OAK RIDGE National Laboratory

EnumeratedString: safety and simplicity for finite string sets

Presentation to mantid 2023
Reece Boston, PhD
RSE at ORNL

ORNL is managed by UT-Battelle LLC for the US Department of Energy

U.S. DEPARTMENT OF ENERGY

The Problem: logical string properties

e.g. GetDetectorOffsets

Properties

Name	Direction	Type	Default	Description
InputWorkspace	Input	MatrixWorkspace	*Mandatory*	A 2D workspace with X values of d-spacing
Step	Input	number	0.001	Step size used to bin d-spacing data
DReference	Input	number	2	Center of reference peak in d-space
XMin	Input	number	0	Minimum of CrossCorrelation data to search for peak, usually negative
XMax	Input	number	0	Maximum of CrossCorrelation data to search for peak, usually positive
GroupingFileName	Input	string		Optional: The name of the output CalFile to save the generated OffsetsWorkspace. Allowed extensions: ['.cal']
OutputWorkspace	Output	OffsetsWorkspace		An output workspace containing the offsets.
MaskWorkspace	Output	MatrixWorkspace	Mask	An output workspace containing the mask.
PeakFunction	Input	string	Gaussian	The function type for fitting the peaks. Allowed values: ['AsymmetricPearsonVII', 'BackToBackExponential', 'Bk2BkExpConvPV', 'DeltaFunction', 'ElasticDiffRotDiscreteCircle', 'ElasticDiffSphere', 'ElasticIsoRotDiff', 'ExamplePeakFunction', 'Gaussian', 'IkedaCarpenterPV', 'Lorentzian', 'PseudoVoigt', 'Voigt']
EstimateFWHM	Input	boolean	False	Whether to esimate FWHM of peak function when estimating fit parameters
MaxOffset	Input	number	1	Maximum absolute value of offsets; default is 1
OffsetMode	Input	string	Relative	Whether to calculate a relative, absolute, or signed offset. Allowed values: ['Relative', 'Absolute', 'Signed']
DIdeal	Input	number	2	The known peak centre value from the NIST standard information, this is only used in Absolute OffsetMode.

The Problem: logical string properties

e.g. GetDetectorOffsets

```
/* Signed mode calculates offset in number of bins */
std::vector<std::string> modes{"Relative", "Absolute", "Signed"};

declareProperty("OffsetMode", "Relative", std::make_shared<StringListValidator>(modes),
                "Whether to calculate a relative, absolute, or signed offset");
```

The Problem: logical string properties

e.g. GetDetectorOffsets

```
void GetDetectorOffsets::exec() {
  inputW = getProperty("InputWorkspace");
  m_Xmin = getProperty("XMin");
  m_Xmax = getProperty("XMax");
  m_maxOffset = getProperty("MaxOffset");
  if (m_Xmin >= m_Xmax)
    throw std::runtime_error("Must specify m_Xmin<m_Xmax");
  m_dreference = getProperty("DReference");
  m_step = getProperty("Step");
  m_estimateFWHM = getProperty("EstimateFWHM");

  std::string mode_str = getProperty("OffsetMode");

  if (mode_str == "Absolute") {
    mode = offset_mode::absolute_offset;
  }

  else if (mode_str == "Relative") {
    mode = offset_mode::relative_offset;
  }

  else if (mode_str == "Signed") {
    mode = offset_mode::signed_offset;
  }
```

(this is smartly casting these to related enums, but still requires this string comparison)

```
enum class offset_mode : int { signed_offset, relative_offset, absolute_offset };
```

The Problem: logical string properties

e.g. GetDetectorOffsets

```
void GetDetectorOffsets::exec() {
  inputW = getProperty("InputWorkspace");
  m_Xmin = getProperty("XMin");
  m_Xmax = getProperty("XMax");
  m_maxOffset = getProperty("MaxOffset");
  if (m_Xmin >= m_Xmax)
    throw std::runtime_error("Must specify m_Xmin<m_Xmax");
  m_dreference = getProperty("DReference");
  m_step = getProperty("Step");
  m_estimateFWHM = getProperty("EstimateFWHM");

  std::string mode_str = getProperty("OffsetMode");

  if (mode_str == "Absolute") {
    mode = offset_mode::absolute_offset;
  }

  else if (mode_str == "Relative") {
    mode = offset_mode::relative_offset;
  }

  else if (mode_str == "Singed") {
    mode = offset_mode::signed_offset;
  }
```

(this is smartly casting these to related enums, but still requires this string comparison)

```
enum class offset_mode : int { signed_offset, relative_offset, absolute_offset };
```

The Problem: logical string properties

- string comparison is slower
- string comparison is more prone to human error
- string comparison is inelegant
- string comparison is not gauranteed to be valid
- string comparison does not correspond to the logical operation being performed

