Tutorial

Overview

This tutorial provides a basic example of how to work with FlatBuffers. We will step through a simple example application, which shows you how to:

- Write a FlatBuffer `schema` file.
- Use the `flatc` FlatBuffer compiler.
- Parse JSON files that conform to a schema into FlatBuffer binary files.
- Use the generated files in many of the supported languages (such as C++, Java, and more.)

During this example, imagine that you are creating a game where the main character, the hero of the story, needs to slay some `orc` s. We will walk through each step necessary to create this monster type using FlatBuffers.

Please select your desired language for our quest:

◉C++ ○Java ○Kotlin ○C# ○Go ○Python ○JavaScript ○TypeScript ○PHP ○C ○Dart ○Lua ○Lobster ○Rust ○Swift

Where to Find the Example Code

Samples demonstating the concepts in this example are located in the source code package, under the `samples` directory. You can browse the samples on GitHub here.

For your chosen language, please cross-reference with:

sample_binary.cpp

Writing the Monsters' FlatBuffer Schema

To start working with FlatBuffers, you first need to create a `schema` file, which defines the format for each data structure you wish to serialize. Here is the `schema` that defines the template for our monsters:

```
// Example IDL file for our monster's schema.
```

```
namespace MyGame.Sample;

enum Color:byte { Red = 0, Green, Blue = 2 }

union Equipment { Weapon } // Optionally add more tables.

struct Vec3 {
  x:float;
  y:float;
  z:float;
}

table Monster {
  pos:Vec3; // Struct.
  mana:short = 150;
  hp:short = 100;
  name:string;
  friendly:bool = false (deprecated);
  inventory:[ubyte];  // Vector of scalars.
  color:Color = Blue; // Enum.
  weapons:[Weapon];   // Vector of tables.
  equipped:Equipment; // Union.
  path:[Vec3];        // Vector of structs.
}

table Weapon {
  name:string;
  damage:short;
}

root_type Monster;
```

As you can see, the syntax for the `schema` Interface Definition Language (IDL) is similar to those of the C family of languages, and other IDL languages. Let's examine each part of this `schema` to determine what it does.

The `schema` starts with a `namespace` declaration. This determines the corresponding package/namespace for the generated code. In our example, we have the `Sample` namespace inside of the `MyGame` namespace.

Next, we have an `enum` definition. In this example, we have an `enum` of type `byte`, named `Color`. We have three values in this `enum`: `Red`, `Green`, and `Blue`. We specify `Red = 0` and `Blue = 2`, but we do not specify an explicit value for `Green`. Since the behavior of an `enum` is to increment if unspecified, `Green` will receive the implicit value of `1`.

Following the `enum` is a `union`. The `union` in this example is not very useful, as it only contains the one `table` (named `Weapon`). If we had created multiple tables that we would want the `union` to be able to reference, we could add more elements to the `union Equipment`.

After the `union` comes a `struct Vec3`, which represents a floating point vector with `3` dimensions. We use a `struct` here, over a `table`, because `struct`s are ideal for data structures that will not change, since they use less memory and have faster lookup.

The `Monster` table is the main object in our FlatBuffer. This will be used as the template to store our `orc` monster. We specify some default values for fields, such as `mana:short = 150`. All unspecified fields will default to `0` or `NULL`. Another thing to note is the line `friendly:bool = false (deprecated);`. Since you cannot delete fields from a `table` (to support backwards compatability), you can set fields as `deprecated`, which will prevent the generation of accessors for this field in the generated code. Be careful when using `deprecated`, however, as it may break legacy code that used this accessor.

The `Weapon` table is a sub-table used within our FlatBuffer. It is used twice: once within the `Monster` table and once within the `Equipment` enum. For our `Monster`, it is used to populate a `vector of tables` via the `weapons` field within our `Monster`. It is also the only table referenced by the `Equipment` union.

The last part of the `schema` is the `root_type`. The root type declares what will be the root table for the serialized data. In our case, the root type is our `Monster` table.

