ATLAS core software and EDM

Scott Snyder

Brookhaven National Laboratory, Upton, NY, USA

July 25, 2018

The facts of life

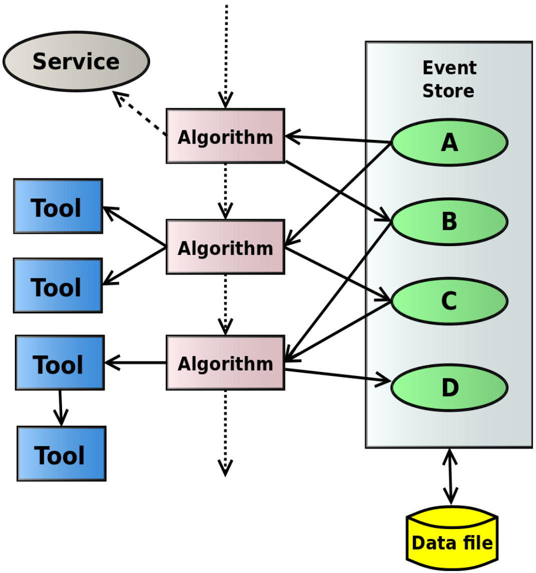
- Atlas offline software is about 5M LOC (excluding external libraries).
  - Roughly 80% C++, 20% Python (mostly job configuration).
- Written by hundreds of contributors over several decades.
  - "... any process which is not forbidden by a conservation law actually does take place with appreciable probability." — Gell-Mann
- Under continual evolution of both core and algorithmic code.
  - There are often several methods/interfaces for doing something, that were introduced at different times.
  - Will generally show here the currently recommended patterns.
  - But some or even most of the code in the repository may use older ways.
- Major evolution in progress: multithreading.
- Currently building with gcc 6; nearly ready to switch to gcc 7.
  - Can compile with gcc8/clang with a few patches.
  - Compiling with icc may be a significant effort. icc appears to generate incorrect code for some core components.

Athena/Gaudi component model

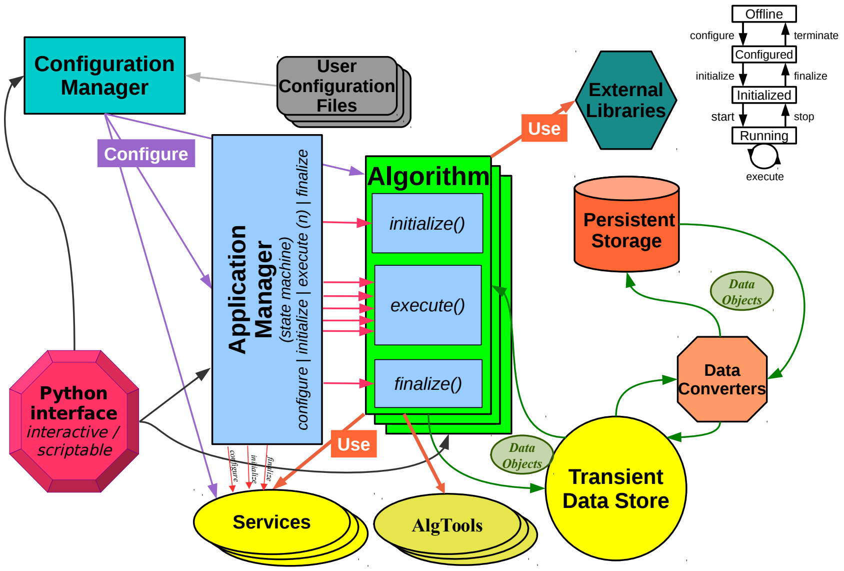
Athena and Gaudi

Athena uses Gaudi, shared with LHCb and other experiments.

- Dynamically loadable *components*.
  - Including algorithms, tools, services.
- Ideally used via abstract interfaces.
- Algorithms and tools can own other tools.
- Services are singletons.
- Identified with a type (C++ type) and name (instance).
- Each component has a list of *properties*.



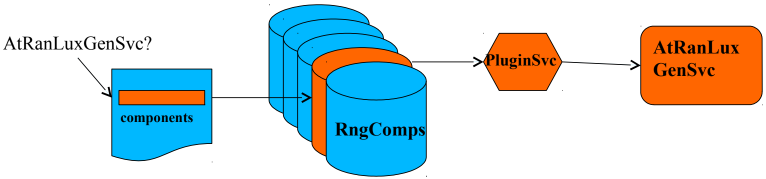
Gaudi/Athena Components and States



Interfaces, Plug-ins, Factories and all that

Gaudi component model inspired by Microsoft COM.

- **Components implement an interface, and use other components through an interface**
- **Components are packaged in Shared Object Libraries (DSO/DLL) and declared in a “components” manifest file.**
- **Gaudi PluginSvc uses manifest to locate which DSO contains a given component dictionary, dlopen-it, and create an instance using a factory method**



Gaudi components

- Four types: Service, algorithm, tool, converter.
- Identified by *type* (C++ class name) and *name* (identifies instance). Ex: `CaloClusterMaker/CaloTopoCluster`.
- Manifest ("components") files generated automatically during build map class names to *component* DSOs.
  - Generally, no other library should link against a component library.
- Resolving component names and loading libraries managed by Gaudi PluginService.
- Each component has a list of named *properties* set during job configuration.
  - Correspond to component data members.
  - Special *handle* property types for references between components and for access to event data.

Services

- A global service. Examples:
  - `StoreGateSvc` — transient data store.
  - `MessageSvc` — logging.
  - `AtRndmGenSvc` — random number generator management.
- Single instance for the entire job.
  - So for MT jobs, services should be explicitly thread-safe.
- Ideally (though not necessarily) accessed via an abstract interface.

Example interface definition:

```
class IMyInterface
  : virtual public IInterface // Gaudi interface base class
{
public:
  // Gaudi boilerplate
  DeclareInterfaceID (IMyInterface, 1, 0);
  virtual int getFoo() const = 0;
};
```

Service example

```
class MyService
  // Service with additional interfaces; can add more.
  : public extends<AthService, IMyInterface>
{
public:
  // Constructor.  (Could inherit instead for this example.)
  MyService::MyService (const std:string& name,
                        ISvcLocator* svcloc)
    : base_class (name, svcloc) {}
  // Also finalize(), start(), stop()
  virtual StatusCode initialize() override {
    ATH_MSG_INFO("Initializing; foo is " << m_foo.value());
    return StatusCode::SUCCESS; } // or FAILURE
  virtual int getFoo() const override { return m_foo; }
private:
  Gaudi::Property<int> m_foo { this, "Foo", "Example" };
};
```

Tools

- Helper component owned by an algorithm, service, or other tool.
  - These are called “private” tools. It has also been possible to have global “public” tools. There is nothing that particularly differentiates public tools from services, so this concept is deprecated. There are still many in the source, though.
- Again, ideally accessed via an abstract interface.
- Coding is nearly the same as for services.
  - Except deriving from AthAlgTool instead of AthService.

Algorithms

- Called once per event to process event data.
- Event data taken from the transient data store.
- Derive from AthAlgorithm (or AthReentrantAlgorithm).
- Provide the interface:

```
virtual StatusCode execute();
```

  (slightly different for reentrant algorithms; see later).
- Can be grouped into sequences, which are themselves algorithms.
- Algorithms can set a flag to stop execution of a sequence early (filtering).
- In serial jobs, algorithms are executed in a fixed order specified in job configuration.
- Common pattern: algorithm fetches input/creates empty output, then executes a configurable list of tools.

