

RapidJSON Documentation

Milo Yip

Published
with GitBook



Table of Contents

1. [Introduction](#)
2. [Features](#)
3. [Tutorial](#)
4. [Stream](#)
5. [Encoding](#)
6. [DOM](#)
7. [SAX](#)
8. [Performance](#)
9. [Internals](#)
10. [FAQ](#)

RapidJSON

release v1.0.1


A fast JSON parser/generator for C++ with both SAX/DOM style API

Tencent is pleased to support the open source community by making RapidJSON available.

Copyright (C) 2015 THL A29 Limited, a Tencent company, and Milo Yip. All rights reserved.

- [RapidJSON GitHub](#)
- [RapidJSON Documentation](#)
 - [English](#)
 - [简体中文](#)
 - [GitBook](#) with downloadable PDF/EPUB/MOBI, without API reference.

Build status

Linux	Windows	Coveralls
build passing	 BUILDING...	coverage 100%

Introduction

RapidJSON is a JSON parser and generator for C++. It was inspired by [RapidXml](#).

- RapidJSON is small but complete. It supports both SAX and DOM style API. The SAX parser is only a half thousand lines of code.
- RapidJSON is fast. Its performance can be comparable to `strlen()`. It also optionally supports SSE2/SSE4.2 for acceleration.
- RapidJSON is self-contained. It does not depend on external libraries such as BOOST. It even does not depend on STL.
- RapidJSON is memory friendly. Each JSON value occupies exactly 16/20 bytes for most 32/64-bit machines (excluding text string). By default it uses a fast memory allocator, and the parser allocates memory compactly during parsing.
- RapidJSON is Unicode friendly. It supports UTF-8, UTF-16, UTF-32 (LE & BE), and their detection, validation and transcoding internally. For example, you can read a UTF-8 file and let RapidJSON transcode the JSON strings into UTF-16 in the DOM. It also supports surrogates and "\u0000" (null character).

More features can be read [here](#).

JSON(JavaScript Object Notation) is a light-weight data exchange format. RapidJSON should be in fully compliance with RFC7159/ECMA-404. More information about JSON can be obtained at

- [Introducing JSON](#)
- [RFC7159: The JavaScript Object Notation \(JSON\) Data Interchange Format](#)

- [Standard ECMA-404: The JSON Data Interchange Format](#)

Compatibility

RapidJSON is cross-platform. Some platform/compiler combinations which have been tested are shown as follows.

- Visual C++ 2008/2010/2013 on Windows (32/64-bit)
- GNU C++ 3.8.x on Cygwin
- Clang 3.4 on Mac OS X (32/64-bit) and iOS
- Clang 3.4 on Android NDK

Users can build and run the unit tests on their platform/compiler.

Installation

RapidJSON is a header-only C++ library. Just copy the `include/rapidjson` folder to system or project's include path.

RapidJSON uses following software as its dependencies:

- [CMake](#) as a general build tool
- (optional)[Doxygen](#) to build documentation
- (optional)[googletest](#) for unit and performance testing

To generate user documentation and run tests please proceed with the steps below:

1. Execute `git submodule update --init` to get the files of thirdparty submodules (google test).
2. Create directory called `build` in rapidjson source directory.
3. Change to `build` directory and run `cmake ..` command to configure your build. Windows users can do the same with `cmake-gui` application.
4. On Windows, build the solution found in the build directory. On Linux, run `make` from the build directory.

On successful build you will find compiled test and example binaries in `bin` directory. The generated documentation will be available in `doc/html` directory of the build tree. To run tests after finished build please run `make test` or `ctest` from your build tree. You can get detailed output using `ctest -v` command.

It is possible to install library system-wide by running `make install` command from the build tree with administrative privileges. This will install all files according to system preferences. Once RapidJSON is installed, it is possible to use it from other CMake projects by adding `find_package(RapidJSON)` line to your CMakeLists.txt.

