ROOT modularization [progress report]

O.Shadura, V.Vassilev, B.Bockelman, R.Isemann
University of Nebraska-Lincoln

I.Updates on modularization proposal

Goal of modularization project

- By making the boundaries and relationships more explicit through modules, **we can better define and implement a “minimal ROOT,”** increasing the chances its functionality can be embedded in other contexts. **This enables ROOT users to interact with the wider data science ecosystem.**
- **Packages and package management provide a mechanism for ROOT users to socialize and and reuse projects built in the context of ROOT**, it helps to make ROOT more flexible and open for new customers. **This allows ROOT to continue to serve as a community nexus.**
- **Having a package format can help define community standards** and allows the community to go in a similar direction, focusing what effort we have and then in the same time to reduces the desire / ability for every experiment and analysis group to ‘invent their own’ set of packaging on top.

ROOT Modularization proposal GoogleDoc: https://goo.gl/N6rzaW

Concepts

- **Module:** A set of interdependent classes implementing coherent functionality and providing well-defined APIs; modules could also contain “data files”, that aren’t necessarily a class implementation.
    - **Library:** a module or set of modules, which makes sense to be together and that can be used in a program or another library.
- **Package:** A distinct, self-describing resource (file, URL) that provide one or more modules.
- **Package database:** A record of all packages currently available in a ROOT installation.
- **Package manager:** An actor that can locate and install packages into a ROOT installation from a package reference, along with their transitive dependencies.

OSPM/LPM/PDM

- **OS package manager (we are interested in dependency resolution, but not in terms of ROOT modularization)**
- **Language package manager (LPM):** an interactive tool that can retrieve and build specified packages of source code for a particular language (or in the case of ROOT particular runtime).
- **Project/application dependency manager (PDM):** an interactive system for managing the source code dependencies of a *single project* in a particular language. That means specifying, retrieving, updating, arranging on disk, and removing sets of dependent source code, in such a way that *collective coherence* is maintained beyond the termination of any single command. Its output, which is precisely reproducible, is a self-contained source tree that acts as the input to a compiler or interpreter. One might think of it as "compiler, phase zero."[1]

ROOT Data Analysis Framework

~ (LPM + PDM) functionality

Modularization

https://medium.com/@sdboyer/so-you-want-to-write-a-package-manager-4ae9c17d9527

Top OSPM/LPM/PPM in Github

- Alcatraz (Obj-C and packages descriptions are in JSON) **10k* PPM**
- HomeBrew (-s) (Ruby) **10k* OSPM**
- Web: components/nmp/bower (JS) **10k***
- Swift-package manager (Swift) **6k* LPM**
- ...
- Portage (Python) **0.2k* OSPM**
- Spack (Python) **0.5k* OSPM**

Inspiration

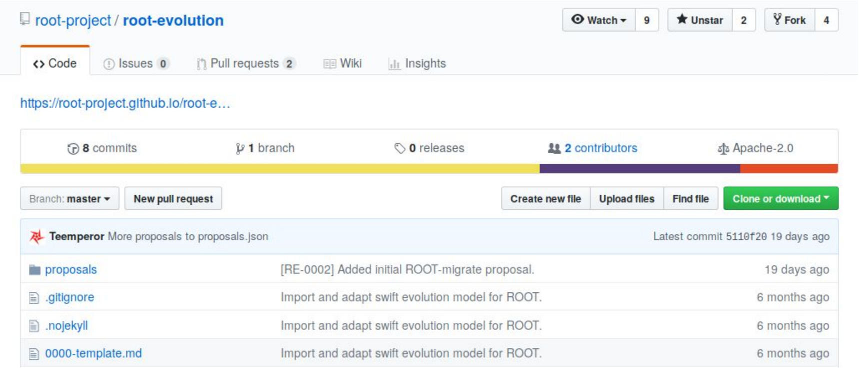
- Swift-package manager (https://github.com/apple/swift-package-manager)
  - Nicely aligned with ROOT needs
  - Swift has easy interoperability with C/C++ based languages through the use of the Clang modules system)

Ideas

- ROOT is “compiler” by itself
- PCH & Dictionaries
  - For better modularization, we need to achieve PCH separability and it could be done via C++ modules, by generating C++ module for each new ROOT component to be plugged in
- C++ modules & module maps
  - Logical continuation of work done on C++ modules for ROOT and their integration in CMSSW

root-evolution proposal: looking forward for community feedback

- https://github.com/root-project/root-evolution



- https://github.com/oshadura/root-evolution/blob/modularization-proposal/proposals/0001-modularization.md