Component references

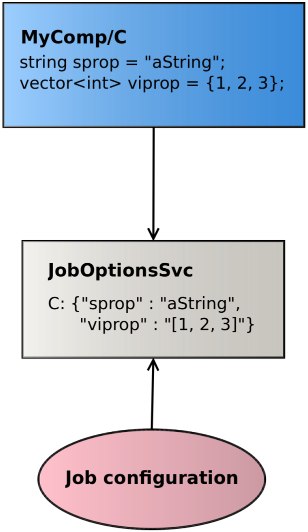
References between components done via properties of type `ServiceHandle<>` and `ToolHandle<>`.

```
class MyAlgorithm : class AthAlgorithm {
  ServiceHandle<IMyInterface> m_svc
    { this, "TheSvc", "MySvc", Comment" };
  ToolHandle<IMyTool> m_tool
    { this, "TheTool", "MyTool/tool1", "Comment" };
public:
  using AthAlgorithm::AthAlgorithm;
  virtual StatusCode initialize() override {
    ATH_CHECK( m_svc.retrieve() );
    ATH_CHECK( m_Tool.retrieve() ); }
  virtual StatusCode execute() override {
    int foo = m_svc->getFoo();
    ATH_CHECK( m_tool->doSomething (foo) );
    return StatusCode::SUCCESS;  }
};
```

Job configuration

Component properties

- Each component has a list of named properties.
  - Of various C++ types.
  - Corresponding to data members in the component C++ class.
- Also can have properties representing references to other components: ToolHandle and ServiceHandle.
- When a component initializes, it queries JobOptionsSvc for its properties.
- JobOptionsSvc holds property settings for each component, as strings.
- JobOptionsSvc populated by job configuration.



Python component configuration interface: Configurable

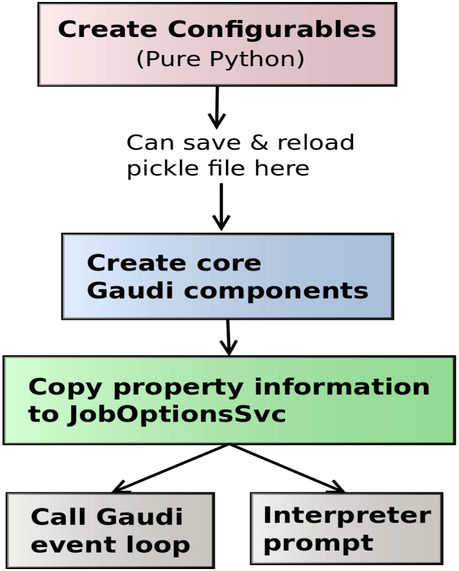
Each component has a Python “Configurable” class to collect information during job configuration:

```
from MyComponents.MyComponentsConf import MyComponent
# Create an instance of MyComponent named mycomp and
# set some properties.
mycomp = MyComponent ('mycomp',
                      sprop = "some string",
                      viprop = [3, 2, 1])
# Can change/set properties later.
mycomp.iprop = 10
mycomp.sprop = "some other string"
```

Configurables

- Configurables are pure Python classes.
  - Can be pickled, etc.
- Generated automatically during build from the shared libraries containing the components.
- Singleton behavior: `MyComponent("foo")` always gives the *same* object.
- Hold property settings, and can later transport them to JobOptionsSvc.

athena.py (python is Athena binary):


Create Configurables
(Pure Python)
Can save & reload pickle file here
Create core Gaudi components
Copy property information to JobOptionsSvc
Call Gaudi event loop
Interpreter prompt

Component references

Configurables can reference other Configurables to build up a complete job configuration.

```
from MyComponents.MyComponentsConf import *

# Create a Service; register with ServiceMgr.
myserv = MyService ('myserv', parm=1)
svcMgr += myserv

# An algorithm using this service plus a tool.
myalg = MyAlgorithm ('myalg',
                     service = myserv,
                     tool = MyTool ('mytool',
                                    arg='something'))

# Schedule this algorithm to run in sequence.
topSequence += myalg
```

Other configuration helpers

- `include`: Textual inclusion of another Python fragment.

```
include('Calorimeter/CaloRec.py')
```

- Job configuration flags:

```
from CaloRec.CaloRecFlags import jobproperties as jp
jp.CaloRecFlags.doTileCorrection = True
if jp.CaloRecFlags.doTileMuID():
    ...
```

- Some properties set automatically from input file metadata:

```
tool = MyTool ('mytool',
               energy = jobproperties.Beam.energy)
```

Done? Maybe not.

- Doesn't scale — becomes unmanageable with thousands of components.
- Everything in the global namespace — easy to have collisions.
- Different pieces of the configuration can try to configure the same component in different ways.
- Many components have prerequisites that must be configured first.
  - Difficult even for experts to get it right.
- Several attempts made in the past to add more structure.
  - Put configuration into imported Python modules.
  - Various registries for looking up configuration information.
- Helped somewhat — but now different parts of the job are configured in different ways.

Run 3 job configuration

- Working on overhauling how job configuration works for run 3.
- Subject of a separate poster; only brief summary here.
- No global namespace; make dependencies explicit.
- Everything in imported modules; no include.
- Configuration of component also configures all its dependencies.
  - Configurations are self-contained.
- Modules can be run stand-alone and concatenated:
  - Need to remove duplicates if the some component is configured multiple times.
- Will probably be used for major parts of the Run 3 configuration.
  - May not have sufficient effort available to convert everything before Run 3, given other migrations also in progress.
- Trying to keep new code compatible with both Python 2 and 3.

Run 3 job configuration: simple example

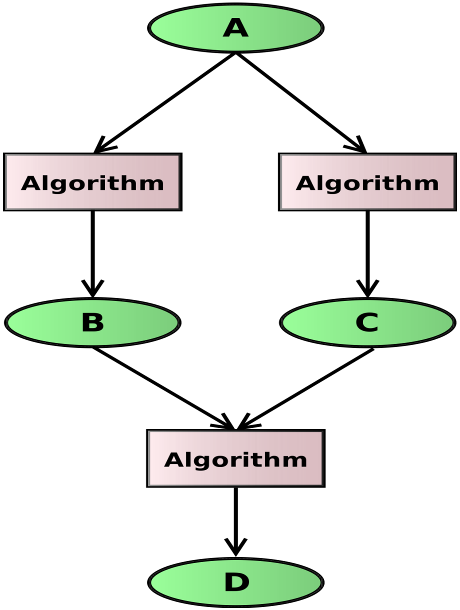
```
def MyAlgoCfg(inputFlags):
   result=ComponentAccumulator()
   isMC=inputFlags.get("AthenaConfiguration.GlobalFlags.isMC"

   # Set up geometry.
   from Geometry.GeomConfig import GeomCfg
   result.addConfig (GeomCfg, inputFlags)

   form MyAlgoPackage.MyAlgoPackageConf import MyAlgo
   myAlgo = MyAlgo(isData = not isMC)
   result.addEventAlgo(myAlgo)
   return result
```

AthenaMT and data access

AthenaMT: Intra-event parallelism



Task scheduling based on the Intel Thread Building Blocks library with a custom graph scheduler.

Algorithms declare their inputs and outputs.

- Scheduler finds an algorithm with all inputs available and runs it as a task.
- "Data flow."

Flexible parallelism within an event.

