A C++ wrapper for HDF5

A new approach to an old problem

Eugen Wintersberger
ICALEPCS 2017, 08.10.2017

HELMHOLTZ
ASSOCIATION

DESY

Motivation

Why C++?

- High performance
- No garbage collection due to destructors
- Type safety due to static typing
- Highly portable

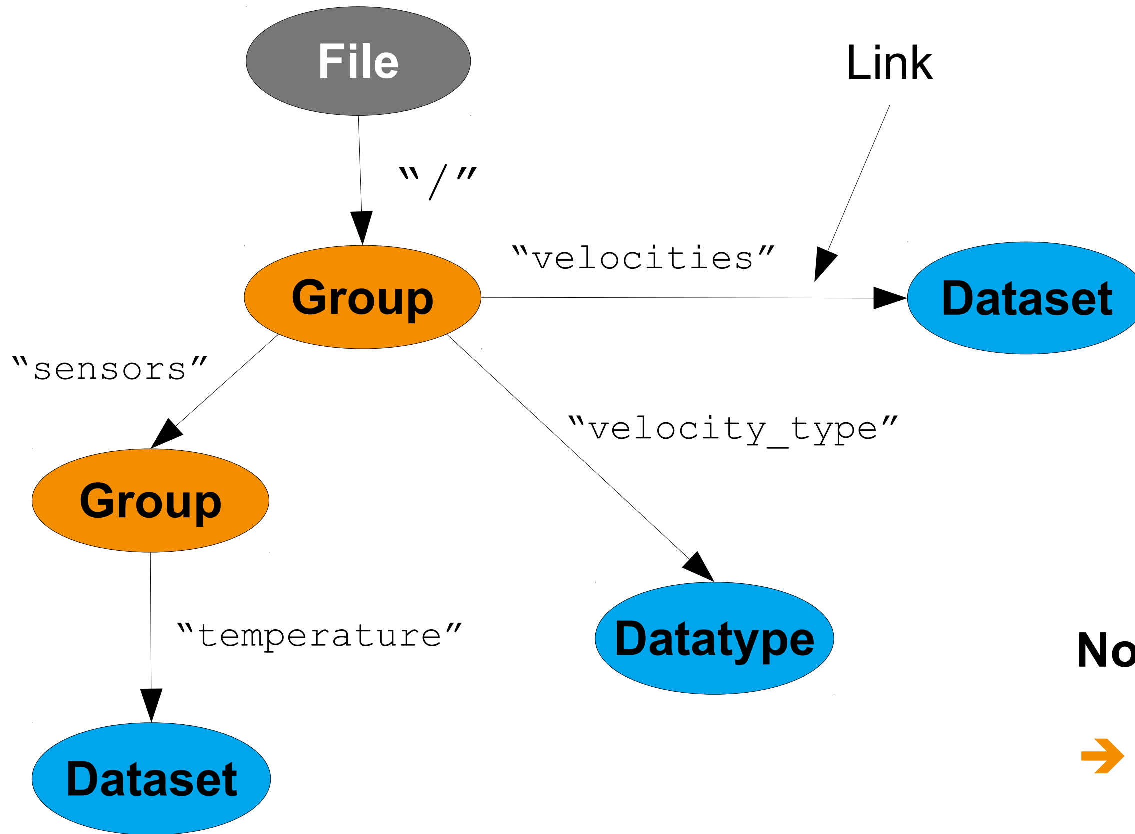
Why a new C++ wrapper?