II. Updates on prototype

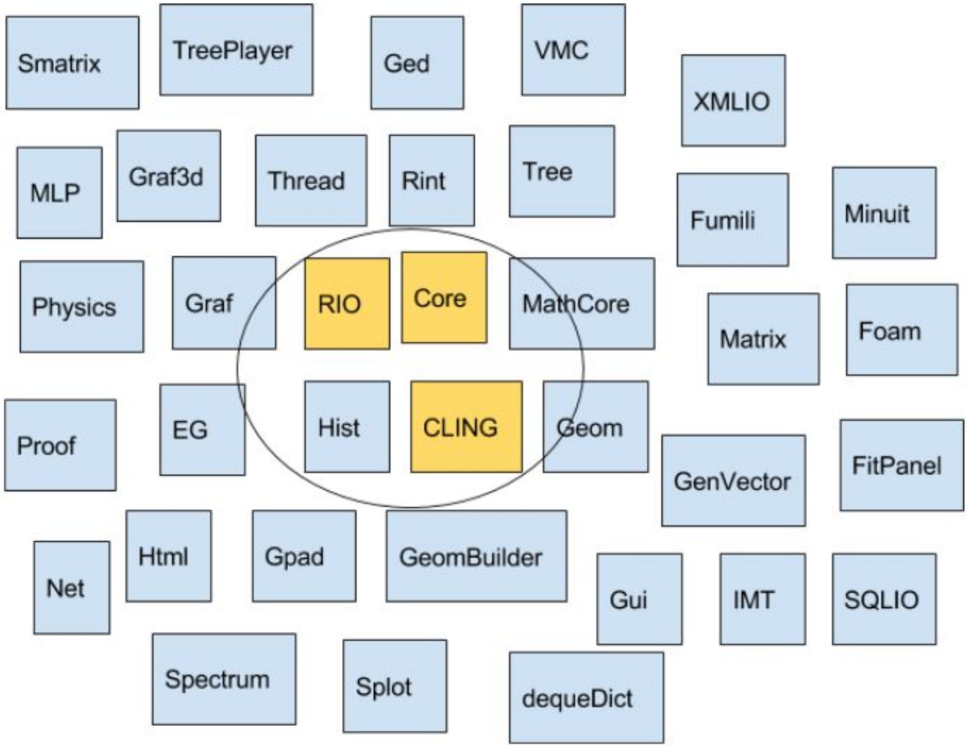
First steps

- “Base” option (available on branch oshadura/rootbase);
    - **Big thanks to Raphael for helping with the initial implementation!!!**
- CMake improvements: introducing new macros, allowing to build ROOT module on the top of ROOT “Base”;
- Better separation of modules ;
    - Fixing cross TU dependencies, usually resulting in linking errors [RIO->MathCore];
    - Fixing header dependencies, they may not result in linking errors if we have all definitions in the headers (in case of inline functions and templates) [MathCore->Hist];
    - Cleanup of unneeded legacy dependencies for ROOT binaries and legacy code cleanup;

From “Minimal” from “Base”

*List of directories to be used in **“Base”**:*

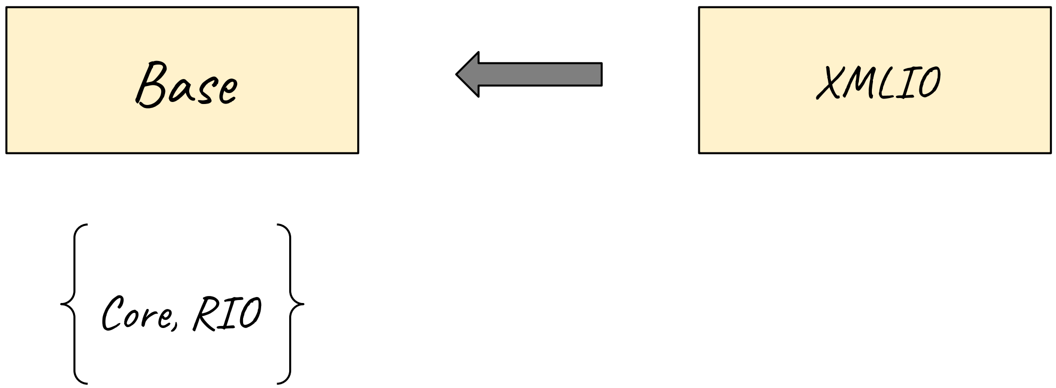
- *core/*all**
- *io/io/*
- *io/pcm/*



Pros and cons of “Base”

1. Separate fundamental part for ROOT that take a longest fraction of time to build ROOT;
2. Define boundaries between “Base” and ROOT modules;
3. Refactoring includes (changes will be transparent);
4. “Base” could be easily released as a Docker container and help to provide Travis-CI support for new ROOT modules and/or externals projects testing.

Example of ROOT Module: XMLIO



Example: YAML manifest for XMLIO

sources:

- *“sources_of_xmlio_library ”*

dependencies: RIO

Should be extended with extra fields:

- *Version of ROOT(6.12/master)*
- *Timestamp*
- *...*

Current results:

with help of bash/python we can prototype an expected behaviour of tool (query already build modules or build missing dependencies)!

