The NANDRAD CodeGenerator

Andreas Nicolai
andreas.nicolai@tu-dresden.de

Version 1.0.0, June 2020

Table of Contents

1. Overview

What is the NANDRAD-Code-Generator? Basically, it is a tool to parse NANDRAD header files, pick up lines with special comments and then generate code for:

- translating enumeration values into strings, descriptions, colors and units (very handy for physical parameter lists)
- reading and writing xml data structures (elements and attributes) for designated member variables
- comparing data structures with default instances (so that only modified members are written to file), and other utility functions

1.1. How does it work?

The code generator is called as part of the CMake build chain. It detects changes in header files, and then updates the respective generated files, including the file `NANDRAD_KeywordList.cpp`. It places the class-specific generated files into a subdirectory `ncg` (for **N**ANDRAD **C**ode **G**enerator files) where they are compiled just as any other cpp file of the project.

Since the generated code is plain C++ code, it can be parsed and debugged with any IDE just as regular code. Just remember:

Any changes made to generated code will be overwritten!

2. Functionality Example

2.1. Keyword support

You may have a class header, that looks as follows:

```
class Interval {
public:
    /*! Parameters. */
    enum para_t {
        IP_START,       // Keyword: Start       [d] 'Start time point'
        IP_END,         // Keyword: End         [d] 'End time point'
        IP_STEPSIZE,    // Keyword: StepSize    [h] 'StepSize'
        NUM_IP
    };

    ...

    /*! The parameters defining the interval. */
    IBK::Parameter   m_para[NUM_IP];
};
```

Here, you have a list of enumeration values. These are used to index one of the parameters in the member variable array `m_para`. In the code, you will use these enumerations like:

```
double endtime = m_para[Interval::IP_END].value;
```

Sometimes, however, you need the keyword as a string, or need to get the unit written in the comment line. So, you write code like:

```
// construct a parameter
m_para[Interval::IP_END].set(
        NANDRAD::KeywordList::Keyword("Interval::para_t", Interval::IP_END), // the keyword is used as name
        24, // value
        NANDRAD::KeywordList::Unit("Interval::para_t", Interval::IP_END) // the unit
    );
```

Normally, the unit is only meaningful for user interface representation, or when the value is printed to the user as default output/disply unit.

Also useful is the description, mostly in informative/error messages. Recommended would be:

```
if (condition_failed) {
    IBK::IBK_Message(IBK::FormatString("Parameter %1 (%2) is required.")
        .arg(KeywordList::Keyword("Interval::para_t", Interval::IP_END))
        .arg(KeywordList::Description("Interval::para_t", Interval::IP_END)) );
}
```

Also, when parsing user data, for example from format:

```
<IBK:Parameter name="Start" unit="d">36</IBK:Parameter>
<IBK:Parameter name="End" unit="d">72</IBK:Parameter>
```

You may need code like:

```
nameAttrib = getAttributeText(xmlTag, "name"); // might be 'End'

// resolve enumeration value
Interval::para_t p = (Interval::para_t)KeywordList::Enumeration("Interval::para_t", nameAttrib);
```

The code generator will now create the implementation of the functions:

- `KeywordList::Enumeration`
- `KeywordList::Description`
- `KeywordList::Keyword`
- `KeywordList::Unit`
- `KeywordList::Color` - meaningful for coloring types in user interfaces
- `KeywordList::DefaultValue` - can be used as default value in user interfaces
- `KeywordList::Count` - returns number of enumeration values in this category
- `KeywordList::KeywordExists` - convenience function, same as comparing result of `Enumeration()` with -1

2.2. Usage

The header file `NANDRAD_KeywordList.h` is always the same and can be included directory. The corresponding implementation file `NANDRAD_KeywordList.cpp` is generated in the same directory as the NANDRAD header files.

2.3. CMake automation

The automatic update of the keyword list is triggered by a custom rule in the NANDRAD CMake project file:

```
# collect a list of all header files of the Nandrad library
file( GLOB Nandrad_HDRS ${PROJECT_SOURCE_DIR}/../../src/*.h )

# run the NandradCodeGenerator tool whenever the header files have changed
# to update the NANDRAD_KeywordList.h and NANDRAD_KeywordList.cpp file
add_custom_command (
    OUTPUT   ${PROJECT_SOURCE_DIR}/../../src/NANDRAD_KeywordList.cpp
    DEPENDS  ${Nandrad_HDRS} NandradCodeGenerator
    COMMAND  NandradCodeGenerator
    ARGS     NANDRAD ${PROJECT_SOURCE_DIR}/../../src
)
```

where `NandradCodeGenerator` is built as part of the tool chain as well. The rule has all header files as dependencies so that any change in any header file will result in a call to the code generator. The code generator will then create the file `NANDRAD_KeywordList.cpp`.

