PODIO

Streaming Readout X
18.5.2022

Benedikt Hegner
CERN

Outline

- **Motivation and Context**
- **Driving Design Considerations**
- **Explanation of the Current Implementation**
- **Open work and future steps**

A little disclaimer:

Given the kind of workshop this presentation goes more into design choices and details rather than giving a simple introduction into the end-user interface.

Why a new data model library?

- LHC experiments show that we overdid on inheritance and polymorphism
  - State of the art when the code was written!
  - Expensive virtual calls and memory operations
- Sometimes deep and absurd object hierarchies
  - A “CaloTower” as a “Particle”
- Many physicists do not feel productive in the existing data models…

  … and leave the official frameworks behind as soon as possible
- During the last 15 years technology evolved a lot!

⇒ Needed to rethink what we did

Driving Design Considerations

1. **Simple Memory Model**
    a. Concrete data are contained within plain-old-data structures (PODs)
    b. Provide vectorization friendly (or at least not unfriendly) interfaces
2. **Simple Class Hierarchies**
    a. Wherever possible use concrete types
    b. Favour composition over inheritance
3. **Simple interfaces on user side**
    a. In particular avoid ownership problems!
4. **Employ code generation**
    a. Quick turn-around for improvements on back-end
    b. Easy creation of new types
5. **Support for both C++ and Python**
6. **Thread-safety**
7. **Use ROOT as first choice for I/O**
    a. Keep transient to persistent layer as thin as possible

Simple Memory Model

Interlude - what is a POD?

A POD combines two concepts

- Support for static initialization (*trivial class*)
- They have *standard layout*
    - No virtual functions and no virtual base classes
    - Same access control for all non-static data members
    - …

In short - a POD is closer to a classical C struct than a C++ object

A POD is good for memory layout and memory operations

⇒ PODIO !

Separation of Concerns

Using PODs is a good idea…

… but they are a little bit too dumb to support all what is needed.

Need smart layers on top of the PODs to

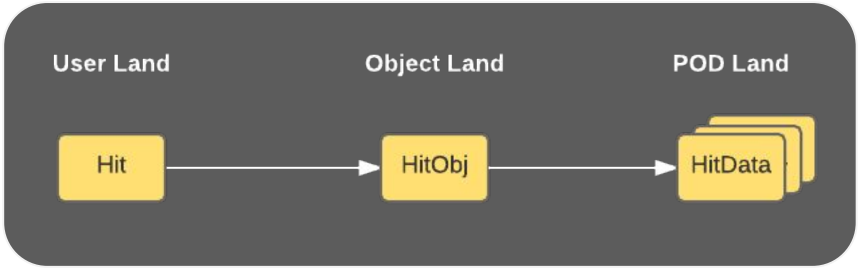
- Deal with object ownership
- Allow referencing between objects
- Deal with non-trivial I/O operations

Whenever performance is a concern - leave possibility to access the bare PODs

The PODIO layers

1. User visible classes (e.g. *Hit*). These act as transparent references to the underlying data
2. A transient object knowing about all the data for a certain physics object, including inter-object references (e.g. *HitObject*)
3. POD holding the persistent object information (e.g. *HitData*)

   and
4. A Collection containing the user’s objects (e.g. *HitCollection*)



Simple Interfaces

Supported Syntax

Objects and collections can be created via factories, ensuring proper ownership:

```
auto& hits = store.create<HitCollection>("hits")
auto hit1 = hits.create(1.4,2.4,3.7,4.2);  // init with values
auto hit2 = hits.create(); // default-construct object
hit2.energy(42.23);
```

Objects can be created in the free - if not attached to a collection, they are automatically garbage collected:

```
auto hit1 = Hit();
auto hit2 = Hit();
...
hits.push_back(hit1);
...
<automatic deletion of hit2>
```

Whenever performance is a concern - leave possibility to access the bare PODs

Object Ownership

Unclear object ownership and memory leaks are a common problem

⇒ **Make it as hard as possible to do mistakes**

In PODIO there are two stages in object ownership

1. Before registering data into an event store ⇒ reference counted
2. After adding data into event store ⇒ ownership up to event store

Additional costs on object creation time and no costs later

Relations between Objects

Providing relations between objects is important. Relations can be

1. Internal: m `mother = particle.mother()`
2. External: `mother = mother_daughter_relations[particle]`

Providing only external ones is puristic, providing internal ones easier to use

PODIO allows both, obviously.

