LA-UR-20-23786
Approved for public release; distribution is unlimited.

Title: GNDStk (GNDS Toolkit)

Author(s): Staley, Martin Frank
Conlin, Jeremy Lloyd

Intended for: Working Party on International Nuclear Data Evaluation Co-operation, 2020-05-12 (Paris (now Webex), France)

Issued: 2020-05-21

GNDStk
(GNDS Toolkit)

13 May 2020

Martin Staley

Jeremy Conlin

Objectives

Create a software toolkit for working with GNDS data

- **Generic internal format** for GNDS
- **GNDS versions**: support existing and potential future versions of GNDS
- **Read** GNDS files (XML, JSON, etc.)
- **Modify** existing GNDS data
- **Create** new GNDS data
- **Write** GNDS files (XML, JSON, etc.)
- **Convert** between file types
- **Notation**: clean, concise, uncluttered
- **Queries**: search down into data tree
- **Extensible "keyword" scheme**
- **Interoperability** made easy: interface with viz, production codes, other GNDS libraries, etc.
- **Modern C++**
- **Header-only library**
- **Leverage** good existing C++ libraries for XML, JSON etc.
- **Compile-time polymorphism**; *use the C++ type system*
- **Tools**: dictionary, "x-ray" etc.

Major Classes/Capabilities: Summary

External Library Wrappers

class XML
…Wraps the “pugi XML” library
class JSON
…Wraps the “nlohmann JSON” library

These provide a uniform look and feel across the external libraries.

Some Capabilities

Read & write XML GNDS
Read & write JSON GNDS
Build new GNDS trees
Modify existing GNDS trees
Queries: basic and “smart”
Conversions between file types

GNDS Hierarchy

class Node<some container types>
class node = Node<default>

class Tree<some container types>
class tree = Tree<default>

Individual nodes vs. entire GNDS tree. Similar, but with some differences that reflect each one’s exact purpose.

Keyword System

class meta_t<type>
class child_t<type,multiplicity>

More on these soon.

Example GNDS: atom-004_Be_000.xml

```
<?xml version="1.0"?>
<PoPs name="ENDF atomic relaxation for Z = 5" version="1.0" format="0.1">
  <styles>
    <evaluated label="eval" date="2017-12-01" library="ENDF/B"
               version="8.0.0"/>
  </styles>
  <documentations>
    <documentation name="endfDoc"><![CDATA[
  5-B -  0 NDS,IAEA   Eval-Dec17 D.E.Cullen
 NDS-IAEA-224         DIST-FEB18
----ENDF/B-VIII.0     MATERIAL  100
-----ATOMIC RELAXATION DATA
------ENDF-6 FORMAT
*** SOME CONTENT DELETED HERE, FOR BREVITY ***
=============================================================
The Livermore Evaluated Atomic Data Library (EADL) in the
ENDF-6 Format. Translated from the Livermore ENDL format
to the ENDF-6 Format.
=============================================================
  Contents
=============================================================
  MF/MT    Description
=============================================================
  28/533   Atomic Relaxation Data for Electrons and Photons
=============================================================
  Definition of Data
=============================================================
  Atomic Relaxation Data for Electrons and Photons
=============================================================
1) An atom can be ionized by a photon or an electron interaction.
2) The atomic relaxation data included here describes how a
   singly ionized atom returns to neutrality, by emitting photons
   (x-rays) and/or electrons (Auger, Coster-Kronig).
3) Data is given to define the transition probabilities between
   subshells for radiative (x-ray emission) and non-radiative
   (Auger, Coster-Kronig) transitions.
4) The complete spectrum of emitted photons and electrons can be
   calculated using the data included here, e.g., see ref. 2,
   below, for am example of a computer code that analytically
   calculates these spectra.
=============================================================
  History
=============================================================
 (1) November, 1991 - Initial release in the ENDL format.
 (2) November, 2001 - Initial release in the ENDF-6 format.
 (3) September,2014 - Insure Standard C, C++, FORTRAN Format
 (4) September,2014 - Updated based on recently publshed data
 (5) September,2017 - New Binding Energies (see: ref.3)
                      Same Transition Probabilities
=============================================================
 ********** Program DICTIN (VERSION 2017-1) **********]]>
    </documentation>
  </documentations>
  <chemicalElements>
    <chemicalElement symbol="B" Z="5" name="Boron">
      <atomic>
        <configurations>
          <configuration subshell="1s1/2" electronNumber="2.0">
            <bindingEnergy>
              <double label="eval" value="192.0" unit="eV"/>
            </bindingEnergy>
          </configuration>
          <configuration subshell="2s1/2" electronNumber="2.0">
            <bindingEnergy>
              <double label="eval" value="11.39" unit="eV"/>
            </bindingEnergy>
          </configuration>
          <configuration subshell="2p1/2" electronNumber="0.33">
            <bindingEnergy>
              <double label="eval" value="8.3" unit="eV"/>
            </bindingEnergy>
          </configuration>
          <configuration subshell="2p3/2" electronNumber="0.67">
            <bindingEnergy>
              <double label="eval" value="8.3" unit="eV"/>
            </bindingEnergy>
          </configuration>
        </configurations>
      </atomic>
    </chemicalElement>
  </chemicalElements>
</PoPs>
```