The scalar types can also use alias type names such as `int16` instead of `short` and `float32` instead of `float`. Thus we could also write the `Weapon` table as:

table Weapon { name:string; damage:int16; }

More Information About Schemas

You can find a complete guide to writing `schema` files in the Writing a schema section of the Programmer's Guide. You can also view the formal Grammar of the schema language.

Compiling the Monsters' Schema

After you have written the FlatBuffers schema, the next step is to compile it.

If you have not already done so, please follow these instructions to build `flatc`, the FlatBuffer compiler.

Once `flatc` is built successfully, compile the schema for your language:

```
cd flatbuffers/samples
./../flatc --cpp monster.fbs
```

For a more complete guide to using the `flatc` compiler, please read the Using the schema compiler section of the Programmer's Guide.

Reading and Writing Monster FlatBuffers

Now that we have compiled the schema for our programming language, we can start creating some monsters and serializing/deserializing them from FlatBuffers.

Creating and Writing Orc FlatBuffers

The first step is to import/include the library, generated files, etc.

```
#include "monster_generated.h" // This was generated by `flatc`.

using namespace MyGame::Sample; // Specified in the schema.
```

Now we are ready to start building some buffers. In order to start, we need to create an instance of the `FlatBufferBuilder`, which will contain the buffer as it grows. You can pass an initial size of the buffer (here 1024 bytes), which will grow automatically if needed:

```
// Create a `FlatBufferBuilder`, which will be used to create our
// monsters' FlatBuffers.
flatbuffers::FlatBufferBuilder builder(1024);
```

After creating the `builder`, we can start serializing our data. Before we make our `orc` Monster, lets create some `Weapon` s: a `Sword` and an `Axe`.

```
auto weapon_one_name = builder.CreateString("Sword");
short weapon_one_damage = 3;

auto weapon_two_name = builder.CreateString("Axe");
short weapon_two_damage = 5;

// Use the `CreateWeapon` shortcut to create Weapons with all the fields set.
auto sword = CreateWeapon(builder, weapon_one_name, weapon_one_damage);
auto axe = CreateWeapon(builder, weapon_two_name, weapon_two_damage);
```

Now let's create our monster, the `orc`. For this `orc`, lets make him `red` with rage, positioned at `(1.0, 2.0, 3.0)`, and give him a large pool of hit points with `300`. We can give him a vector of weapons to choose from (our `Sword` and `Axe` from earlier). In this case, we will equip him with the `Axe`, since it is the most powerful of the two. Lastly, let's fill his inventory with some potential treasures that can be taken once he is defeated.

Before we serialize a monster, we need to first serialize any objects that are contained there-in, i.e. we serialize the data tree using depth-first, pre-order traversal. This is generally easy to do on any tree structures.

```
// Serialize a name for our monster, called "Orc".
auto name = builder.CreateString("Orc");

// Create a `vector` representing the inventory of the Orc. Each number
// could correspond to an item that can be claimed after he is slain.
unsigned char treasure[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
auto inventory = builder.CreateVector(treasure, 10);
```

We serialized two built-in data types (`string` and `vector`) and captured their return values. These values are offsets into the serialized data, indicating where they are stored, such that we can refer to them below when adding fields to our monster.

Note: To create a `vector` of nested objects (e.g. `table`s, `string`s, or other `vector`s), collect their offsets into a temporary data structure, and then create an additional `vector` containing their offsets.

If instead of creating a vector from an existing array you serialize elements individually one by one, take care to note that this happens in reverse order, as buffers are built back to front.

