Augmenting PODIO Serialisation

CERN Summer Student Program 2018

Ali Fatemi
Graeme Stewart, Benedikt Hegner

PODIO

- support the creation and handling of data models in particle physics
- plain-old-data (POD) data structures
- avoiding deep-object hierarchies
- virtual inheritance

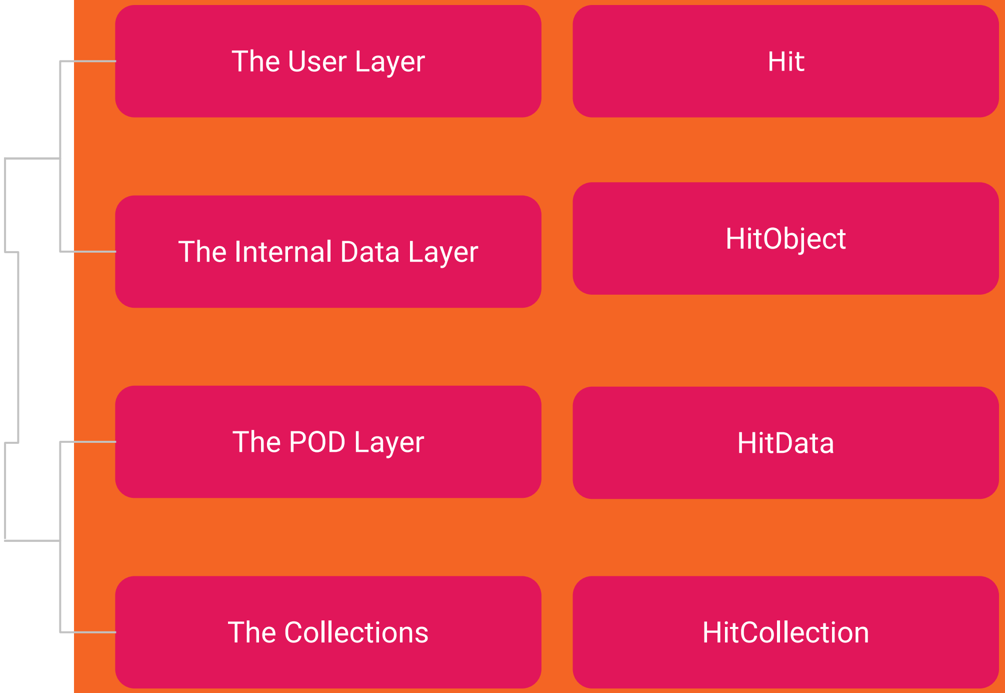
runtime performance & persistency services

Design

1. the concrete data are contained within plain-old-data structures (PODs)
2. user-exposed data types are concrete and do not use inheritance
3. the C++ and Python interface should look as close as possible
4. the user does not do any explicit memory management
5. classes are generated using a higher-level abstraction and code generators

Layout of Objects

four different kind of objects and layers



The User Layer

Hit

The Internal Data Layer

HitObject

The POD Layer

HitData

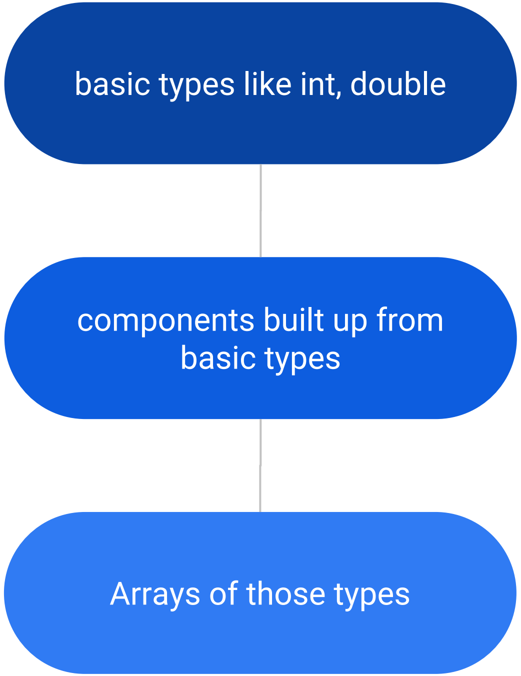
The Collections

HitCollection

Mutability

- data collections may be read-only after creation, or may be still altered
- however, created collections are always immutable after leaving the scope of the creator

