Use in C++

Before you get started

Before diving into the FlatBuffers usage in C++, it should be noted that the Tutorial page has a complete guide to general FlatBuffers usage in all of the supported languages (including C++). This page is designed to cover the nuances of FlatBuffers usage, specific to C++.

Prerequisites

This page assumes you have written a FlatBuffers schema and compiled it with the Schema Compiler. If you have not, please see Using the schema compiler and Writing a schema.

Assuming you wrote a schema, say `mygame.fbs` (though the extension doesn't matter), you've generated a C++ header called `mygame_generated.h` using the compiler (e.g. `flatc -c mygame.fbs`), you can now start using this in your program by including the header. As noted, this header relies on `flatbuffers/flatbuffers.h`, which should be in your include path.

FlatBuffers C++ library code location

The code for the FlatBuffers C++ library can be found at `flatbuffers/include/flatbuffers`. You can browse the library code on the FlatBuffers GitHub page.

Testing the FlatBuffers C++ library

The code to test the C++ library can be found at `flatbuffers/tests`. The test code itself is located in test.cpp.

This test file is built alongside `flatc`. To review how to build the project, please read the Building documentation.

To run the tests, execute `flattests` from the root `flatbuffers/` directory. For example, on Linux, you would simply run: `./flattests`.

Using the FlatBuffers C++ library

Note: See Tutorial for a more in-depth example of how to use FlatBuffers in C++.

FlatBuffers supports both reading and writing FlatBuffers in C++.

To use FlatBuffers in your code, first generate the C++ classes from your schema with the `--cpp` option to `flatc`. Then you can include both FlatBuffers and the generated code to read or write FlatBuffers.

For example, here is how you would read a FlatBuffer binary file in C++: First, include the library and generated code. Then read the file into a `char *` array, which you pass to `GetMonster()`.

```
#include "flatbuffers/flatbuffers.h"
#include "monster_test_generate.h"
#include <iostream> // C++ header file for printing
#include <fstream> // C++ header file for file access


std::ifstream infile;
infile.open("monsterdata_test.mon", std::ios::binary | std::ios::in);
infile.seekg(0,std::ios::end);
int length = infile.tellg();
infile.seekg(0,std::ios::beg);
char *data = new char[length];
infile.read(data, length);
infile.close();

auto monster = GetMonster(data);
```

`monster` is of type `Monster *`, and points to somewhere *inside* your buffer (root object pointers are not the same as `buffer_pointer`!). If you look in your generated header, you'll see it has convenient accessors for all fields, e.g. `hp()`, `mana()`, etc:

```
std::cout << "hp : " << monster->hp() << std::endl;            // `80`
std::cout << "mana : " << monster->mana() << std::endl;        // default
      value of `150`
std::cout << "name : " << monster->name()->c_str() << std::endl;        //
      "MyMonster"
```

Note: That we never stored a `mana` value, so it will return the default.

The following attributes are supported:

- `shared` (on a field): For string fields, this enables the usage of string pooling (i.e. `CreateSharedString`) as default serialization behavior.

  Specifically, `CreateXxxDirect` functions and `Pack` functions for object based API (see below) will use `CreateSharedString` to create strings.

FlatBuffers is all about memory efficiency, which is why its base API is written around using as little as possible of it. This does make the API clumsier (requiring pre-order construction of all data, and making mutation harder).

For times when efficiency is less important a more convenient object based API can be used (through `--gen-object-api`) that is able to unpack & pack a FlatBuffer into objects and standard STL containers, allowing for convenient construction, access and mutation.

To use:

```
// Autogenerated class from table Monster.
MonsterT monsterobj;

// Deserialize from buffer into object.
GetMonster(flatbuffer)->UnPackTo(&monsterobj);

// Update object directly like a C++ class instance.
cout << monsterobj->name;  // This is now a std::string!
monsterobj->name = "Bob";  // Change the name.

// Serialize into new flatbuffer.
FlatBufferBuilder fbb;
fbb.Finish(Monster::Pack(fbb, &monsterobj));
```

The following attributes are specific to the object-based API code generation:

- `native_inline` (on a field): Because FlatBuffer tables and structs are optionally present in a given buffer, they are best represented as pointers (specifically std::unique_ptrs) in the native class since they can be null. This attribute changes the member declaration to use the type directly rather than wrapped in a unique_ptr.
- `native_default` : "value" (on a field): For members that are declared "native_inline", the value specified with this attribute will be included verbatim in the class constructor initializer list for this member.
- `native_custom_alloc` :"custom_allocator" (on a table or struct): When using the object-based API all generated NativeTables that are allocated when unpacking your flatbuffer will use "custom allocator". The allocator is also used by any std::vector that appears in a table defined with `native_custom_alloc`. This can be used to provide allocation from a pool for example, for faster

unpacking when using the object-based API.