For example, take a look at the two `Weapon`s that we created earlier (`Sword` and `Axe`). These are both FlatBuffer `table`s, whose offsets we now store in memory. Therefore we can create a FlatBuffer `vector` to contain these offsets.

```
// Place the weapons into a `std::vector`, then convert that into a
      FlatBuffer `vector`.
std::vector<flatbuffers::Offset<Weapon>> weapons_vector;
weapons_vector.push_back(sword);
weapons_vector.push_back(axe);
auto weapons = builder.CreateVector(weapons_vector);
```

Note there's additional convenience overloads of `CreateVector`, allowing you to work with data that's not in a `std::vector`, or allowing you to generate elements by calling a lambda. For the common case of `std::vector<std::string>` there's also `CreateVectorOfStrings`.

Note that vectors of structs are serialized differently from tables, since structs are stored in-line in the vector. For example, to create a vector for the `path` field above:

```
Vec3 points[] = { Vec3(1.0f, 2.0f, 3.0f), Vec3(4.0f, 5.0f, 6.0f) };
auto path = builder.CreateVectorOfStructs(points, 2);
```

We have now serialized the non-scalar components of the orc, so we can serialize the monster itself:

```
// Create the position struct
auto position = Vec3(1.0f, 2.0f, 3.0f);

// Set his hit points to 300 and his mana to 150.
int hp = 300;
int mana = 150;

// Finally, create the monster using the `CreateMonster` helper function
// to set all fields.
auto orc = CreateMonster(builder, &position, mana, hp, name, inventory,
                        Color_Red, weapons, Equipment_Weapon, axe.Union(),
                        path);
```

Note how we create `Vec3` struct in-line in the table. Unlike tables, structs are simple combinations of scalars that are always stored inline, just like scalars themselves.

Important: Unlike structs, you should not nest tables or other objects, which is why we created all the strings/vectors/tables that this monster refers to before `start`. If you try to create any of them between `start` and `end`, you will get an assert/exception/panic depending on your language.

Note: Since we are passing `150` as the `mana` field, which happens to be the default value, the field will not actually be written to the buffer, since the default value will be returned on query anyway. This is a nice space savings, especially if default values are common in your data. It also means that you do not need to be worried of adding a lot of fields that are only used in a small number of instances, as it will not bloat the buffer if unused.

If you do not wish to set every field in a `table`, it may be more convenient to manually set each field of your monster, instead of calling `CreateMonster()`. The following snippet is functionally equivalent to the above code, but provides a bit more flexibility.

```
// You can use this code instead of `CreateMonster()`, to create our orc
// manually.
MonsterBuilder monster_builder(builder);
monster_builder.add_pos(&position);
```

```
monster_builder.add_hp(hp);
monster_builder.add_name(name);
monster_builder.add_inventory(inventory);
monster_builder.add_color(Color_Red);
monster_builder.add_weapons(weapons);
monster_builder.add_equipped_type(Equipment_Weapon);
monster_builder.add_equipped(axe.Union());
auto orc = monster_builder.Finish();
```

Before finishing the serialization, let's take a quick look at FlatBuffer `union Equipped`. There are two parts to each FlatBuffer `union`. The first, is a hidden field `_type`, that is generated to hold the type of `table` referred to by the `union`. This allows you to know which type to cast to at runtime. Second, is the `union`'s data.

In our example, the last two things we added to our `Monster` were the `Equipped Type` and the `Equipped` union itself.

Here is a repetition these lines, to help highlight them more clearly:

```
monster_builder.add_equipped_type(Equipment_Weapon); // Union type
monster_builder.add_equipped(axe); // Union data
```

After you have created your buffer, you will have the offset to the root of the data in the `orc` variable, so you can finish the buffer by calling the appropriate `finish` method.

```
// Call `Finish()` to instruct the builder that this monster is complete.
// Note: Regardless of how you created the `orc`, you still need to call
// `Finish()` on the `FlatBufferBuilder`.
builder.Finish(orc); // You could also call `FinishMonsterBuffer(builder,
                     //                                          orc);`.
```

The buffer is now ready to be stored somewhere, sent over the network, be compressed, or whatever you'd like to do with it. You can access the buffer like so:

```
// This must be called after `Finish()`.
uint8_t *buf = builder.GetBufferPointer();
int size = builder.GetSize(); // Returns the size of the buffer that
                              // `GetBufferPointer()` points to.
```

Now you can write the bytes to a file, send them over the network.. **Make sure your file mode (or transfer protocol) is set to BINARY, not text.** If you transfer a FlatBuffer in text mode, the buffer will be corrupted, which will lead to hard to find problems when you read the buffer.