3. Read/write to XML support, and utility macros

A second task for the code generator is to create functions for serialization of data structures to XML files. Hereby, the TinyXML-library is used.

3.1. Read/write code

Since reading/writing XML code is pretty straight forward, much of this code writing can be generalized. Let's take a look at a simple example.

Class Sensor, with declarations of `readXML()` and `writeXML()` functions

```
class Sensor {
public:
    // *** PUBLIC MEMBER FUNCTIONS ***

    void readXML(const TiXmlElement * element);
    TiXmlElement * writeXML(TiXmlElement * parent) const;

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id = NANDRAD::INVALID_ID;     // XML:A:required
    /*! Name of the measured quantity */
    std::string                         m_quantity;                     // XML:E
};
```

Since we use C++11 code, member variable initialization with the = assignment in header is ok and saves creating default constructors. Do this!

The two members are written into file as follows:

```
<Sensor id="12">
    <Quantity>Temperature</Quantity>
</Sensor>
```

The implementation looks as follows:

Implementation of `Sensor::readXML()`

```
void Sensor::readXML(const TiXmlElement * element) {
    FUNCID(Sensor::readXML);

    try {
        // search for mandatory attributes
        if (!TiXmlAttribute::attributeByName(element, "id"))
            throw IBK::Exception( IBK::FormatString(XML_READ_ERROR).arg(element->Row()).arg(
                IBK::FormatString("Missing required 'id' attribute.") ), FUNC_ID);

        // reading attributes
        const TiXmlAttribute * attrib = element->FirstAttribute();
        while (attrib) {
            const std::string & attribName = attrib->NameStr();
            if (attribName == "id")
                m_id = readPODAttributeValue<unsigned int>(element, attrib);
            else {
                IBK::IBK_Message(IBK::FormatString(XML_READ_UNKNOWN_ATTRIBUTE).arg(attribName).arg(element->Row()),
IBK::MSG_WARNING, FUNC_ID, IBK::VL_STANDARD);
            }
            attrib = attrib->Next();
        }
        // search for mandatory elements
        // reading elements
        const TiXmlElement * c = element->FirstChildElement();
        while (c) {
            const std::string & cName = c->ValueStr();
            if (cName == "Quantity")
                m_quantity = c->GetText();
            else {
                IBK::IBK_Message(IBK::FormatString(XML_READ_UNKNOWN_ELEMENT).arg(cName).arg(element->Row()), IBK
::MSG_WARNING, FUNC_ID, IBK::VL_STANDARD);
            }
            c = c->NextSiblingElement();
        }
    }
    catch (IBK::Exception & ex) {
        throw IBK::Exception( ex, IBK::FormatString("Error reading 'Sensor' element."), FUNC_ID);
    }
    catch (std::exception & ex2) {
        throw IBK::Exception( IBK::FormatString("%1\nError reading 'Sensor' element.").arg(ex2.what()), FUNC_ID);
    }
}
```

In this function there is a lot of code that is repeated nearly identical in all files of the data model. For example, reading of attributes, converting them to number values (including error checking), testing for known child

elements (and error handling) and the outer exception catch clauses. Similarly, this looks for the `writeXML()` function.

Implementation of `Sensor::writeXML()`

```
TiXmlElement * Sensor::writeXML(TiXmlElement * parent, bool /*detailedOutput*/) const {
    TiXmlElement * e = new TiXmlElement("Sensor");
    parent->LinkEndChild(e);

    e->SetAttribute("id", IBK::val2string<unsigned int>(m_id));
    if (!m_quantity.empty())
        TiXmlElement::appendSingleAttributeElement(e, "Quantity", nullptr, std::string(), m_quantity);
    return e;
}
```

In order for the code generator to create these two functions, we need to add some *annotations* to original class declaration:

Class Sensor, with annotations for read/write code generation

```
class Sensor {
public:
    // *** PUBLIC MEMBER FUNCTIONS ***

    void readXML(const TiXmlElement * element);
    TiXmlElement * writeXML(TiXmlElement * parent) const;

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id = NANDRAD::INVALID_ID;     // XML:A:required
    /*! Name of the measured quantity */
    std::string                         m_quantity;                     // XML:E
};
```

The `// XML:A` says: make this an attribute. The `// XML:E` says: make this a child-element. The additional `required` keyword means: this attribute (or element) must be provided, otherwise `readXML()` will throw an exception.