Minimal Example:

schema:

table mytable(native_custom_alloc:"custom_allocator") { ... }

with custom_allocator defined before flatbuffers.h is included, as:

template <typename t=""> struct custom_allocator : public std::allocator<T> {

typedef T *pointer;

template <class u=""> struct rebind { typedef custom_allocator<U> other; };

pointer allocate(const std::size_t n) { return std::allocator<T>::allocate(n); }

void deallocate(T* ptr, std::size_t n) { return std::allocator<T>::deallocate(ptr,n); }

custom_allocator() throw() {} template <class u=""> custom_allocator(const custom_allocator<U>&) throw() {} };

- `native_type` ' "type" (on a struct): In some cases, a more optimal C++ data type exists for a given struct. For example, the following schema:

  struct Vec2 { x: float; y: float; }

  generates the following Object-Based API class:

  struct Vec2T : flatbuffers::NativeTable { float x; float y; };

  However, it can be useful to instead use a user-defined C++ type since it can provide more functionality, eg.

  struct vector2 { float x = 0, y = 0; vector2 operator+(vector2 rhs) const { ... } vector2 operator-(vector2 rhs) const { ... } float length() const { ... } // etc. };

  The `native_type` attribute will replace the usage of the generated class with the given type. So, continuing with the example, the generated code would use |vector2| in place of |Vec2T| for all generated code.

  However, becuase the native_type is unknown to flatbuffers, the user must provide the following functions to aide in the serialization process:

  namespace flatbuffers { FlatbufferStruct Pack(const native_type& obj); native_type UnPack(const FlatbufferStruct& obj); }

Finally, the following top-level attribute

- `native_include` : "path" (at file level): Because the `native_type` attribute can be used to introduce types that are unknown to flatbuffers, it may be necessary to include "external" header files in the generated code. This attribute can be used to directly add an include directive to the top of the generated code that includes the specified path directly.

- `force_align` : this attribute may not be respected in the object API, depending on the aligned of the allocator used with `new` .

External references.

An additional feature of the object API is the ability to allow you to load multiple independent FlatBuffers, and have them refer to eachothers objects using hashes which are then represented as typed pointers in the object API.

To make this work have a field in the objects you want to referred to which is using the string hashing feature (see `hash` attribute in the schema documentation). Then you have a similar hash in the field referring to it, along with a `cpp_type` attribute specifying the C++ type this will refer to (this can be any C++ type, and will get a `*` added).

Then, in JSON or however you create these buffers, make sure they use the same string (or hash).

When you call `UnPack` (or `Create`), you'll need a function that maps from hash to the object (see `resolver_function_t` for details).

Using different pointer types.

By default the object tree is built out of `std::unique_ptr` , but you can influence this either globally (using the `--cpp-ptr-type` argument to `flatc`) or per field (using the `cpp_ptr_type` attribute) to by any smart pointer type (`my_ptr<T>`), or by specifying `naked` as the type to get `T *` pointers. Unlike the smart pointers, naked pointers do not manage memory for you, so you'll have to manage their lifecycles manually. To reference the pointer type specified by the `--cpp-ptr-type` argument to `flatc` from a flatbuffer field set the `cpp_ptr_type` attribute to `default_ptr_type` .

Using different string type.

By default the object tree is built out of `std::string` , but you can influence this either globally (using the `--cpp-str-type` argument to `flatc`) or per field using the `cpp_str_type` attribute.

The type must support T::c_str(), T::length() and T::empty() as member functions.

Further, the type must be constructible from std::string, as by default a std::string instance is constructed and then used to initialize the custom string type. This behavior impedes efficient and zero-copy construction of custom string types; the `--cpp-str-flex-ctor` argument to `flatc` or the per field attribute `cpp_str_flex_ctor` can be used to change this behavior, so that the custom string type is constructed

by passing the pointer and length of the FlatBuffers String. The custom string class will require a constructor in the following format: custom_str_class(const char *, size_t). Please note that the character array is not guaranteed to be NULL terminated, you should always use the provided size to determine end of string.

