PODIO - Applying plain-old-data for defining physics data models

F.Gaede, DESY B. Hegner, P. Mato CERN

CHEP2016, Oct 10-14, 2016

Outline

- Introduction and Motivation
- Design of PODIO
- Implementation
    - Vectorization
    - Object Ownership
    - Relations
    - I/O
- Summary and Outlook

Why a new event data model (EDM) library?

- LHC experiments have overly complex EDMs
  - strong use of inheritance and polymorphism
    - state of the art when the code was written
  - rather expensive virtual calls and memory operations
  - deep object hierarchies
- LCIO used in LC-community has much simpler (less complex) EDM, but:
  - uses inheritance and virtual function calls (*abstract interfaces*)
  - actual (non-ROOT) I/O suffers from similar issues as LHC solutions
- new projects - like the FCC - are an opportunity to do better this time
  - *use what worked well - throw away what didn't*

General Context of PODIO

- PODIO is developed in context of the **FCC** study
  - addressing the problem in a generic way
  - allowing potential re-use by other HEP groups
  - use the application to LCIO as immediate cross check for generality
- supported by the **AIDA2020** EU-project
  - together with other generic HEP software tools (DD4hep, VecGeom, . . .)
- one of the first projects adopted by the **HEP Software Foundation**

Driving Design Considerations

- simple memory model
    - by default use array of structs (PODs)
    - allow for easy vectorization
- simple user API and class hierarchies
    - use concrete types: favor composition over inheritance
    - clear ownership design
- apply code generation
    - avoid user mistakes
    - quick turn-around for improvements on back-end and extensions of EDM
- support for C++ and Python
- thread-safety
- allow for different I/O layers
    - use ROOT for first (reference) implementation

Interlude: what is a POD ?

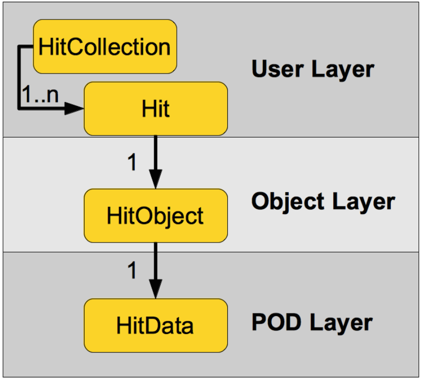
- **P**lain-**O**ld-**D**ata object
- in C++11/14 a POD combines two concepts
    - support for static initialization (trivial class)
    - standard layout
        - no virtual functions and no virtual base classes
        - same access control (i.e. public,private,protected) for all non-static data members
        - ...

think of a POD as a classical C struct rather than a normal C++ class

PODs are good for memory layout and operations as well as for **I/O**
=> **PODIO** !

The three PODIO layers

- user layer (API):
  - handles to EDM objects (e.g. **Hit**)
  - collection of EDM object handles (e.g. **HitCollection**).
- object layer
  - transient objects (e.g. **HitObject**) handling vector members and *references* to other objects
- POD layer
  - the actual data structures holding the persistent information (e.g. **HitData**)



Support for Vectorization

- key for vectorization is **struct-of-arrays (SoA)** vs. of **arrays-of-structs (AoS)**
  - which representation is better heavily depends on the use case
- PODIO allows to choose one representation at the implementation of the *POD layer*
  - choice hidden from the non-expert user (no effect on *user layer*)
- on demand transformation between complete SoA vs. AoS representations is highly inefficient → *the decision for the representation of a given data type has to be made at* **compile time**
- provide convenience methods for on demand transformation

```
auto& hit_x_array = hits.x<10>();
```

- need proper performance measurements on real use cases

Object ownership in PODIO

clear design of ownership (hard to make mistakes) in two stages:

objects added to event store are *owned by event store*

```
auto& hits = store.create<HitCollection>("hits") ;
auto h1 = hits.create( 1.,2.,3.,42.) ; // init w/ values
auto h2 = hits.create() ; // default construct
h2.energy( 42.) ;
```

objects created standalone are *reference counted* and automatically garbage collected:

```
auto h3 = Hit() ;
auto h4 = Hit() ;
hits.push_back( h3 ) ;
//  h1,h2,h3 are automatically deleted with collection
//  h4 is garbage collected
```