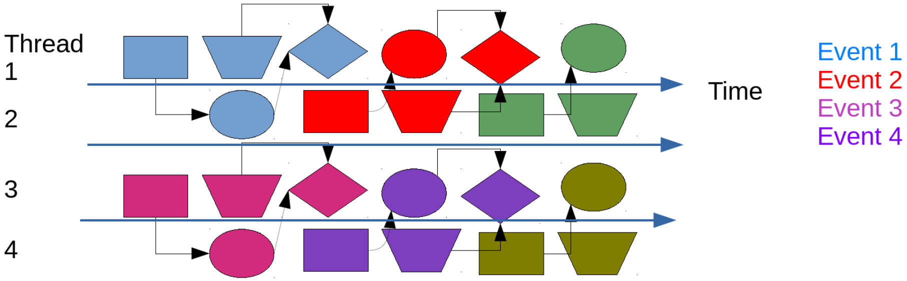
Can still declare sequences of algorithms that must execute in fixed order ("control flow").

AthenaMT: Inter-event parallelism

Allow *multiple* event stores ("slots").

Allows parallelism both within and event and between events.

Number of simultaneous events in flight is configurable.



Different shapes: different algorithms; different colors: different events.

Algorithm dependency declarations

Algorithm data dependencies declared via special properties (HandleKey).

```
SG::WriteHandleKey<X> m_xKey { this, "XKey", "x" };
```

Used together with an *event context* that identifies the particular event slot being used:

```
SG::WriteHandle<X> x (m_xKey, ctx);
ATH_CHECK( x.record (std::make_unique<X>()) );
// Can modify the object until the WriteHandle is deleted.
x->set (something);
```

Context argument may be omitted — will then be read from a thread-local global.

Dependencies of tools will be propagated up to their owning algorithms.

Algorithm types

By default Algorithms are singletons, and a given algorithm cannot be executing simultaneously in more than one thread.

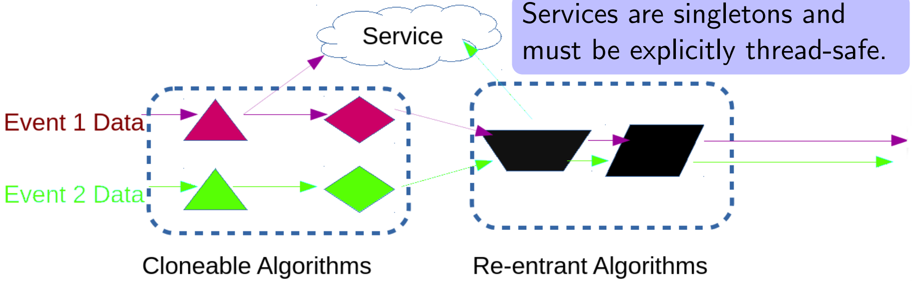
Algorithms may be declared *clonable*. Multiple copies of the Algorithm are made and can be executing simultaneously.

```
    virtual StatusCode execute();
```

Algorithms declared *reentrant* are singletons but may execute in multiple threads. Any internal mutable state must be thread-safe.

```
    virtual StatusCode execute_r(const EventContext&) const;
```

Services are singletons and must be explicitly thread-safe.



Example: algorithm class declaration

```
class ExampleAlg
  : public AthReentrantAlgorithm
{
public:
  using AthReentrantAlgorithm::AthReentrantAlgorithm;
  virtual StatusCode initialize() override;
  virtual StatusCode execute_r (const EventContext& ctx)
    const override;

private:
  // Declare the keys used to access the data: one for reading
  // and one for writing.
  SG::ReadHandleKey<MyDataObj> m_readKey
    { this, "ReadKey", "in" };
  SG::WriteHandleKey<MyDataObj> m_writeKey
    { this, "WriteKey", "out" };
};
```

Example: algorithm initialize

```
StatusCode ExampleAlg::initialize()
{
  // Can make changes to the key properties here.

  // This will check that the properties were initialized
  // properly by job configuration.
  ATH_CHECK( m_readKey.initialize() );
  ATH_CHECK( m_writeKey.initialize() );
  return StatusCode::SUCCESS;
}
```

Example: algorithm execute

```
StatusCode ExampleAlg::execute_r (const EventContext& ctx) const
{
  // Construct handles from the keys.
  // Since this is a reentrant algorithm, we have an explicit
  // event context, which we pass to the handles.
  SG::ReadHandle<MyDataObj> h_read (m_readKey, ctx);
  SG::WriteHandle<MyDataObj> h_write (m_writeKey, ctx);

  // Now we can dereference the read handle to access input data.
  int newval = h_read->val()+1;

  // We make a new object, held by a unique_ptr, and record it
  // in the store using the record method of the handle.
  auto newobj = std::make_unique<MyDataObj> (newval);
  ATH_CHECK( h_write.record (std::move (newobj)) );
  return StatusCode::SUCCESS;
}
```

Conditions

Calibrations, etc. depending on event number or time.

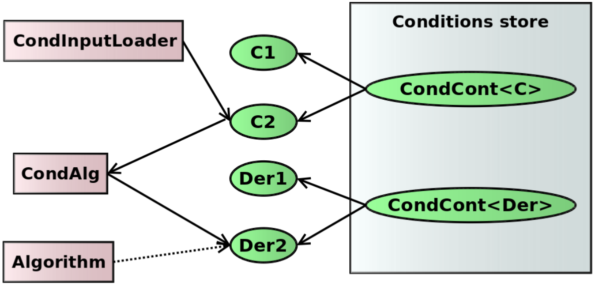
Different events may use different conditions versions.

Conditions store holds potentially multiple versions of conditions objects.

CondInputLoader algorithm loads needed conditions for each event.

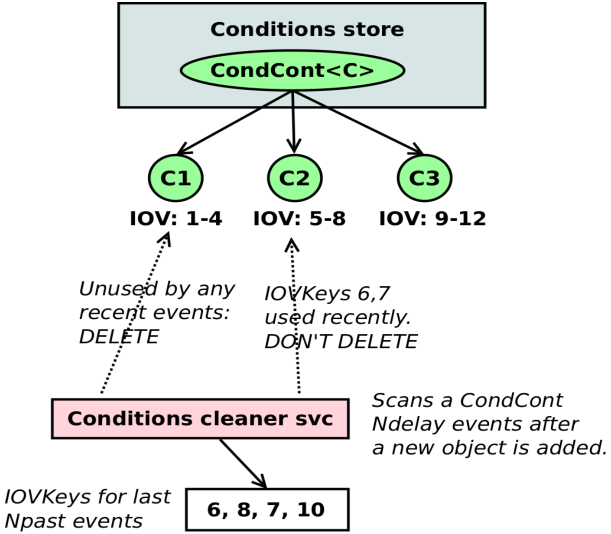
To apply a transformation to conditions data, use a ‘conditions algorithm’ acting on data in the conditions store.

Algorithms access conditions data via handles.



```
ReadCondHandle<Der> h (key, ctx);
```

Scheduler knows about dependencies and schedules them accordingly.


Conditions store
CondCont<C>
C1
C2
C3
IOV: 1-4
IOV: 5-8
IOV: 9-12
Unused by any recent events: DELETE
IOVKeys 6,7 used recently. DON'T DELETE
Conditions cleaner svc
Scans a CondCont Ndelay events after a new object is added.
IOVKeys for last Npast events
6, 8, 7, 10

After an object is added to a conditions container, we clean old conditions in that container some number of events later. The oldest objects in the container that have not been used by any recent events are deleted.

Conditions example

Reading conditions data works just like reading event data except for the handle type.

```
class ExampleAlg : public AthReentrantAlgorithm
{ ...
  SG::ReadCondHandleKey<CondObj> m_condKey
    { this, "CondKey", "condobj" };

StatusCode ExampleAlg::execute_r (const EventContext& ctx) const
{
  SG::ReadCondHandle<CondObj> condObj (m_condKey, ctx);
  float scaleFac = condObj->scaleFac();
```

Conditions objects being read from the database need to be associated with the proper database key in job configuration.