- The current implementation is not complete
- Current wrapper is standard C++ 2003 at best
- The HDF group has to remain a stable interface for their contractors
- Not STL compliant

Design goals

- Complete functionality (on the long run)
- **STL compliant interfaces**
- Easy to use (*keep simple things simple and make complex possible*)
- High performance
- Supported platforms: Windows, Linux, macOS
- **Avoid handler leaks**
- **Easy to extend** (user defined data types while keeping type safety)
- **Preserve the access history of an object**
- Provide facilities for signal handling
- Keep thread safety in mind

A very high level view on HDF5

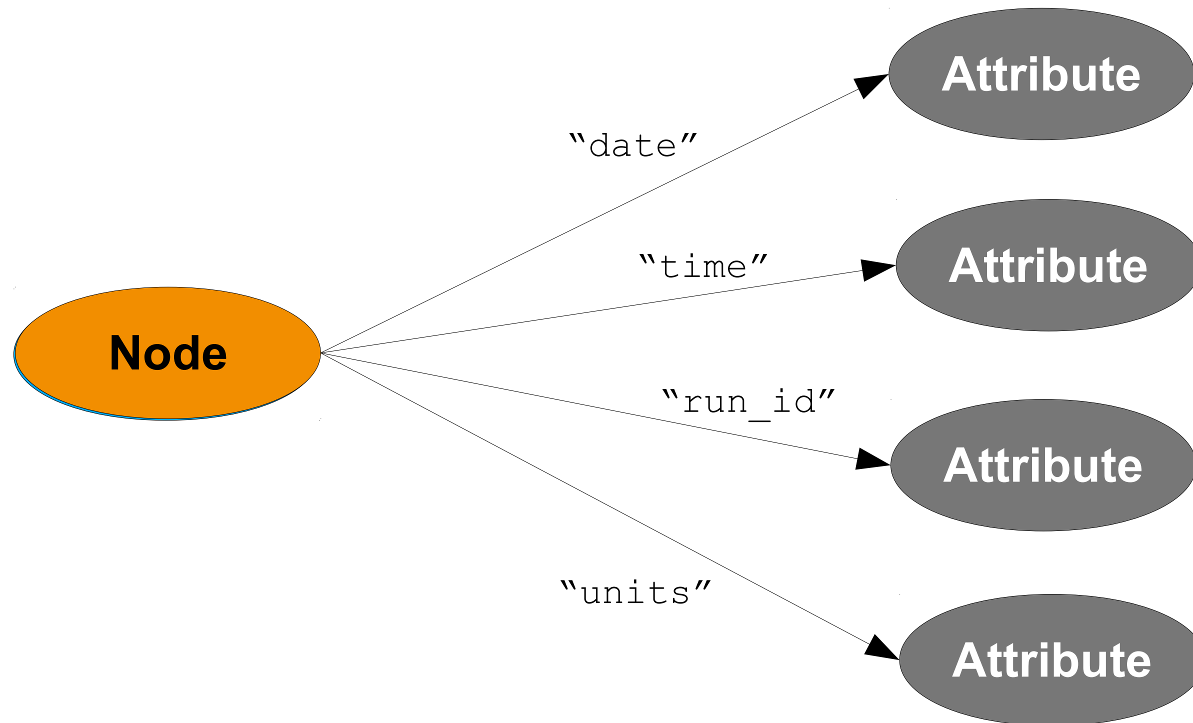


Nodes:

- **Group**
- **Dataset**
- **Commited Datatype**

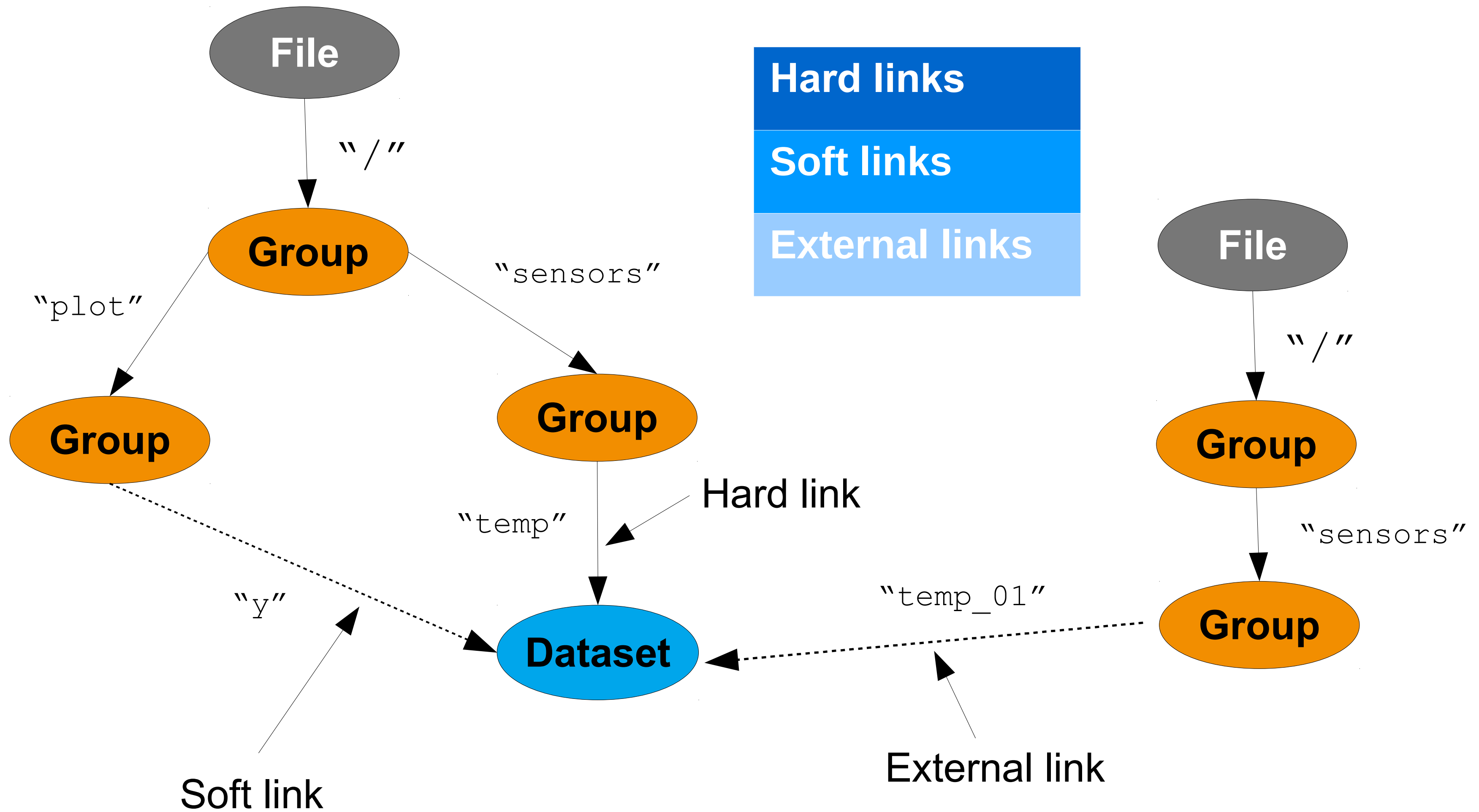
DESY

Adding metadata with attributes



DESY

Three kinds of links



A node path

Path := a ‘/’ separated *list* of *link names* used to access a particular node

File

"/"

The root group link has the special name ‘/’

Group

"plot"

"sensors"

Path = /plot

Group

Group

Hard link

"temp"