Usage at a glance

This simple example parses a JSON string into a document (DOM), make a simple modification of the DOM, and finally stringify the DOM to a JSON string.

```
// rapidjson/example/simplesdom/simplesdom.cpp`
#include "rapidjson/document.h"
#include "rapidjson/writer.h"
#include "rapidjson/stringbuffer.h"
#include <iostream>

using namespace rapidjson;

int main() {
    // 1. Parse a JSON string into DOM.
    const char* json = "{\"project\":\"rapidjson\",\"stars\":10}";
```

```

Document d;
d.Parse(json);

// 2. Modify it by DOM.
Value& s = d["stars"];
s.SetInt(s.GetInt() + 1);

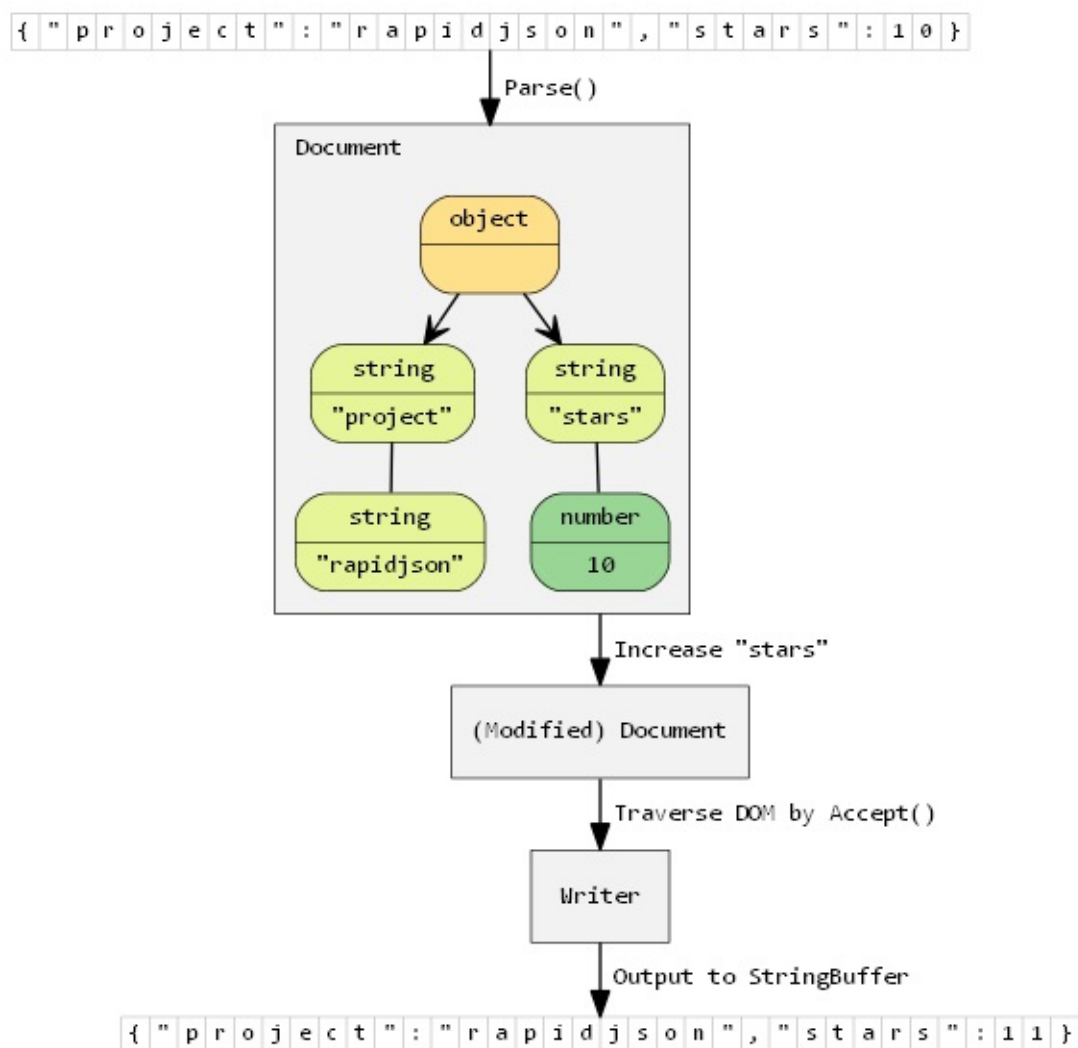
// 3. Stringify the DOM
StringBuffer buffer;
Writer<StringBuffer> writer(buffer);
d.Accept(writer);

// Output {"project":"rapidjson","stars":11}
std::cout << buffer.GetString() << std::endl;
return 0;
}

```

Note that this example did not handle potential errors.

The following diagram shows the process.



More [examples](#) are available.

Features

General

- Cross-platform
 - Compilers: Visual Studio, gcc, clang, etc.
 - Architectures: x86, x64, ARM, etc.
 - Operating systems: Windows, Mac OS X, Linux, iOS, Android, etc.
- Easy installation
 - Header files only library. Just copy the headers to your project.
- Self-contained, minimal dependences
 - No STL, BOOST, etc.
 - Only included `<cstdio>`, `<cstdlib>`, `<cstring>`, `<inttypes.h>`, `<new>`, `<stdint.h>`.
- Without C++ exception, RTTI
- High performance
 - Use template and inline functions to reduce function call overheads.
 - Internal optimized Grisu2 and floating point parsing implementations.
 - Optional SSE2/SSE4.2 support.

