# VarHandle work

- Trying to move to using `VarHandle` classes for most access to StoreGate.
- Issue for multithreading: they contain mutable state, so cannot be used as properties for reentrant code.
- Solution: introduce separate `VarHandleKey` classes that hold the CLID/key/store and are usable as properties. Initialize `VarHandle` from key and optional event context.
- New `AthReentrantAlgorithm` with `const` execute method explicitly passing an event context.
- Other handle changes:
  - Add `record()` methods.
  - Rationalize const issues. Remove `mutable` qualifications of members.
  - Explicitly prohibit null keys.
- Details in following slides.

# Summary: Reentrant VarHandle

```
class WriteDataR : public AthReentrantAlgorithm { ...
  virtual StatusCode execute_r (const EventContext& ctx)
     const override final;
  SG::WriteHandleKey<MyDataObj> m_dobjKey;
...
WriteDataR::WriteDataR(const std::string& name,
                       ISvcLocator* pSvcLocator) :
  AthReentrantAlgorithm(name, pSvcLocator) {
  declareProperty ("DObjKey", m_dobjKey = "dobj");
...
StatusCode WriteDataR::initialize() {
  ATH_CHECK( m_dobjKey.initialize() );
...
StatusCode WriteDataR::execute_r (const EventContext& ctx) co
  SG::WriteHandle<MyDataObj> dobj (m_dobjKey, ctx);
  ATH_CHECK( dobj.record
             (CxxUtils::make_unique<MyDataObj>(1)) );
```

# Key classes

- Templated classes `SG::ReadHandleKey<T>`, `SG::WriteHandleKey<T>`, `SG::UpdateHandleKey<T>`.
- Derive from `SG::VarHandleKey`, which derives from `Gaudi::DataHandle`.
- `Gaudi::DataHandle` stores CLID, object key, and mode (r/w/u).
- `SG::VarHandleKey` stores handle to event store.
- Templated derived classes pass CLID and mode to base class. (No longer virtual methods.)
- Update methods of `Gaudi::DataHandle` blocked.
- `StatusCode assign(const std::string&)` parses key string: `"MyKey"` or `"StoreName/MyKey"`.
- `initialize()` method may be called during algorithm/tool `initialize()` to check key validity and retrieve the event store pointer.

# Key classes (summary)

```
template <class T>
class ReadHandleKey  : public VarHandleKey ...
class VarHandleKey : public Gaudi::DataHandle {
  virtual void setKey(const DataObjID& key) override final;
  virtual void updateKey(const std::string& key) override fin
  ServiceHandle<IProxyDict> m_storeHandle;
public:
  VarHandleKey (CLID clid,
                const std::string& sgkey,
                Gaudi::DataHandle::Mode a,
                const std::string& storeName = "StoreGateSvc"
  VarHandleKey& operator= (const std::string& sgkey);
  StatusCode assign (const std::string& sgkey);
  StatusCode initialize();
  CLID clid() const;
  const std::string& key() const;
  ServiceHandle<IProxyDict> storeHandle() const;
```

# WriteHandle<T>

- `WriteHandle` can only be used to *write* objects into the store.
  - Does not do SG lookup; only return the pointer that has been recorded.
- Key may not be null.
- Old assignment method removed: `operator= (const& T)`
  Instead of `handle = obj`, use

```
*handle = obj;
```

  (Better match to smart pointer semantics.)
- `operator=(std::unique_ptr<T>)` still present, but prefer:

```
StatusCode record(std::unique_ptr<T>)
```

- Also `recordNonConst` to skip declaring object as const. (Can still modify the object through the `WriteHandle` as long as it exists.)

# WriteHandle<T>

- Record an object + aux store (should already be associated):

```
template <class AUXSTORE>
StatusCode record (std::unique_ptr<T> data,
                   std::unique_ptr<AUXSTORE> store);
```

- Record DataObject passing shared ownership.

```
StatusCode record (SG::DataObjectSharedPtr<T> data);
```

  - `DataObjectSharedPtr` is a reference-counting smart pointer.
  - In AthenaKernel.
  - Based on `boost::intrusive_ptr`.

# ReadHandle<T>

- Can *only* retrieve const objects.
- Retrieval methods are non-const.
  - (Change from existing version.)
  - Because data members may change.
  - Take advantage of compiler const rules for thread-safety/reentrancy.
- Key may not be blank.