Dataset

Path = /sensors/temp

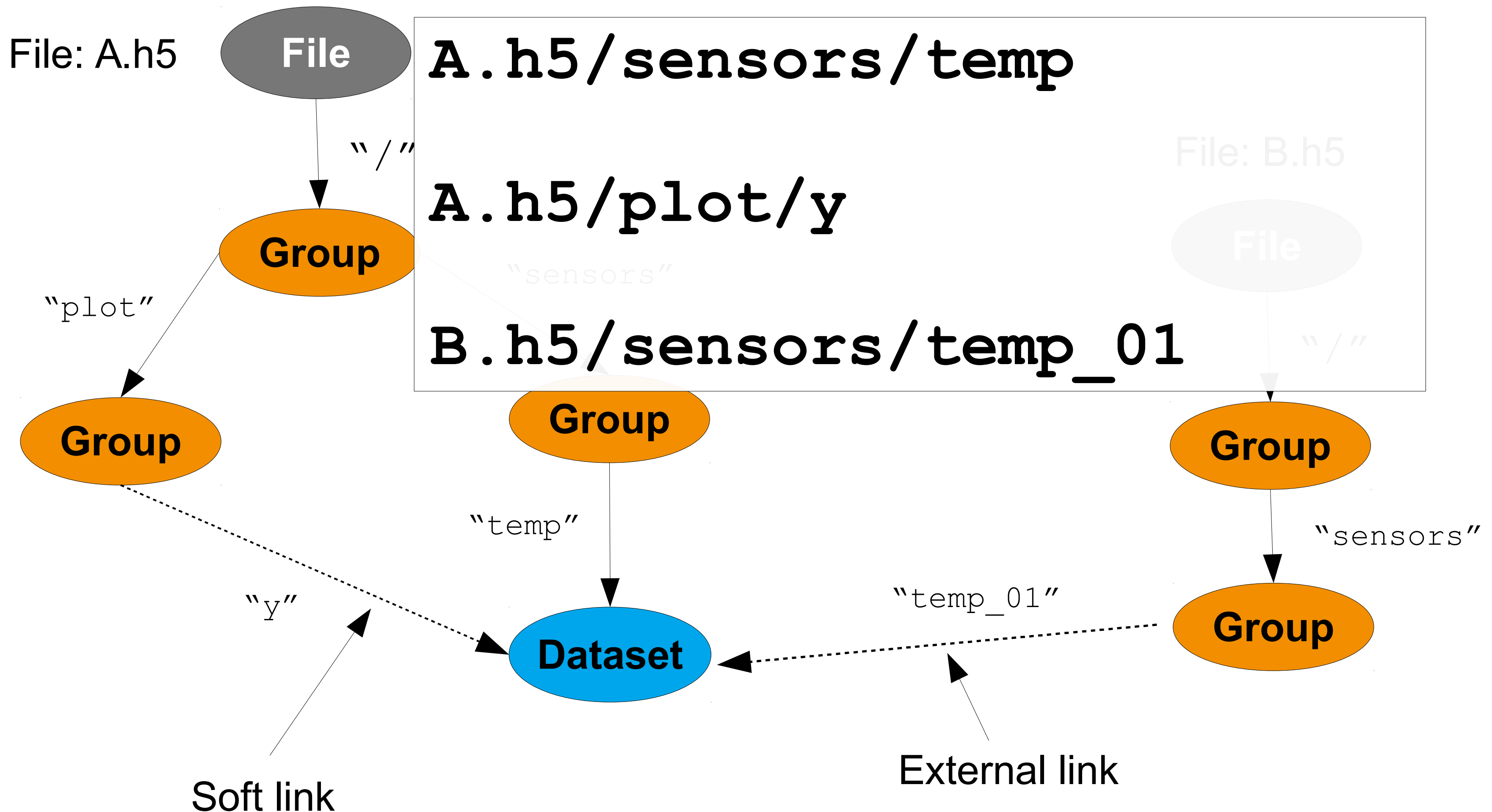
Every node can be accessed by following the links mentioned in its path!

We end up with two problems

I. What's the path of a node?

II. How to avoid accessing a node multiple times during iteration?

The name problem – whats the name of the dataset?



DESY

The name problem – its getting worser!

There is not solution:

all three names are equally valid and can be used to access the same object!