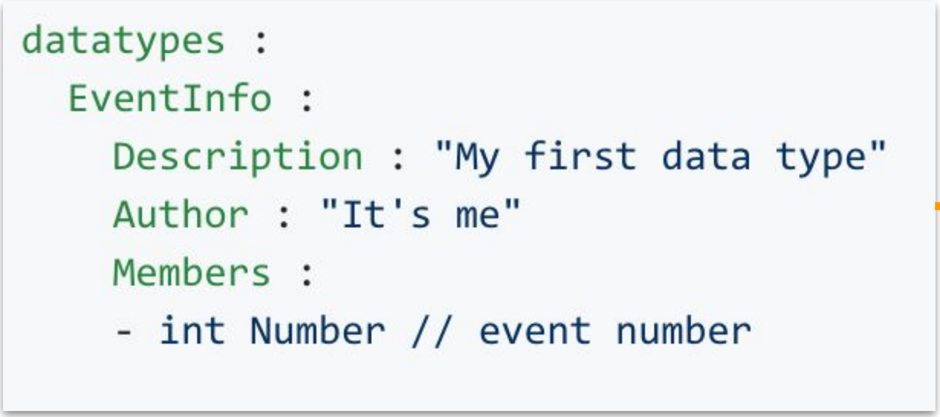
Data Models and Data Model Definitions



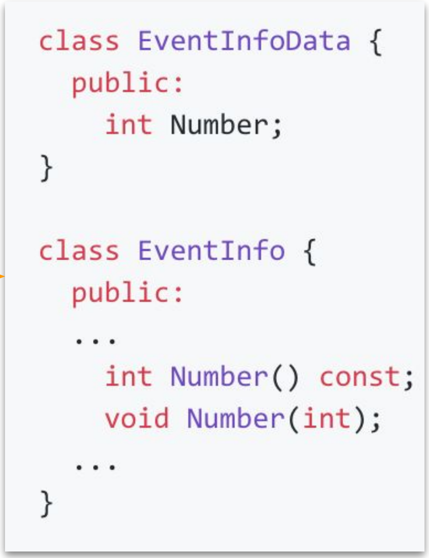
basic types like int, double

components built up from basic types

Arrays of those types



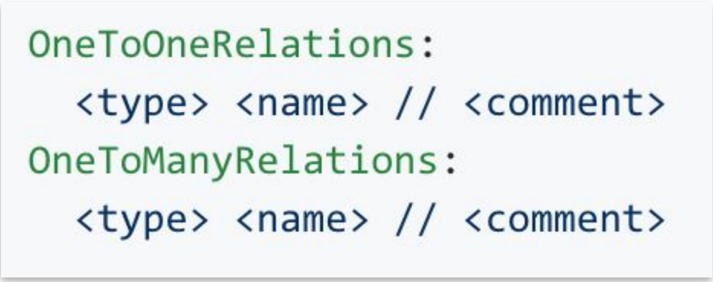
```
datatypes :
  EventInfo :
    Description : "My first data type"
    Author : "It's me"
    Members :
    - int Number // event number
```



```
class EventInfoData {
  public:
    int Number;
}

class EventInfo {
  public:
  ...
    int Number() const;
    void Number(int);
  ...
}
```

Definition of custom data classes



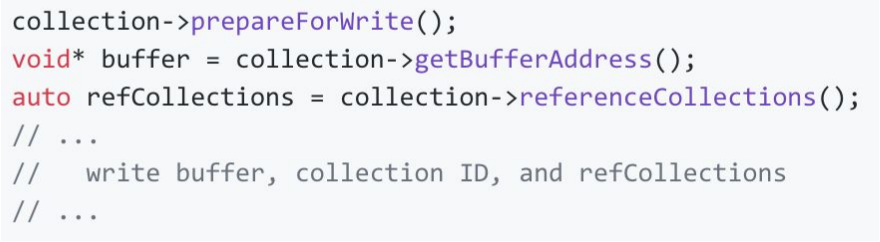
```
OneToOneRelations:
  <type> <name> // <comment>
OneToManyRelations:
  <type> <name> // <comment>
```