Examples 1, 2, 3

Example 1

```
#include "GNDStk.hpp"

int main()
{
}
```

Example 2

```
#include "GNDStk.hpp"

// For brevity
using namespace GNDStk;

int main()
{
   tree t("atom-005_B_000.xml");
}
```

Example 3

```
#include "GNDStk.hpp"

// For brevity
// May eventually be njoy::GNDStk
using namespace GNDStk;

int main()
{
   tree t("atom-005_B_000.xml");

   // Another way (there are several)
   {
      tree t;
      t.read("atom-005_B_000.xml");
   }

   // Write as JSON
   t.write("atom-005_B_000.json");
}
```

Example 4

```
#include "GNDStk.hpp"
using namespace GNDStk;

int main()
{
    // Read from xml
    tree x("atom-005_B_000.xml");

    // Write to json
    x.write("atom-005_B_000.json");

    // Read back from json
    tree j("atom-005_B_000.json");

    // Both of the trees should have the
    // same content. Let's verify this...

    // However, GNDS has no "ordering", and
    // content may read or write differently
    // in XML as compared with JSON. So, we
    // provide sort() as a convenience.
    x.sort();
    j.sort();

    // Stream output (<<) writes the tree
    // in our internal "debugging" format
    std::ostringstream ossx;
    ossx << x;
    std::ostringstream ossj;
    ossj << j;

    // Compare
    assert(ossx.str() == ossj.str());
}
```

Example 5

Raw data access. It's available, but beware of [] indexing! Remember, GNDS data have no guaranteed ordering.

```
#include "GNDStk.hpp"
using namespace GNDStk;

int main()
{
   tree t("atom-005_B_000.xml");
   node n = t.top(); // top-level GNDS node
   // A node has:
   //    string name
   //    vector<pair of strings> metadata
   //    vector<pointers to nodes> children

   // name
   assert(n.name == "PoPs");

   // metadata: name, version, format
   assert(n.metadata.size() == 3);
   assert(n.metadata[0].first ==
      "name");
   assert(n.metadata[0].second ==
      "ENDF atomic relaxation for Z = 5");
   assert(n.metadata[1].first ==
      "version");
   assert(n.metadata[1].second ==
      "1.0");
   assert(n.metadata[2].first  ==
      "format");
   assert(n.metadata[2].second ==
      "0.1");

   // child nodes: <styles>,
   // <documentations>, <chemicalElements>
   assert(n.children.size() == 3);
   assert(n.children[0]->name ==
      "styles");
   assert(n.children[1]->name ==
      "documentations");
   assert(n.children[2]->name ==
      "chemicalElements");

   // Let's improve on the raw access!
}
```