Consequences:

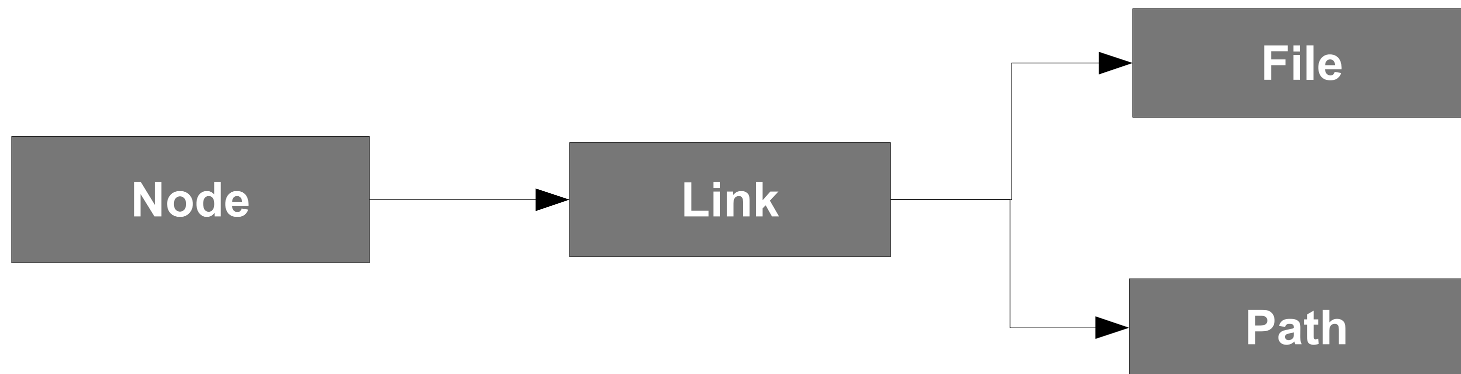
1) We cannot ask a node for its path!

2) Thus we cannot ask a node for its parent!

3) A path does not uniquely identify an object!

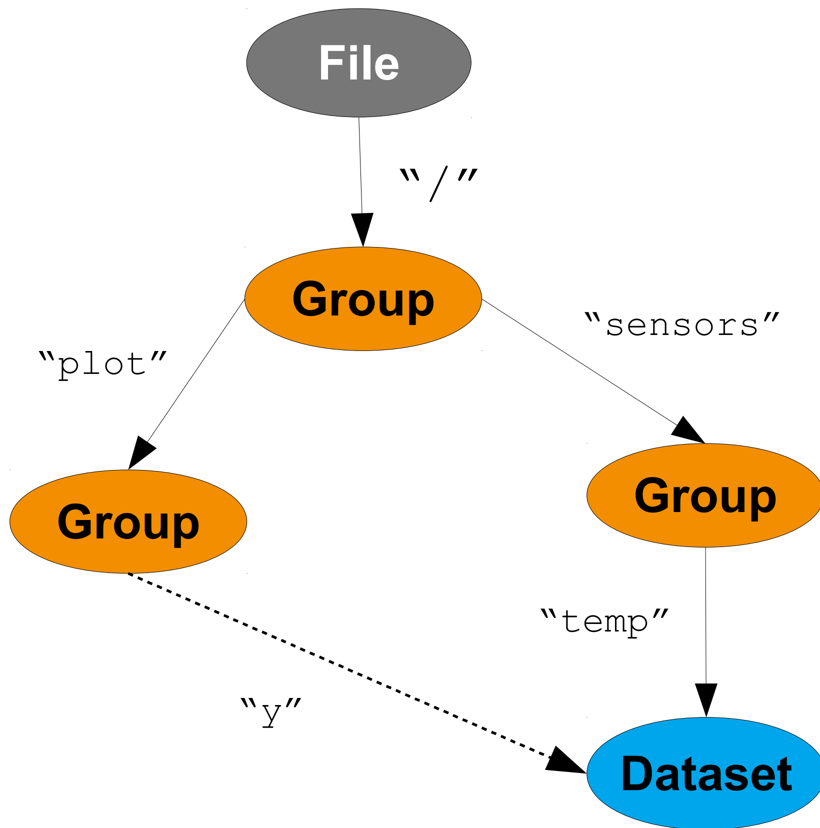
The name problem – a pragmatic solution!

