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. 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 squences 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.

The model of YAML I/O is that you define your YAML schema in C++ using some types defined by llvm/Support/YAMLIO.h. Then YAML I/O will be able to write those types as YAML and parse YAML into those types.

YAML mappings are represented in C++ as a struct with one field for each possible key. In addition you must define a method name yamlMapping which binds the fields to keys. For example:

```
using llvm::yaml::YamlMap;
using llvm::yaml::IO;

struct Person : public YamlMap {
  StringRef   name;
  uint8_t     hatSize;

  void yamlMapping(IO &io) {
    requiredKey(io, name,    "name");
    optionalKey(io, hatSize, "hat-size");
  }
};
```

YAML Sequences are represented in C++ using a templated named Sequence which is a subclass of std::vector. So if your YAML documents could have a sequence of Person mapppings, that would be represented in C++ as:

```
using llvm::yaml::Sequence;

typedef Sequence<Person>  PersonList;
```

Once your schema is defined, you can programmatically build a PersonList, then 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 = "Tom";
dan.hatSize = 7;
PersonList 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::DocumentList;
using llvm::yaml::Input;

DocumentList<PersonList> docs;
Input yin(document.getBuffer());
yin >> docs;

if ( yin.error() )
  return;

// Process read document
for (unsigned i=0; i < docs.size(); ++i) {
  PersonList &pl = docs[i];
  for (unsigned j=0; j < pl(); ++j) {
    cout << "name=" << pl[j].name;
  }
}
```

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

Overall, the model of YAML I/O is to translate C++ maps and sequences to YAML and back. It is not intended to translate between your existing data structures and YAML. You will need to write C++ code to translate your native data structures into the llvm::yaml based data structures. But that is usually just copying fields.

Error Handling

When parsing a yaml document, if the input does not match your schema (as expressed in C++ YamlMap and Sequence<>). 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:

- StringRef
- int
- uint64_t
- uint32_t
- uint16_t
- uint8_t
- bool

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

Hex types

Sometimes it is more readable to print some numberic values in hexadecimal. For those cases YAML I/O defines:

- 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.

UniqueValue

YAML I/O supports translating between in-memory enumerations and a set of string values in YAML documents. This is done by supply a mapping table between a enumeration values and the strings that you'd like to be used in YAML documents. For instance suppose you had an enumeration of CPUs:

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

To support reading and writing of this enumeration, you can define a mapping table. Then bind the mapping table to the field in the YamlMap like this:

```
using llvm::yaml::UniqueValue;
using llvm::yaml::YamlMap;
using llvm::yaml::IO;

static const UniqueValue<CPUs> cpuConversion[] = {
  {cpu_x86_64,  "x86_64"},
  {cpu_x86,     "x86"},
  {cpu_PowerPC, "PowerPC"},
  {cpu_x86,      NULL}  // default cpu
};

struct Info : public YamlMap {
  CPUs        cpu;
  uint32_t    flags;

  void yamlMapping(IO &io) {
    requiredKey(io, cpu,   "cpu", cpuConversion);
    optionalKey(io, flags, "flags");
  }
};
```

The requiredKey() method has an optional parameter which is a conversion table. In this example the conversion table maps between in-memory enumeration values and string literals to be used in YAML documents.

The UniqueValue table is terminated with a NULL where the string literal normally is. The enumeration value in that entry also has a special meaning. When a key is marked optional (i.e. optionalKey() is used instead of requiredKey()), if the YAML document being read does not have that key, then then that field of the struct will be filled in with that default value.

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
};
```

To support reading and writing of this enumeration, you can define a mapping table. Then bind the mapping table to the field in the YamlMap like this:

```
using llvm::yaml::UniqueValue;
using llvm::yaml::YamlMap;
using llvm::yaml::IO;

static const BitValue<uint32_t> flagsConversion[] = {
  {flagsPointy, "pointy"},
  {flagsHollow, "hollow"},
  {flagsFlat,   "flat"},
  {flagsRound,  "round"},
  {0,           NULL}  // default flags
};

struct Info : public YamlMap {
  StringRef   name;
  uint32_t    flags;

  void yamlMapping(IO &io) {
    requiredKey(io, name,  "name");
    requiredKey(io, flags, "flags", flagsConversion);
  }
};
```

With the above, YAML I/O when writing will test mask each value in the flagsConversion table against the flags field and each that matches will cause the coresponding 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 ]
```

YamlMap

To be part of a schema that YAML I/O can use a struct must sublcass llvm::yaml::YamlMap and implement the “void yamlMapping(IO &io)” method. The yamlMapping method is a series of calls to either requiredkey() or optionalKey() which binds a YAML document key to a field in the struct.

When writing out a YAML document, the keys are written in the order that the calls to requiredkey/optionalKey are made in the yamlMapping method.

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 requiredkey/optionalKey are made in the yamlMapping 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 the mapping table used based on the cpu. This works for both reading and writing. For example:

```
using llvm::yaml::UniqueValue;
using llvm::yaml::YamlMap;
using llvm::yaml::IO;


struct Info : public YamlMap {
  CPUs        cpu;
  uint32_t    flags;

  void yamlMapping(IO &io) {
    requiredKey(io, cpu,   "cpu");
    requiredKey(io, flags, "flags", getConverter()); // must be after cpu
  }

  // Supply cpu specific converstion table
  const BitValue<uint32_t> getConverter() {
    switch ( cpu ) {
      case x86_64:
        return flagsConversionx86_64;
      case x86:
        return flagsConversionx86;
    }
  }
};
```

Sequence

An llvm::yaml::Sequence<XX> is just a std::vector<XX> with extra functionality. You can build up a sequence using push_back() and iterate through a sequence using standard iteration techniques.

```
using llvm::yaml::Sequence;

typedef Sequence<MyType>   MyTypeList;

MyType element = ...;
MyTypeList list;
list.push_back(element);

// C++11
for(MyType &elem : list) {
  // ...
}
```

Flow Sequence

An llvm::yaml::FlowSequence is just like a Sequence except that it is written and read as a YAML flow sequence (e.g [foo, bar]).

Document List

YAML allows you to define multiple documents in one file/stream. The beginning of a new document is denoted with “—”. So in order for Input to handle multiple documents, it operators on an llvm::yaml::Document<>.

Note: This assumes homogenous documents.

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 table 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 yamlMapping() method and retrive those values in the inner map’s yamlMapping() method.

Output

The llvm::yaml::Output class is used to generate a YAML document from an instance of in-memory YamlMaps and Sequences. 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 to write a DocumentList, YamlMap, or Sequence as YAML, but only the DocumentList will write out the initial “—” and trailing ”…”.

```
using llvm::yaml::Output;
using llvm::yaml::Sequence;
using llvm::yaml::DocumentList;

Output yout(llvm::outs());

Person                            p;
Sequence<Person>                  people;
DocumentList<Sequence<Person>>   allPeople;
...

// Legal to write out just person, but --- and ... are missing
yout << p;

// Legal to write out list of persons, but --- and ... are missing
yout << people;

// Legal to write out list of persons
yout << allPeople;
```

Input

The llvm::yaml::Input class is used to parse a YAML document into an instance of in-memory YamlMaps and Sequences. 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 DocumentList and the Input’s error() method to check if there was an error parsing. For example:

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

Input yin(mb.getBuffer());

DocumentList<Sequence<Person>>  allPeople;

// Read list of persons
yin >> allPeople;

// check for error
if ( yin.error() )
  return;
```