The Solution: enum

- enums have discrete values
- enums have finite allowed values
- enums compare like integers
- enums can branch with switch statements
- enum class can only be set with enums, not primitives (no mistyping values)

The Problem with the Solution

how enums work:

```
enum offset_mode: int {
    signed_offset=0,
    relative_offset=1,
    absolute_offset=2
};
```

how we want enums to work:

```
enum offset_mode: std::string {
    signed_offset="Signed",
    relative_offset="Relative",
    absolute_offest = "Absolute"
};
```

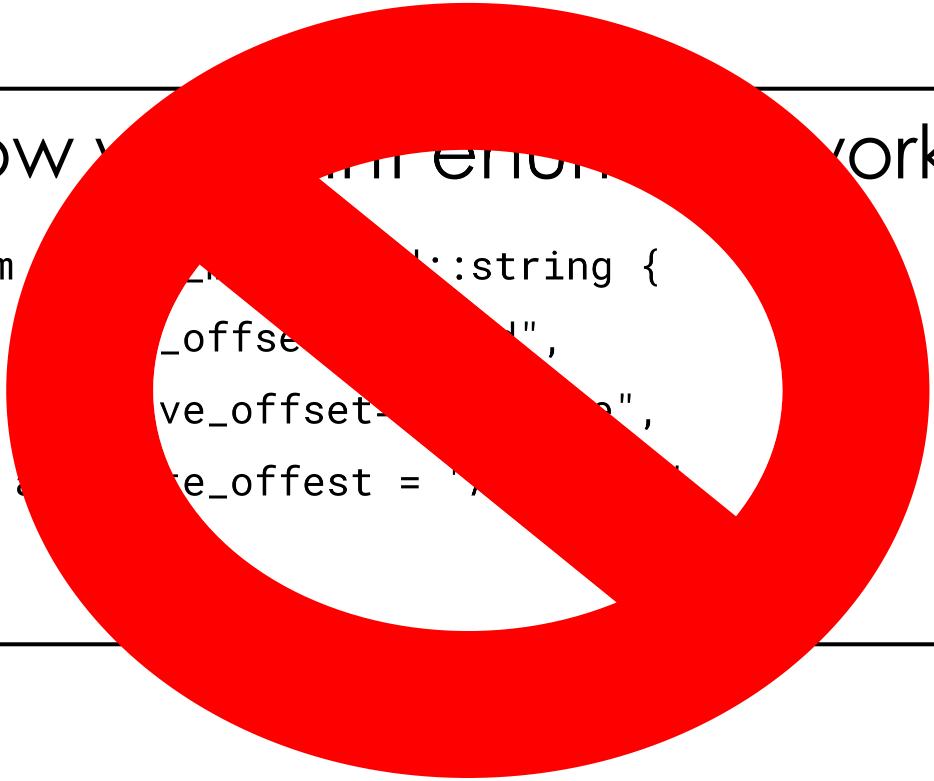
The Problem with the Solution

how enums work:

```
enum offset_mode: int {
    signed_offset=0,
    relative_offset=1,
    absolute_offset=2
};
```

how y... enu... ork:

```
enum ... _... ::string {
    ... _offse... ",
    ... ve_offset... ",
    a... e_offest = ... 
};
```



The Workaround: EnumeratedString

- Create the enums, associated vector of strings (use the StringListValidator)
- Bind these together in EnumeratedString
- Can be natively set from the string property
- Can be natively compared to:
  - enum values
  - std::string objects
  - string literals
  - integer types (if not `enum class`, else easily cast)
- Can be used in `if/else`, `switch`, and `for` structures

EnumeratedString

Definition and Constructors

```
template <class E, const std::vector<std::string> *names,
          std::function<bool(const std::string &, const std::string &)> *stringComparator = &compareStrings>
class EnumeratedString {
  /**
   * @tparam class E an `enum`, the final value *must* be `enum_count`
   *              (i.e. `enum class Fruit {apple, orange, enum_count}`)
   * @tparam a pointer to a static vector of string names for each enum
   * The enum and string array *must* have same order.
   *
   * @tparam an optional pointer to a statically defined string comparator.
   */
public:
  EnumeratedString() { ensureCompatibleSize(); }

  EnumeratedString(const E e) {
    ensureCompatibleSize();
    this->operator=(e);
  }

  EnumeratedString(const std::string &s) {
    ensureCompatibleSize();
    this->operator=(s);
  }

  EnumeratedString(const EnumeratedString &es) : value(es.value), name(es.name) {}
```

EnumeratedString

Comparisons

```
// for comparison of the object to either enums or strings
bool operator==(const E e) const { return value == e; }
bool operator!=(const E e) const { return value != e; }

bool operator==(const std::string &s) const { return (*stringComparator)(name, s); }
bool operator!=(const std::string &s) const { return !(*stringComparator)(name, s); }

bool operator==(const char *s) const { return (*stringComparator)(name, std::string(s)); }
bool operator!=(const char *s) const { return !(*stringComparator)(name, std::string(s)); }

bool operator==(const EnumeratedString &es) const { return value == es.value; }
bool operator!=(const EnumeratedString &es) const { return value != es.value; }

const char *c_str() const { return name.c_str(); }
static size_t size() { return names->size(); }
```