Reading Orc FlatBuffers

Now that we have successfully created an `Orc` FlatBuffer, the monster data can be saved, sent over a network, etc. Let's now adventure into the inverse, and access a FlatBuffer.

This section requires the same import/include, namespace, etc. requirements as before:

```
#include "monster_generated.h" // This was generated by `flatc`.

using namespace MyGame::Sample; // Specified in the schema.
```

Then, assuming you have a buffer of bytes received from disk, network, etc., you can create start accessing the buffer like so:

Again, make sure you read the bytes in BINARY mode, otherwise the code below won't work

```
uint8_t *buffer_pointer = /* the data you just read */;

// Get a pointer to the root object inside the buffer.
auto monster = GetMonster(buffer_pointer);

// `monster` is of type `Monster *`.
// Note: root object pointers are NOT the same as `buffer_pointer`.
// `GetMonster` is a convenience function that calls `GetRoot<Monster>`,
// the latter is also available for non-root types.
```

If you look in the generated files from the schema compiler, you will see it generated accessors for all non-`deprecated` fields. For example:

```
auto hp = monster->hp();
auto mana = monster->mana();
auto name = monster->name()->c_str();
```

These should hold `300`, `150`, and `"Orc"` respectively.

Note: The default value `150` *wasn't stored in* `mana`*, but we are still able to retrieve it.*

To access sub-objects, in the case of our `pos`, which is a `Vec3`:

```
auto pos = monster->pos();
auto x = pos->x();
auto y = pos->y();
auto z = pos->z();
```

`x`, `y`, and `z` will contain `1.0`, `2.0`, and `3.0`, respectively.

Note: Had we not set `pos` *during serialization, it would be a* `NULL` *-value.*

Similarly, we can access elements of the inventory `vector` by indexing it. You can also iterate over the length of the array/vector representing the FlatBuffers `vector`.

```
auto inv = monster->inventory(); // A pointer to a `flatbuffers::Vector<>`.
auto inv_len = inv->size();
auto third_item = inv->Get(2);
```

For `vector` s of `table` s, you can access the elements like any other vector, except your need to handle the result as a FlatBuffer `table`:

```
auto weapons = monster->weapons(); // A pointer to a `flatbuffers::Vector<>`.
auto weapon_len = weapons->size();
auto second_weapon_name = weapons->Get(1)->name()->str();
auto second_weapon_damage = weapons->Get(1)->damage()
```

Last, we can access our `Equipped` FlatBuffer `union`. Just like when we created the `union`, we need to get both parts of the `union`: the type and the data.

We can access the type to dynamically cast the data as needed (since the `union` only stores a FlatBuffer `table`).

```
auto union_type = monster.equipped_type();

if (union_type == Equipment_Weapon) {
  auto weapon = static_cast<const Weapon*>(monster->equipped()); // Requires
      `static_cast`
                                                                 // to type
      `const Weapon*`.

  auto weapon_name = weapon->name()->str(); // "Axe"
  auto weapon_damage = weapon->damage();    // 5
}
```

Mutating FlatBuffers

As you saw above, typically once you have created a FlatBuffer, it is read-only from that moment on. There are, however, cases where you have just received a FlatBuffer, and you'd like to modify something about it before sending it on to another recipient. With the above functionality, you'd have to generate an entirely new FlatBuffer, while tracking what you modified in your own data structures. This is inconvenient.

For this reason FlatBuffers can also be mutated in-place. While this is great for making small fixes to an existing buffer, you generally want to create buffers from scratch whenever possible, since it is much more efficient and the API is much more general purpose.

To get non-const accessors, invoke `flatc` with `--gen-mutable`.

