Google Summer of Code 2023 RooFit project

Jonas Rembser

February 16, 2023

Pre-exercises

The following are the evaluation exercises for prospective Google Summer of Code (GSoC) 2023 students interested in the RooFit project. The exercises are designed to see if you are sufficiently knowledgeable with the programming languages used in RooFit, which are:

- C++
- Python

They also make you familiar with the parts of the ROOT framework that will be used in the project:

- ROOT's built-in C++ interpreter cling
- PyROOT based on cppyy

If you have any questions while solving these exercises, please don't hesitate to get back to Jonas Rembser (jonas.rembser@cern.ch). Being able to ask the right questions at the right time is just as important as technical expertise! In particular, the ROOT documentation still has some rough edges, and your questions can help to guide us in improving the documentation.

Setting up ROOT - Forking the git repository and building ROOT from source

The goal of this exercise is to clone the ROOT repository on GitHub and build ROOT from the source code, which is necessary to collaborate on ROOT.

The ROOT project uses the CMake build system. You can learn more about CMake in these two tutorials:

- Tutorial in the CMake documentation
- CMake by example tutorial

ROOT uses git for version control. Note that for creating commits, please follow the best practices to structure commits and write commit messages.

The necessary steps to fork and build ROOT are:

1. Fork and clone the ROOT repo
2. Build ROOT from source following these instructions.

Warmup 1 - You first ROOT script in C++

With ROOT, you can interpret C++ scripts (or "macros") as described here in the ROOT manual.

Your first warmup task is to write a macro that instantiates an object of the class `MyClass` (code below) and prints it via the provided function.

```
class MyClass : public TObject {
public:
   inline MyClass(std::string const &name, std::string const &title, int value)
      : name_{name}, title_{title}, value_{value}
   {
   }

   inline void Print(Option_t *option = "") const override
   {
      std::cout << name_ << "  " << title_ << "  " << value_ << std::endl;
   }

private:
   std::string name_;
   std::string title_;
   int value_;
};
```

The output should look like below:

```
root [0]
Processing solution1.C...
name  title  1
root [1] .q
```

Warmup 2 - Your first ROOT script in Python

In the next step, you should familiarize yourself with writing ROOT scripts in Python, using PyROOT. You can find an introduction to PyROOT again in the manual on the ROOT website.

The essential information there is that you can run arbitrary code via the built-in ROOT interpreter (cling):

```
ROOT.gInterpreter.ProcessLine('#include <iostream>')
ROOT.gInterpreter.ProcessLine('std::cout << "Hello World" << std::endl;')
```

Try to use ROOT's automatic Python bindings to instantiate `MyClass` again in Python like `obj = ROOT.MyClass(..)` and call `Print()` on it!

Exercise - JSON factory interface

This is the actual evaluation exercise. Please send back all the files necessary to run your solution.

You know now how to instantiate a C++ class in Python via the automatic PyROOT bindings.

For this exercise, you should make a little detour over a JSON object. This seems overly complicated to instantiate a simple class, but in RooFit, we will do something similar as the JSON factory in C++ already exists. That means if you solve this exercise, you know all the necessary ingredients to the technology stack used in the project!

The goal is to implement the following chain of functions:

1. A function that you defined on the C++ side with the following signature:

   `std::unique_ptr<TObject> createFromJsonString(std::string const &jsonStr)`

   - This function should not call the class constructor directly but internally parse the JSON to create the right constructor call as a `std::string`:

     `"new MyClass{\"first\", \"The First Instance\", 1};"`

   - This string is used to instantiate the returned object via cling, using again `gInterpreter->ProcessLine` like:

     ```
     std::unique_ptr<TObject> instance{
         reinterpret_cast<TObject *>(gInterpreter->ProcessLine(...))
     };
     ```

2. On the Python side, implement a function that takes the name of a class plus a list of constructor arguments and returns an instance of that class, for example:

   `create_from_json("MyClass", ["first", "The First Instance", 1])`

   - Internally, this function will create a JSON string like this (still in Python):

     ```
     {
         "class": "MyClass",
         "args" : ["first", "The First Instance", 1]
     }
     ```

   - This string is passed to `createFromJsonString()` that you defined on the C++ side

At the end of your Python script, you should yet again instantiate a `MyClass` object and print it, but this time using your JSON factory:

```
obj_1 = create_from_json("MyClass", ["first", "The First Instance", 1])
obj_1.Print()
```

There are only two libraries that you need to use for this task:

- The Python json module to generate the JSON string in Python easily
- The nlohmann-json library for C++ to parse that string again

The final solution will have approximately 100 lines of code (C++ and Python combined).

I'm very curious to see how your solution will look like, and hope that in any case you have fun learning a bit more about Python and C++ while solving this exercise!