Contents

Introduction

PODIO, or plain-old-data I/O, is a C++ library to support the creation and handling of data models in particle physics. It is based on the idea of employing plain-old-data (POD) data structures wherever possible, while avoiding deep-object hierarchies and virtual inheritance. This is to both improve runtime performance and simplify the implementation of persistency services.

At the same time it provides the necessary high-level functionality to the physicist, such as support for inter-object relations, and automatic memory-management. In addition, it provides a (ROOT assisted) Python interface. To simplify the creation of efficient data models, PODIO employs code generation from a simple yaml-based markup syntax.

To support the usage of modern software technologies, PODIO was written with concurrency in mind and gives basic support for vectorization technologies.

This document describes first how to define and create your own data model, then how to use the created data.

Afterwards it will explain the overall design and a few of the technical details of the implementation.

Quick-start

An up-to-date installation and quick start guide for the impatient user can be found on the PODIO github page.

PODIO class definition syntax

PODIO uses a user-written data model definition file

Definition of custom components

A component is just a flat struct containing data. it can be defined via:

```
components:
  # My example component
  MyComponent:
    x : float
    y : float
    z : float
    a : AnotherComponent
```

The purpose of components is to support the patter of composition rather than inheritance when building higher level data classes. Components can only contain simple data types and other components.

Definition of custom data classes

This package allows the definition of data types using a simple syntax. This is to ease the creation of optimised data formats. Here an excerpt from "datamodel.yaml" for a simple class, just containing one member of the type `int`.

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

Using this definition, three classes will be created: `EventInfo`, `EventInfoData` and `EventInfoCollection`. These have the following signature:

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

class EventInfo {
```

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

Defining members

The definition of a member is done in the `Members` section in the form:

```
Members:
  <type> <name> // <comment>
```

where `type` can be any buildin-type or a `component`.

Definition of references between objects:

There can be one-to-one-relations and one-to-many relations being stored in a particular class. This happens either in the `OneToOneRelations` or `OneToManyRelations` section of the data definition. The definition has again the form:

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

Explicit definition of methods

In a few cases, it makes sense to add some more functionality to the created classes. Thus this library provides two ways of defining additional methods and code. Either by defining them inline or in external files

```
ExtraCode:
  declaration: <string>
  implementation : <string>
  declarationFile: <string> (to be implemented!)
  implementationFile: <string> (to be implemented!)
```

The code being provided has to use the macro `{name}` in place of the concrete name of the class.

Examples for Supported Interface

The following snippets show the support of PODIO for the different use cases. The event `store` used in the examples is just an example implementation, and has to be replaced with the store used in the framework of your choice.

Object Ownership

Every data created is either explicitly owned by collections or automatically garbage-collected. There is no need for any `new` or `delete` call on user side.

Object Creation and Storage

Objects and collections can be created via factories, which ensure proper object 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);
```

In addition, individual objects can be created in the free. If they aren't attached to a collection, they are automatically garbage-collected:

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

In this respect all objects behave like objects in Python.

Object References

The library supports the creation of one-to-many relations:

```
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);
```

The references can be accessed via iterators on the referencing objects

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

or via direct accessors

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

If asking for an entry outside bounds, a std::out_of_range exception is thrown.

Looping through Collections

Looping through collections is supported in two ways. Via iterators:

```
for(auto i = hits.begin(), end = hits.end(); i != end; ++i) {
  std::cout << i->energy() << std::endl;
}
```

and via direct object access:

```
for(int i = 0, end = hits.size(), i != end, ++i){
  std::cout << hit[i].energy() << std::endl;
}
```

Support for Notebook-Pattern

The `notebook pattern` uses the assumption that it is better to create a small copy of only the data that are needed for a particular calculation. This pattern is supported by providing access like

```
auto x_array = hits.x<10>(); // returning values of
auto y_array = hits.y<10>(); // the first 10 elements
```

The resulting `std::array` can then be used in (auto-)vectorizable code. If less objects than requested are contained in the collection, the remaining entries are default initialized.

EventStore functionality

The event store contained in the package is for *educational* purposes and kept very minimal. It has two main methods:

```
/// create a new collection
template<typename T>
T& create(const std::string& name);

/// access a collection.
template<typename T>
const T& get(const std::string& name);
```

Please note that a `put` method for collections is not foreseen.

Object Retrieval

Collections can be retrieved explicitly:

```
auto& hits = store.get<HitCollection>("hits");
if (hits.isValid()) { ... }
```

Or implicitly when following an object reference. In both cases the access to data that has been retrieved is `const`.

