YAML I/O

Introduction to YAML

YAML is a human readable data serialization language. The full YAML language spec can be read at yaml.org. The simplest form of yaml is just “scalars”, “mappings”, and “sequences”. A scalar is any number or string. The pound/hash symbol (#) begins a comment line. A mapping is a set of key-value pairs where the key ends with a colon. For example:

```
# a mapping
name:      Tom
hat-size:  7
```

A sequence is a list of items where each item starts with a leading dash (‘-‘). For example:

```
# a sequence
- x86
- x86_64
- PowerPC
```

You can combine mappings and sequences by indenting. For example a sequence of mappings in which one of the mapping values is itself a sequence:

```
# a sequence of mappings with one key's value being a sequence
```

```
- name:      Tom
  cpus:
   - x86
   - x86_64
- name:      Bob
  cpus:
   - x86
- name:      Dan
  cpus:
   - PowerPC
   - x86
```

Sometime sequences are known to be short and the one entry per line is too verbose, so YAML offers an alternate syntax for sequences called a “Flow Sequence” in which you put comma separated sequence elements into square brackets. The above example could then be simplified to :

```
# a sequence of mappings with one key's value being a flow sequence
- name:      Tom
  cpus:      [ x86, x86_64 ]
- name:      Bob
  cpus:      [ x86 ]
- name:      Dan
  cpus:      [ PowerPC, x86 ]
```

Introduction to YAML I/O

The use of indenting makes the YAML easy for a human to read and understand, but having a program read and write YAML involves a lot of tedious details. The YAML I/O library structures and simplifies reading and writing YAML documents.

YAML I/O assumes you have some “native” data structures which you want to be able to dump as YAML and recreate from YAML. The first step is to try writing example YAML for your data structures. You may find after looking at possible YAML representations that a direct mapping of your data structures to YAML is not very readable. Often the fields are not in the order that a human would find readable. Or the same information is replicated in multiple locations, making it hard for a human to write such YAML correctly.

In relational database theory there is a design step called normalization in which you reorganize fields and tables. The same considerations need to go into the design of your YAML encoding. But, you may not want to change your exisiting native data structures. Therefore, when writing out YAML there may be a normalization step, and when reading YAML there would be a corresponding denormalization step.

YAML I/O uses a non-invasive, traits based design. YAML I/O defines some abstract base templates. You specialize those templates on your data types. For instance, if you have an eumerated type FooBar you could specialize ScalarEnumerationTraits on that type and define the enumeration() method:

```
using llvm::yaml::ScalarEnumerationTraits;
using llvm::yaml::IO;

template <>
struct ScalarEnumerationTraits<FooBar> {
  static void enumeration(IO &io, FooBar &value) {
  ...
  }
};
```

As with all YAML I/O template specializations, the ScalarEnumerationTraits is used for both reading and writing YAML. That is, the mapping between in-memory enum values and the YAML string representation is only in place. This assures that the code for writing and parsing of YAML stays in sync.

To specify a YAML mappings, you define a specialization on llvm::yaml::MapppingTraits. If your native data structure happens to be a struct that is already normalized, then the specialization is simple. For example:

```
using llvm::yaml::MapppingTraits;
using llvm::yaml::IO;

template <>
struct MapppingTraits<Person> {
  static void mapping(IO &io, Person &info) {
    io.mapRequired("name",         info.name);
    io.mapOptional("hat-size",     info.hatSize);
  }
};
```

A YAML sequence is automatically infered if you data type has begin()/end() iterators and a push_back() method. Therefore any of the STL containers (such as std::vector<>) will automatically translate to YAML sequences.

Once you have defined specializations for your data types, you can programmatically use YAML I/O to write a YAML document:

```
using llvm::yaml::Output;

Person tom;
tom.name = "Tom";
tom.hatSize = 8;
Person dan;
dan.name = "Dan";
dan.hatSize = 7;
std::vector<Person> persons;
persons.push_back(tom);
persons.push_back(dan);

Output yout(llvm::outs());
yout << persons;
```

This would write the following:

```
- name:      Tom
  hat-size:  8
- name:      Dan
  hat-size:  7
```

And you can also read such YAML documents with the following code:

```
using llvm::yaml::Input;

typedef std::vector<Person> PersonList;
std::vector<PersonList> docs;

Input yin(document.getBuffer());
yin >> docs;

if ( yin.error() )
  return;

// Process read document
for ( PersonList &pl : docs ) {
  for ( Person &person : pl ) {
    cout << "name=" << person.name;
  }
}
```

One other feature of YAML is the ability to define multiple documents in a single file. That is why reading YAML produces a vector of your document type.

