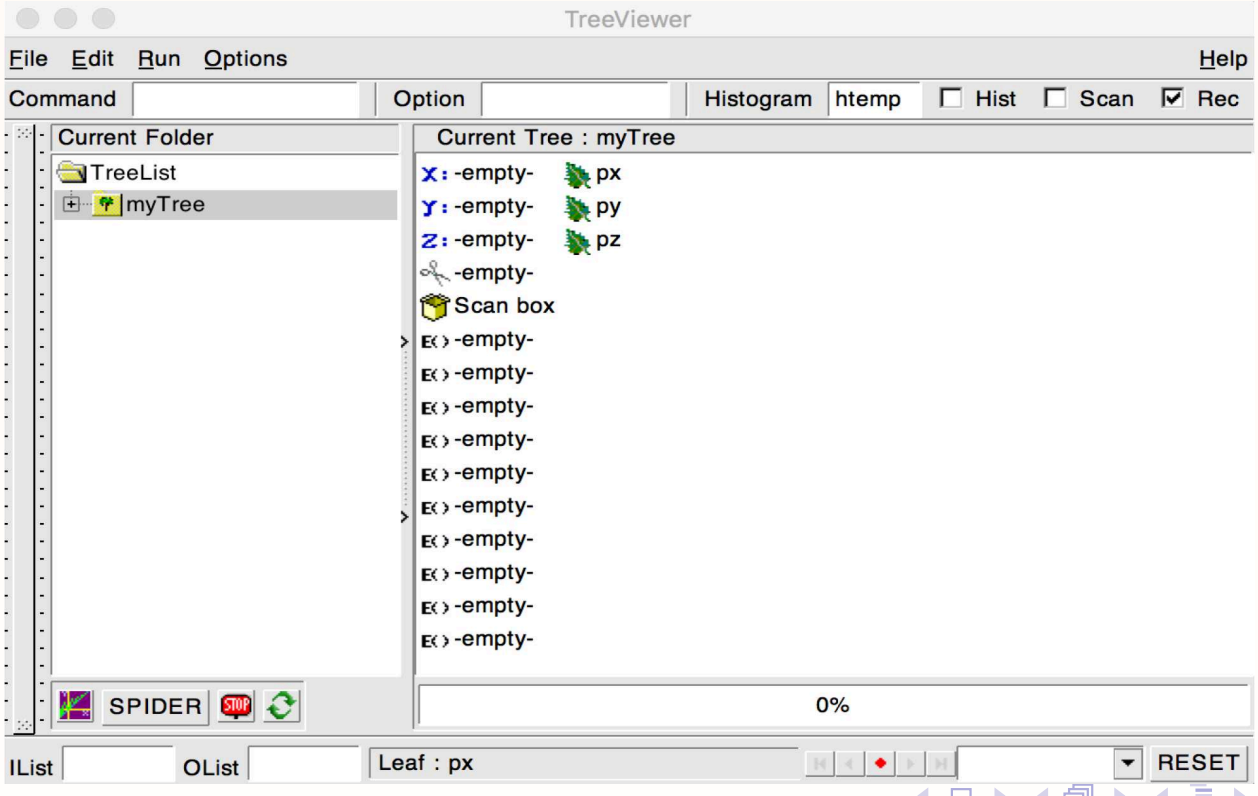


Tuples : TTree et TNtuple

Quelques méthodes de TTree : TTree : :StartViewer()

Scaner les valeurs de certaines variables :

```
.....

// Viewer
   --> myTree->StartViewer();
```