# ReadHandle<T> example

```
class MyAlg ...
  SG::ReadHandleKey<MyObj> m_objKey;

MyAlg::MyAlg ...
  declareProperty ("ObjKey", m_objKey = "obj");

MyAlg::initialize...
  ATH_CHECK( m_objKey.initialize() );

MyAlg::execute_r (const EventContext& ctx) const ...
  SG::ReadHandle<MyObj> objH (m_objKey, ctx);
  const MyObj& obj = *objH;
```

# UpdateHandle<T>

- Has non-const retrieval methods returning non-const objects.
- Additional `cptr` method for retrieving a const pointer.
- But can *only* retrieve non-const proxies.
- Assignment from object instance removed. Again, use:

```
*handle = obj;
```

- Obtaining a non-const pointer will flag the object as updated in Hive.
  - `UpdateHandle` maintains a flag so that this is only done once.
  - Use `cptr` to avoid flagging the object as modified.

# UpdateHandle<T> example

```
class MyAlg ...
  SG::UpdateHandleKey<MyObj> m_objKey;

MyAlg::MyAlg ...
  declareProperty ("ObjKey", m_objKey = "obj");

MyAlg::initialize...
  ATH_CHECK( m_objKey.initialize() );

MyAlg::execute_r (const EventContext& ctx) const ...
  SG::UpdateHandle<MyObj> objH (m_objKey, ctx);
  if (needsUpdate (*objH.cptr())
    *objH = makeUpdatedObj (*objH);
```

# makeHandle

- For each handle type, there are helper functions `makeHandle` to return a handle of the proper type from the key and optionally the event context.

```
class MyAlg ...
  SG::WriteHandleKey<MyObj> m_objKey;

MyAlg::execute_r (const EventContext& ctx) const ...
  // type of these variables is SG::WriteHandle<MyObj>.
  auto objH1 = SG::makeHandle (m_objKey);
  auto objH2 = SG::makeHandle (m_objKey, ctx);
```

# EventContext and AthReentrantAlgorithm

- EventContext object in Gaudi.
- Dumb container.
- Contains an `IProxyDict` member:

```
IProxyDict* proxy() const { return m_proxy; }
void setProxy(IProxyDict* prx) {
  m_proxy = prx;
}
IProxyDict* m_proxy {0};
```

  (Not the best name since it's not a proxy...)
- Forward declaration of `IProxyDict` only (`IProxyDict` definition remains in AthenaKernel).
- Maybe better to just include an explicitly experiment-dependent `void*`?

# AthReentrantAlgorithm

execute is declared final to prevent overriding. execute_r is const. (Has to have a different name to prevent warnings about execute being hidden.)

```
class AthReentrantAlgorithm : public AthAlgorithm { ...
public:
  virtual StatusCode execute_r (const EventContext& ctx)
     const = 0;
  virtual StatusCode execute() override final
  { // To replace with fetch of context from Gaudi
    EventContext ctx;
#ifdef ATHENAHIVE
    ctx.setProxy (evtStore()->hiveProxyDict());
#else
    ctx.setProxy (&*evtStore());
#endif
    return execute_r (ctx);
  }
```

# Update/overwrite

Limit use of update as much as possible. By default, objects considered as const as soon as they are recorded. Main cases to handle:

- Reprocessing: If there is a `WriteHandle` for a key, then that key is not read from the input file.
- Fix: apply corrections to objects read from the input file. Done with `UpdateHandle`, but restrict use: algorithms using `UpdateHandle` can only depend on data directly from the input file.
- Scheduler will ensure that update algorithms run before anything else.

# Thread safety and data classes

Default requirements for data classes:

- `const` methods may be called simultaneously from multiple threads.
- non-`const` methods may only be called from a single thread (and not at the same time as any `const` methods).
- Following const-correctness should get you most of the way there. Using `const_cast` or `mutable` will usually mean you need to think about synchronization (so best not to use them!).

Objects may be declared as thread-safe:

- non-`const` methods may be called simultaneously from multiple threads.
- Relax restrictions on update for thread-safe object.
- Multiple updates may be scheduled simultaneously.
- Have one use-case for this (related to lazy fetching of partial trigger data). Hope there won't be many more.