Conditions algorithm example: definition

```
class CondAlg : public AthReentrantAlgorithm
{
public:
  using AthReentrantAlgorithm::AthReentrantAlgorithm;
  virtual StatusCode initialize() override;
  virtual StatusCode execute_r (const EventContext& ctx) const over
private:
  SG::ReadCondHandleKey<CondObj1> m_condObj1Key
    { this, "CondObj1Key", "condobj1" };
  SG::WriteCondHandleKey<CondObj2> m_condObj2Key
    { this, "CondObj2Key", "condobj2" };
};
```

Conditions algorithm example: initialize

```
StatusCode CondAlg::initialize()
{
  ATH_CHECK( m_condObj1Key.initialize() );
  ATH_CHECK( m_condObj2Key.initialize() );
  return StatusCode::SUCCESS;
}
```

Conditions algorithm example: execute

```
StatusCode CondAlg::execute_r (const EventContext& ctx) const
{
  SG::ReadCondHandle<CondObj1> condObj1 (m_condObj1Key, ctx);
  auto c2 = std::make_unique<CondObj2> (condObj1->something());

  // Propagate validity range from input to output.
  // For the case of multiple inputs, there are helpers to find
  // the intersection of all input validity ranges.
  EventIDRange range;
  ATH_CHECK( condObj1.range(range) );

  SG::WriteCondHandle<CondObj2> condObj2 (m_condObj2Key, ctx);
  ATH_CHECK( condObj2.record (range, std::move(c2)) );
}
```

StoreGate

- Service that holds transient data.
- Different instances corresponding to different data lifetimes:
  - Event store: Data for the current event. Internally split into a separate store for each slot.
  - Detector store: Data describing the detector which doesn't change during the job. (Currently also contains some conditions data.)
  - Conditions core: Data with a limited range of validity (run/event or time based).
  - Metadata stores: Used to record and propagate data describing a set of events.
  - Pileup stores: Used for pileup simulation, which overlaps multiple input events.
- Essentially a map from (type,key) pairs to DataProxy instances.
  - Same proxy can be entered with different types/keys (symlinks/aliases).
- type represented by an integer ID (Class ID), assigned via a macro in the class headers.
- DataProxy implements deferred reading/creation of objects.
- Stores provide abstract IProxyDict interface, used by handles.

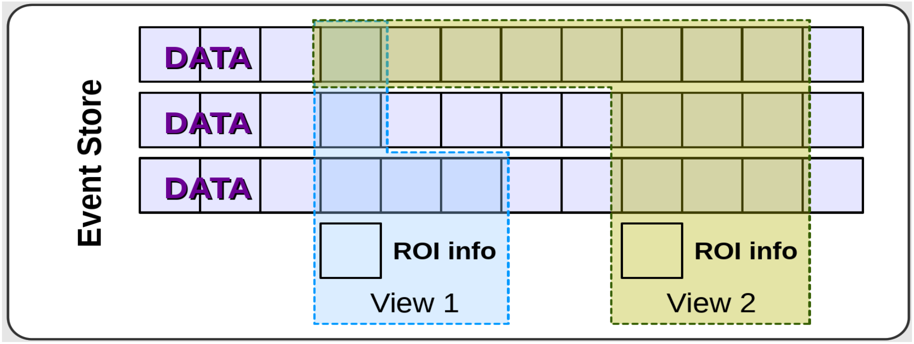
Trigger

Want high-level trigger to use the same algorithms as the offline code.

For performance, trigger does reconstruction only within geometrically limited regions of interest (ROI).

EventView: Stores data for one ROI and implements the same interface as the full event store.

Algorithms that access data via handles can transparently run in an EventView rather than the full store.



Algorithm/tool/service migration

- Algorithms/tools need to change to handles to access event/conditions data.
- Change to using conditions algorithms rather than callbacks.
- Event data must not be modified once recorded in the event store.
- Avoid thread-unfriendly code: use of statics, const-correctness violations.
- Services need to be explicitly thread-safe.
- Reentrant algorithms that have mutable data must be explicitly thread-safe.

Thread-safety static checker

- Have a static checker available to flag some thread-safety problems.
    - Mostly relating to const-correctness and use of static data.
- Set of checks similar to that done by the CMS checker.
    - But implemented within gcc rather than clang, so they can run as part of the default build.
- Gives warnings like:

```
ArenaSharedHeapSTLAllocator.icc:499:10: warning: Static
 expression 'SG::ArenaSharedHeapSTLAllocator<Payload>::s_index'
 passed to pointer or reference function argument of
 'SG::ArenaHeapAllocator* SG::ArenaSharedHeapSTLHeader::get_pool(si
 within function 'void SG::ArenaSharedHeapSTLAllocator<T>::get_pool
 may not be thread-safe.
   m_pool = m_header->get_pool<T> (s_index);
```

- Also has checks related to naming conventions and other coding style issues.

Data model

DataVector

- Objects stored StoreGate may be of arbitrary type.
- Aside from a few singleton objects and some special cases for raw data, most event data objects are specializations of `DataVector`.
- `DataVector<T>` acts like `std::vector<T*>` with some additional features:
  - Optional strict-ownership semantics.
  - Container covariance.
  - `const` propagation to elements.
  - Auxiliary store.

DataVector: Ownership semantics

- By default, a DataVector takes ownership of objects it contains:

```
auto v = std::make_unique<DataVector<int> >();
// DataVector takes ownership of this object.
v->push_back (std::make_unique<int> (10));
// Deletes existing object.
(*v)[0] = std::make_unique<int> (12);
```

- Copying between owning containers is not allowed:

```
DataVector<int> v1;
v1.push_back (std::make_unique<int> (10));
DataVector<int> v2 (10);
v2[0] = v1[0];  // Will cause an assertion failure.
```

DataVector: View containers

- A DataVector can be initialized so that it doesn't own its elements.

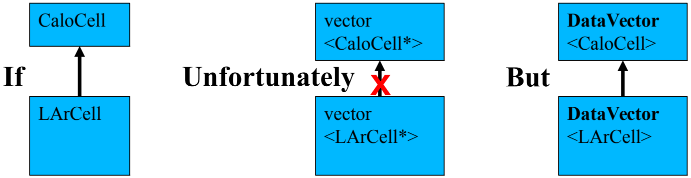
```
DataVector<int> v1 = ...;
DataVector<int> v2 (SG::VIEW_ELEMENTS);
// OK --- v2 doesn't take ownership.
v2.push_back (v1->front());
// Copying gives a view container.
DataVector<int> v3 (v1);
```

- Can be used to hold results of selections, or to logically merge together multiple containers.
- Same algorithmic code works in all cases.
- Implies that implementation of DataVector cannot be simply a collection of `unique_ptr`.
- `ViewVector<T>` (deriving from `DataVector<T>`) used for view containers that must be made persistent.

DataVector

Support for "Container Inheritance"

- If LArCell inherits from CaloCell we would like vector<LArCell*> to inherit from vector<CaloCell*>
- Very useful when writing generic Algorithms/AlgTools that loop over cells coming from different sub-detectors
- DataVector provides this functionality with minimal performance impact, if any



On Thursday, in Scott's talk you'll learn much more about DataVector structure, in the context of xAOD containers

DataVector: Container covariance

- With these declarations:

```
    class B {};
    class D : public B {};
    DATAVECTOR_BASE (D, B);
```

  Then DataVector<D> derives from DataVector<B>.

- For type-safety, can only insert via the most-derived type.

```
  DataVector<D>* d = ...;
  DataVector<B>* b = d;
  // Neither of these are allowed (will throw exceptions).
  b->push_back (std::make_unique<D>());
  b->push_back (std::make_unique<B>());
```

- When recorded in SG, can retrieve by any base type.