Reflection (& Resizing)

There is experimental support for reflection in FlatBuffers, allowing you to read and write data even if you don't know the exact format of a buffer, and even allows you to change sizes of strings and vectors in-place.

The way this works is very elegant; there is actually a FlatBuffer schema that describes schemas (!) which you can find in `reflection/reflection.fbs`. The compiler, `flatc`, can write out any schemas it has just parsed as a binary FlatBuffer, corresponding to this meta-schema.

Loading in one of these binary schemas at runtime allows you traverse any FlatBuffer data that corresponds to it without knowing the exact format. You can query what fields are present, and then read/write them after.

For convenient field manipulation, you can include the header `flatbuffers/reflection.h` which includes both the generated code from the meta schema, as well as a lot of helper functions.

And example of usage, for the time being, can be found in `test.cpp/ReflectionTest()`.

Mini Reflection

A more limited form of reflection is available for direct inclusion in generated code, which doesn't any (binary) schema access at all. It was designed to keep the overhead of reflection as low as possible (on the order of 2-6 bytes per field added to your executable), but doesn't contain all the information the (binary) schema contains.

You add this information to your generated code by specifying `--reflect-types` (or instead `--reflect-names` if you also want field / enum names).

You can now use this information, for example to print a FlatBuffer to text:

```
auto s = flatbuffers::FlatBufferToString(flatbuf, MonsterTypeTable());
```

`MonsterTypeTable()` is declared in the generated code for each type. The string produced is very similar to the JSON produced by the `Parser` based text generator.

You'll need `flatbuffers/minireflect.h` for this functionality. In there is also a convenient visitor/iterator so you can write your own output / functionality based on the mini reflection tables without having to know the FlatBuffers or reflection encoding.

Storing maps / dictionaries in a FlatBuffer

FlatBuffers doesn't support maps natively, but there is support to emulate their behavior with vectors and binary search, which means you can have fast lookups directly from a FlatBuffer without having to unpack your data into a `std::map` or similar.

To use it:

- Designate one of the fields in a table as they "key" field. You do this by setting the `key` attribute on this field, e.g. `name:string (key)` . You may only have one key field, and it must be of string or scalar type.
- Write out tables of this type as usual, collect their offsets in an array or vector.
- Instead of `CreateVector` , call `CreateVectorOfSortedTables` , which will first sort all offsets such that the tables they refer to are sorted by the key field, then serialize it.
- Now when you're accessing the FlatBuffer, you can use `Vector::LookupByKey` instead of just `Vector::Get` to access elements of the vector, e.g.: `myvector->LookupByKey("Fred")` , which returns a pointer to the corresponding table type, or `nullptr` if not found. `LookupByKey` performs a binary search, so should have a similar speed to `std::map` , though may be faster because of better caching. `LookupByKey` only works if the vector has been sorted, it will likely not find elements if it hasn't been sorted.

Direct memory access

As you can see from the above examples, all elements in a buffer are accessed through generated accessors. This is because everything is stored in little endian format on all platforms (the accessor performs a swap operation on big endian machines), and also because the layout of things is generally not known to the user.

For structs, layout is deterministic and guaranteed to be the same across platforms (scalars are aligned to their own size, and structs themselves to their largest member), and you are allowed to access this memory directly by using `sizeof()` and `memcpy` on the pointer to a struct, or even an array of structs.

To compute offsets to sub-elements of a struct, make sure they are a structs themselves, as then you can use the pointers to figure out the offset without having to hardcode it. This is handy for use of arrays of structs with calls like `glVertexAttribPointer` in OpenGL or similar APIs.

It is important to note is that structs are still little endian on all machines, so only use tricks like this if you can guarantee you're not shipping on a big endian machine (an `assert(FLATBUFFERS_LITTLEENDIAN)` would be wise).

Access of untrusted buffers

The generated accessor functions access fields over offsets, which is very quick. These offsets are not verified at run-time, so a malformed buffer could cause a program to crash by accessing random memory.

When you're processing large amounts of data from a source you know (e.g. your own generated data on disk), this is acceptable, but when reading data from the network that can potentially have been modified by an attacker, this is undesirable.

For this reason, you can optionally use a buffer verifier before you access the data. This verifier will check all offsets, all sizes of fields, and null termination of strings to ensure that when a buffer is accessed, all reads will end up inside the buffer.

