1. Given project description

Run-time-provisioning of ROOT’s I/O type binding: ROOT is the LHC physicists’ common tool for data analysis; almost all data is stored using ROOT’s I/O system. This system benefits from a custom description of types (a so-called dictionary) that is optimised for the I/O. Until now, the dictionary cannot be provided at run-time; it needs to be prepared in a separate prerequisite step. This project will move the generation of the dictionary to run-time, making use of ROOT 6’s new just-in-time compiler. It allows a more dynamic and natural access to ROOT’s I/O features especially for user code.

1.1. Clarifications to the project description. The objective of this summer student project was to provide dictionary generation for collections of objects at run-time, not general dictionaries as stated in the project description - this is already possible.

2. Preliminary Knowledge

To understand the work done, I will briefly describe what a dictionary is, what it contains and why we use them in ROOT.

2.1. What is a dictionary? A dictionary is an extension of the Run-Time Type Information (RTTI). RTTI is a C++ mechanism that exposes information about an object’s data type at run-time. This can for example be used to doing safe typecasts at run-time. In less technical terms, a dictionary is a description of how objects in ROOT are constructed, and descriptions of how they work.

2.2. What is in a dictionary? A dictionary for a class contains a set of things. First of all it contains a description of the data contained in the class, including names, types and access information. Secondly it includes a description of the methods for the class (the functionality of the class) including method-names, return types, description of method-parameters and access information. The dictionary also contains a title and a version for the class and a function for creating new objects of this class.

Date: September 6, 2013.
2.3. **Why do we use dictionaries?** There are many use-cases for dictionaries in ROOT. I will try to name a few of the more important ones here.

One of the more fundamental ones is when saving objects to the hard-disk (so-called serialisation). When we want to save an object to the disk, we need to know how the object is structured.

When using the Graphical User Interface in ROOT (for example for plotting graphs and histograms) one can right click items - for example an axis or some part of the data - and get a right click menu with a set of options. These options corresponds to member methods for the class corresponding to the object you are right clicking. These member methods are looked up using the dictionary.

When we want to read an object from disc into ROOT, we first need to create the object in memory. This is done using the dictionary which describes the structure of the object.

A more specific use-case closely related to my work is when using collections defined in the C++ Standard Library (STL), like for example a vector. The implementations of STL-collections differ from platform to platform (Windows, Mac, Linux, etc.), so when saving and loading STL-collections, we need to store it in a separate home-made format. We need to generate dictionaries for STL-collections, and these contain a specific method called the Collection-Proxy which translates the platform specific STL object to the on-disk format and back.

3. **My work**

One of the issues that I have solved is described in this thread on the ROOT forums: [http://root.cern.ch/phpBB3/viewtopic.php?t=4936](http://root.cern.ch/phpBB3/viewtopic.php?t=4936). In the thread the user gets an error like this:

```
Error in <TTree::Branch>: The class requested (vector<MyClass>) for
the branch "checked_value" refer to an stl collection and do not
have a compiled CollectionProxy. Please generate the dictionary
for this class (vector<MyClass>)
```

stating that the user needs to generate a dictionary for the class `vector<MyClass>`. As noted in the discussion on the forum, the old solution was to generate a file `loader.C` with the following contents:

```
#include <vector>
#ifdef __MAKECINT__
#pragma link C++ class vector<MyClass>++;
#endif
```

and then - in ROOT - run:

```
.L loader.C+
```

which will generate the dictionary for `vector<MyClass>`.

With my changes, this issue will no longer appear since the dictionary for `vector<MyClass>` will be automatically generated and loaded into ROOT.
3.1. **How did I solve this.** Before my work, we would issue the warning stated above when we noticed a missing dictionary. Now we instead generate this dictionary as a string and feed it to the run-time compiler **cling**. Conceptually, this can be formulated as

```c
if (itMakesSenseToGenerateThisDictionary())
{
    dictionaryString = GetDictionarySource();
    LoadText(dictionaryString);
}
```

The work I have been doing can be split into two main parts. The first part was to construct the logic that tells whether or not a dictionary can be generated for the type we encounter. The second part was to move/refactor the source code for generating dictionaries so that it is accessible to the run-time compiler **cling**.

3.2. **Additional gains.** An additional gain from the solution is a large speedup when reading and writing to the disk for collection types that used to not have a dictionary. Input/Output is much faster for types for which we only have the header file but no dictionary - because we generate that dictionary at runtime, and that allows ROOTs I/O to make use of all the I/O shortcuts in the dictionary.

For example, if we are loading in a large collection of MyClass objects, then before the emulated collection proxy would use interpreted calls to create each MyClass object in the collection. Now that we have a dictionary and a collection proxy for a collection of MyClass, then we can load the entire collection in one go, consequently saving a lot of time.

4. **What have I learnt?**

During my time working on this project, I have learnt a lot of new tools and techniques. First of all, I have gained a lot of experience and knowledge about programming in C/C++, especially since ROOT is a software project that utilises a lot of low-level functionality in C++. As I am studying a master in mathematics, most of my programming experience is with high-level concepts of programming - so this project has been a tough challenge for me technically.

I have learnt to use the tools **make** and **gdb** to handle compilation and debugging of C/C++ programs in a terminal. By using **gdb** for debugging memory issues, I have also gained a deeper understanding of how C/C++ and operating systems works under the hood. I have used **valgrind** to debug more difficult memory issues, consequently going even deeper down the rabbit hole of memory management in operation systems and C/C++.

For version control we have used **Git**, and even though I have been exposed to this before, I now have a lot better understanding of the concepts, strengths and weaknesses of using **Git**.

Finally I have been using the functionality in the **clang/llvm** library a lot, and by this gained knowledge and experience with C/C++ run-time compilers.
5. Conclusions

My project work this summer has been to provide dictionary generation for collections of objects at run-time in ROOT. This was achieved, and by doing this ROOT users will have to write less code and this makes it even easier to read ROOT files (and thus the experiments’ data). My changes are currently included in the source code of ROOT.

On the personal level I have gained a significant amount of experience with C/C++ programming, tools like gdb and make and maintaining large complex software systems.

Supervisor: Axel Naumann