- When asking a node for its path we expect to get the path used to access the node
- When asking for the nodes parent we expect to get the top level group of the final link used to access the object



DESY

The unique ID problem



When iterating recursively over all nodes in below the root group we get two references to the same dataset!

1) /sensors/temp

2) /plot/y

This is bad!

DESY

The ObjectId class

```
namespace hdf5 {
namespace node {

class DLL_EXPORT Node
{
  public:

    Node(ObjectHandle &&handle,const Link &link);
    Node();
    Node(const Node &);
    Node &operator=(const Node &node);
    virtual ~Node();

    Type type() const;

    ObjectId id() const;

    explicit operator hid_t() const
    {
      return static_cast<hid_t>(handle_);
    }

    bool is_valid() const;
    const Link &link() const;
    attribute::AttributeManager attributes;

  private:
    ObjectHandle handle_; //!< access handle to the object
    Link link_;           //!< stores the link to the object
};

DLL_EXPORT bool operator==(const Node &lhs, const Node &rhs);
DLL_EXPORT bool operator!=(const Node &lhs, const Node &rhs);

} // namespace node
} // namespace hdf5
```

```
namespace hdf5
{

class ObjectId
{
  public:
    ObjectId();
    ObjectId(hid_t object);

    bool operator== (const ObjectId& other) const;
    bool operator!= (const ObjectId& other) const;
    bool operator< (const ObjectId& other) const;
    …
  private:
    …
};
}
```

ObjectId **allows to uniquely identify an object within a program context!**

Avoiding handler (hid_t) leaks

```
namespace hdf5
{
class DLL_EXPORT ObjectHandle
{
  public:
    enum class Type {…};

    enum class Policy
    {
      WITH_WARD = 1,
      WITHOUT_WARD = 2
    };
  private:
    hid_t handle_; //!< ID of the object

    void increment_reference_count() const;
    void decrement_reference_count() const;
  public:
    explicit ObjectHandle(hid_t id,Policy policy=Policy::WITH_WARD);
    explicit ObjectHandle() noexcept;
    ObjectHandle(const ObjectHandle &o);
    ObjectHandle(ObjectHandle &&o) noexcept;
    ~ObjectHandle();

    ObjectHandle &operator=(const ObjectHandle &o);
    ObjectHandle &operator=(ObjectHandle &&o) noexcept;

    explicit operator hid_t() const
    {
      return handle_;
    }

    void close();
    bool is_valid() const;
    ObjectHandle::Type get_type() const;
    int get_reference_count() const;
    ObjectHandle get_file() const;
};
} // namespace hdf5
```

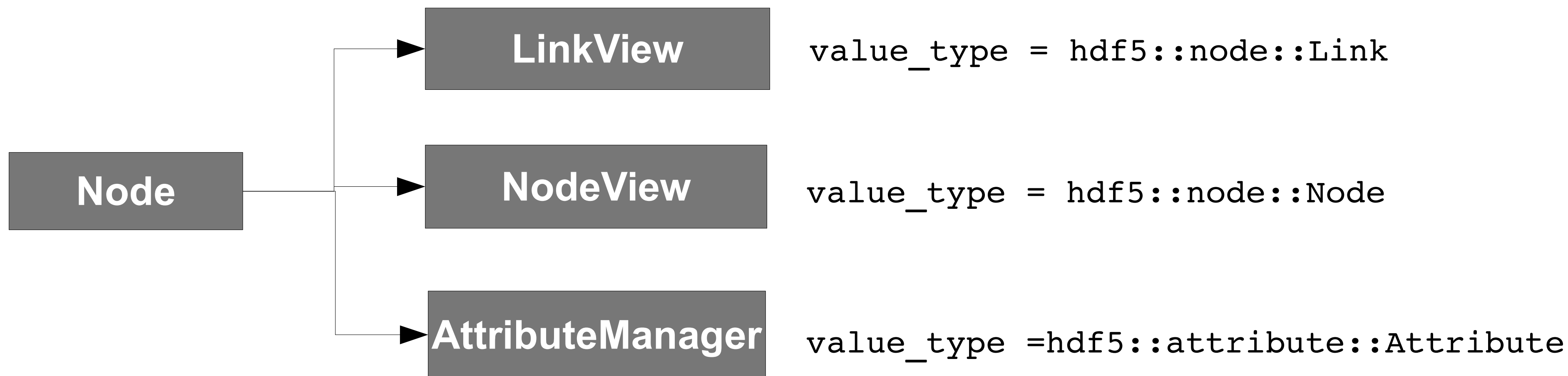
- Access to an object in HDF5 is provided via a handler (hid_t)
- When handlers are not closed the file an object belongs to is not closed