Relations between Objects

allow to have 1-1, 1-N or N-M relationships, e.g.

```
auto& hits = store.create<HitCollection>("hits");
auto& clusters = store.create<ClusterCollection>("clusters");
auto hit1 = hits.create(); auto hit2 = hits.create();
auto cluster = clusters.create();
cluster.addHit(hit1);
cluster.addHit(hit2);
```

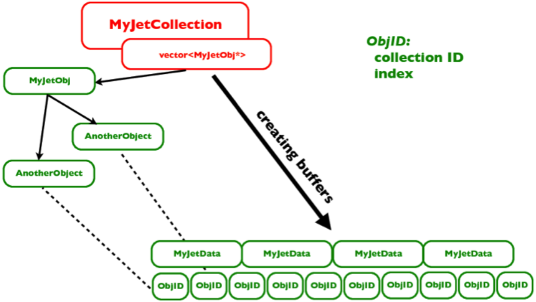
referenced objects can be accessed via iterator or directly

```
for( auto h = cluster.Hits_begin(),end = cluster.Hits_end(); h!=end ++h){
    std::cout << h->energy() << std::endl;
}
auto hit  = cluster.Hits(42);
```

also standalone relations between arbitrary EDM objects

Relations Details

- relations are handled outside the PODs
- the “Object Land” manages the lookup in memory
- every object in PODIO is uniquely identified by **ObjectID = collectionID + collectionIndex**
- during I/O every reference is being replaced by its ObjectID
    - independent of the specific I/O system



ObjID:
collection ID
index

Code generation

- code (C++/Python) for the EDM classes is generated from yaml files
- EDM objects (data structures) are built from
  - basic type data members
  - components (structs of basic types)
  - references to other objects
- additional user code (member functions) can be defined in the yaml files

```
# LCIO MCParticle
 MCParticle:
  Description: "LCIO MC Particle"
  Author : "F.Gaede, B. Hegner"
  Members:
   - int pDG               // PDG code of the particle
   - int generatorStatus   // status as defined by the generator
   - int simulatorStatus   // status from the simulation
    #...
  OneToManyRelations:
   - MCParticle parents // The parents of this particle.
   - MCParticle daughters // The daughters this particle.
  ExtraCode:
   const_declaration:
   "bool isCreatedInSimulation() const {
     return simulatorStatus() != 0 ;
    } \n"
```

Python Interface

- Python is treated as first class citizen in the provided library
- can use *pythonic* code for iterators etc.
- implemented with PyROOT and some special usability code in Python

Python code example:

```
store = EventStore(filenames)
for i, event in enumerate(store):
  hits = store.get("hits")
  for h in hits:
    print h.energy()
```

I/O implementation

- PODIO's I/O is rather trivial at the moment
- PODs are directly stored using ROOT I/O
    - auto generated streamer code via dictionary
    - not properly optimized for PODs yet
- object references are translated into *ObjectIDs* before being stored

Major To-Do-item:

implement a direct binary I/O (storing array of structs) for performance comparison with ROOT and to demonstrate the potential performance advantage of storing PODs

PODIO in use

- PODIO is actively used by the FCC study efforts
  - in combination with Gaudi
  - "Standalone" for other C++ and Python applications
  - current data model definitions are in fcc-edm
- currently investigating the use of PODIO as evolution of LCIO
  - improve the I/O performance - keep the EDM (plcio)
  - need quite some extra code for backwards compatibility
- LHCb is interested in PODIO for their data model upgrade
  - → lhcbio demonstrator created during a coding sprint

Summary and Outlook

- EDM toolkit PODIO developed in context of FCC/LC with general HEP in mind
  - also AIDA2020 and HSF project
  - storing EDM objects in arrays of PODs
  - currently using ROOT I/O - others to follow
  - code automatically generated for C++ and Python
  - first implementation in use by FCC
  - under evaluation for LC

Outlook

- finalize the user API
- implement alternative binary I/O

Links and Pointers

- GitHub repository + docs:
  - `https://github.com/hegner/podio`
- doxygen page:
  - `https://fccsw.web.cern.ch/fccsw/podio/index.html`
- issue tracker:
  - `https://sft.its.cern.ch/jira/projects/PODIO`
- plcio (EDM for LCIO w/ podio) git repository:
  - `https://stash.desy.de/projects/IL/repos/plcio`
- PODIO Library Design Document:
  - `http://cds.cern.ch/record/2212785`