The annotations can be used for quite a few data types. Rules for these are given in section XML Serialization.

3.2. Utility Macros

Since the declaration for the `readXML()` and `writeXML()` functions are always the same, we can avoid typing errors by using a define:

Global code generator helpers

```
#define NANDRAD_READWRITE \
    void readXML(const TiXmlElement * element); \
    TiXmlElement * writeXML(TiXmlElement * parent) const;
```

The header is now very short:

Class Sensor, using code generator

```
class Sensor {
public:
    // *** PUBLIC MEMBER FUNCTIONS ***

    NANDRAD_READWRITE

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id = NANDRAD::INVALID_ID;     // XML:A:required
    /*! Name of the measured quantity */
    std::string                         m_quantity;                     // XML:E
};
```

The implementation file `NANDRAD_Sensor.cpp` is no longer needed and can be removed.

The code generator will create a file: `ncg_NANDRAD_Sensor.cpp` with the functions `Sensor::readXML()` and `Sensor::writeXML()`.

To avoid regenerating (and recompiling) all `ncg_*` files whenever *one* header file is modified, the code generator inspects the file creation times of the `ncg_XXX.cpp` file with the latest modification/creation data of the respective `ncg_XXX.h` file. The code is only generated, if the header file is newer than the generated file.

3.2.1. Comparison operator macro

When checking if the content of an object is effectively the same as that of another (possibly freshly constructed) object, we need a comparison operator. Actually, we usually need both `operator==` and `operator!=` (depending on the alorithm used, either of the two is needed). The code for the class `Sensor` normally looks like that:

Comparison operator (inequality)

```
bool Sensor::operator!=(const Sensor & other) const {
    if (m_id != other.m_id)             return true;
    if (m_quantity != other.m_quantity) return true;
    return false;
}
```

The other comparison operator is normally just implemented using the other:

Comparison operator (equality)

```
bool operator==(const Sensor & other) const { return !operator!=(other); }
```

The declaration and the definition of the equality operator can be replaced by a define:

Global code generator helpers

```
#define NANDRAD_COMP(X) \
    bool operator!=(const X & other) const;
```

So the class declaration becomes:

Class Sensor, with comparison function declarations

```
class Sensor {
public:
    // *** PUBLIC MEMBER FUNCTIONS ***

    NANDRAD_READWRITE
    NANDRAD_COMP(Sensor)

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id = NANDRAD::INVALID_ID;     // XML:A:required
    /*! Name of the measured quantity */
    std::string                         m_quantity;                     // XML:E
};
```

3.2.2. Custom read/write functionality

Sometimes, the default read/write code is not enough, because something special needs to be written/read as well. Here, you can simply use an alternative define `NANDRAD_READWRITE_PRIVATE`:

Global code generator helpers

```
#define NANDRAD_READWRITE_PRIVATE \
    void readXMLPrivate(const TiXmlElement * element); \
    TiXmlElement * writeXMLPrivate(TiXmlElement * parent) const;
```

which tells the code generator to generate the read/write code inside the `XXXPrivate`-functions.

You can now implement `readXML()` and `writeXML()` manually, hereby re-using the auto-generated functionality. Below is an example:

Class Sensor, using code generator with private read/write functions

```
class Sensor {
    NANDRAD_READWRITE_PRIVATE
public:
    // *** PUBLIC MEMBER FUNCTIONS ***

    NANDRAD_READWRITE
    NANDRAD_COMP(Sensor)

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id = NANDRAD::INVALID_ID;     // XML:A:required
    /*! Name of the measured quantity */
    std::string                         m_quantity;                     // XML:E
};
```

Implementation file `NANDRAD_Sensor.cpp`

```
void Sensor::readXML(const TiXmlElement * element) {
    // simply reuse generated code
    readXMLPrivate(element);

    // ... read other data from element
}


TiXmlElement * Sensor::writeXML(TiXmlElement * parent, bool detailedOutput) const {
    TiXmlElement * e = writeXMLPrivate(parent, detailedOutput);

    // .... append other data to e
    return e;
}
```