=> very hard to debug failures!

ObjectHandle **is a guard object which ensures that a handler gets destroyed whenever it looses scope!**

STL compliance

We want to use all the algorithms provided by the STL to search through an HDF5 tree.



The LinkView interface

```
namespace hdf5 {
namespace node {

class Node;
class NodeIterator;

class DLL_EXPORT NodeView : public GroupView
{
  public:
    using value_type = Node;
    using const_iterator = NodeIterator;

    NodeView() = delete;
    NodeView(const NodeView &) = default;
    NodeView(Group &node);
    ~NodeView();

    Node operator[](size_t index) const;

    bool exists(const std::string &name,
                const property::LinkAccessList &lapl =
                property::LinkAccessList()) const;

    Node operator[](const std::string &name) const;

    const_iterator begin() const;
    const_iterator end() const;
};

} // namespace node
} // namespace hdf5
```

```
namespace hdf5 {
namespace node {

class DLL_EXPORT NodeIterator : public Iterator
{
  public:
    using value_type = Node;
    using pointer = value_type*;
    using reference = value_type&;
    using difference_type = ssize_t;
    using iterator_category = std::random_access_iterator_tag;

    NodeIterator() = delete;
    NodeIterator(const NodeIterator&) = default;
    NodeIterator(const NodeView &view,ssize_t index);

    explicit operator bool() const
    {
      return !(index()<0 || index()>=view_.get().size());
    }

    Node operator*() const;
    Node *operator->();

    NodeIterator &operator++();
    NodeIterator operator++(int);
    NodeIterator &operator--();
    NodeIterator operator--(int);

    NodeIterator &operator+=(ssize_t i);
    NodeIterator &operator-=(ssize_t i);
    bool operator==(const NodeIterator &a) const;
    bool operator!=(const NodeIterator &a) const;

  private:
    std::reference_wrapper<const NodeView> view_;
    Node current_node_;

};

} // namespace node
} // namespace hdf5
```

Examples

Where are we today – what is implemented

- Classes for virtually all property lists
- File class
- Attribute facilities are complete
  - Attribute
  - AttributeManager
  - AttributeIterator
- Node classes
  - Dataset (currently no partial IO)
  - Group
  - NodeView
  - LinkView
- Datatype
  - Integer
  - Float
  - Compound
- Dataspace
  - Scalar
  - Simple
- Utility classes
  - Path
  - ObjectHandle
  - ObjectId

Todo

For the first milestone 0.0.1

- Link creation functions
- Partial IO for Datasets
- String Datatype (in all favors)

In general

- Documentation update (user manual)
- Implement better error management
- Signal handling support
- Test coverage
- Utility functions to access nodes and links

Collaboration

EUROPEAN
SPALLATION
SOURCE

- ➔ **Martin Shetty**
- ➔ **Jonas Nilsson**

Science & Technology Facilities Council
ISIS

- ➔ **Matthew Jones**

Github Project of the HDF5 C++ wrapper