Similar to how we read fields using the accessors above, we can now use the mutators like so:

```
auto monster = GetMutableMonster(buffer_pointer);  // non-const
monster->mutate_hp(10);                            // Set the table `hp` field.
monster->mutable_pos()->mutate_z(4);               // Set struct field.
monster->mutable_inventory()->Mutate(0, 1);        // Set vector element.
```

We use the somewhat verbose term `mutate` instead of `set` to indicate that this is a special use case, not to be confused with the default way of constructing FlatBuffer data.

After the above mutations, you can send on the FlatBuffer to a new recipient without any further work!

Note that any `mutate` functions on a table will return a boolean, which is `false` if the field we're trying to set is not present in the buffer. Fields that are not present if they weren't set, or even if they happen to be equal to the default value. For example, in the creation code above, the `mana` field is equal to `150`, which is the default value, so it was never stored in the buffer. Trying to call the corresponding `mutate` method for `mana` on such data will return `false`, and the value won't actually be modified!

One way to solve this is to call `ForceDefaults` on a FlatBufferBuilder to force all fields you set to actually be written. This, of course, increases the size of the buffer somewhat, but this may be acceptable for a mutable buffer.

If this is not sufficient, other ways of mutating FlatBuffers may be supported in your language through an object based API (`--gen-object-api`) or reflection. See the individual language documents for support.

Using `flatc` as a JSON Conversion Tool

If you are working with C, C++, or Lobster, you can parse JSON at runtime. If your language does not support JSON at the moment, `flatc` may provide an alternative. Using `flatc` is often the preferred method, as it doesn't require you to add any new code to your program. It is also efficient, since you can ship with the binary data. The drawback is that it requires an extra step for your users/developers to perform (although it may be able to be automated as part of your compilation).

JSON to binary representation

Lets say you have a JSON file that describes your monster. In this example, we will use the file `flatbuffers/samples/monsterdata.json`.

Here are the contents of the file:

```
{
  pos: {
    x: 1.0,
    y: 2.0,
    z: 3.0
  },
  hp: 300,
  name: "Orc",
  weapons: [
    {
      name: "axe",
      damage: 100
    },
    {
      name: "bow",
      damage: 90
    }
  ],
  equipped_type: "Weapon",
  equipped: {
    name: "bow",
    damage: 90
  }
}
```

You can run this file through the `flatc` compiler with the `-b` flag and our `monster.fbs` schema to produce a FlatBuffer binary file.

```
./../flatc -b monster.fbs monsterdata.json
```

The output of this will be a file `monsterdata.bin`, which will contain the FlatBuffer binary representation of the contents from our `.json` file.

Note: If you're working in C++, you can also parse JSON at runtime. See the Use in C++ section of the Programmer's Guide for more information.

FlatBuffer binary to JSON

Converting from a FlatBuffer binary representation to JSON is supported as well:

```
./../flatc --json --raw-binary monster.fbs -- monsterdata.bin
```

This will convert `monsterdata.bin` back to its original JSON representation. You need to pass the corresponding FlatBuffers schema so that flatc knows how to interpret the binary buffer. Since `monster.fbs` does not specify an explicit `file_identifier` for binary buffers, `flatc` needs to be forced into reading the `.bin` file using the `--raw-binary` option.

The FlatBuffer binary representation does not explicitly encode default values, therefore they are not present in the resulting JSON unless you specify `--defaults-json`.

If you intend to process the JSON with other tools, you may consider switching on `--strict-json` so that identifiers are quoted properly.

*Note: The resulting JSON file is not necessarily identical with the original JSON. If the binary representation contains floating point numbers, floats and doubles are rounded to 6 and 12 digits, respectively, in order to represent them as decimals in the JSON document. *

Advanced Features for Each Language

Each language has a dedicated `Use in XXX` page in the Programmer's Guide to cover the nuances of FlatBuffers in that language.

For your chosen language, see:

Use in C++