Error Handling

When parsing a YAML document, if the input does not match your schema (as expressed in your XxxTraits<> specializations). YAML I/O will print out an error message and your Input object's error() method will return true. For instance the following document:

```
- name:      Tom
  shoe-size: 12
- name:      Dan
  hat-size:  7
```

Has a key (shoe-size) that is not defined in the schema. YAML I/O will automatically generate this error:

```
YAML:2:2: error: unknown key 'shoe-size'
  shoe-size:       12
  ^~~~~~~~~
```

Similar errors are produced for other input not conforming to the schema.

Scalars

YAML scalars are just strings (i.e. not a sequence or mapping). The YAML I/O library provides support for translating between YAML scalars and specific C++ types.

Built-in types

The following types have built-in support in YAML I/O:

- bool
- float
- double
- StringRef
- int64_t
- int32_t
- int16_t
- int8_t
- uint64_t
- uint32_t
- uint16_t
- uint8_t

That is, you can use those types in fields of MapppingTraits or as element type in sequence. When reading, YAML I/O will validate that the string found is convertible to that type and error out if not.

Unique types

Given that YAML I/O is trait based, the selection of how to convert your data to YAML is based on the type of your data. But in C++ type matching, typedefs do not generate unique type names. That means if you have two typedefs of unsigned int, to YAML I/O both types look exactly like unsigned int. To facilitate make unique type names, YAML I/O provides a macro which is used like a typedef on built-in types, but expands to create a class with conversion operators to and from the base type. For example:

```
LLVM_YAML_STRONG_TYPEDEF(uint32_t, MyFooFlags)
LLVM_YAML_STRONG_TYPEDEF(uint32_t, MyBarFlags)
```

This generates two classes MyFooFlags and MyBarFlags which you can use in your native data structures instead of uint32_t. They are implicitly converted to and from uint32_t. The point of creating these unique types is that you can now specify traits on them to get different YAML conversions.

Hex types

An example use of a unique type is that YAML I/O provides fixed sized unsigned integers that are written with YAML I/O as hexadecimal instead of the decimal format used by the built-in integer types:

- Hex64
- Hex32
- Hex16
- Hex8

You can use llvm::yaml::Hex32 instead of uint32_t and the only different will be that when YAML I/O writes out that type it will be formatted in hexadecimal.

ScalarEnumerationTraits

YAML I/O supports translating between in-memory enumerations and a set of string values in YAML documents. This is done by specializing ScalarEnumerationTraits<> on your enumeration type and define a enumeration() method. For instance, suppose you had an enumeration of CPUs and a struct with it as a field:

```
enum CPUs {
  cpu_x86_64  = 5,
  cpu_x86     = 7,
  cpu_PowerPC = 8
};

struct Info {
  CPUs      cpu;
  uint32_t  flags;
};
```

To support reading and writing of this enumeration, you can define a ScalarEnumerationTraits specialization on CPUs, which can then be used as a field type:

```
using llvm::yaml::ScalarEnumerationTraits;
using llvm::yaml::MapppingTraits;
using llvm::yaml::IO;

template <>
struct ScalarEnumerationTraits<CPUs> {
  static void enumeration(IO &io, CPUs &value) {
    io.enumCase(value, "x86_64",  cpu_x86_64);
    io.enumCase(value, "x86",     cpu_x86);
    io.enumCase(value, "PowerPC", cpu_PowerPC);
  }
};

template <>
struct MapppingTraits<Info> {
  static void mapping(IO &io, Info &info) {
    io.mapRequired("cpu",       info.cpu);
    io.mapOptional("flags",     info.flags, 0);
  }
};
```

When reading YAML, if the string found does not match any of the the strings specified by enumCase() methods, an error is automatically generated. When writing YAML, if the value being written does not match any of the values specified by the enumCase() methods, a runtime assertion is triggered.