Python Interface The class `EventStore` provides all the necessary (read) access to event files. It can be used as follows:

```
from EventStore import EventStore
store = EventStore(<list of files>)
for event in store:
  hits = store.get("hits")
  for hit in hits:
    ...
```

Design and Implementation Details

The driving considerations for the PODIO design are:

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

The following sections give some more technical details and explanations for the design choices. More concrete implementation details can be found in the doxygen documentation.

Layout of Objects

The data model is based on four different kind of objects and layers, namely

1. user visible (physics) classes (e.g. `Hit`). These act as transparent references to the underlying data,
2. a transient object knowing about all data for a certain physics object, including inter-object references (e.g. `HitObject`),
3. a plain-old-data (POD) type holding the persistent object information (e.g. `HitData`), and
4. a user-visible collection containing the physics objects (e.g. `HitCollection`).

These layers are described in the following.

The User Layer

The user visible objects (e.g. `Hit`) act as light-weight references to the underlying data, and provide the necessary user interface. For each of the data-members and one-to-one relations declared in the data model definition, corresponding setters and getters are created. For each of the one-to-many relations a vector-like interface is provided.

With the chosen interface, the code written in C++ and Python looks almost identical, if taking proper advantage of the `auto` keyword.

The Internal Data Layer

The internal objects give access to the object data, i.e. the POD, and the references to other objects. These objects inherit from `podio::ObjBase`, which takes care of object identification (`podio::ObjectID`), and object-ownership. The `ObjectID` consists of the index of the object and an ID of the collection it belongs to. If the object does not belong to a collection yet, the data object owns the POD containing the real data, otherwise the POD is owned by the respective collection. For details about the inter-object references and their handling within the data objects please see below.

The POD Layer

The plain-old-data (POD) contain just the data declared in the `Members` section of the datamodel definition. Ownership and lifetime of the PODs is managed by the other parts of the infrastructure, namely the data objects and the data collections.

The Collections

The collections created serve three purposes:

1. giving access to or creating the data items
2. preparing objects for writing into PODs or preparing them after reading
3. support for the so-called notebook pattern

Vectorization support / notebook pattern

As an end-user oriented library, PODIO provides only a limited support for struct-of-arrays (SoA) memory layouts. In the vision, that the data used for heavy calculations is best copied locally, the library provides convenience methods for extracting the necessary information from the collections. More details can be found in the examples section of this document.

References between objects

The existence of relations between objects has been mentioned several times.

Transient representation TODO

Handling const-correcteness

As a peculiarity, PODIO provides dedicated const and non-const classes, instead of fully trusting the C++ `const` keyword. This is done for two reasons - to avoid accidental drop of the const-keyword on user side. And to hide the non-const methods in Python. A case for accidental dropping of the `const` keyword in C++ are implicit copies when using the auto keyword.

Advanced Topics

Persistency

The library is build such that it can support multiple storage backends. However, the tested storage system being used is ROOT.

Thread-safety

PODIO was written with thread-safety in mind and avoids the usage of globals and statics. However, a few assumptions about user code and use-patterns were made. The following lists the caveats of the library when it comes to parallelization.

Setting user data

The non-const variants of the user classes are in no way protected for concurrent update operations. However, if only dealing with the const-variants it is guaranteed that no internal state or cache prevents concurrent operations.

Serialization

During the calls of `prepareForWriting` and `prepareAfterReading` on collections other operations like object creation or addition will lead to an inconsistent state.

Not-thread-safe components

The example event store provided with PODIO is as of writing not thread-safe. Neither is the chosen serialization.

Implementing a transient Event Class

PODIO contains one example `podio::EventStore` class. To implement your own transient event store, the only requirement is to set the collectionID of each collection to a unique ID on creation.

Implementing a persistency layer on top of PODIO

PODIO contains one example persistency implementation, based on ROOT. However, it is possible to implement other serialization solution on top of PODIO.

Writing Back-End

There is no interface a writing class has to fulfill. It only needs to take advantage of the interfaces provided in PODIO. To persistify a collection, three pieces of information have to be stored:

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

Before writing out a collection, the data need to be put into the proper structure. For this, the method `prepareForWrite` needs to be invoked. In the case of trivial POD members this results in a no-op. In case, the collection contains references to other collections, the pointers are being translated into `collID:objIndex` pairs. A serialization solution would - in principle - then look like this:

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

Reading Back-End

There are two possibilities to implement a reading-back end. In case one uses the `podio::EventStore`, one simply has to implement the `IReader` interface.

If not taking advantage of this implementation, the data reader or the event store have to implement the `ICollectionProvider` interface. Reading of a collection happens then similar to:

```
...
  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 );
```

The strong assumption here is that all references are being followed up directly and no later on-demand reading is done.

Necessary reflection information

To be written