Example 6

Data access via string lookup. For metadata, use **meta()**. Same-named child nodes can appear more than once. Use **one()** for one, **all()** for all.

```
#include "GNDStk.hpp"
using namespace GNDStk;

int main()
{
    tree t("atom-005_B_000.xml");

    // Child's metadatum
    assert(t.one("PoPs")
            .meta("version") == "1.0");

    // Child's child
    assert(t.one("PoPs")
            .one("styles")
            .metadata.size() == 0);

    // Child's child's child
    assert(t.one("PoPs")
            .one("styles")
            .one("evaluated")
            .metadata.size() == 4);

    // Child's child's child's metadatum
    assert(t.one("PoPs")
            .one("styles")
            .one("evaluated")
            .meta("date") == "2017-12-01");

    // From child() and meta(), we get back,
    // respectively, node and string. So:

    node &n = t.one("PoPs").one("styles");
    assert(n.one("evaluated")
            .meta("date") == "2017-12-01");

    node &eval = n.one("evaluated");
    std::string datestr = eval.meta("date");
    assert(datestr == "2017-12-01");

    // This is preferable over raw access,
    // but let's try to make it even better!
}
```

Data Query Improvements

Recall the first few lines of our example XML file:

```
<?xml version="1.0"?>
<PoPs name="ENDF atomic relaxation for Z = 5" version="1.0" format="0.1">
  <styles>
    <evaluated label="eval" date="2017-12-01" library="ENDF/B" version="8.0.0"/>
  </styles>
  ...
```

And, consider this data query, on a `GNDStk::tree` object called `tree`, as illustrated in our previous example:

```
tree.one("PoPs").one("styles").one("evaluated").meta("date")
```

The above query gives us the string `"2017-12-01"`.

Question: How can we improve this?

Improvement #1: Predefined Strings

This one is **super simple**, but useful. We can provide **predefined strings**, like these, for the **names** of all possible GNDS metadata and child nodes:

```
const std::string PoPs      = "PoPs";
const std::string styles    = "styles";
const std::string evaluated = "evaluated";
const std::string date      = "date";
```

These allow us to take our previous query,

```
tree.one("PoPs").one("styles").one("evaluated").meta("date")
```

and replace it with:

```
tree.one(PoPs).one(styles).one(evaluated).meta(date)
```

That's hardly earth shattering, but it's useful...

You **save four keystrokes** (shift quote … shift quote) per name.
A **typo** (evalauted) **presents at compile time**, not at run time ("evalauted").
We can **provide all valid GNDS names**, so it's harder to use one that isn't valid.

Improvement #2: Metadata vs. Children

Let's take the "predefined strings" concept further. Consider two simple classes, which we'll call **meta_t** (metadatum type) and **child_t** (child node type). Briefly:

```
struct meta_t {
   std::string name;
   // and a constructor
};
```

```
struct child_t {
   std::string name;
   // and a constructor
};
```

Now consider some simple C++ **function overloading**. We'll make two functions, both called f( ). The first takes a **meta_t**, while the second takes a **child_t**.

```
string &f(const meta_t  &arg) { /* calls .meta(arg.name) */ }
node   &f(const child_t &arg) { /* calls .one (arg.name) */ }

const meta_t foo("foo");
f(foo); // ultimately, this calls meta("foo")

const child_t bar("bar");
f(bar); // ultimately, this calls one("bar")
```

Essentially, we've embedded our strings into types that control where they're sent.

Improvement #2 (continued)

GNDStk's **meta_t** and **child_t** classes are a bit more complicated than that, but, for now, imagine that we'll provide predefined ones like these:

```
const meta_t  date      = "date";
const child_t PoPs      = "PoPs";
const child_t styles    = "styles";
const child_t evaluated = "evaluated";
```

These allow us to take our previous query,

```
tree.one(PoPs).one(styles).one(evaluated).meta(date)
```

and replace it with:

```
tree(PoPs)(styles)(evaluated)(date)
```

Well, that's a nice improvement. You no longer need to write **.one** and **.meta** all over the place. Those still work, but our overloaded parentheses operator, (), accepts either a **meta_t** or a **child_t**, and knows how to handle each of them properly.

Improvement #3: One Call

Now that we've **disappeared the `.one` and `.meta` clutter**, leaving just a series of calls to (), we'll **also allow for a single call with comma-separated arguments**.

Before, we had this:

```
tree(PoPs)(styles)(evaluated)(date)
```

but now we can write:

```
tree(PoPs,styles,evaluated,date)
```

Not a major improvement, but slightly shorter and easier to type.

