Overview on C++, ROOT (CERN) and Make

presentation and exercises

Marco Colonna

Programming Curses 2025
March 20th

Overview

- **C++**
  - Why should you understand and learn C++?
  - C++ methods and fundamental topics
- **ROOT**
  - Basic concepts of ROOT
  - Using TTrees and TFiles
  - Using the TBrowser
  - What is a Macro?
- **Make**
  - Fundamentals of `make`

What is C++?

- One of the **most popular programming languages**, an evolution of C
- **Object-Oriented**: Supports modular, reusable code through classes and objects:
  - powerful development features like inheritance, polimorphisms...
- Good for **High-Level** (create object and play with them) and **Low-Level** (direct memory manipulation and hardware access) programming:
  - it is extremely flexible.

If you ever used:

- Used for, e.g., operating systems (Windows), Photoshop, Spotify, software in medical physics...
- ROOT, Geant4, Bash, make...

Than you trust C++ already!

Pros and Cons of C++

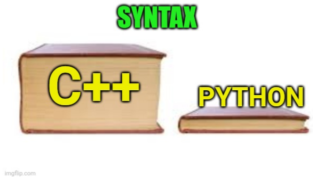
Why to use C++?

- Very efficient since it is not interpreted (unlike Python).
- Often around 100 times faster than pure Python code.

What do you have to pay?

- Not the most beginner-friendly; writing good code takes time.
- (Often) More difficult debugging, syntax is not very forgiving.

Use as much C++ as needed, as much python as possible!



Compilation

- **C++ Code Execution:**
  - C++ scripts cannot be executed directly.
  - A compiler (e.g., g++) is required to generate an executable file.
  - Pre-installed on Linux.
- **What is Compilation?**
  - Compilation is the process of converting C++ source code into an executable program.
- **Steps of Compilation:**
  - **Preprocessing:** Handles directives like `#include` and `#define`.
  - **Compilation:** Translates source code into assembly code.
  - **Assembly:** Converts assembly code into machine code (object files).
  - **Linking:** Combines object files and libraries to create the final executable.
- **Compilation Syntax:**
  - `g++ <input file> <maybe additional files> -o <output file>`
- → **Produces the final executable** `<output file>`

Compilation Example: Triangle Class

- **Why Separate Compilation?**
  - Improves compilation speed—only modified files need recompiling.
  - Helps manage large projects with multiple source files.
- **Compile Separately:**
  - `g++ -c main.cpp` # generates standalone object file (.o)
  - `g++ -c triangle.cpp`
- **Link Object Files:**
  - `g++ main.o triangle.o -o runMain`
- **Execute:**
  - `./runMain`
- **Alternatively, Compile and Link in One Step:**
  - `g++ -o runMain main.cpp triangle.cpp`
  - → **Does not create .o files separately**

Brief C++ overview... what do we have?

- **Variables and Types:**
  - C++ requires explicit variable type declaration: `int`, `float`, `char`, `bool`, etc.
  - Example: `int x = 5;`
- **Control Structures:**
  - Similar to Python, C++ supports `if`, `else`, `for`, `while` loops.
  - Syntax: `if (x > 0) { ...}`
- **Arrays:**
  - C++ arrays are of fixed size: `int arr[5] = {1, 2, 3, 4, 5};`
  - Unlike Python lists, arrays cannot change size after initialization.
- **Error Handling:**
  - C++ uses `try`, `catch`, and `throw` for exception handling.
  - Example: `try { throw 10; } catch (int e) { std::cout << "Error: " << e << std::endl; }`

Find everything in the https://en.cppreference.com/w/

Console: Input/Output

- **C++ Input/Output (I/O) requires:**
  - `#include <iostream>` at the beginning of the program.
  - The `std` namespace can be used or omitted with `std::`.
- **Standard Output (`cout`):**
  - Used to display output on the console.
  - Example: `std::cout << "Hello World" << std::endl;`
- **Standard Input (`cin`):**
  - Used to read input from the keyboard.
  - Example: `int x; std::cin >> x;`

Example Program:

```
#include <iostream>
int main() {
  int age;
  std::cout << "Enter your age:  ";
  std::cin >> age;
  std::cout << "You are " << age << " years old." << std::endl;
  return 0;
}
```

Pointers

- A pointer stores the **memory address of an object**.
- **Dereferencing** a pointer with '*' to **access the object pointed**.
- Grants **efficient direct memory manipulation**.
- Pointer arithmetic allows **navigation through memory** by adding offsets.