Relations between Objects

N-to-M relations look like this:

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

auto& clusters = store.create<ClusterCollection>("clusters");
auto cluster = clusters.create();

cluster.addHit(hit1);
cluster.addHit(hit2);

<...>

auto hit  = cluster.Hits(<aNumber>);
```

Using smart references avoids confusion when to pass values, pointers, etc

Const correctness on smart references

It is easy to strip constness from a smart reference, e.g. by doing implicit copies

`auto myRef` instead of `const auto& myRef`
`const auto myRef`

Need to preserve constness state within the smart reference by either:

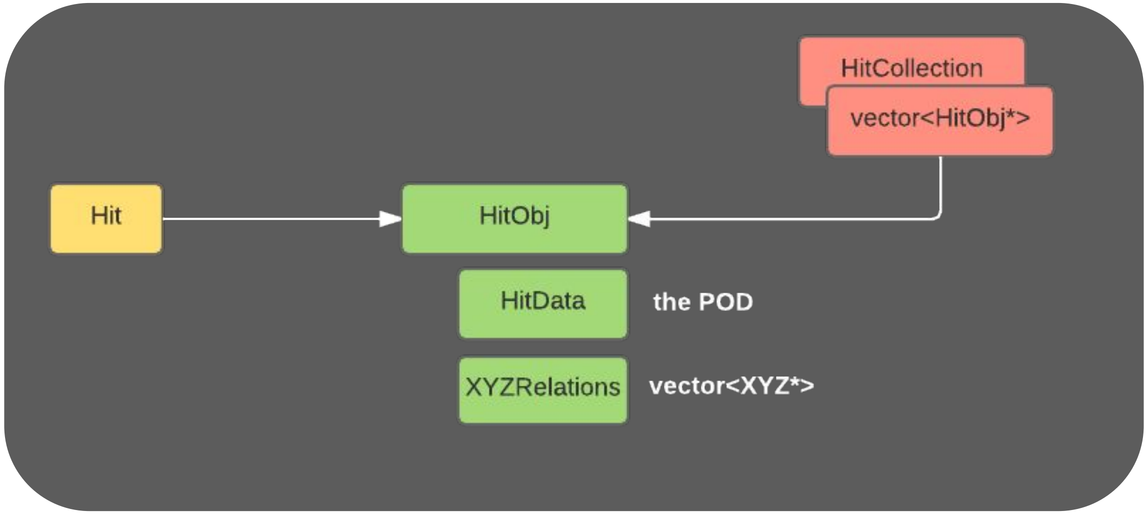
1. Having two classes (*Ref* and *MutableRef*) - compile time!
2. Having an internal flag (a la *isMutable*) - runtime!

In the prototyping stage we played with both. The first is more puristic, the second more user-friendly.

⇒ **So far we are using the compile-time option**

Relations - Details

1. Relations are handled outside the PODs
2. The “Object Land” manages the lookup in memory
3. Every object in PODIO is uniquely identified by *collectionID* + *index*
4. During I/O ever reference is being replaced by its Object ID



Code generation

Defining Data Types

In PODIO data model classes are not written manually, but defined in yaml-files

As PODIO encourages composition over inheritance, there are two high-level categories of definition:

1. Components - can contain any types that allow them to be PODs (including other components)
2. Classes - can contain the same types as components, but in addition references to other objects

The PODIO class generator creates the data model from the yaml file

If users make a choice destroying pure PODness, they are notified at data model creation time

Data Model Definition

Simple Members:

```
RawCalorimeterHit:
  description : “raw calorimeter hit”
  author : “B. Hegner”
  members :
     -  int cellID // The detector specific (geometrical) cell ID.
     -  int amplitude // The amplitude of the hit in ADC counts.
     -  int timeStamp // The time stamp for the hit.
```

Relation to other objects

```
SimCalorimeterHit:
...
OneToOneRelations:
 -  MCParticle particle // The MC particle that caused the hit.
OneToManyRelations:
 -  MCParticle daughters // the daughters of this particle
 -  MCParticle parents   // the parents of this particle
```

Thread-safety

Thread Safety 1/2

Thread-safety is about states, their change, and parallel executions getting an inconsistent view on those

⇒ **keep the number of states to the bare minimum**

⇒ **don't play with globals**

In PODIO there are the following *local states*

1. The actual data in the PODs
2. The reference counting
3. What's contained in the whiteboard

And in case of reading/writing the *I/O with plenty of local and global states*

There are no smart caches and on-demand operations done or triggered by PODIO itself

Thread Safety 2/2

Some of the states can be tackled by protocols/conventions, others by protecting code

1. The actual data in the PODs
   ⇒ follow a convention, e.g. create once, don't change afterwards
2. The reference counting
   ⇒ using atomics
3. Whiteboard
   ⇒ users provide their own thread-safe whiteboard
4. Input/output
   ⇒ Back-end dependent

Now time for discussion…