In general, you can provide any number (possibly zero) of **`child_t`** objects, and end the sequence with either a **`meta_t`** object or a **`child_t`** object:

```
tree ( [child_t, child_t, ... ,] meta_t )
tree ( [child_t, child_t, ... ,] child_t )
```

Expect – *for now* – a **`string`** from the first, and a **`node`** from the second.

Improvement #4: Custom Types

The improvements we've shown so far constitute part of what we call our **"smart keyword"** system. **("Keywords" here really just means variables of our `meta_t` and `child_t` types.)** Names of allowable GNDS metadata and child nodes are encapsulated into `meta_t` and `child_t` objects, then combined with a flexible parentheses operator `(...)` to allow ***succinct, intuitive node and tree queries***. **Now, perhaps, the most fun part.**

Let's again revisit our example GNDS file:

```
<?xml version="1.0"?>
<PoPs name="ENDF atomic relaxation for Z = 5" version="1.0" format="0.1">
  <styles>
    <evaluated label="eval" date="2017-12-01" library="ENDF/B" version="8.0.0"/>
  </styles>
  ...
```

Consider `date="2017-12-01"`. It'll serve as a simple example of **fitting a custom metadata structure into our existing query system**.

We'll illustrate with a complete example.

Improvement #4 (continued)

```
#include "GNDStk.hpp"
using namespace GNDStk;

// Define a simple type to hold a date
struct date_t {
   int year, month, day;
};

// Define stream input for date_t. (There
// are alternatives, if you don't want to
// have operator>>, or if it already exists
// and does something else.)
std::istream &operator>>(
   std::istream &s, date_t &d
) {
   char ch; // for '-' in "yyyy-mm-dd"
   return s >> d.year >> ch >> d.month
            >> ch >> d.day;
}

// Now make a GNDStk::meta_t object, here
// called date, in the following way.
meta_t<date_t> date("date");
```

```
int main()
{
   tree t("atom-005_B_000.xml");

   // GNDStk provides built-in child_t
   // objects in its child:: namespace.
   // We'll use these below.

   // Now, use GNDStk's query system, along
   // with your own "date" object, to get
   // the date as a date_t object.
   date_t d = t(
      child::PoPs,      // from GNDStk
      child::styles,    // from GNDStk
      child::evaluated, // from GNDStk
      date // <== our own
   );

   // We got a date_t, not a string, so:
   assert(d.year  == 2017);
   assert(d.month == 12);
   assert(d.day   == 1);
}
```

Improvements (so far)

In summary, we've taken the original long-winded query into our tree, whittled it down into something **much shorter**, and then provided a way to get a **custom type** instead of the string that's in the original GNDS.

Original:

```
tree.one("PoPs").one("styles").one("evaluated").meta("date")
```

With predefined strings:

```
tree.one(PoPs).one(styles).one(evaluated).meta(date)
```

With overloading via **`meta_t`** and **`child_t`** types:

```
tree(PoPs)(styles)(evaluated)(date)
```

One call with comma-separated values:

```
tree(PoPs,styles,evaluated,date) // <== return is string
```

And then finally:

```
tree(PoPs,styles,evaluated,date) // <== return is our custom date_t type
```

if the parameter **`date`** is formulated as an object of type **`meta_t<date_t>`**.

Improvement #5

For metadata retrieval, we've seen how anyone can define a custom type T, create an object of type `meta_t<T>`, and use the object seamlessly, in GNDStk's query system, to extract a metadata value as an object of type T.

What about extracting child nodes?

For this we can similarly create objects, now of type **`child_t<T>`**, to extract child nodes into a custom type T.

However, `child_t<>` also takes a second argument, with two possible values:

```
child_t< T, find::one >
child_t< T, find::all >
```

`find::one` tells GNDStk to look for **one such child node**, and return it as a T.

`find::all` tells GNDStk to look for **all such child nodes**, and return them in a `vector<T>`. (We do allow for containers other than `vector`, but the notation for that is a bit cumbersome right now. GNDStk is a work in progress.)