Examples of code:

```
int main() {
int myArray[5] = {0, 1, 2, 3, 4};
int *ptr = myArray;
for (int i = 0; i < 5; i++) {
*(ptr + i) = i * 2;
}
for (int i = 0; i < 5; i++) {
std::cout << myArray[i] << " ";
}
return 0;
}
```

```
int main() {
int *ptr = new int;
*ptr = 42;
std::cout << "Value:  " << *ptr << std::endl;
delete ptr;
return 0;
}
```

Functions in C++

- A function is a **block of code** that performs a specific task.
- **Function syntax** consists of:
  - **Return type**: The type of value the function will return (e.g., `int`, `float`, ...).
  - **Function name**: The name of the function (e.g., `add`).
  - **Parameters** (optional): The values passed to the function (e.g., `int a, int b`).
  - **Return statement**: The value returned by the function (optional depending on the return type).

Example: Function that adds two integers

```
#include <iostream>
int add(int a, int b) {
  return a + b;
}
int main() {
  int result = add(3, 4);
  std::cout << "Sum:  " << result << std::endl;
}
```

Classes in C++

- A class is a blueprint for creating objects, defining attributes and methods.
  - **Attributes:** Variables that store the object's data.
  - **Methods:** Functions that define object behavior.
- Access specifiers control visibility:
  - **Public:** Accessible from anywhere.
  - **Private:** Accessible only inside the class.
  - **Protected:** Similar to private, but accessible in derived classes.
- A **constructor** is a method that runs when an object is created, to initialize attributes.
- Classes can **inherit** from other classes, allowing code reuse and hierarchy creation.
  - Example: A general `Animal` class can be inherited by a `Dog` class, which is further inherited by a `Dalmatian` class.
  - The derived class (`Dog`) has all methods from the base class (`Animal`) and can define additional ones.

Classes: Implementation

- Class implementation is typically divided into:
  - A **header file** (`.h`) that defines the class structure.
  - A **source file** (`.cpp`) that implements the class methods.
- Example:
  - `MyClass.h` - Header file defining the class structure.
  - `MyClass.cpp` - Implementation file defining the methods.

Header File (MyClass.h)

```
class MyClass {
private:
float myVar1, myVar2;
public:
MyClass();
void setFloats(float, float);
float add();
};
#endif
```

Implementation File (MyClass.cpp)

```
#include "MyClass.h"
void MyClass::setFloats(float myVar1, float myVar2)
{
this->myVar1 = myVar1;
this->myVar2 = myVar2;
}
float MyClass::add() {
return this->myVar1 + this->myVar2;
}
```

What is ROOT?

- **Open-source** data analysis framework
- Primarily **C++**-based, with support for **Python** (PyROOT)
- Developed at **CERN** to efficiently handle **large-scale data**
- Widely used in **particle physics**, but also in **finance, medicine, and big data**

Why learn ROOT?

- Standard tool in High Energy Physics
- Essential for working with experimental data
- Powerful features: plotting, statistical analysis, ML
- **Developed from physicists for physicists!**

ROOT is a **toolbox** that provides multiple **complex methods for data analysis**.

Using ROOT via LCG-Releases

- ROOT includes **Cling**, a C++ interpreter with Just-In-Time (JIT) compilation
- This allows executing C++ code **interactively**, like a calculator
- Example: Evaluating a geometric series sum in ROOT

$$s_N = \sum_{i=0}^{N-1} q^i = \frac{1-q^N}{1-q}$$

Try this in your terminal:

```
root -l
double q = 0.5;
int N = 10;
double s_N = (1 - pow(q, N)) / (1 - q);
double s_N2=0;
for (int i=0;i<N;++i) s_N2 += pow(q,i);
std::cout << s_N << " " << s_N2 <<std::endl;
```

Where to Find Information on ROOT?

- **Official Documentation** (ROOT Reference Guide)
  - Comprehensive but technical; best for in-depth understanding
- **User Guide** (ROOT User Guide)
  - Step-by-step tutorial with explanations
- **Tutorials and Code Examples** (ROOT Tutorials)
  - Ready-to-run scripts for learning by example
- **Community and Support**
  - ROOT Forum – Ask questions and discuss with users
  - GitHub Repository – Report issues and contribute

Application Example: `TLorentzVector`

- A ROOT class for **relativistic four-vectors** (`(x, y, z, t)`)
- Used for **kinematic calculations** in particle physics
- Can be used **standalone** or in **data analysis frameworks**

Example usage:

```
TLorentzVector v1;
v1.SetXYZT(1.0, 2.0, 3.0, 4.0);
TLorentzVector v2 = v1 + v1;
v1.Boost(0.1, 0.2, 0.3);
double scalar_product = v1 * v2;
v1.Print();
```

ROOT provides **Classes to describe physical concepts** (like 4-vectors) including multiple **methods ready to be used.**

What are TTrees?

