session 15 - Serialization Formats

There are a couple of formats which we need to learn how to read and write so that we can be productive. The big ones
for us are Yaml, Json, Xml. Lets start with YAML.

YAML

Yaml is a markup language which we have been using for quite some time. It's "clever" title stands for "YAML Ain't
Markup Language". It's spec may be found at yaml.org. In order to read and write yaml, we are going
to use a popular library - yaml-cpp. So, open your favorite browser, and go over to the yaml-cpp project on github, because the first step is going to be getting the library.

Downloading and Building yaml-cpp

In order to use yaml-cpp, we need to pull it down. When you go to the url, you will notice a couple of
things. First, this is not a header only library, which means we have to build and install it somewhere. Second,
its last major tagged release is dependent upon BOOST. Now, we love boost, but we don't want the hassle of dealing
with a boost dependency if we don't need to. Fortunately, the trunk code has attempted to excise boost. It just needs
some additional testing. Well, that's what we are going to do. So, click on the *clone or download* button and do
as it says. Navigate to the place you want to run the build from (I do it it ~/src on my machine) in a shell and
type the following:

```
git clone https://github.com/jbeder/yaml-cpp.git
```

Now, follow the directions on github for building it. Navigate into the project, create a *build* directory, and go
into it. Then run `cmake ..` with appropriate flags to build the library. You might be wondering what those
appropriate flags are. Well, there are at least two that I can thing of:

If we want to build a shared library, we need to specify -DBUILD_SHARED_LIBS=ON . Otherwise, we will build a static
library (which is fine by the way).

If we want to control where the install directive puts the results of the build (assuming we are not copying files
out by hand or we are just relying on the default pathing), we need to use the -DCMAKE_INSTALL_PREFIX=.
This is going to prepend the provided path to the location of the install. So, if the project in question normally
installs to */usr/local/bin*, and you use `-DCMAKE_INSTALL_PREFIX=/home/jlgerber` (or better yet your own
home directory instead of mine), then you will end up installing to */home/jlgerber/usr/local/bin*. It is important
to note this, as you will need to subsequently USE these paths to configure our upcoming project. Also, if you are on
windows, this **ISN'T** going to work, due to those pesky drive letters. Anyway, I am going to run the following:

```
cmake .. -DCMAKE_INSTALL_PREFIX=/home/<your homedir name>
make
make install
```

If you installed to your home directory, please note the addition of two subdirectories - *include* for all of the
headers, and *lib/* where it puts your libyaml-cpp.a file.

While you are in the build directory, cd into the *test* subdirectory and run the provided tests. You might as well,
they took longer to build than the actual library.

```
cd test
./run-tests
```

Setting up a simple yaml file for reading

Before we can read a yaml file, we need one. Lets create a dummy file to go over the fun things we can do with
YAML. Create a books.yaml somewhere with the following contents:

```
---
-
    name: Remembrance of Things Past
    author: Marcel Proust
    opening:
        For a long time I used to go to bed early. Sometimes, when I had put out my candle,
        quickly that I had not even time to say I'm going to sleep.
    cost: 36.95
```

```
-
    name: Look Homeward Angel
    author: Thomas Wolfe
    opening:
        A destiny that leads the English to the Dutch is strange enough; but one that leads
        and thence into the hills that shut Altamont over the proud coral cry of the cock, a
        an angel, is touched by the dark miracle of chance which makes new magic in a dusty
    cost: 15.00
-
    name: Clia
    author: Lawrence Durrell
    opening:
        The oranges were more plentiful than usual that year. They glowed in their arbours c
        lanterns, flickering up there among the sunny woods.
    cost: 4.99
-
    name: Hunger
    author: Knut Hamsun
    opening:
        All of this happened while I was walking around starving in Christiania -- that stra
        until it has left its mark on him....
    cost: 10.00
-
    name: Speak, Memory
    author: Vladimir Nabokov
    opening:
      The cradle rocks above an abyss, and common sense tells us that our existence is but a
      two eternities of darkness. Although the two are identical twins, man, as a rule, view
      more calm than the one he is heading for.
    cost: 12.00
```

So now that you (a) know a least four books sitting on my shelf, and (b) have a suitably complex yaml example, lets
learn how to deserialize yaml.