Definition of references between objects

Persistency

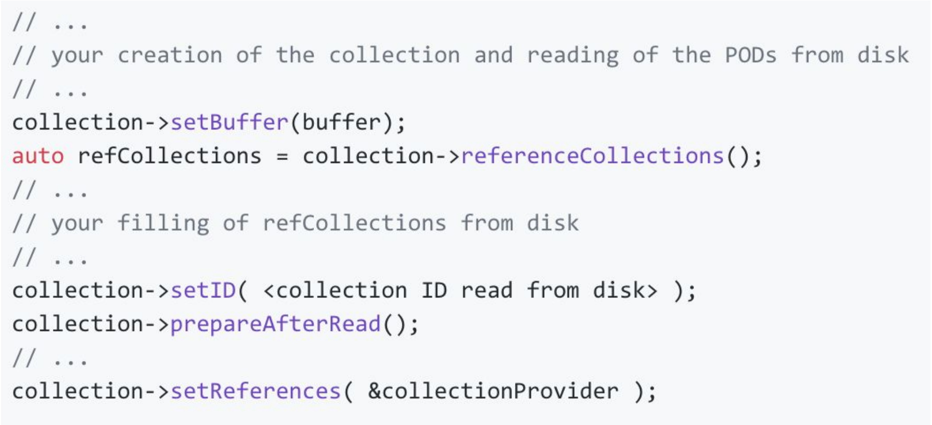
Writing Back-End

1. the ID of the collection,
2. the vector of PODs in the collection, and
3. the relation information in the collection



```
collection->prepareForWrite();
void* buffer = collection->getBufferAddress();
auto refCollections = collection->referenceCollections();
// ...
//   write buffer, collection ID, and refCollections
// ...
```

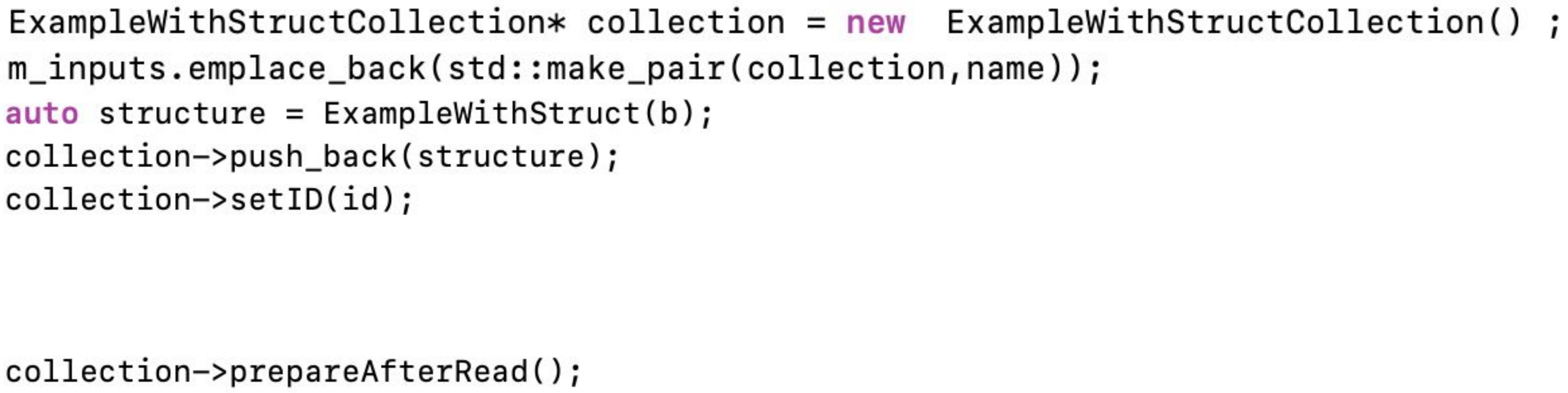
Reading Back-End



```
// ...
// your creation of the collection and reading of the PODs from disk
// ...
collection->setBuffer(buffer);
auto refCollections = collection->referenceCollections();
// ...
// your filling of refCollections from disk
// ...
collection->setID( <collection ID read from disk> );
collection->prepareAfterRead();
// ...
collection->setReferences( &collectionProvider );
```