- A **TTree** is the main data structure in ROOT for storing large datasets
- It organizes data into **branches**, where each branch can hold a specific variable (e.g., a number, vector, or object)
- Each branch is made up of **entries** (rows of data), and the `TTree` allows efficient access to these entries
- Typically used for storing data from experiments or simulations, like particle collision events
- **Benefits:** Efficient storage, fast reading, and writing of data, especially for large datasets

ROOT Files

- ROOT uses its own file format: `.root`
- Files are loaded into ROOT using the `TFile` class
- `.root` files store structured data, often used for **sequential data** storage
- ROOT files often contain **NTuples** for efficient data organization and access

Example: Loading a ROOT file:

- `TFile *file = TFile::Open("data.root");`
- `TTree *tree = (TTree*)file->Get("myTree");`
- `tree->Scan("variable1:variable2");`

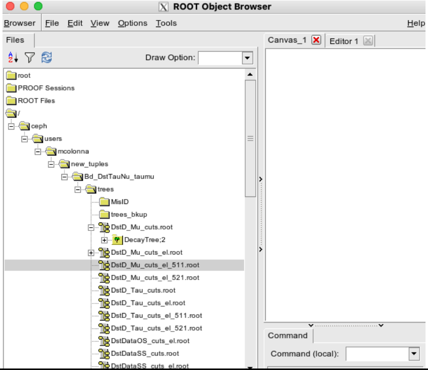
How to Handle TTrees

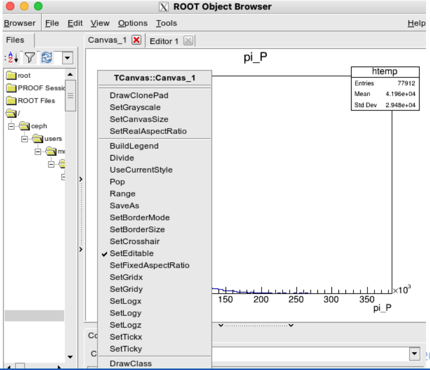
Example of handling TFiles and TTree:

- `TFile *file = TFile::Open("data.root", "UPDATE");`
- `TTree *tree = (TTree*)file->Get("myTree");`
- `int var, newVar;`
- `tree->SetBranchAddress("branchName", &var);`
- `tree->Branch("newBranch", &newVar, "newBranch/I");`
- ```
  for (int i = 0; i < tree->GetEntries(); ++i){
   tree->GetEntry(i);
   newVar = 2*var;
   tree->Fill(); }
  ```
- `tree->Write();`
- `file->Close();`
  ```

The TBrowser

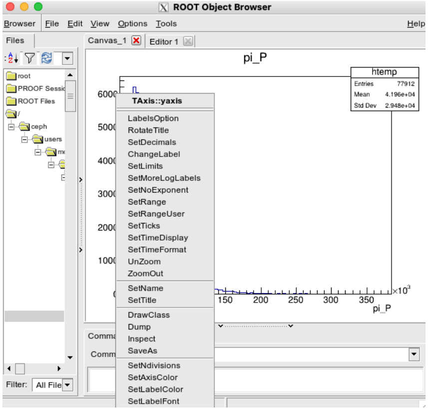
- **ROOT's file browser**: A GUI tool to view and interact with ROOT files
- **Double-clicking on a ROOT file**: In most cases, it's not the best way to inspect the contents of a ROOT file
- `TBrowser`: The graphical interface within ROOT to explore and plot data from a ROOT file
- To launch it within a ROOT session: `new TBrowser();`

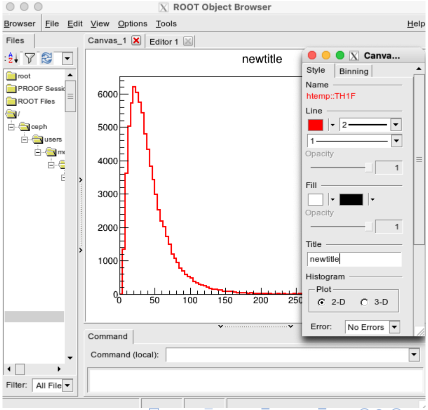




First Steps with the TBrowser - Options

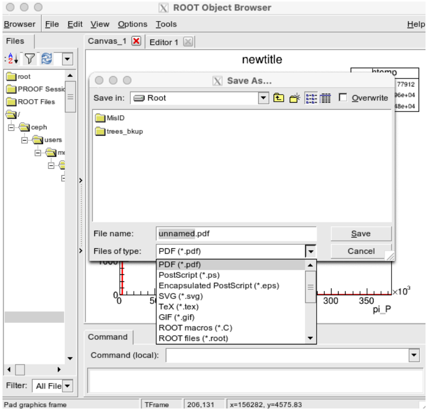
- Clicking on a leaf directly generates a histogram of the data
- The default action displays the histogram on a `TCanvas` with a **statistics box**
- Right-click on the `TCanvas` for customization options:
  - Example: `SetLogy` to enable logarithmic scaling on the y-axis

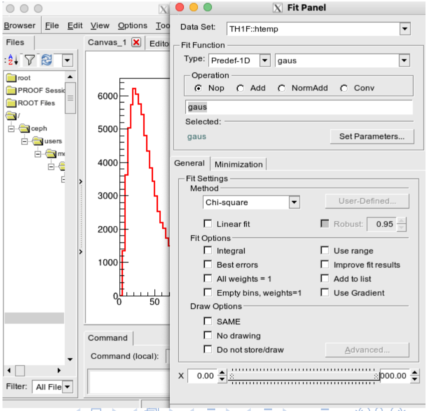




First Steps with the TBrowser - Saving

- After customizing your plot and data view, you can save the result
- Save your plot as a **PDF** or **PS** file using the TBrowser's GUI options
- The TBrowser also provides functions like fitting curves to histograms, accessible via right-click options





ROOT Macros: Introduction

- Working in the session can be a bit tedious, despite ROOT history
- A sequence of ROOT commands should be reliably reproducible
- 2nd level after ROOT session: Macros
- Small standalone files that perform simple tasks
- A macro consists of a .C file with the following structure:

```
void MacroName() {
 ...
 <your lines of C++ code>
 code line ends with;
 ...
}
```

ROOT Macros: Usage

- ROOT macros can be executed in three different ways:
- 1. Shell:
  `root MacroName.C`
- 2. ROOT session:
  `root [0] .x MacroName.C`
- or:
  ```
  root [0] .L MacroName.C
  root [1] MacroName()
  ```
- Note: In the second case, the macro is compiled; otherwise, it is interpreted by Cling
- There is even more possible with macros, see the documentation as always

Introduction to `make`

- `make` is a build automation tool used to compile and link programs
- It reads a special file called `Makefile` that defines build rules
- With `make`, you define dependencies between files and commands for building them
- It's widely used to manage large projects and automate repetitive tasks
- ROOT can also benefit from `make` to streamline building and running macros

Creating a Makefile for ROOT Macros

- A `Makefile` is a simple text file that specifies how to compile and link programs
- In the context of ROOT macros, a `Makefile` helps automate the compilation of macros and related libraries
- Example of a basic `Makefile`:

```
ROOTCFLAGS = $(shell root-config --cflags)
ROOTLIBS = $(shell root-config --libs)
CXX = g++
SRC = macro.C
OUT = macro.exe
all:  $(OUT)
$(OUT): $(SRC)
 g++ $(SRC) $(ROOTCFLAGS) -o $(OUT) $(ROOTLIBS)
```

- This Makefile compiles `macro.C` and links it with the necessary ROOT libraries

Running the Makefile

- After creating the Makefile, you can run it by executing:
  - make in the terminal
- This will compile your ROOT macro and produce an executable, for example macro.exe
- Once compiled, you can run the macro with:
  - ./macro.exe
- If you make any changes to the .C file, just run make again to recompile the updated code

Advanced Usage with `Makefile`

- You can add more complex features to your `Makefile`, such as:
  - Specifying multiple source files
  - Defining separate rules for cleaning up compiled files (e.g., `make clean`)
  - Automatically linking against shared libraries
- Example for cleaning up compiled files:

```
clean:
	rm -f $(OUT)
```

- You can also use `make` with multi-step processes, for example, building a library before compiling the macro

Thank You for Your Attention

- **Thank you** for your attention throughout this presentation!
- We've covered a lot:
  - C++ fundamentals + 1 phrase on polymorphism of classes
  - Using C++ in ROOT for data analysis
  - Navigating and manipulating ROOT files, classes, and macros
  - Automating tasks with `make` and creating efficient workflows
- It was **a lot to digest**, come back to these concepts and explore them in time.
- Don't be scared of trying out new programming methods in your projects.
- Don't hesitate to reach out if you have any questions later on.

HAPPY CODING!