3.2.3. Only writing data if not still default content

To avoid writing empty tags or default values, you can write code like:

Implmenetation of writeXML with default check

```
TiXmlElement * Sensor::writeXML(TiXmlElement * parent, bool detailedOutput) const {
    // check if we still have default data
    if (*this == Sensor())
        return; // still default, do not write anything

    TiXmlElement * e = new TiXmlElement("Sensor");
    parent->LinkEndChild(e);

    e->SetAttribute("id", IBK::val2string<unsigned int>(m_id));
    if (!m_quantity.empty())
        TiXmlElement::appendSingleAttributeElement(e, "Quantity", nullptr, std::string(), m_quantity);
    return e;
}
```

However, the code generator cannot write this automatically, because sometimes it is desired to write even default content. Also, a comparison-operator is not always available.

You can, however, use the macro `NANDRAD_READWRITE_IFNOTEMPTY(X)` instead of the regular `NANDRAD_READWRITE` macro for this:

Macro with check for default values

```
#define NANDRAD_READWRITE_IFNOTEMPTY(X) \
    void readXML(const TiXmlElement * element) { readXMLPrivate(element); } \
    TiXmlElement * writeXML(TiXmlElement * parent) const { if (*this != X()) return writeXMLPrivate(parent); else
return nullptr; }
```

Since this macro uses the functions `readXMLPrivate()` and `writeXMLPrivate()` you also need to tell the code generator to use the private function versions, as in the following example:

Class Sensor, using code generator with private read/write functions and check to not write default data

```
class Sensor {
    NANDRAD_READWRITE_PRIVATE
public:
    // *** PUBLIC MEMBER FUNCTIONS ***

    NANDRAD_READWRITE_IFNOTEMPTY(Sensor)
    NANDRAD_COMP(Sensor)

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id = NANDRAD::INVALID_ID;     // XML:A:required
    /*! Name of the measured quantity */
    std::string                         m_quantity;                     // XML:E
};
```

4. Specifications

4.1. Keyword List Support

The parse requires fairly consistent code to be recognized, with the following rules. Look at the following example:

```
class MyClass {
public:


    enum parameterSet {
        PS_PARA1,      // Keyword: PARA1     'some lengthy description'
        PS_PARA2,      // Keyword: PARA2     [K] <#4512FF> {273.15} 'A temperature parameter'
        NUM_PS
    }

    enum otherPara_t {
        OP_P1,         // Keyword: P1
        OP_P2,         // Keyword: P2
        OP_P3,         // Keyword: P3
        NUM_OP
    }
...
}
```

Here are the rules/conventions (how the parser operates):

- a class scope is recognized by a string `class xxxx` (same line)
- an enum scope is recognized by a string `enum yyyy` (same line)
- a keyword specification is recognized by the string `// Keyword:` (with space between `//` and `Keyword:`!)
- either *all* enumeration values (except the line with `NUM_XXX`) must have a keyword specification, or *none* (the keyword spec is used to increment the enum counter)
- you **must not** assign a value to the enumeration like `MY_ENUM = 15,` - the parser does not support this format. With proper scoping, you won't need such assignments for parameter lists.

The parser isn't a c++ parser and does not know about comments. If the strings mentioned above are found inside a comment, the parser will not know the difference. As a consequence, the following code will confuse the parser and generate wrong keyword categories:

```
class MyClass {
public:

    /* Inside this
       class my stuff will work
       perfectly!
    */

    enum para_t {
    ...
    }
...
}
```

This will generate the keyword category `my::para_t` because `class my` is recognized as class scope. So, **do not do this**! Same applies to enum documentation.

Thankfully, documentation is to be placed above the class/enum declaration lines and should not interfere with the parsing.

When using class forward declarations, always put only the class declaration on a single line without comments afterwards:

```
// forward declarations
class OtherClass;
class OtherParentClass;
class YetAnotherClass;
```

The parser will detect forward declarations when the line is ended with a `;` character. Again, this should normally not be an issue, unless someone uses a forward declaration of a class *inside* a class scope.