Standard compliance

- RapidJSON should be fully RFC4627/ECMA-404 compliance.
- Support Unicode surrogate.
- Support null character (`"\u0000"`)
 - For example, `["Hello\u0000World"]` can be parsed and handled gracefully. There is API for getting/setting lengths of string.

Unicode

- Support UTF-8, UTF-16, UTF-32 encodings, including little endian and big endian.
 - These encodings are used in input/output streams and in-memory representation.
- Support automatic detection of encodings in input stream.
- Support transcoding between encodings internally.
 - For example, you can read a UTF-8 file and let RapidJSON transcode the JSON strings into UTF-16 in the DOM.
- Support encoding validation internally.
 - For example, you can read a UTF-8 file, and let RapidJSON check whether all JSON strings are valid UTF-8 byte sequence.
- Support custom character types.
 - By default the character types are `char` for UTF8, `wchar_t` for UTF16, `uint32_t` for UTF32.
- Support custom encodings.

API styles

- SAX (Simple API for XML) style API
 - Similar to [SAX](#), RapidJSON provides a event sequential access parser API (`rapidjson::GenericReader`). It also provides a generator API (`rapidjson::Writer`) which consumes the same set of events.
- DOM (Document Object Model) style API
 - Similar to [DOM](#) for HTML/XML, RapidJSON can parse JSON into a DOM representation

- (`rapidjson::GenericDocument`), for easy manipulation, and finally stringify back to JSON if needed.
- The DOM style API (`rapidjson::GenericDocument`) is actually implemented with SAX style API (`rapidjson::GenericReader`). SAX is faster but sometimes DOM is easier. Users can pick their choices according to scenarios.

Parsing

- Recursive (default) and iterative parser
 - Recursive parser is faster but prone to stack overflow in extreme cases.
 - Iterative parser use custom stack to keep parsing state.
- Support *in situ* parsing.
 - Parse JSON string values in-place at the source JSON, and then the DOM points to addresses of those strings.
 - Faster than convention parsing: no allocation for strings, no copy (if string does not contain escapes), cache-friendly.
- Support 32-bit/64-bit signed/unsigned integer and `double` for JSON number type.
- Support parsing multiple JSONs in input stream (`kParseStopWhenDoneFlag`).
- Error Handling
 - Support comprehensive error code if parsing failed.
 - Support error message localization.

DOM (Document)

- RapidJSON checks range of numerical values for conversions.
- Optimization for string literal
 - Only store pointer instead of copying
- Optimization for "short" strings
 - Store short string in `value` internally without additional allocation.
 - For UTF-8 string: maximum 11 characters in 32-bit, 15 characters in 64-bit.
- Optionally support `std::string` (define `RAPIDJSON_HAS_STDSTRING=1`)

Generation

- Support `rapidjson::PrettyWriter` for adding newlines and indentations.

Stream

- Support `rapidjson::GenericStringBuffer` for storing the output JSON as string.
- Support `rapidjson::FileReadStream` and `rapidjson::FileWriteStream` for input/output `FILE` object.
- Support custom streams.

Memory

- Minimize memory overheads for DOM.
 - Each JSON value occupies exactly 16/20 bytes for most 32/64-bit machines (excluding text string).
- Support fast default allocator.
 - A stack-based allocator (allocate sequentially, prohibit to free individual allocations, suitable for parsing).
 - User can provide a pre-allocated buffer. (Possible to parse a number of JSONs without any CRT allocation)
- Support standard CRT(C-runtime) allocator.
- Support custom allocators.

Miscellaneous

- Some C++11 support (optional)
 - Rvalue reference
 - `noexcept` specifier

Tutorial

This tutorial introduces the basics of the Document Object Model(DOM) API.

As shown in [Usage at a glance](#), a JSON can be parsed into DOM, and then the DOM can be queried and modified easily, and finally be converted back to JSON.

[TOC]

Value & Document

Each JSON value is stored in a type called `Value`. A `Document`, representing the DOM, contains the root `Value` of the DOM tree. All public types and functions of RapidJSON are defined in the `rapidjson` namespace.

Query Value

In this section, we will use excerpt of `example/tutorial/tutorial.cpp`.