BitValue

Another common data structure in C++ is a field where each bit has a unique meaning. This is often used in a “flags” field. YAML I/O has support for converting such fields to a flow sequence. For instance suppose you had the following bit flags defined:

```
enum {
  flagsPointy = 1
  flagsHollow = 2
  flagsFlat   = 4
  flagsRound  = 8
};

LLVM_YAML_UNIQUE_TYPE(MyFlags, uint32_t)
```

To support reading and writing of MyFlags, you specialize ScalarBitSetTraits<> on MyFlags and provide the bit values and their names.

```
using llvm::yaml::ScalarBitSetTraits;
using llvm::yaml::MapppingTraits;
using llvm::yaml::IO;

template <>
struct ScalarBitSetTraits<MyFlags> {
  static void bitset(IO &io, MyFlags &value) {
    io.bitSetCase(value, "hollow",  flagHollow);
    io.bitSetCase(value, "flat",    flagFlat);
    io.bitSetCase(value, "round",   flagRound);
    io.bitSetCase(value, "pointy",  flagPointy);
  }
};

struct Info {
  StringRef   name;
  MyFlags     flags;
};

template <>
struct MapppingTraits<Info> {
  static void mapping(IO &io, Info& info) {
    io.mapRequired("name",  info.name);
    io.mapRequired("flags", info.flags);
   }
};
```

With the above, YAML I/O (when writing) will test mask each value in the bitset trait against the flags field, and each that matches will cause the corresponding string to be added to the flow sequence. The opposite is done when reading and any unknown string values will result in a error. With the above schema, a same valid YAML document is:

```
name:    Tom
flags:   [ pointy, flat ]
```

Custom Scalar

Sometimes for readability a scalar needs to be formatted in a custom way. For instance your internal data structure may use a integer for time (seconds since some epoch), but in YAML it would be much nicer to express that integer in some time format (e.g. 4-May-2012 10:30pm). YAML I/O has a way to support custom formatting and parsing of scalar types by specializing ScalarTraits<> on your data type. When writing, YAML I/O will provide the native type and your specialization must create a temporary llvm::StringRef. When reading, YAML I/O will provide a llvm::StringRef of scalar and your specialization must convert that to your native data type. An outline of a custom scalar type looks like:

```
using llvm::yaml::ScalarTraits;
using llvm::yaml::IO;

template <>
struct ScalarTraits<MyCustomType> {
  static void output(const T &value, llvm::raw_ostream &out) {
    out << value;  // do custom formatting here
  }
  static StringRef input(StringRef scalar, T &value) {
    // do custom parsing here.  Return the empty string on success,
    // or an error message on failure.
    return StringRef();
  }
};
```

Mappings

To be translated to or from a YAML mapping for your type T you must specialize llvm::yaml::MapppingTraits on T and implement the “void mapping(IO &io, T&)” method. If your native data structures use pointers to a class everywhere, you can specialize on the class pointer. Examples:

```
using llvm::yaml::MapppingTraits;
using llvm::yaml::IO;

// Example of struct Foo which is used by value
template <>
struct MapppingTraits<Foo> {
  static void mapping(IO &io, Foo &foo) {
    io.mapOptional("size",      foo.size);
  ...
  }
};

// Example of struct Bar which is natively always a pointer
template <>
struct MapppingTraits<Bar*> {
  static void mapping(IO &io, Bar *&bar) {
    io.mapOptional("size",    bar->size);
  ...
  }
};
```

No Normalization

The mapping() method is responsible, if needed, for normalizing and denormalizing. In a simple case where the native data structure requires no normalization, the mapping method just uses mapOptional() or mapRequired() to bind the struct's fields to YAML key names. For example:

```
using llvm::yaml::MapppingTraits;
using llvm::yaml::IO;

template <>
struct MapppingTraits<Person> {
  static void mapping(IO &io, Person &info) {
    io.mapRequired("name",         info.name);
    io.mapOptional("hat-size",     info.hatSize);
  }
};
```

Normalization

When [de]normalization is required, the mapping() method needs a way to access normalized values as fields. To help with this, there is a template MappingNormalization<> which you can then use to automatically do the normalization and denormalization. The template is used to create a local variable in your mapping() method which contains the normalized keys.