4.2. Keyword Parameters

A keyword specification line has the following format:

```
KW_ENUM_VALUE,  // Keyword:   Keyword-Name  [unit]  <color>  {default value} 'description'
```

The `Keyword-Name` can be actually a list of white-space separated keywords that are used to convert to the enumeration value: for example:

```
SP_HEATCONDCOEFF, // Keyword: HEATCONDCOEFF ALPHA [W/m2K] 'Heat conduction coefficient'
```

Allows to convert strings `HEATCONDCOEFF` and `ALPHA` to enum value `SP_HEATCONDCOEFF`, but conversion from `SP_HEATCONDCOEFF` to string always yields the first keyword `HEATCONDCOEFF` in the list.

The remaining parameters *unit*, *color*, *default value* and *description* are **optional**. But if present, they must appear in the order shown above. This is just to avoid nesting problems and is strictly only required from the description, since this may potentially contain the characters `<>[]{}`.

The *default value* must be a floating point number in C locale format. Similarly as color and unit, this parameter is meaningful for user interfaces with somewhat generic parameter input handling.

4.3. XML Serialization

In order for the CodeGenerator to work correct, we need a ~~few~~ lots of conventions:

- only one class per file
- only member variables with `// XML:A` or `// XML:E` annotations are written/read (code generated for them)
- all member variables must be prefixed `m_`
- only the types used in the following test class are currently supported. Complex types with own `readXML()` and `writeXML()` functions are always supported (see section Reading/writing custom complex types)

Example class with different types currently supported by code generator

```
class SerializationTest {
public:

    NANDRAD_READWRITE

    enum test_t {
        t_x1,                                           // Keyword: X1
        t_x2,                                           // Keyword: X2
        NUM_test
    };

    enum intPara_t {
        IP_i1,                                          // Keyword: I1
        IP_i2,                                          // Keyword: I2
        NUM_IP
    };

    int                 m_id1       = 5;                // XML:A:required
    unsigned int        m_id2       = 10;               // XML:A
    bool                m_flag1     = false;            // XML:A
    double              m_val1      = 42.42;            // XML:A
    test_t              m_testBla   = t_x1;             // XML:A
    std::string         m_str1      = "Blubb";          // XML:A
    IBK::Path           m_path1     = IBK::Path("/tmp");// XML:A
    IBK::Unit           m_u1        = IBK::Unit("K");   // XML:A

    int                 m_id3       = 10;               // XML:E:required
    unsigned int        m_id4       = 12;               // XML:E
    bool                m_flag2     = true;             // XML:E
    double              m_val2      = 41.41;            // XML:E
    test_t              m_testBlo   = t_x2;             // XML:E
    std::string         m_str2      = "blabb";          // XML:E
    IBK::Path           m_path2     = IBK::Path("/var");// XML:E
    IBK::Unit           m_u2        = IBK::Unit("C");   // XML:E
    double              m_x5;                           // XML:E
    IBK::Flag           m_f;                            // XML:E
    IBK::Time           m_time1;                        // XML:E
    IBK::Time           m_time2;                        // XML:E

    DataTable           m_table;                        // XML:E

    std::vector<double>     m_dblVec;                   // XML:E
    std::vector<Interface>  m_interfaces;               // XML:E

    IBK::Parameter      m_para[NUM_test];               // XML:E
    IBK::IntPara        m_intPara[NUM_IP];              // XML:E
    IBK::Flag           m_flags[NUM_test];              // XML:E

    IBK::LinearSpline   m_spline;                       // XML:E

    // example for a generic class with own readXML() and writeXML() function
    Schedule            m_sched;                        // XML:E

};
```

The following conventions are used when composing the XML content:

1. parent XML-Element name is always the same as the class name, so in the example above the xml-tag is `SerializationTest`.
2. child tag names are composed of the capitalized variable name without `m_` prefix, so `m_testParameter` becomes

`TestParameter`

3. attribute names are composed of the variable name without `m_` prefix, so `m_flagFive` becomes attribute `flagFive`
4. for vector quantities (for example `std::vector<Interface> m_interfaces`, the variable name is used to generate the list-type XML tag, here `Interfaces` (again just by capitalizing the variable name string). Inside the list the actual members are written, hereby calling `writeXML()` in the child elements (`Interface::writeXML()` in the example above)
5. static arrays are supported, but only with enumeration index where the enum is parametrized with keyword list and `NUM_xxx` enumeration value as last enum value. The xml-tags are named as the keywords for the corresponding enum type).