Assumes we have a JSON stored in a C string (`const char* json`):

```
{
  "hello": "world",
  "t": true ,
  "f": false,
  "n": null,
  "i": 123,
  "pi": 3.1416,
  "a": [1, 2, 3, 4]
}
```

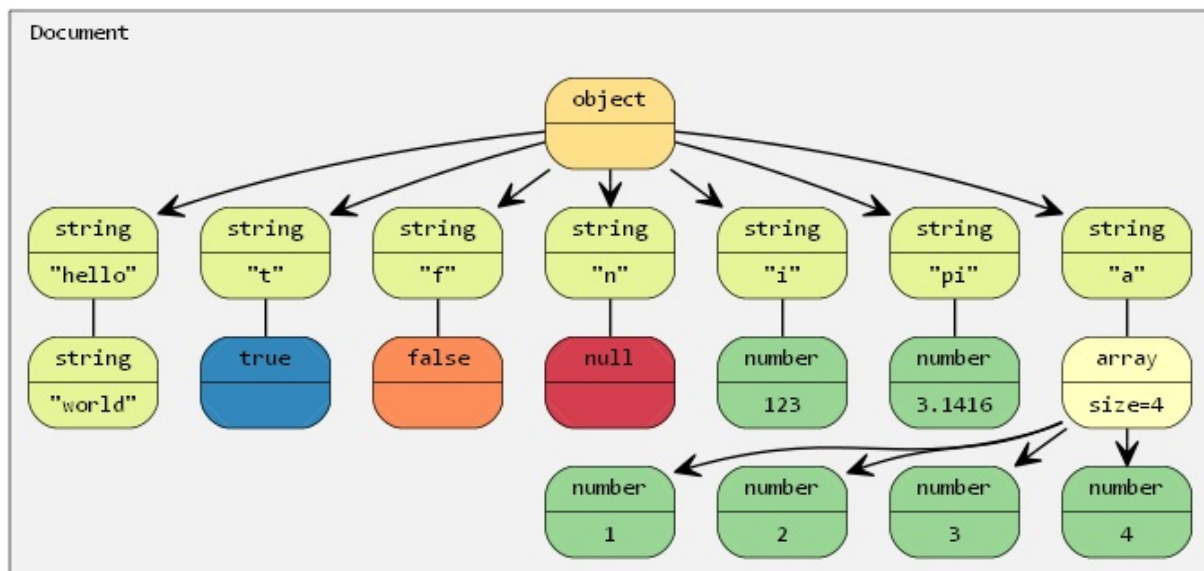
Parse it into a `Document` :

```
#include "rapidjson/document.h"

using namespace rapidjson;

// ...
Document document;
document.Parse(json);
```

The JSON is now parsed into `document` as a *DOM tree*:



Since the update to RFC 7159, the root of a conforming JSON document can be any JSON value. In earlier RFC 4627, only objects or arrays were allowed as root values. In this case, the root is an object.

```
assert(document.IsObject());
```

Let's query whether a "hello" member exists in the root object. Since a `Value` can contain different types of value, we may need to verify its type and use suitable API to obtain the value. In this example, "hello" member associates with a JSON string.

```
assert(document.HasMember("hello"));
assert(document["hello"].IsString());
printf("hello = %s\n", document["hello"].GetString());
```

```
world
```

JSON true/false values are represented as `bool` .

```
assert(document["t"].IsBool());
printf("t = %s\n", document["t"].GetBool() ? "true" : "false");
```

```
true
```

JSON null can be queried by `IsNull()` .

```
printf("n = %s\n", document["n"].IsNull() ? "null" : "?");
```

```
null
```

JSON number type represents all numeric values. However, C++ needs more specific type for manipulation.

```
assert(document["i"].IsNumber());

// In this case, IsUint()/IsInt64()/IsUInt64() also return true.
assert(document["i"].IsInt());
printf("i = %d\n", document["i"].GetInt());
// Alternative (int)document["i"]

assert(document["pi"].IsNumber());
assert(document["pi"].IsDouble());
printf("pi = %g\n", document["pi"].GetDouble());
```

```
i = 123
pi = 3.1416
```

JSON array contains a number of elements.

```
// Using a reference for consecutive access is handy and faster.
const Value& a = document["a"];
assert(a.IsArray());
for (SizeType i = 0; i < a.Size(); i++) // Uses SizeType instead of size_t
    printf("a[%d] = %d\n", i, a[i].GetInt());
```

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
```

Note that, RapidJSON does not automatically convert values between JSON types. If a value is a string, it is invalid to call `GetInt()`, for example. In debug mode it will fail an assertion. In release mode, the behavior is undefined.

In the following, details about querying individual types are discussed.

Query Array

By default, `SizeType` is typedef of `unsigned`. In most systems, array is limited to store up to $2^{32}-1$ elements.

You may access the elements in array by integer literal, for example, `a[0]`, `a[1]`, `a[2]`.

Array is similar to `std::vector`, instead of using indices, you may also use iterator to access all the elements.