DataVector: const propagation

- Unlike vector<T*>, const accessors of DataVector<T> will return only const T*.

```
    const DataVector<C>& v = ...;
    v[0]->changeSomething(); // Not allowed.
```

- But

```
  const DataVector<C>& v = ...;
  auto vnew = std::make_unique<DataVector<C> >
    (SG::VIEW_ELEMENTS);
  vnew->push_back (v.front()); // Compilation error
```

DataVector: const propagation

Solve using `ConstDataVector`:

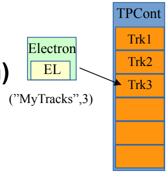
```
const DataVector<C>& v = ...;
auto vnew = std::make_unique<ConstDataVector<
                                DataVector<C> > >
  (SG::VIEW_ELEMENTS);
vnew->push_back (v.front()); // OK
SG::WriteHandle<DataVector<C> > h (key, ctx);
// OK -- object can subsequently be retrieved as
// a const DataVector<C>.
ATH_CHECK( h.record (std::move (vnew));
```

ElementLink

Persistable pointer to an element of a container

- **Usually a data member of a Data Object (e.g. Electron)**
- **Filled when creating the host object (Electron)**

```
ElementLink< TrackParticleContainer > m_trackParticle;
```



Different ways to set an ElementLink

```
m_trackParticle("MyTracks", 3)                   Fastest
m_trackParticle(pTrkColl, 3)                       OK
m_trackParticle (pTrackParticle, *pTrkColl)  Careful! O(N)
```

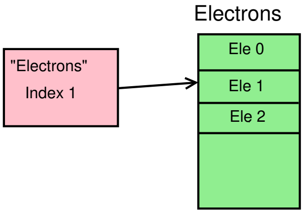
Like ReadHandle, ElementLink only provides const access

```
 float chi2 = (*m_trackParticle)->chiSquared(); //OK
 (*m_trackParticle)->setFitQuality(chi2, nDOF); //ERROR
```

DataLink persistable pointer to a Data Object in SG

ElementLink

- Persistable pointer to a collection element.
- Holds collection name and element index.
- Dereferencing the link returns a reference to the element (a pointer in most cases).
- DataLink points to a single object.



Internally contains a pointer to the DataProxy and a cached copy of the element.

```
const xAOD::Electron& ele = ...;
// Doesn't force the TrackParticle to be constructed.
ElementLink<xAOD::TrackParticleContainer> link =
   ele.trackParticleLink();
std::cout << link.dataID() << "/" << link.index() << "\n";
// This forces the TrackParticle to be read.
const xAOD::TrackParticle* part = *link;
// This does the same thing.
part = ele->trackParticle();
```

Desires for Run 2

Several Run 1 types supported adding extra named pieces of data — "decorations" — to elements of containers.

- Originally done to be able to separate pieces of the structure for I/O.
- Several different, incompatible implementations.
- Can we unify this?

Data stored as, essentially, "array of structures."

- Poor locality of reference.
- "Structure of arrays" might be better.
- Can we still make it look like a collection of structures?

Make data easily and efficiently readable from ROOT.

- Avoid copies.
- Partial object reading / writing.
- User extensibility for analysis.

Keep existing interfaces as must as possible.

Run 2 event data model

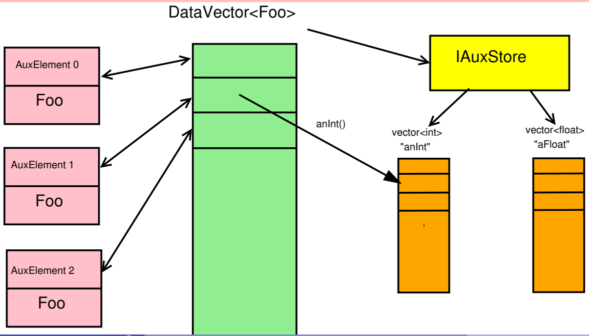
Analysis-level event data model redesigned for Run 2. ("xAOD")

- Simplify, and make more directly usable with ROOT.

Based on new "auxiliary data" feature of `DataVector`.

- Attach data of arbitrary type to elements of `DataVector`.
- Data are stored as vectors, managed via separate "auxiliary store" object via abstract interface.
- Object data stored as auxiliary data rather than in the object itself.

Almost all object data stored as auxiliary data.



Auxiliary data

- Most xAOD object data are not stored in the xAOD objects themselves, but in a separate "auxiliary store".
- Object data stored as vectors of values.
  - ("Structure of arrays" versus "array of structures.")
- Allows for better interaction with root, partial reading of objects, and user extension of objects.
- The store is associated with a container.
  - When creating a container, need to create a store too.
  - Generally can't call getters/setters for objects that are not part of a container.
  - Use care when moving objects between containers.
- Use the xAOD class interface to access standard data members.
- User analysis code can add custom "decorations" to objects.

Usage example 1

```
struct C : public AuxElement {
  int anInt() const {
    static const Accessor<int> acc("anInt");
    return acc(*this); }
  int setAnInt (int x) {
    static const Accessor<int> acc("anInt");
    acc(*this) = x; }
}; ...
auto vc = std::make_unique<DataVector<C> >();
// Aux data store.
auto store = std::make_unique<CAuxContainer>();
vc->setStore (store.get());

// Record both vector and store.  Does setStore() if not
// already done.  Aux store key ends with Aux.
SG::ReadHandle<DataVector<C> > h (key, ctx);
ATH_CHECK( h.record (std::move (vc), std::move (store)) );
```

Usage example 2

```
vc->push_back (std::make_unique<C>());
vc->push_back (std::make_unique<C>());

// Set/get auxiliary data through class members.
(*vc)[0]->setAnInt (3);
std::cout << (*vc)[0]->anInt() << "\n";

// Attach additional auxiliary data to objects.
// The Accessor object caches the lookup of an internal
// identifier from the data item name.
static const C::Accessor<int> myInt ("myInt");
myInt((*vc)[0]) = 2;
myInt(*vc, 1) = myInt(vc[0]) + 1;

// Alternate interface that does not cache the lookup.
(*vc)[1]->auxdata<float> ("myFloat") = 1.5;
```

Auxiliary data decorations

```
// A typedef for DataVector<xAOD::Electron>
const xAOD::ElectronContainer& eles = ...;

// Set a decoration on the electron.  (All other objects
// in eles will now have fdecor == 0.)
eles[0]->auxdecor<float>("fdecor") = 1.2;

// Test if a decoration is available and retrieve it.
if (ele.isAvailable<int> ("idecor"))
  i = eles[0]->auxdecor<int> ("idecor");

// If used in a loop, best to cache the name lookup.
SG::AuxElement::Decorator<int> idecor ("idecor");
for (const xAOD::Electron* e : eles) {
  if (idecor.isAvailable (*e))
    i += idecor(*e);
```

Data access: Decoration handles

An xAOD object in the event store may have additional auxiliary data added to it, even if it is locked. “decorations.” Decorations are in some respect like independent data objects, and the scheduler should be aware of them.

WriteDecorHandle

```
class MyAlg { ...
  SG::WriteDecorHandleKey<MyCont> m_key;  }

MyAlg::MyAlg(...) {
  declareProperty ("Key", m_key = "Obj.d"); ...

StatusCode MyAlg::initialize() { ...
  ATH_CHECK( m_key.initialize() );

StatusCode MyAlg::execute_r (const EventContext& ctx) const { ..
  SG::WriteDecorHandle<MyCont, float> h (m_key, ctx);
  for (const MyObj& o : *h) {  // Access the container.
    h (o) = calculate (o);  // Add the decoration.
```

Data access: Decoration handles

ReadDecorHandle

```
class MyAlg { ...
  SG::ReadDecorHandleKey<MyCont> m_key;  }

MyAlg::MyAlg(...) {
  declareProperty ("Key", m_key = "Obj.d"); ...

StatusCode MyAlg::initialize() { ...
  ATH_CHECK( m_key.initialize() );

StatusCode MyAlg::execute_r (const EventContext& ctx) const { ..
  SG::ReadDecorHandle<MyCont, float> h (m_key, ctx);
  for (const MyObj& o : *h) {  // Access the container.
    doSomething (h (o));  // Access the decoration.
```

See `MultiThreadingEventDataAccess` twiki page for more details.