Reading Yaml

Time to create a new project. Remember to configure the include path and library path to look at the yaml library we
just installed. If you are using cmake, your CMakeLists.txt will look something like this:

```
cmake_minimum_required(VERSION 3.2)

project(ReadingYaml)
```

```
set(CMAKE_CXX_STANDARD 11)

include_directories( /home/jlgerber/include )

link_directories( /home/jlgerber/lib )

file(GLOB cpps src/*.cpp)
file(GLOB hpps src/*.hpp)

add_executable( read-yaml ${cpps} ${hpps})
target_link_library( read-yaml yaml-cpp)
```

Now, create your main function and lets get down to brass tacks.

```
#include <iostream>
#include <cassert>

#include "yaml-cpp/yaml.h"

int main() {
    readYaml();
    return 0;
}
```

Reading from a String Ok, lets ease into reading some yaml. Before we tackle the file above, we are going to get our feet wet with some basics.
First, Yaml is stored in memory as a tree of YAML::Nodes. Each document has a root node, and child nodes. Let's create a
sequence:

```
void readYaml() {
    YAML::Node node = YAML::Load("[1, 2, 3]");
    assert(node.Type() == YAML::NodeType::Sequence);
    assert(node.IsSequence()); // a shortcut to the code above
}
```

Sequences and Maps are contained in special Collection nodes, which act a bit like STL vectors amd maps. In our example
above, we can iterate over the sequence one of two ways:

```
for(std::size_t i=0; i < node.size(); i++) {
    std::cout << node[i].as<int>() << "\n";
}
```

Or using iterators:

```
for(YAML::const_iterator it=node.begin(); it != node.end(); ++it) {
    std::cout << it->as<int>() << "\n";
}
```

In either case, we have to fetch the contents of the child, and we have to provide type information when we do so. This
is handled by the `as` template function.

Reading from a File We are going to read that yaml file from above, starting with a blank readYaml function, and filling out out slowly.

```
void readYamlFile() {

}
```

Ok, well lets load the file. We can do this using the `YAML::LoadFile function`

```
YAML::Node books_root = YAML::LoadFile("../../chapter_15/books.yaml");
```

Ok, we now have a YAML::Node. Let's check to make sure it is what we think it is. Looking at our document, the top node
should be a sequence type.

```
assert(books_root.IsSequence());
```

Great, now we have the top node of a yaml file. We can iterate through it using YAML::const_iterator.
Remember, we are expecting a sequence of maps. Both sequences and maps can be accessed via iterators,
so this should be simple.

```
for(YAML::const_iterator i = boos_root.begin(); i != books_root.end(); ++i) {
    for(YAML::const_iterator mit = i->begin(); mit != i->end(); ++mit) {
        std::cout << "key: " << mit->first.as<std::string>() << " value: "
        << mit->second.as<std::string>() << std::endl;
    }
    std::cout << std::endl;
}
```

We can also access values using bracket notation. As a bonus, accessing non-extant values does not
raise an exception. In fact, a pretty nice pattern is as follows:

```
if(books_root[0] && books_root[0]["author"])
    std::cout << books_root[0]["author}] << std::endl;
```

What Type do we have Here? As mentioned above, there are a couple ways of introspecting node type. The first is by using the
method `type()` and testing against `YAML::NodeType`, which provides a set of enums which
are appropriate fodder for switch statements, and the like.

Additionally, yaml-cpp provides a number of Is* methods (IsNull, IsSequence, IsMap, etc) which are
more convenient than calling Type().

Emitting Yaml

Of course, it would be nice if we could actually emit yaml as well eh? Well, this is pretty simple
too. Yaml-cpp implements a stream style operator for us to use.

No matter what data type we want to emit we first need to create an emitter.

```
YAML::Emitter out;
```

Once created, we can use it like any other stream instance (more or less).

Scalars The simplest type of data we can encode is a scalar. We do this trivially, once we have an emitter:

```
out << "Hello, World!";
```

We can always convert to a c string by calling ‘`c_str()`:

```
std::cout << out.c_str() << std::endl;
```

Sequences Yaml-cpp has special stream manipulators to indicate beginning and ending of sequences. You begin
outputting a sequence using `YAML::BeginSeq` and end it using `YAML::EndSeq`. Any output
between these two manipulators is treated as elements of the sequence.

```
out << YAML::BeginSeq;
out << "eggs";
out << "bread";
out << "milk" ;
out << "cheese";
out << YAML::EndSeq;
```

And of course, you can nest sequences, as long as you balance BeginSeq and EndSeq.