Example of in-source modularization: module example XMLIO

https://github.com/root-project/root/tree/master/io/xml

```
-- Performing Test CXX_HAS_fno_rtti - Success
-- Performing Test found_setresuid
-- Performing Test found_setresuid - Success
-- Performing Test found_stdstringview
-- Performing Test found_stdstringview - Failed
-- Performing Test found_stdexpstringview
-- Performing Test found_stdexpstringview - Success
-- Performing Test found_stod_stringview
-- Performing Test found_stod_stringview - Failed
-- Performing Test found_stdapply
-- Performing Test found_stdapply - Failed
-- Performing Test found_stdinvoke
-- Performing Test found_stdinvoke - Failed
-- Performing Test found_attribute_always_inline
-- Performing Test found_attribute_always_inline - Success
Running /home/oksana/CERN_sources/root/build/unix/compiledata.sh
Making /home/oksana/CERN_sources/root/builds/include/compiledata.h
-- Configuring done
-- Generating done
-- Build files have been written to: /home/oksana/CERN_sources/root/builds
[1028/3269] Performing download step (verify and extract) for 'LZMA'
-- LZMA download command succeeded.  See also /home/oksana/CERN_sources/root/builds/On/Stamp/LZMA/LZMA-download-*.log
[2694/3269] Performing configure step for 'LZMA'
-- LZMA configure command succeeded.  See also /home/oksana/CERN_sources/root/builds/On/Stamp/LZMA/LZMA-configure-*.log
[3131/3269] Performing build step for 'LZMA'
-- LZMA build command succeeded.  See also /home/oksana/CERN_sources/root/builds/On/Stamp/LZMA/LZMA-build-*.log
[3132/3269] Performing install step for 'LZMA'
-- LZMA install command succeeded.  See also /home/oksana/CERN_sources/root/builds/On/Stamp/LZMA/LZMA-install-*.log
[3268/3269] Generating etc/dictpch/allLinkDefs.h, etc/dictpch/allHeaders.h, etc/dictpch/allCppflags.txt

Generating PCH for core/base core/clingutils core/rint core/thread io/io

[3269/3269] Generating etc/allDict.cxx.pch
ROOTSYS: /home/oksana/CERN_sources/root/builds
-- The C compiler identification is GNU 6.2.0
-- The CXX compiler identification is GNU 6.2.0
-- Check for working C compiler: /usr/lib/ccache/cc
-- Check for working C compiler: /usr/lib/ccache/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/lib/ccache/c++
-- Check for working CXX compiler: /usr/lib/ccache/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found ROOT: /home/oksana/CERN_sources/root/builds/bin/root-config
-- Configuring done
-- Generating done
-- Build files have been written to: /home/oksana/CERN_sources/root/builds/xml-build
[9/9] Linking CXX shared library libXMLIO.so
```

ROOT "Base"

XMLIO module [includes XMLIO basic functionality test]

Plan of work

1. *Prototype 1*
   - Map “minimal ROOT” to a module called “Base”
   - Work on prototype tool
     - List all modules installed
     - Other extra queries on ROOT dependencies
   - Define package format (logical and concrete format)
   - Given an installable package, install it and be able to load the module

This provides an installation “ROOT Package Database”

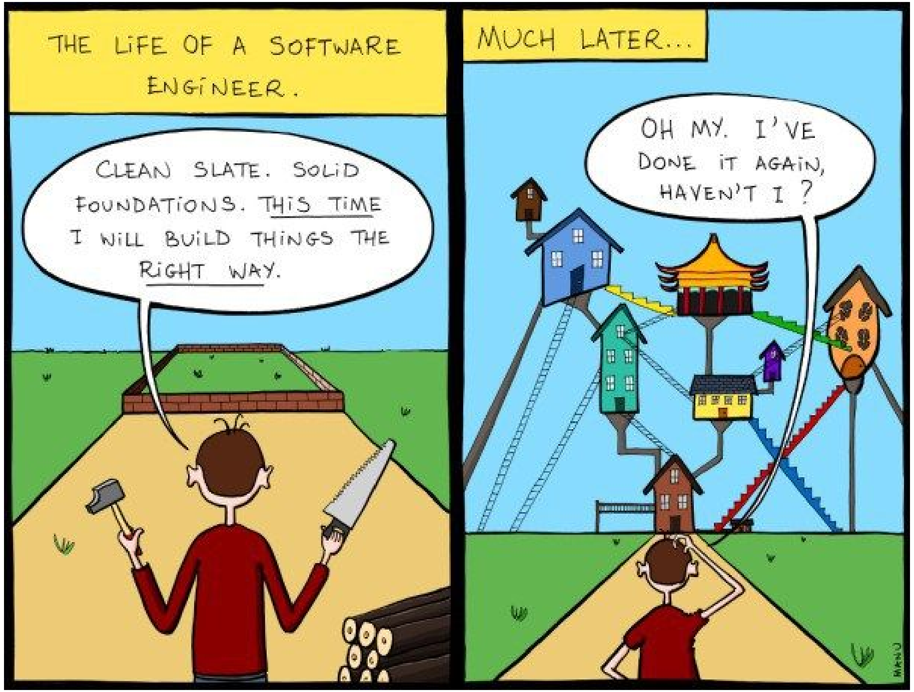
2. *Next step: Prototype 1 + Prototype 2*
   - Lazy install based on missing package
   - External package resolution

This provides an “ROOT package Resolver” or “ROOT Indexer”

3. *Next step: Prototype 1 + Prototype 2 + Prototype 3*
   - Tackle building of various packages (in-source and external modularization)

It is just a beginning !

Thank you!



THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?