Decoration will be locked when the `WriteDecorHandle` goes away.

Standalone objects

- The auxiliary data for an object are associated with the container. But what if you want to have an object that is not part of a container?
- Call `makePrivateStore` on the object to create a private aux data store associated with the object itself. If the object is later added to a container, the aux data will be copied and the private store deleted. If it is later removed from the container, the private store will be recreated and repopulated.

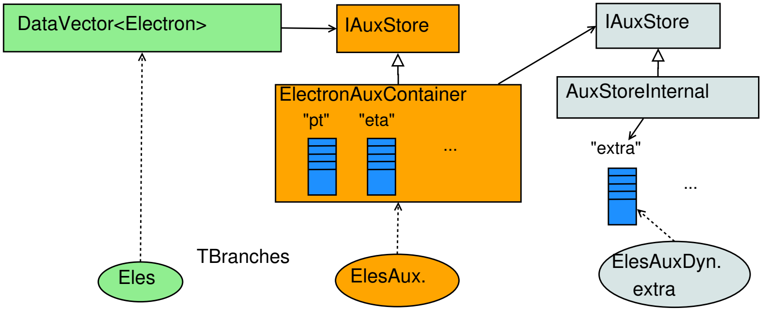
```
class C : public AuxElement {};
auto c = std::make_unique<C>();    c->makePrivateStore();
static const C::Accessor<int> anInt ("anInt");
anInt(*c) = 5;  // In c's private store.
DataVector<C>& vc = ...;
vc. push_back (std::move(c));  // c's private store deleted,
// data copied to container's store.
...
// Get container to give up ownership:
vc.swapElement (vc.begin(), nullptr, c);
// c's private store recreated.
```

I/O interaction

`DataVector<T>` itself saved to ROOT like `vector<T>`

The aux store for xAOD types is a “static” class containing the aux variable vectors as members. Saved and loaded as a single object.

The static store references a “dynamic” store, which can manage arbitrary aux data. New aux data items are added here. Items can be saved and loaded individually from ROOT. Branch objects are set to the vectors managed here.



xAOD implementation

- Each xAOD data class (`DataVector<xAOD::Muon_v1>`) has an associated aux store object (`xAOD::MuonAuxContainer_v1`).
- Both are recorded in StoreGate. The key for the aux store should be the same as the data object with '`Aux.`' appended.
- The xAOD aux store object contains the 'static' aux variables.
- It also holds a `SG::AuxStoreInternal` object which manages any additional 'dynamic' variables.
- Brief rundown of important interfaces:
  - `IConstAuxStore` — Methods for reading data from an aux store (and adding decorations).
  - `IAuxStore` — Methods for changing data in an aux store.
  - `IAuxStoreIO` — Methods for accessing aux data items for writing.
  - `IAuxStoreHolder` — An aux store that can hold another one.

IAuxStore

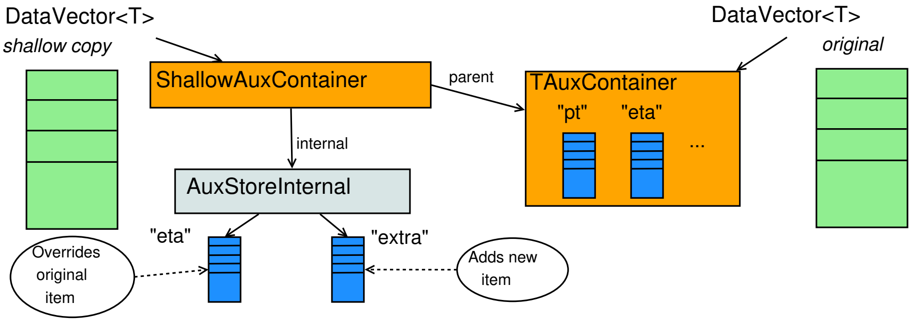
A key feature is the ability to change the auxiliary store implementation through the abstract interface. Some types used:

Each xAOD type has a static auxiliary store chained to a dynamic store.

In trigger: implementation specialized for storage in raw data stream.

On input: implementation allowing on-demand reading of items.

“Shallow copy” store: records writes, forwards reads for unknown items to another store.



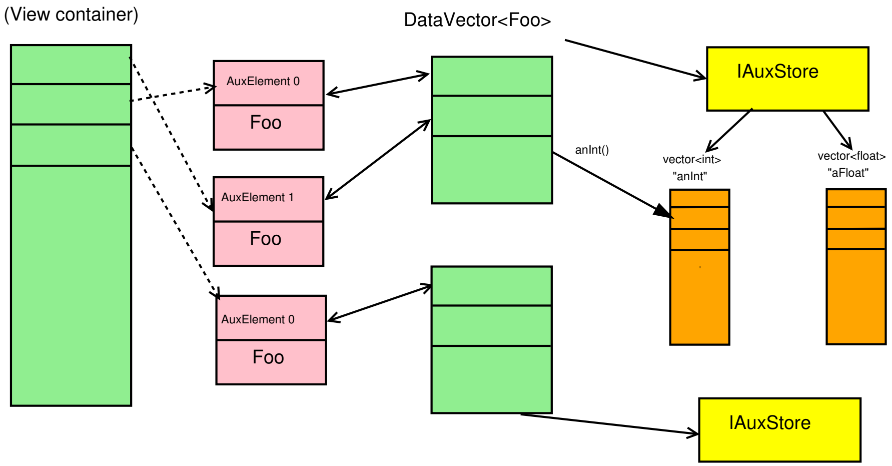
Shallow copy

- Can make a “shallow copy” of a container:

```
const DataVector<Foo>* vec = ...;
auto ret = xAOD::shallowCopyContainer (*vec);
//ret.first is a pointer to a new DataVector.
//ret.second is a pointer to the corresponding aux store.
```

- The copied aux store will contain a reference to the original store.
- Writes will go to the copied aux store.
- Reads will be searched for first in the copied store, then in the original store.
- Reference between stores is via `DataLink`, so a shallow-copied container can be written without problem.
- Limitation: you cannot change the ordering of the elements in the copied container (as indices must remain consistent with the original).
  - May be possible to relax this, but non-trivial. Under consideration.

Auxiliary data and view containers



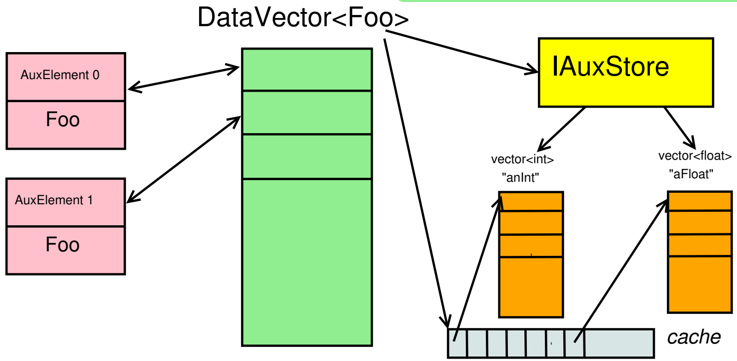
View containers do not have their own aux store; rather, they reference elements that live in other containers. View containers of objects with aux data cannot be safely written as-is.