Sequential Files (ASCII)

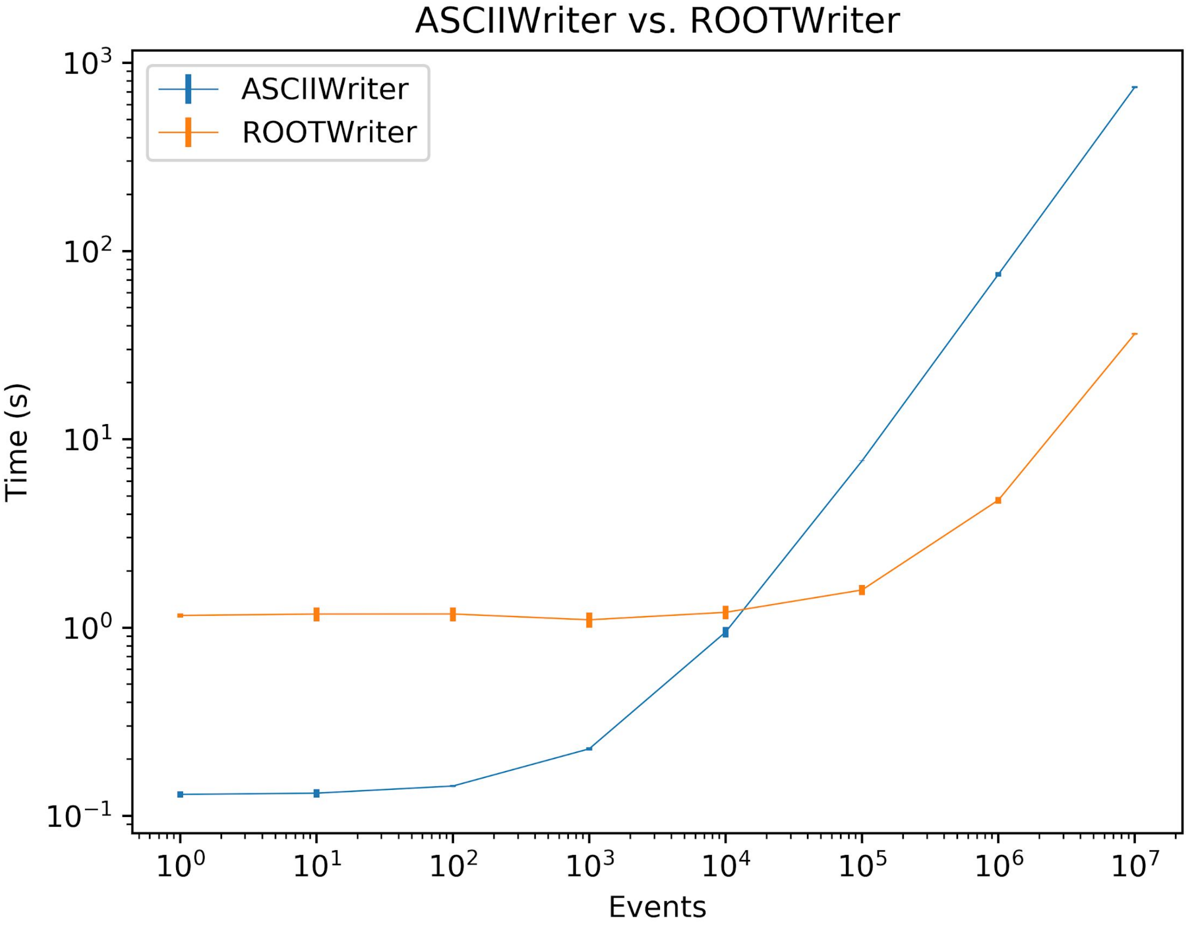
ASCIIReader

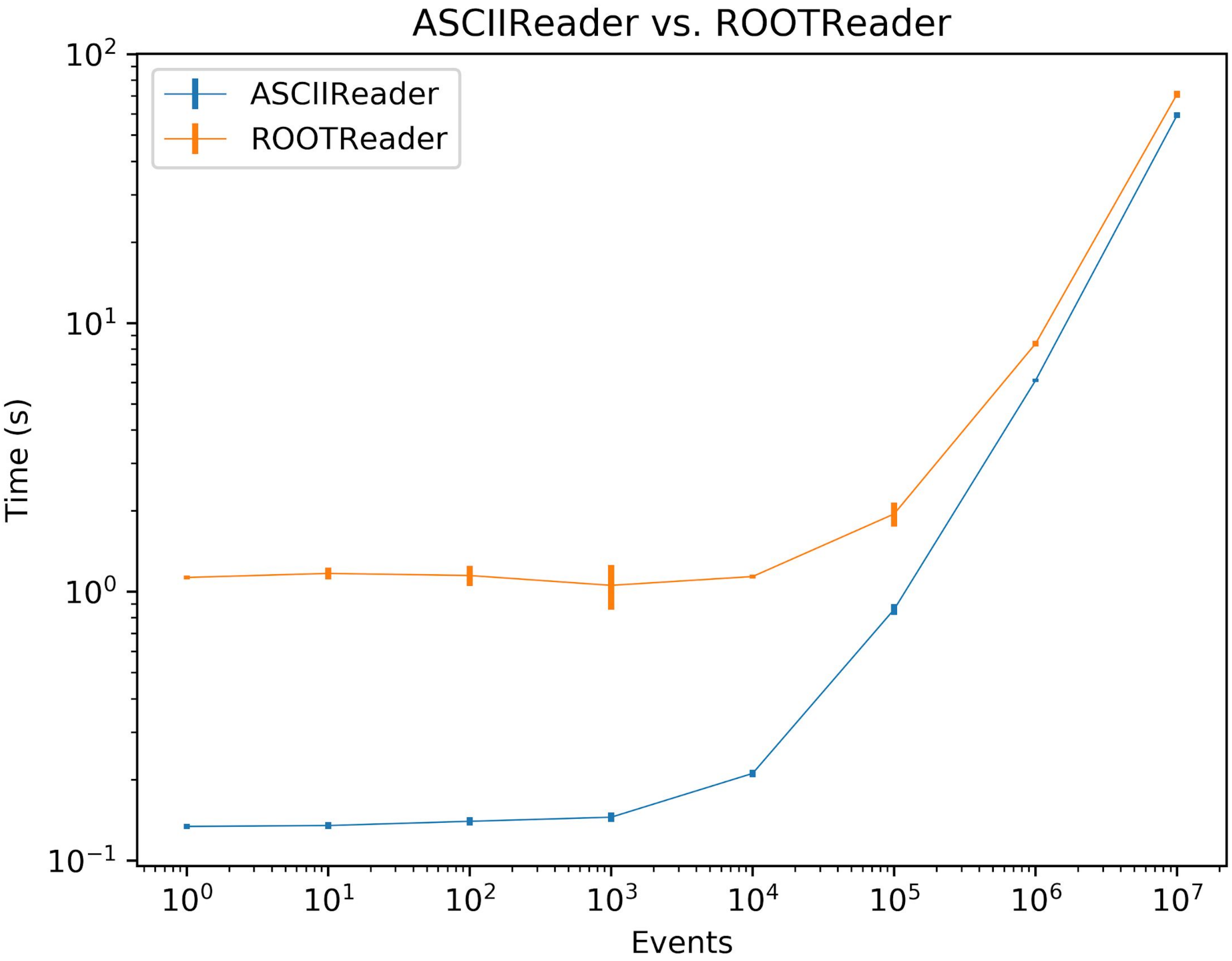


```
ExampleWithStructCollection* collection = new  ExampleWithStructCollection() ;
m_inputs.emplace_back(std::make_pair(collection,name));
auto structure = ExampleWithStruct(b);
collection->push_back(structure);
collection->setID(id);




collection->prepareAfterRead();
```

ASCIIWriter vs. ROOTWriter
ASCIIWriter
ROOTWriter
Time (s)
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
Events
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}


ASCIIReader vs. ROOTReader
ASCIIReader
ROOTReader
Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
Events
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}

ROOT Writing

~20x

ASCII Reading*

~1.2x

ROOT File Size

~5x

Less Memory Consumption

What’s next?

- Automatically handling data types in reader
- Serialising/deserialising object links
- An binary writer and reader
- More tests

Thanks