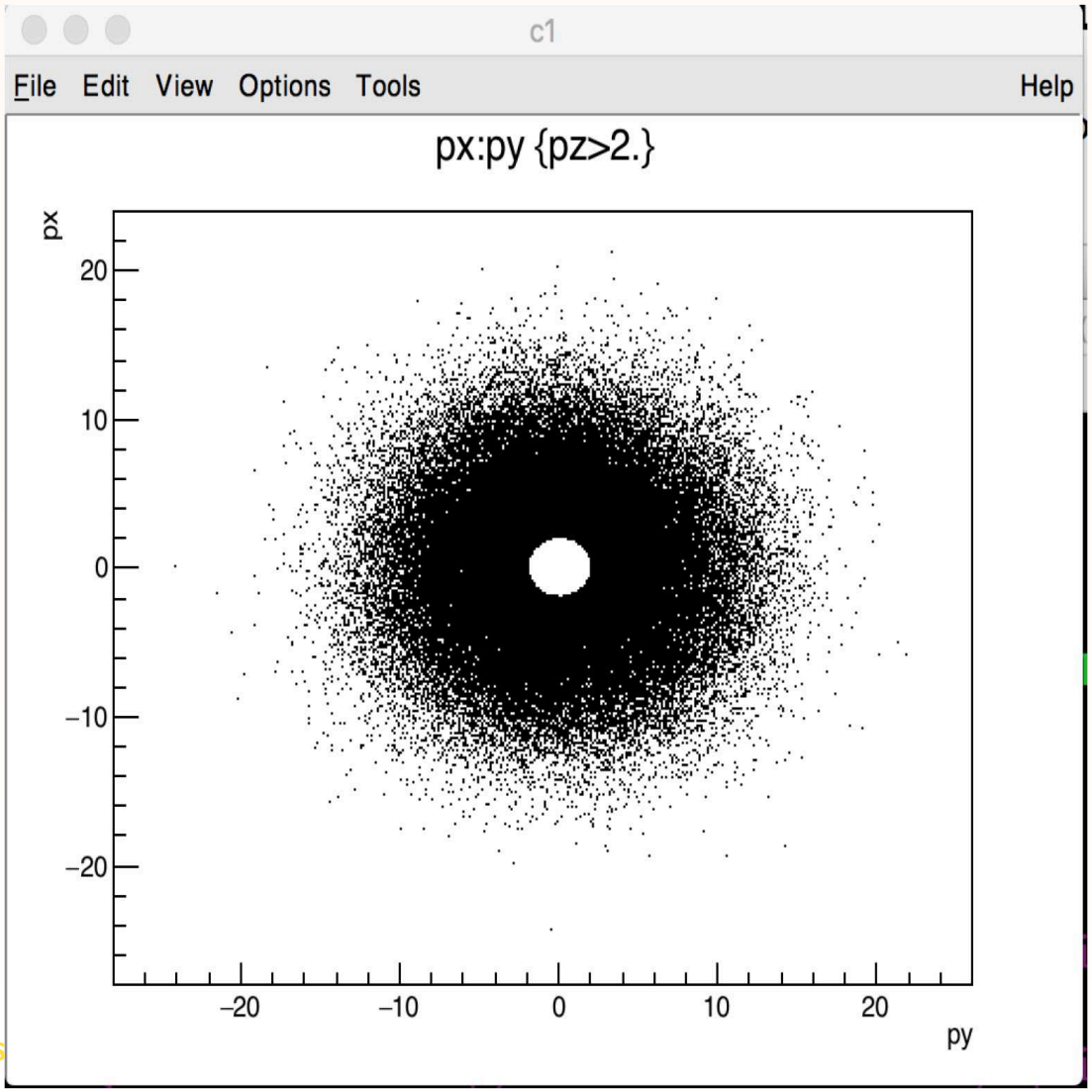


Tuples : TTree et TNtuple

Quelques méthodes de TTree : TTree : :Draw(....)

Tracer un graphique **TTree : :Draw (const char* varexp, const TCut& selection, Option_t* option = "", Long64_t nentries = 1000000000, Long64_t firstentry = 0)**

```
myTree->Draw("px:py","pz>2.","",100000);
```



Tuples : TTree et TNtuple

Quelques méthodes de TTree : TTree : :MakeClass

Pour faire une analyse hors ligne : création de classe associée au Ntuple :

```
...
// Création de la classe associée au Ntuple
   --> myTree->MakeClass()
Info in <TTreePlayer::MakeClass>: Files: myTree.h and myTree.C generated from TTree: myTree
```

myTree.h :

```
//////////////////////////////////////////////////////////
// This class has been automatically generated on
// Thu Jun  7 16:04:37 2018 by ROOT version 5.34/36
// from TTree myTree/Composantes de l'impulsion px, py et pz
// found on file: dataTree.root
//////////////////////////////////////////////////////////

#ifndef myTree_h
#define myTree_h

#include <TROOT.h>
#include <TChain.h>
#include <TFile.h>

// Header file for the classes stored in the TTree if any.

// Fixed size dimensions of array or collections stored
// in the TTree if any.

class myTree {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain
.....
```

myTree.cxx :

```
#define myTree_cxx
#include "myTree.h"
#include <TH2.h>
#include <TStyle.h>
#include <TCanvas.h>

void myTree::Loop()
{
//   In a ROOT session, you can do:
//      Root > .L myTree.C
//      Root > myTree t
//      Root > t.GetEntry(12); // Fill t data members with
//                             entry number 12
//      Root > t.Show();       // Show values of entry 12
//      Root > t.Show(16);     // Read and show values of entry 16
//      Root > t.Loop();       // Loop on all entries
//
.....
```