Implementation notes

Types with aux data derive from `AuxElement`: holds a reference to the container and the element index.

- Updated by vector operations.
- No overhead for types that do not derive from `AuxElement`.

Aux data items identified by small integers via a registry. `DataVector` holds cache of pointers to start of data for each item. If pointer is in cache, access is entirely inlined. Otherwise, calls to out-of-line code.



Vectorizable data access

- Accessing aux data via dereferencing an element pointer is not vectorizable.
- Alternate interfaces exist that could allow vectorization.

```
const DataVector<C>& vc = ...;
static const C::Accessor<float> f ("var");

// Access by index.
float x = 0;
for (size_t i = 0; i < vc.size(); i++) {
  x += f (vc, i);
}

// Get array directly.
const float* fvar = f.getDataArray (vc);
```

Not currently used by any algorithmic code.

xAOD class example

xAODPackage/xAODPackage/versions/MyClass_v1.h

```
namespace xAOD {
class MyClass_v1 : public SG::AuxElement  {
public:
  float x() const;
  void setX (float x);
};
} // namespace xAOD
SG_BASE( xAOD::MyClass_v1, SG::AuxElement );
```

xAODPackage/Root/MyClass_v1.cpp

```
#include "xAODPackage/versions/MyClass_v1.h"
#include "xAODCore/AuxStoreAccessorMacros.h"
namespace xAOD {
AUXSTORE_PRIMITIVE_SETTER_AND_GETTER (MyClass_v1, float,
                                      x, setX)
}
```

xAODPackage/xAODPackage/versions/MyClassAuxContainer_v1.h

```
namespace xAOD {
class MyClassAuxContainer_v1 : public xAOD::AuxContainerBase
public:
  MyClassAuxContainer_v1();
  float x;
};
} // namespace xAOD
SG_BASE( xAOD::MyClassAuxContainer_v1,
         xAOD::AuxContainerBase );
```

xAODPackage/Root/MyClassAuxContainer_v1.cxx

```
namespace xAOD {
MyClassAuxContainer_v1::MyClassAuxContainer_v1() {
  AUX_VARIABLE(x)
} }
```

xAODPackage/xAODPackage/MyClass.h

```
#include "Package/versions/MyClass_v1.h"
namespace xAOD {
  typedef MyClass_v1 MyClass;
}
CLASS_DEF( xAOD::MyClass, 345346456, 1)
```

xAODPackage/xAODPackage/MyClassAuxContainer.h

```
#include "Package/versions/MyClassAuxContainer_v1.h"
namespace xAOD {
  typedef MyClassAuxContainer_v1 MyClassAuxContainer;
}
CLASS_DEF( xAOD::MyClassAuxContainer, 345346457, 1)
```

Extras

Input renaming

Recall that for the MT framework, we want to nearly eliminate the use of overwrite/update.

- Thread safety much more difficult to achieve if data can change.

Need to have a different way of dealing with situations that currently use overwrite/update.

Example: filtering job.

- Takes a file, transform it, and write out a new file.
- Output file should use the same keys as the input file.
- How to set this up without update/overwrite?
- Suggestion: allow changing the names by which input objects are seen in SG.
  - So can rename, for example, `Foo` to `Foo_in` and then have an algorithm that produces `Foo` from `Foo_in`.

May also be useful for fix-type jobs.

Data access: Object/decoration renaming and hiding

Reminder: objects may be renamed on input. Useful for reprocessing jobs.

```
from SGComps.AddressRemappingSvc import addInputRename
# Object CVec/cvec in the input file will appear in SG as cvec2.
addInputRename ('CVec', 'cvec', 'cvec2')
```

Decorations may now be renamed in exactly the same way.

```
# Rename decoration anInt of dvec to anInt2.
addInputRename ('DVec', 'dvec.anInt', 'dvec.anInt2')
# Renaming both object and decoration.
addInputRename ('CVec', 'cvec.anInt', 'cvec_renamed.anInt2')
```

New helper function to hide objects/decorations on input (by renaming):

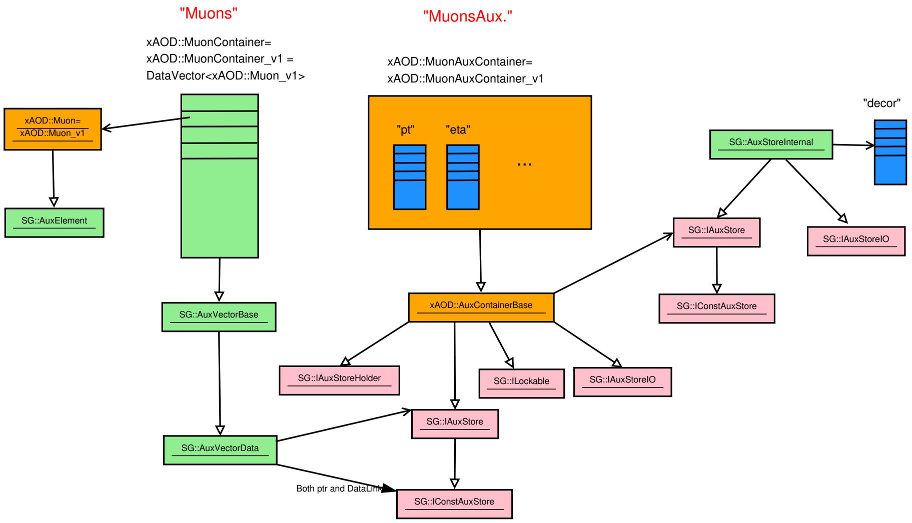
```
from RecExConfig.hideInput import hideInput
hideInput ('CVec', 'cvec')  # Hide object
hideInput ('DVec', 'dvec.anInt')  # Hide decoration
```

Input hiding

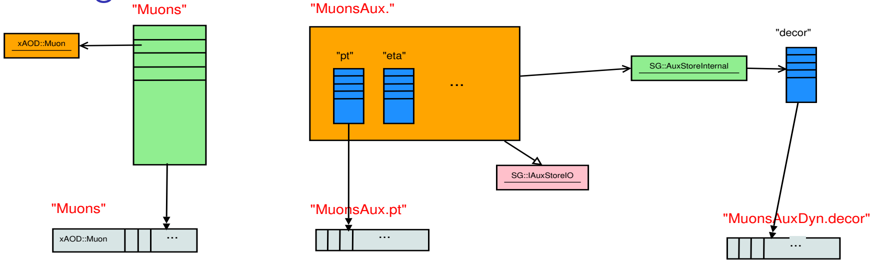
- If an object is listed in a `WriteHandle`, then an object of that name will not be read from the input file.
- Useful for configuring reprocessing jobs.
- In the case where you want to modify an object from an input file, rename the existing object:

```
from SGComps.AddressRemappingSvc import addInputRename
addInputRename ("Foo", "foo", "foo_old")
```

xAOD implementation



Writing aux data



- Container and aux container should both be stored separately in StoreGate. Both should be listed in the item list for writing:

```
fillItemList += ['xAOD::Muon_v1#Muons']
fillItemList += ['xAOD::MuonAuxContainer_v1#MuonsAux.']
```

- `DataVector<T>` will be saved using a custom collection proxy that make makes it appear to root like `vector<T>`.
- Aux store object saved using normal ROOT I/O. Split level fixed to 1.
- Objects supporting the `SG::IAuxStoreIO` interface will have additional “dynamic” branches written (by `RootTreeContainer`).