Suppose you have native data type Polar which specifies a position in polar coordinates (distance, angle):

```
struct Polar {
  float distance;
  float angle;
};
```

but you've decided the normalized YAML for should be in x,y coordinates. That is, you want the yaml to look like:

```
x:   10.3
y:   -4.7
```

You can support this by defining a MapppingTraits that normalizes the polar coordinates to x,y coordinates when writing YAML and denormalizes x,y coordindates into polar when reading YAML.

```
using llvm::yaml::MapppingTraits;
using llvm::yaml::IO;

template <>
struct MapppingTraits<Polar> {

  class NormalizedPolar {
  public:
    NormalizedPolar(IO &io)
      : x(0.0), y(0.0) {
    }
    NormalizedPolar(IO &, Polar &polar)
      : x(polar.distance * cos(polar.angle)),
        y(polar.distance * sin(polar.angle)) {
    }
    Polar denormalize(IO &) {
      return Polar(sqrt(x*x+y*y), arctan(x,y));
    }
    float        x;
    float        y;
  };

  static void mapping(IO &io, Polar &polar) {
    MappingNormalization<NormalizedPolar, Polar> keys(io, polar);

    io.mapRequired("x",    keys->x);
    io.mapRequired("y",    keys->y);
  }
};
```

When writing YAML, the local variable “keys” will be a stack allocated instance of NormalizedPolar, constructed from the suppled polar object which initializes it x and y fields. The mapRequired() methods then write out the x and y values as key/value pairs.

When reading YAML, the local variable “keys” will be a stack allocated instance of NormalizedPolar, constructed by the empty constructor. The mapRequired methods will find the matching key in the YAML document and fill in the x and y fields of the NormalizedPolar object keys. At the end of the mapping() method when the local keys variable goes out of scope, the denormalize() method will automatically be called to convert the read values back to polar coordinates, and then assigned back to the second parameter to mapping().

In some cases, the normalized class may be a subclass of the native type and could be returned by the denormalize() method, except that the temporary normalized instance is stack allocated. In these cases, the utility template MappingNormalizationHeap<> can be used instead. It just like MappingNormalization<> except that it heap allocates the normalized object when reading YAML. It never destroyes the normalized object. The denormalize() method can this return “this”.

Default values

Within a mapping() method, calls to io.mapRequired() mean that that key is required to exist when parsing YAML documents, otherwise YAML I/O will issue an error.

On the other hand, keys registered with io.mapOptional() are allowed to not exist in the YAML document being read. So what value is put in the field for those optional keys? There are two steps to how those optional fields are filled in. First, the second parameter to the mapping() method is a reference to a native class. That native class must have a default constructor. Whatever value the default constructor initially sets for an optional field will be that field’s value. Second, the mapOptional() method has an optional third parameter. If provided it is the value that mapOptional() should set that field to if the YAML document does not have that key.

There is one important difference between those two ways (default constructor and third parameter to mapOptional). When YAML I/O generates a YAML document, if the mapOptional() third parameter is used, if the actual value being written is the same as (using ==) the default value, then that key/value is not written.

Order of Keys

When writing out a YAML document, the keys are written in the order that the calls to mapRequired()/mapOptional() are made in the mapping() method. This gives you a chance to write the fields in an order that a human reader of the YAML document would find natural. This may be different that the order of the fields in the native class.