Each root type will have a verification function generated for it, e.g. for `Monster`, you can call:

```
bool ok = VerifyMonsterBuffer(Verifier(buf, len));
```

if `ok` is true, the buffer is safe to read.

Besides untrusted data, this function may be useful to call in debug mode, as extra insurance against data being corrupted somewhere along the way.

While verifying a buffer isn't "free", it is typically faster than a full traversal (since any scalar data is not actually touched), and since it may cause the buffer to be brought into cache before reading, the actual overhead may be even lower than expected.

In specialized cases where a denial of service attack is possible, the verifier has two additional constructor arguments that allow you to limit the nesting depth and total amount of tables the verifier may encounter before declaring the buffer malformed. The default is `Verifier(buf, len, 64 /* max depth */, 1000000, /* max tables */)` which should be sufficient for most uses.

Text & schema parsing

Using binary buffers with the generated header provides a super low overhead use of FlatBuffer data. There are, however, times when you want to use text formats, for example because it interacts better with source control, or you want to give your users easy access to data.

Another reason might be that you already have a lot of data in JSON format, or a tool that generates JSON, and if you can write a schema for it, this will provide you an easy way to use that data directly.

(see the schema documentation for some specifics on the JSON format accepted).

Schema evolution compatibility for the JSON format follows the same rules as the binary format (JSON

formatted data will be forwards/backwards compatible with schemas that evolve in a compatible way).

There are two ways to use text formats:

Using the compiler as a conversion tool

This is the preferred path, as it doesn't require you to add any new code to your program, and is maximally efficient since you can ship with binary data. The disadvantage is that it is an extra step for your users/developers to perform, though you might be able to automate it.

```
flatc -b myschema.fbs mydata.json
```

This will generate the binary file `mydata_wire.bin` which can be loaded as before.

Making your program capable of loading text directly

This gives you maximum flexibility. You could even opt to support both, i.e. check for both files, and regenerate the binary from text when required, otherwise just load the binary.

This option is currently only available for C++, or Java through JNI.

As mentioned in the section "Building" above, this technique requires you to link a few more files into your program, and you'll want to include `flatbuffers/idl.h`.

Load text (either a schema or json) into an in-memory buffer (there is a convenient `LoadFile()` utility function in `flatbuffers/util.h` if you wish). Construct a parser:

```
flatbuffers::Parser parser;
```

Now you can parse any number of text files in sequence:

```
parser.Parse(text_file.c_str());
```

This works similarly to how the command-line compiler works: a sequence of files parsed by the same `Parser` object allow later files to reference definitions in earlier files. Typically this means you first load a schema file (which populates `Parser` with definitions), followed by one or more JSON files.

As optional argument to `Parse`, you may specify a null-terminated list of include paths. If not specified, any include statements try to resolve from the current directory.

If there were any parsing errors, `Parse` will return `false`, and `Parser::error_` contains a human readable error string with a line number etc, which you should present to the creator of that file.

After each JSON file, the `Parser::fbb` member variable is the `FlatBufferBuilder` that contains the binary buffer version of that file, that you can access as described above.

`samples/sample_text.cpp` is a code sample showing the above operations.

Threading

Reading a FlatBuffer does not touch any memory outside the original buffer, and is entirely read-only (all const), so is safe to access from multiple threads even without synchronisation primitives.

Creating a FlatBuffer is not thread safe. All state related to building a FlatBuffer is contained in a FlatBufferBuilder instance, and no memory outside of it is touched. To make this thread safe, either do not share instances of FlatBufferBuilder between threads (recommended), or manually wrap it in synchronisation primites. There's no automatic way to accomplish this, by design, as we feel multithreaded construction of a single buffer will be rare, and synchronisation overhead would be costly.

Advanced union features

The C++ implementation currently supports vectors of unions (i.e. you can declare a field as `[T]` where `T` is a union type instead of a table type). It also supports structs and strings in unions, besides tables.

For an example of these features, see `tests/union_vector`, and `UnionVectorTest` in `test.cpp`.

Since these features haven't been ported to other languages yet, if you choose to use them, you won't be able to use these buffers in other languages (`flatc` will refuse to compile a schema that uses these features).

These features reduce the amount of "table wrapping" that was previously needed to use unions.

To use scalars, simply wrap them in a struct.