Reading aux data

- DataVector<T> objects read as usual, again through the custom collection proxy.
  - xAOD Gaudi converter will usually set a DataLink in the DataVector pointing to the correct aux store object (via name).
- Aux store object read via root as usual.
- If the aux store object derives from SG::IAuxStoreHolder:
  - I/O system will create a special aux store object for reading dynamic branches (AuxStoreAPR).
  - This will be attached to the primary store object through the SG::IAuxStoreHolder interface.
  - AuxStoreAPR will collect information about "Dyn." branches and read them on demand.
  - If type information is not available statically, root metadata will be used to handle the branches dynamically.
- In Athena, reading currently reads the entire static aux store object.
  - Should look into providing an option to read it piecewise.

Implementation notes: aux data type registry

- Each distinct aux data item is represented internally by a small integer.
- A singleton registry class maps from aux data item name to id and type (`type_info`).
- `Accessor` class caches the id lookup.
- Registry class also maps from `type_info` to factory objects that can create STL vectors of a given type and provide interfaces to manipulate them generically.
- Such factory objects can come from:
  - Factory objects are pre-defined for built-in C++ types.
  - The template instantiation of an `Accessor`. Instantiating `Accessor<Foo>` will provide a factory object for `Foo`.
  - When reading a file if a factory for `Foo` is needed, the ROOT reflection mechanism will be used to try to dynamically load `AuxVectorFactory<Foo>`. If that fails, a completely generic implementation based on ROOT's I/O mechanism is used.

Implementation notes: index tracking

- To access auxiliary data given a pointer to an collection element, the element needs to know its container and the index within the container.
- Stored within the `AuxElement` base class.
- If a `DataVector` owns its elements, and the element class derives from `AuxElement`, then inserting an element in the container automatically sets the container/index within `AuxElement`.
- All vector operations extended to maintain this index information. (Vectors of aux data are also manipulated as required.)
- If the element class does not derive from `AuxElement`, template techniques are used to elide the index tracking operations. (No overhead if you don't use it.)

Implementation notes: aux data dereferencing 1

- Want as little overhead as possible in accessing aux data.
- Would like to at least preserve the possibility of vectorizing loops over the container that access aux data. Not really possible if one goes though the pointer dereference, but would like to be able to vectorize something like:

```
DataVector<C>& vc = ...;
static const C::Accessor<float> a ("a");
static const C::Accessor<float> b ("b");
static const C::Accessor<float> c ("c");
for (size_t i = 0; i < vc.size(); i++)
  c(vc,i) = a(vc,i) + b(vc,i);
```

- Container class contains a vector of pointers to the starting addresses of the variable vector, indexed by aux data id.
- If aux data item is in cache, dereference is entirely inline.
- Otherwise, call out-of-line code that fetches the vector from the store.

Implementation notes: aux data dereferencing 2

Actual dereferencing written as something like this (simplified a bit):

```
template <class T>
T& getData (size_t auxid, size_t i)
{
  return reinterpret_cast<T*>(getDataArray(auxid))[i];
}
void* getDataArray (size_t auxid)
{
  if (auxid >= m_cache_len || m_cache[auxid] == 0) {
    getDataArrayOol (auxid);
    // Inform compiler of postcondition
    if (auxid >= m_cache_len || m_cache[auxid] == 0)
      __builtin_unreachable();
  }
  return m_cache[auxid];
}
```

Implementation notes: aux data dereferencing 2

- `getDataArrayOol` has a postcondition that the cache entry is valid. So we only need to do the test the *first* time a given aux item is accessed (assuming nothing happens that clobbers memory).
- The `__builtin_unreachable` call is to inform the compiler of this postcondition.
- Can compilers take advantage of this?
- If no loops, then both gcc and icc can: they compile this

```
v.getData<float>(auxid,0) + v.getData<float>(auxid,1);
```

to something that ends with (after possibly calling the `Ool` function once):

```
        movss        (%rdx), %xmm0
        addss        4(%rdx), %xmm0
```

Implementation notes: aux data dereferencing 3

- Loops aren't quite there yet though. This:

```
float sum = v.getData<float>(auxid,0);
for (size_t i = 1; i < 10; i++)
  sum += v.getData<float>(auxid,i);
```

compiles to an inner loop like this with icc (-O3):

```
        movss     (%rcx), %xmm0
..B3.9: addss     (%rcx,%r12,4), %xmm0
        incq      %r12      ; loop index
        cmpq      $10, %r12
        jae       ..B3.18
        testq     %rcx, %rcx ; m_cache[auxid]
        jne       ..B3.9
```

- The code generated by gcc is similar, except that it elides the test on `m_cache[auxid]` but retains the test on `m_cache_len`.

Special-purpose allocators

Efficiently allocating blocks of varying sizes is a difficult problem.
But allocating blocks of a single size is *much* easier.
It can also help performance to group objects of like type together.

Block allocators

Allocate memory from `malloc` in big chunks.
Divide these into many fixed-size blocks.

Also allow all objects to be freed at once, rather than individually.

Arena allocators

One implementation is the `Arena*` classes in DataModel.
Only a brief overview given here — for more details, start with DataModel/Arena.h.

Basic Arena use

```
#include "DataModel/ArenaHande.h"
#include "DataModel/ArenaPoolAllocator.h"
struct MyType { MyType(int); };
...
// Create during initialization.
SG::ArenaHandle<MyType, SG::ArenaPoolAllocator> handle;

// Allocate
Mytype* p = new (handle.allocate()) MyType (10);
```

All `MyType` objects allocated this way will be grouped together in memory. They will be automatically deleted at the end of the event.

Arena Allocators

Second Handle template argument is the type of the memory allocator.

ArenaHeapAllocator

Allows deleting individual elements.

```
p = new (handle.allocate()) MyType;
...
handle.free (p);
```

ArenaPoolAllocator

Cannot delete individual elements, just the entire pool. Slightly less overhead than heap.

```
p = new (handle.allocate()) MyType;
...
handle.reset(); // Elements freed, underlying memory kept.
handle.erase(); // Elements + underlying memory freed.
```

Multiple arenas

```
SG::ArenaHandle<MyType, SG::ArenaPoolAllocator> handle;
handle.reset();
```

Will free all instances of `MyType` that have been allocated with any handle of this type.
To have objects with differing lifetimes, use another Arena:

```
SG::Arena myarena;
{
  SG::Arena::Push apush (myarena);
  // Allocate from myarena,
  p = new (handle.allocate());
}
// Back to previous arena.
// Free all elements in myarena.
myarena.reset();
```

STL allocators

ArenaPoolSTLAllocator

Simple block-based allocator for STL types. Memory is owned by the container. Individual elements cannot be freed; memory is released only when the container is deleted.
Suitable for containers that allocate a single fixed-size node type, such as `list`, `map`, `set`. `unordered_map/set` will also work, but the variable-length allocations of those types will use the normal allocator.

```
#include "DataModel/tools/ArenaPoolSTLAllocator.h"
  typedef SG::ArenaPoolSTLAllocator<int> alloc_t;
  typedef std::list<int, alloc_t> list_t;
  list_t c;
  c.push_back (1);
```

STL allocators 2

ArenaSharedHeapSTLAllocator

A memory heap that can be shared between containers. Elements may be deleted individually. Elements may be moved between containers using the same heap.
Suitable for containers that allocate a single fixed-size node type, such as `list`, `map`, `set`. `unordered_map`/`set` will also work, but the variable-length allocations of those types will use the normal allocator.

```
typedef SG::ArenaSharedHeapSTLAllocator<int> alloc_t;
typedef std::map<int, float, std::less<int>, alloc_t> map_t

alloc_t alloc;
map_t c1 (std::less<int>(), alloc);
map_t c2 (std::less<int>(), alloc);
c1[1] = 2;
c2[4] = 6;
```