When reading in a YAML document, the keys in the document can be in any order, but they are processed in the order that the calls to mapRequired()/mapOptional() are made in the mapping() method. That enables some interesting functionality. For instance, if the first field bound is the cpu and the second field bound is flags, and the flags are cpu specific, you can programmatically switch how the flags are converted to and from YAML based on the cpu. This works for both reading and writing. For example:

```
using llvm::yaml::MapppingTraits;
using llvm::yaml::IO;

struct Info {
  CPUs        cpu;
  uint32_t    flags;
};

template <>
struct MapppingTraits<Info> {
  static void mapping(IO &io, Info &info) {
    io.mapRequired("cpu",       info.cpu);
    // flags must come after cpu for this to work when reading yaml
    if ( info.cpu == cpu_x86_64 )
      io.mapRequired("flags",  *(My86_64Flags*)info.flags);
    else
      io.mapRequired("flags",  *(My86Flags*)info.flags);
 }
};
```

Sequence

Any type that conforms to having begin(), end(), and push_back() methods, as well as, types name iterator and value_type will automatically be classified as a sequence. Thus, std::vector<>, std::list<>, and llvm::SmallVector<> are automatically sequences.

Flow Sequence

A YAML “flow sequence” is a squence that when written to YAML it uses the inline notation (e.g [foo, bar]). You can force a sequence to be formatted as a flow sequence by adding “static const bool flow = true;” to the class. For instance:

```
struct IntVector : public std::vector<int> {
  // The existence of this member causes YAML I/O to use a flow sequence
  static const bool flow = true;
};
```

With the above, if you used IntVector as the data type in your native data strucutures, then then when converted to YAML, a flow sequence of integers will be used (e.g. [10, -3, 4]).

User Context Data

When an llvm::yaml::Input or llvm::yaml::Output object is created their constructors take an optional “context” parameter. This is a pointer to whatever state information you might need.

For instance, in a previous example we showed how the conversion type for a flags field could be determined at runtime based on the value of another field in the mapping. But what if an inner mapping needs to know some field value of an outer mapping? That is where the “context” parameter comes in. You can set values in the context in the outer map’s mapping() method and retrieve those values in the inner map’s mapping() method.

The context value is just a void*. All your traits which use the context and operate on your native data types, need to agree what the context value actually is. It could be a pointer to an object or struct which your various traits use to shared context sensitive information.

Output

The llvm::yaml::Output class is used to generate a YAML document from your in-memory data structures, using traits defined on your data types. To instantiate an Output object you need an llvm::raw_ostream, and optionally a context pointer:

```
class Output : public IO {
public:
  Output(llvm::raw_ostream &, void *context=NULL);
```

Once you have an Output object, you can use the C++ stream operator on it to write your native data as YAML. One thing to recall is that a YAML file can contain multiple “documents”. If the top level data structure you are streaming as YAML is a mapping, then Output assumes you are generating one document and wraps the mapping output with “`---`” and trailing “`...`”.

```
using llvm::yaml::Output;

void dumpMyMapDoc(const MyMapType &info) {
  Output yout(llvm::outs());
  yout << info;
}
```

The above could produce output like:

```
---
name:      Tom
hat-size:  7
...
```

On the other hand, if the top level data structure you are streaming as YAML is a sequence, Output assumes that each element of the sequence is a document. In order to generate a single document that is itself a sequence, you need to wrap the data in another sequence.

```
using llvm::yaml::Output;

void dumpMySequenceDoc(const MySequenceType &info) {
  // Wrap in another sequence to disambiguate that MySequenceType is
  // not itself a sequence of documents.
  std::vector<MySequenceType> docList;
  docList.push_back(info);

  Output yout(llvm::outs());
  yout << docList;
}
```

The above could produce output like:

```
---
- name:   Tom
  talent: singing
- name:   Bob
  talent: golf
...
```

Input

The llvm::yaml::Input class is used to parse a YAML document into your native data structures. To instantiate an Input object you need a StringRef to the entire YAML file, and optionally a context pointer:

```
class Input : public IO {
public:
  Input(StringRef inputContent, void *context=NULL);
```

Once you have an Input object, you can use the C++ stream operator to read a document sequence, and be sure to check the Input's error() method to see if there was an error parsing. For example:

```
 using llvm::yaml::Input;

 Input yin(mb.getBuffer());

 std::vector<MyMapType>  myMapList;

 // Parse the YAML file
 yin >> myMapList;

 // Check for error
 if ( yin.error() )
   return;

// If needed, verify there was exactly one document in the file
if ( myMapList.size() != 1 )
  return;
```