# Additional information

# Example of recording aux store

```
class MyAlg ...
  SG::WriteHandleKey<xAOD::ObjContainer> m_objKey;

MyAlg::MyAlg ...
  declareProperty ("ObjKey", m_objKey = "obj");

MyAlg::initialize...
  ATH_CHECK( m_objKey.initialize() );

MyAlg::execute_r (const EventContext& ctx) const ...
  SG::WriteHandle<xAOD::ObjContainer> objH (m_objKey, ctx);
  auto obj = make_unique<xAOD::ObjContainer>();
  auto objstore = make_unique<xAOD::ObjAuxContainer>();
  obj->setStore (objstore.get());
  ... fill ...
  ATH_CHECK(objH.record (std::move(obj),std::move(objstore)));
```

# Example of use IDC-like type

Avoid creating a new object instance for each event.

```
class MyAlg ...
  SG::DataObjectSharedPtr<ObjContainer> m_obj;
  SG::WriteHandleKey<ObjContainer> m_objKey;

MyAlg::MyAlg ...
  declareProperty ("ObjKey", m_objKey = "obj");

MyAlg::initialize...
  ATH_CHECK( m_objKey.initialize() );
  m_obj = new Obj(...);

MyAlg::execute_r (const EventContext& ctx) const ...
  m_obj->reset(); // or whatever
  SG::WriteHandle<ObjContainer> objH (m_objKey, ctx);
  ATH_CHECK(objH.record (m_obj);
```

# WriteHandle<T> summary 1

```
template <class T> WriteHandle : public VarHandleBase {
public:
  explicit WriteHandle (const std::string& name);
  WriteHandle (const std::string& name,
               const std::string& store);
  explicit WriteHandle (const WriteHandleKey<T>& key);
  WriteHandle (const WriteHandleKey<T>& key,
               const EventContext& ctx);

  // All return cached ptr only; no SG lookup
  pointer_type operator->();
  reference_type operator*();
  const_pointer_type cptr();
  pointer_type ptr();
```

# WriteHandle<T> summary 2

```
// Has cached ptr been set?
virtual bool isValid() override final;

pointer_type cachedPtr() const;

// Primary record method.
StatusCode record (std::unique_ptr<T> data);
StatusCode recordNonConst (std::unique_ptr<T> data);

// Same const variants for remaining record methods.
template <class AUXSTORE>
StatusCode record (std::unique_ptr<T> data,
                   std::unique_ptr<AUXSTORE> store);
StatusCode record (SG::DataObjectSharedPtr<T> data);
```

# ReadHandle<T> summary

```
template <class T> ReadHandle : public VarHandleBase {
public:
  explicit ReadHandle(const std::string& name);
  ReadHandle(const std::string& name,
     const std::string& store);
  explicit ReadHandle (const ReadHandleKey<T>& key);
  explicit ReadHandle (const ReadHandleKey<T>& key,
                       const EventContext& ctx);

  // All return const object only.  Methods are non-const.
  const_pointer_type  operator->();
  const_reference_type operator*();
  const_pointer_type cptr();
  const_pointer_type ptr();

  const_pointer_type cachedPtr() const;
  virtual bool isValid() override final;
```

# UpdateHandle<T> summary

```
template <class T> ReadHandle : public VarHandleBase {
public:
  explicit UpdateHandle(const std::string& name);
  UpdateHandle(const std::string& name,
       const std::string& store);
  explicit UpdateHandle (const UpdateHandleKey<T>& key);
  explicit UpdateHandle (const UpdateHandleKey<T>& key,
                         const EventContext& ctx);

  pointer_type operator->();
  reference_type operator*();
  const_pointer_type cptr();
  pointer_type ptr();

  pointer_type cachedPtr() const;
  virtual bool isValid() override final;
```

# VarHandleBase summary 1

```
class VarHandleBase : public VarHandleKey, public IResetable
  void* m_ptr;
  SG::DataProxy* m_proxy;
  IProxyDict* m_store;

public:
  VarHandleBase(CLID clid, Gaudi::DataHandle::Mode mode);
  VarHandleBase(CLID clid,
                const std::string& sgkey,
                Gaudi::DataHandle::Mode mode,
                const std::string& storename = "StoreGateSvc"
  explicit VarHandleBase (const VarHandleKey& key);
  VarHandleBase (const VarHandleKey& key,
                 const EventContext& ctx);
```

# VarHandleBase summary 2

```
// Inherited from VarHandleKey:
//  CLID clid() const;
//  ServiceHandle<IProxyDict> storeHandle() const;
  virtual const std::string& key() const override final;
  const std::string& name() const; // same as key()
  std::string store() const;

  virtual bool isValid() = 0; // Can it be dereferenced?
  bool isPresent() const; // Is object present in store?
  bool isInitialized() const; // Has proxy been retrieved?
  virtual bool isSet() const override final;  // Same
  bool isConst() const; // Is the proxy set as const?
  StatusCode initialize(); // Fetch proxy
  StatusCode setState();  // same
  StatusCode setConst(); // Set const bit in proxy
  StatusCode setStore (IProxyDict* store);
```