Improvement #5 (continued)

Consider part of our example XML:

```
<?xml version="1.0"?>
<PoPs name="ENDF atomic relaxation for Z = 5" version="1.0" format="0.1">
  ...
  <chemicalElements>
    <chemicalElement symbol="B" Z="5" name="Boron">
      <atomic>
        <configurations>
          <configuration subshell="1s1/2" electronNumber="2.0">
            <bindingEnergy>
              <double label="eval" value="192.0" unit="eV"/>
            </bindingEnergy>
          </configuration>
          <configuration subshell="2s1/2" electronNumber="2.0">
            <bindingEnergy>
              <double label="eval" value="11.39" unit="eV"/>
            </bindingEnergy>
          </configuration>
          <configuration subshell="2p1/2" electronNumber="0.33">
            <bindingEnergy>
              <double label="eval" value="8.3" unit="eV"/>
            </bindingEnergy>
          </configuration>
          <configuration subshell="2p3/2" electronNumber="0.67">
            <bindingEnergy>
              <double label="eval" value="8.3" unit="eV"/>
            </bindingEnergy>
          </configuration>
        </configurations>
      </atomic>
    </chemicalElement>
  </chemicalElements>
</PoPs>
```

There's **just one `bindingEnergy`** in any given context, so we might have:

```
struct bindingEnergy_type { ... };

child_t<bindingEnergy_type,find::one>
   binden("bindingEnergy");
```

There are **many `configuration` nodes** in one place, however, so we might have:

```
struct configuration_type { ... };

child_t<configuration_type,find::all>
   config("configuration");
```

Each type that's embedded in a `child_t` **will also need a certain conversion function** from a tree node.

Example

```
#include "GNDStk.hpp"
using namespace GNDStk;

// For accessing one "bindingEnergy"
struct bindingEnergy_type { };
void node2type(
   const node &, bindingEnergy_type &
) { }
child_t<bindingEnergy_type,find::one>
binden("bindingEnergy");

// For accessing all "configuration"s
struct configuration_type { };
void node2type(
   const node &, configuration_type &
) { }
child_t<configuration_type,find::all>
config("configuration");

int main()
{
   tree gnds("atom-005_B_000.xml");

   // GNDStk's child::chemicalElement
   // and our configuration_type are
   // both encoded with find::all, so
   // their use gives us vector<>s...
   auto c = gnds(
      child::PoPs,
      child::chemicalElements,
      child::chemicalElement
   )[0](
      child::atomic,
      child::configurations,
      config // set up above
   );
   // c is vector<configuration_type>
   assert(c.size() == 4);

   // configuration_type as defined
   // earlier is a stub, so we can't do
   // much with it here. In a complete
   // example, it would probably contain
   // values for its metadata, and a
   // bindingEnergy_type object (also
   // remade as more than a stub) for
   // its <bindingEnergy> child node.
}
```

Classes/Capabilities: Sketch

Let’s sketch some of GNDStk’s capabilities in terms of its major classes:

- **XML**
- **JSON**
- **Node<>**
- **Tree<>**

Information on the following pages is by no means complete.

In particular, I’ll omit some of the more-obscure functionality in some of the above classes, and some of the extra niceties you can get from some of their functions, e.g. by sending optional parameters.

Note: the template parameters on **`Node`** and **`Tree`** allow you to specify **container types** for metadata and for child nodes. We won’t worry about these for now.

XML Class

```
struct XML {

   // external XML-library document
   pugi::xml_document doc;

   // clear
   void clear();

   // constructors
   XML() = default;
   XML(XML &&) = default;
   XML(const XML &x);
   explicit XML(const JSON &j);
   explicit XML(const Tree<> &t);
   explicit XML(const std::string &file);
   explicit XML(std::istream &is);

   // assignment
   XML &operator=(XML &&) = default;
   XML &operator=(const XML &x);

   // read
   std::istream &read(std::istream &is);
   bool read(const std::string &file);

   // write
   std::ostream &write(std::ostream &os) const;
   bool write(const std::string &file) const;

}; // struct XML


// stream input
std::istream &operator>>
   (std::istream &is, XML &obj);

// stream output
std::ostream &operator<<
   (std::ostream &os, const XML &obj);
```