Depth limit of nested objects and stack-overflow control

The parser of Flatbuffers schema or json-files is kind of recursive parser. To avoid stack-overflow problem the parser has a built-in limiter of recursion depth. Number of nested declarations in a schema or number of nested json-objects is limited. By default, this depth limit set to `64`. It is possible to override this limit with `FLATBUFFERS_MAX_PARSING_DEPTH` definition. This definition can be helpful for testing purposes or embedded applications. For details see build of CMake-based projects.

The Flatbuffers flatbuffers grammar "grammar" uses ASCII character set for identifiers, alphanumeric literals, reserved words.

Internal implementation of the Flatbuffers depends from functions which depend from C-locale: `strtod()` or `strtof()`, for example. The library expects the dot `.` symbol as the separator of an integer part from the fractional part of a float number. Another separator symbols (`,` for example) will break the compatibility and may lead to an error while parsing a Flatbuffers schema or a json file.

The Standard C locale is a global resource, there is only one locale for the entire application. Some modern compilers and platforms have locale-independent or locale-narrow functions `strtof_l`, `strtod_l`, `strtoll_l`, `strtoull_l` to resolve this dependency. These functions use specified locale rather than the global or per-thread locale instead. They are part of POSIX-2008 but not part of the C/C++ standard library, therefore, may be missing on some platforms. The Flatbuffers library try to detect these functions at configuration and compile time:

- CMake `"CMakeLists.txt"`:
  - Check existence of `strtol_l` and `strtod_l` in the `<stdlib.h>`.
- Compile-time `"/include/base.h"`:
  - `_MSC_VER >= 1900`: MSVC2012 or higher if build with MSVC.
  - `_XOPEN_SOURCE>=700`: POSIX-2008 if build with GCC/Clang.

After detection, the definition `FLATBUFFERS_LOCALE_INDEPENDENT` will be set to `0` or `1`. To override or stop this detection use CMake `-DFLATBUFFERS_LOCALE_INDEPENDENT={0|1}` or predefine `FLATBUFFERS_LOCALE_INDEPENDENT` symbol.

To test the compatibility of the Flatbuffers library with a specific locale use the environment variable `FLATBUFFERS_TEST_LOCALE`:

```
>FLATBUFFERS_TEST_LOCALE="" ./flattests
>FLATBUFFERS_TEST_LOCALE="ru_RU.CP1251" ./flattests
```

Support of floating-point numbers

The Flatbuffers library assumes that a C++ compiler and a CPU are compatible with the `IEEE-754` floating-point standard. The schema and json parser may fail if `fast-math` or `/fp:fast` mode is active.

Support of hexadecimal and special floating-point numbers

According to the grammar `fbs` and `json` files may use hexadecimal and special (`NaN`, `Inf`) floating-point literals. The Flatbuffers uses `strtof` and `strtod` functions to parse floating-point literals. The Flatbuffers library has a code to detect a compiler compatibility with the literals. If necessary conditions are met the preprocessor constant `FLATBUFFERS_HAS_NEW_STRTOD` will be set to `1`. The support of

floating-point literals will be limited at compile time if `FLATBUFFERS_HAS_NEW_STRTOD` constant is less than `1`. In this case, schemas with hexadecimal or special literals cannot be used.

Comparison of floating-point NaN values

The floating-point `NaN` (`not a number`) is special value which representing an undefined or unrepresentable value. `NaN` may be explicitly assigned to variables, typically as a representation for missing values or may be a result of a mathematical operation. The `IEEE-754` defines two kind of `NaNs`:

- Quiet NaNs, or `qNaNs`.
- Signaling NaNs, or `sNaNs`.

According to the `IEEE-754`, a comparison with `NaN` always returns an unordered result even when compared with itself. As a result, a whole Flatbuffers object will be not equal to itself if has one or more `NaN`. Flatbuffers scalar fields that have the default value are not actually stored in the serialized data but are generated in code (see Writing a schema). Scalar fields with `NaN` defaults break this behavior. If a schema has a lot of `NaN` defaults the Flatbuffers can override the unordered comparison by the ordered: `(NaN==NaN)->true`. This ordered comparison is enabled when compiling a program with the symbol `FLATBUFFERS_NAN_DEFAULTS` defined. Additional computations added by `FLATBUFFERS_NAN_DEFAULTS` are very cheap if GCC or Clang used. These compilers have a compile-time implementation of `isnan` checking which MSVC does not.