The following XML-output is generated from the class declaration above:

```
<?xml version="1.0" encoding="UTF-8" ?>
<NandradProject>
    <SerializationTest id1="5" id2="10" flag1="0" val1="42.42" testBla="X1" str1="Blubb" path1="/tmp" u1="K">
        <Id3>10</Id3>
        <Id4>12</Id4>
        <Flag2>1</Flag2>
        <Val2>41.41</Val2>
        <TestBlo>X2</TestBlo>
        <Str2>blabb</Str2>
        <Path2>/var</Path2>
        <U2>C</U2>
        <X5>43.43</X5>
        <IBK:Flag name="F">true</IBK:Flag>
        <Time1>01.01.07 12:47:12</Time1>
        <Schedule type="Friday">
            <StartDate>01.01.00  0:00:00</StartDate>
            <EndDate>01.01.00  0:00:00</EndDate>
            <DailyCycles>
                <DailyCycle />
            </DailyCycles>
        </Schedule>
        <Table>Col1:1,5,3;Col2:7,2,2;</Table>
        <DblVec>0,12,24</DblVec>
        <Interfaces>
            <Interface id="1" location="A" zoneId="0">
                <!--Interface to outside-->
            </Interface>
        </Interfaces>
        <IBK:Parameter name="X1" unit="C">12</IBK:Parameter>
        <IBK:IntPara name="I1">13</IBK:IntPara>
        <IBK:IntPara name="I2">15</IBK:IntPara>
        <IBK:Flag name="X2">true</IBK:Flag>
        <IBK:LinearSpline name="Spline">
            <X unit="-">0 1 1.4 2 </X>
            <Y unit="-">1 2 3.4 5 </Y>
        </IBK:LinearSpline>
    </SerializationTest>
</NandradProject>
```

When writing custom types like `Schedule` in the example above, you **must only have one object** declared as member variable, since the xml-tag is generated based on the variable type name. This is due to the fact, that the code generator currently just calls `writeXML()` inside such complex types and these classes (currently) set the child xml tag name to the class name. In the example above, the class name is `Schedule` and hence the xml-tag is named `Schedule` and not `Sched` as it would be according to the standard naming rules.

For types `IBK::Parameter`, `IBK::IntPara`, `IBK::LinearSpline` and `IBK::Flag` the name must be set exactly to the name of the generated xml-tag name. So, a parameter with member variable `m_transferCoefficient` must be given the name `TransferCoefficient`. In case of static arrays, where the enumeration value determines keyword and thus xml-tag, the name is ignored.

The code generator creates additional code to prevent writing of undefined data:

- `IBK::Parameter`, `IBK::IntPara` and `IBK::Flag` with empty name are not written
- enumeration values where the value matches the corresponding `NUM_xxx` value are not written
- `IBK::Time` with invalid time/date are not written
- empty strings/paths are not written
- undefined units (id=0) are not written

4.3.1. Reading/writing custom complex types

Any data type not listed in the example above and with `// XML:E` annotation is treated by the code generator as a complex type with own functions `readXML()` and `writeXML()` according to the `NANDRAD_READWRITE` macro. The code generator with create code to simply call these functions when writing such code.

When reading an XML-file, the tag is compared with the typename of the member variable (`Schedule` in the example above for member variable `m_sched`) and if matched, an object of said type is created and the `readXML()` function is called for this child tag. Then, the variable is *assigned* to the member variable. Hence, the complex type also requires an assignment operator. This is usually automatically generated, but for classes with pointers or special resource management, you may need to provide this assignment operator in addition to the `readXML()` and `writeXML()` functions.

4.3.2. Unsupported data member types

For any kind of special data types, like `std::map<std::string, std::vector<double> >` you cannot use the code generator to create read/write code for. When you add a read/write annotation to such variables, the code generator will complain about unsupported types and may generate not compiling code.

In such cases you have two options:

1. create your own `readXML()` and `writeXML()` functions (possibly by copy&pasting other generated functions from `ncg_*` files and adjusting the code to your needs). For other member variables whose types are supported by the code generator, you may still use the code generator, but you must use the `NANDRAD_READWRITE_PRIVATE` macro (see example in section Custom read/write functionality).
2. change the type to something different, possibly creating another class with standardized behavior. So, for example, you could store `std::map<std::string, std::vector<double> >` data in `std::vector<NamedDblVector>` where `NamedDblVector` contains a `std::string` and `std::vector<double>` members, both of which are fully supported by the code generator. You may need to code the check for duplicate names yourself.