JSON Class

```
struct JSON {

   // external JSON-library document
   nlohmann::json doc;

   // clear
   void clear();

   // constructors
   JSON() = default;
   JSON(JSON &&) = default;
   JSON(const JSON &j);
   explicit JSON(const XML &x);
   explicit JSON(const Tree<> &t);
   explicit JSON(const std::string &file);
   explicit JSON(std::istream &is);

   // assignment
   JSON &operator=(JSON &&) = default;
   JSON &operator=(const JSON &) = default;

   // read
   std::istream &read(std::istream &is);
   bool read(const std::string &file);

   // write
   std::ostream &write(std::ostream &os) const;
   bool write(const std::string &file) const;

}; // struct JSON


// stream input
std::istream &operator>>
   (std::istream &is, JSON &obj);

// stream output
std::ostream &operator<<
   (std::ostream &os, const JSON &obj);
```

Node Class

```
// <METADATA_CONTAINER,CHILDREN_CONTAINER>
struct Node {
   using metaPair = pair<string,string>;
   using childPtr = unique_ptr<Node>;
   string name;
   METADATA_CONTAINER<metaPair> metadata;
   CHILDREN_CONTAINER<childPtr> children;

   // clear()
   Node &clear();

   // empty()
   bool empty() const;

   // constructors
   Node() { }
   Node(Node &&) = default;
   Node(const Node &from);

   // assignment
   Node &operator=(Node &&) = default;
   Node &operator=(const Node &from);

   // add()
   // Add metadatum/child; many variations

   // meta()
   // Access metadatum

   // one()
   // Access once-appearing child node

   // all()
   // Access many child nodes ==> container

   // child()
   // Access child nodes by using smart
   // keywords. find::one gets one, find::all
   // gets a container of zero or more

   // operator()
   // Access metadatum/child; many variations

   // sort()
   // Orders node's content in a predictable
   // way, for whatever that may help with
   Node &sort();

}; // struct Node
```

Tree Class, Part 1

```
// <METADATA_CONTAINER,CHILDREN_CONTAINER>
struct Tree {

   // Initial node of the tree
   using nodeType = Node<METADATA_CONTAINER,CHILDREN_CONTAINER>;
   std::unique_ptr<nodeType> root;

   // clear()
   void clear();

   // empty()
   bool empty() const;

   // reset()
   // Re-start a tree, with a particular top-level GNDS node and optionally
   // some XML boilerplate if you plan to write as XML. A couple of variations.

   // constructors
   Tree() = default;
   Tree(Tree &&) = default;
   Tree(const Tree &from);
   explicit Tree(const Tree<different> &from);
   explicit Tree(const XML &x);
   explicit Tree(const JSON &j);
   explicit Tree(const std::string &file, const format form = format::null);
```

Tree Class, Part 2

```
Tree(const std::string &file, const std::string &type);
explicit Tree(std::istream &is, const format form = format::null);
Tree(std::istream &is, const std::string &type);
// ...and a couple of additional ctors that are more complicated

// assignment
Tree &operator=(Tree &&) = default;
Tree &operator=(const Tree &t);
Tree &operator=(const Tree<different> &t);

// read
bool read(const std::string &file, const format form = format::null);
std::istream &read(std::istream &is, format form = format::null);
bool read(const std::string &file, const std::string &form);
std::istream &read(std::istream &is, const std::string &form);

// write
bool write(const std::string &file, format form = format::null) const;
std::ostream &write(std::ostream &os, const format form = format::null) const;
bool write(const std::string &file, const std::string &form) const;
std::ostream &write(std::ostream &os, const std::string &form) const;
```

Tree Class, Part 3

```
    // decl()
    // Access "declaration node" if applicable
    const nodeType &decl() const;
    nodeType &decl();

    // top()
    // Access top-level GNDS node (reactionSuite, PoPs, ...)
    const nodeType &top() const;
    nodeType &top();

    // Similar to those for node...
    // meta()
    // one()
    // all()
    // child()
    // operator()
    // sort()

}; // struct Tree


// I/O
std::istream &operator>>(std::istream &is, Tree<> &obj);
std::ostream &operator<<(std::ostream &os, const Tree<> &obj);
```