EnumeratedString

Checks

```
private:
  E value;
  std::string name;

  // given a string, find the corresponding enum value
  E findEFromString(const std::string &s) {
    E e = E(0);
    for (; size_t(e) < names->size(); e = E(size_t(e) + 1))
      if ((*stringComparator)(s, names->at(size_t(e))))
        break;
    return e;
  }

  void ensureCompatibleSize() {
    if (size_t(E::enum_count) != names->size()) {
      std::stringstream msg;
      msg << "Size of " << typeid(E).name() << " incompatible with vector of names: ";
      msg << size_t(E::enum_count) << " vs. " << names->size() << std::endl;
      throw std::runtime_error(msg.str());
    }
  }
```

EnumeratedString

Casting and setting

```
// treat the object as either the enum, or a string
operator E() const { return value; }
operator std::string() const { return name; }

// assign the object either by the enum, or by string
EnumeratedString &operator=(E e) {
  if (int(e) >= 0 && size_t(e) < names->size()) {
    value = e;
    name = names->at(size_t(e));
  } else {
    std::stringstream msg;
    msg << "Invalid enumerator " << int(e) << " for enumerated string " << typeid(E).name();
    throw std::runtime_error(msg.str());
  }
  return *this;
}
EnumeratedString &operator=(const std::string &s) {
  E e = findEFromString(s);
  if (e != E::enum_count) {
    value = e;
    name = s;
  } else {
    std::stringstream msg;
    msg << "Invalid string " << s << " for enumerated string " << typeid(E).name();
    throw std::runtime_error(msg.str());
  }
  return *this;
}
```

Using EnumeratedString

e.g. GetDetectorOffsets

Using EnumeratedString

e.g. GetDetectorOffsets

Step 1: create the enum, the string list, and a typedef

```
#include "MantidKernel/EnumeratedString.h"

namespace Mantid::Algorithms {

namespace {
  enum class offset_mode : int { signed_offset, relative_offset, absolute_offset, enum_count };
  std::vector<std::string> modes{"Relative", "Absolute", "Signed"};
  typedef Mantid::Kernel::EnumeratedString<offset_mode, &modes> OFFSETMODE;
}

// Register the class into the algorithm factory
DECLARE_ALGORITHM(GetDetectorOffsets)
```

header only!

Last element must be called `enum_count` to check size against the string list

Using EnumeratedString

e.g. GetDetectorOffsets

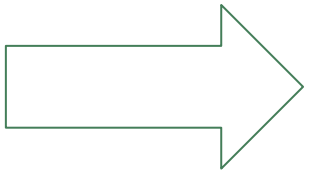
Step 2: initialize an EnumeratedString variable

```
std::string mode_str = getProperty("OffsetMode");

if (mode_str == "Absolute") {
  mode = offset_mode::absolute_offset;
}

else if (mode_str == "Relative") {
  mode = offset_mode::relative_offset;
}

else if (mode_str == "Signed") {
  mode = offset_mode::signed_offset;
}
```



```
OFFSETMODE mode = getPropertyValue("OffsetMode");
```

Using EnumeratedString

e.g. GetDetectorOffsets

Step 3: use in branching logic – if/else or switch, as you prefer

```
double offset = function->getParameter(3); // params[3]; // f1.PeakCentre
if (mode == offset_mode::signed_offset) {
  offset *= -1;
}
else if (mode == offset_mode::relative_offset) {
  offset = -1. * offset * m_step / (m_dreference + offset * m_step);
}
else if (mode == offset_mode::absolute_offset) {
  offset = -1. * offset * m_step / (m_dreference + offset * m_step);
  offset += (m_dideal - m_dreference) / m_dreference;
}
```

```
double offset = function->getParameter(3); // params[3]; // f1.PeakCentre
switch(mode){
  case offset_mode::signed_offset:
    offset *= -1;
    break;
  case offset_mode::relative_offset:
    offset = -1. * offset * m_step / (m_dreference + offset * m_step);
    break;
  case offset_mode::absolute_offset:
    offset = -1. * offset * m_step / (m_dreference + offset * m_step);
    offset += (m_dideal - m_dreference) / m_dreference;
    break;
}
```

you can compare to an enum, to a string, or to an EnumeratedString

Algos with EnumeratedString

- CalculateDIFC
- ConvertDiffCal
- Rebin
- LoadDiffCal
- LoadEmptyInstrument
- more to come!

More info

- See dev documentation (near bottom of page)
- See the test package (inside Framework/Kernel/tests)
- See the previous-listed algos

Thanks!