Maps Emitting maps is nearly as simple as emitting sequences. Like sequences, maps provide a begin and
end manipulator to delineate it. Additionally, yaml-cpp provides a Key and Value stream manipulator
to encode key and value:

```
out << YAML::BeginMap;
out << YAML::Key << "author";
out << YAML::Value << "Haruki Mirukami";
out << YAML::Key << "name";
out << YAML::Value << "South of the Border, West of the Sun";
out << YAML::EndMap;
```

Additional Manipulators

Literal (|) You can use `YAML::Literal` to emit a literal string:

```
out << YAML::Literal << "A\n B\n  C"
```

Flow You can also produce more compact map and sequence output by using the `YAML::FLow` manipulator.

```
out << YAML::FLow;
out << YAML::BeginSeq << 2 << 3 << 4 << 5 << YAML::EndSeq;
```

Comments You can embed comments into the document using the `YAML::Comment` manipulator like so:

```
YAML::Emitter out;
out << YAML::BeginMap;
out << YAML::Key << "author";
```

```
out << YAML::Value << "Henry Miller";
out << YAML::Key << "name";
out << YAML::Value << "The Air-Conditioned Nightmare";
out << YAML::Comment("An oft overlooked Miller Novel");
out << YAML::EndMap;
```

Aliases and Anchors Yaml has the ability to name a section and refer to it later in the document. yaml-cpp supports this through the
Achor and Alias tags.

```
YAML::Emitter out;
out << YAML::BeginSeq;
out << YAML::Anchor("fred");
out << YAML::BeginMap;
out << YAML::Key << "name" << YAML::Value << "Fred";
out << YAML::Key << "age" << YAML::Value << "42";
out << YAML::EndMap;
out << YAML::Alias("fred");
out << YAML::EndSeq;
```

Manipulator Lifetimes Manipulators affect the **next** output item in the stream. If that item is a `BeginSeq` or a
`BeginMap`, the manipulator lasts until the corresponding `EndSeq` or `EndMap`. Of course,
nesting works here as well.

You can perminaently chagne a setting by using a global setter. There are setters corresponding to each
manipulator. EG:

```
YAML::Emitter out;
out.SetIndent(8);
out.SetMapFormat(YAML::Flow);
...
out.SetSeqFormat(YAML::FLow);
...
```

Overloaded Conveniences Yaml-cpp overloads the `operator <<` for `std::vector`, `std::list`, and `std::map`,
allowing us to do things like this:

```
std::vector <int> squares;
squares.push_back(1);
```

```
squares.push_back(4);
squares.push_back(9);
squares.push_back(16);

std::map <std::string, int> ages;
ages["Daniel"] = 26;
ages["Jesse"] = 24;

YAML::Emitter out;
out << YAML::BeginSeq;
out << YAML::Flow << squares;
out << YAML::Flow << ages;
out << YAML::EndSeq;
```

Custom Overloading You can support custom data types for encoding and decoding as long as they implement `operator==`.

You accomplish this through template specialization. For example, say we have the following Vec3
struct:

```
struct Vec3 {
    double m_x, m_y, m_z;

     Vec3() : m_x{0}, m_y{0}, m_z{0} {};

     Vec3(double x, double y, double z) : m_x{x}, m_y{y}, m_z{z} {};

    bool operator==(const Vec3& lhs) const {
        return m_x == lhs.m_x && m_y == lhs.m_y && m_z == lhs.m_z;
    }

    // although not necessary, if we want to use the Emitter, we also have to implement oper
    Emitter& operator<<(Emitter& out, const Vec3 &v) {
            out << YAML::Flow << YAML::BeginSeq  << v.m_x << v.m_y << v.m_z << YAML::EndSeq;
            return out;
    }
};
```

We can provide support with the following template specialization:

```
namespace YAML {
template<>
struct convert<Vec3> {
    static Node encode(const Vec3& rhs) {
```

```
        Node node;
        node.push_back(rhs.m_x);
        node.push_back(rhs.m_y);
        node.push_back(rhs.m_z);
        return node;
    }

    static bool decode(const Node &node, Vec3 &rhs) {
        if(!node.IsSequence() || node.size() != 3) {
            return false;
        }

        rhs.m_x = node[0].as<double>();
        rhs.m_y = node[1].as<double>();
        rhs.m_z = node[2].as<double>();
        return true;
    }
};

}
```

Now we should be able to use Vec3 anywhere we want:

First reading.

```
YAML::Node node = YAML::Load("start: [1, 3, 0]");
Vec3 v = node["start"].as<Vec3>();
```

Then writing.

```
YAML::Emitter out;
out << YAML::BeginMap;
out << YAML::Key << "start";
out << YAML::Value << Vec3(1, -2, 0);
out << YAML::EndMap;
```

Stream State - Detecting Errors If you happen to screw up the stream (like if you forget a YAML::EndSeq, or misplace a YAML::Key),
then yaml-cpp will set an error flag on the Emitter. You can check the state using the `good()`
method.

If the Emitter's state is not good, then you can outptu the last known error using the `GetLastError()`
method.

```
YAML::Emitter out;
assert(out.good());
out << YAML::Key;
assert(!out.good());
std::cout << "Emitter error: " << out.GetLastError() << "\n";
```