```
for (Value::ConstValueIterator itr = a.Begin(); itr != a.End(); ++itr)
    printf("%d ", itr->GetInt());
```

And other familiar query functions:

- `SizeType Capacity() const`
- `bool Empty() const`

Query Object

Similar to array, we can access all object members by iterator:

```
static const char* kTypeNames[] =
{ "Null", "False", "True", "Object", "Array", "String", "Number" };

for (Value::ConstMemberIterator itr = document.MemberBegin();
     itr != document.MemberEnd(); ++itr)
{
    printf("Type of member %s is %s\n",
          itr->name.GetString(), kTypeNames[itr->value.GetType()]);
}
```

```
Type of member hello is String
Type of member t is True
Type of member f is False
Type of member n is Null
Type of member i is Number
Type of member pi is Number
Type of member a is Array
```

Note that, when `operator[](const char*)` cannot find the member, it will fail an assertion.

If we are unsure whether a member exists, we need to call `HasMember()` before calling `operator[](const char*)`. However, this incurs two lookup. A better way is to call `FindMember()`, which can check the existence of member and obtain its value at once:

```
Value::ConstMemberIterator itr = document.FindMember("hello");
if (itr != document.MemberEnd())
    printf("%s %s\n", itr->value.GetString());
```

Querying Number

JSON provide a single numerical type called Number. Number can be integer or real numbers. RFC 4627 says the range of Number is specified by parser.

As C++ provides several integer and floating point number types, the DOM tries to handle these with widest possible range and good performance.

When a Number is parsed, it is stored in the DOM as either one of the following type:

Type	Description
<code>unsigned</code>	32-bit unsigned integer
<code>int</code>	32-bit signed integer
<code>uint64_t</code>	64-bit unsigned integer
<code>int64_t</code>	64-bit signed integer
<code>double</code>	64-bit double precision floating point

When querying a number, you can check whether the number can be obtained as target type:

Checking	Obtaining
<code>bool IsNumber()</code>	N/A

<code>bool IsUint()</code>	<code>unsigned GetUint()</code>
<code>bool IsInt()</code>	<code>int GetInt()</code>
<code>bool IsUint64()</code>	<code>uint64_t GetUint64()</code>
<code>bool IsInt64()</code>	<code>int64_t GetInt64()</code>
<code>bool IsDouble()</code>	<code>double GetDouble()</code>

Note that, an integer value may be obtained in various ways without conversion. For example, A value `x` containing 123 will make `x.IsInt() == x.IsUint() == x.IsInt64() == x.IsUint64() == true`. But a value `y` containing -3000000000 will only makes `x.IsInt64() == true`.

When obtaining the numeric values, `GetDouble()` will convert internal integer representation to a `double`. Note that, `int` and `unsigned` can be safely convert to `double`, but `int64_t` and `uint64_t` may lose precision (since mantissa of `double` is only 52-bits).

Query String

In addition to `GetString()`, the `Value` class also contains `GetStringLength()`. Here explains why.

According to RFC 4627, JSON strings can contain Unicode character `U+0000`, which must be escaped as `"\u0000"`. The problem is that, C/C++ often uses null-terminated string, which treats `'\0'` as the terminator symbol.

To conform RFC 4627, RapidJSON supports string containing `U+0000`. If you need to handle this, you can use `GetStringLength()` API to obtain the correct length of string.

For example, after parsing a the following JSON to `Document d`:

```
{ "s" : "a\u0000b" }
```

The correct length of the value `"a\u0000b"` is 3. But `strlen()` returns 1.

`GetStringLength()` can also improve performance, as user may often need to call `strlen()` for allocating buffer.

Besides, `std::string` also support a constructor:

```
string(const char* s, size_t count);
```

which accepts the length of string as parameter. This constructor supports storing null character within the string, and should also provide better performance.

Comparing values

You can use `==` and `!=` to compare values. Two values are equal if and only if they are have same type and contents. You can also compare values with primitive types. Here is an example.

```
if (document["hello"] == document["n"]) /*...*/; // Compare values
if (document["hello"] == "world") /*...*/; // Compare value with literal string
if (document["i"] != 123) /*...*/; // Compare with integers
if (document["pi"] != 3.14) /*...*/; // Compare with double.
```

Array/object compares their elements/members in order. They are equal if and only if their whole subtrees are equal.

Note that, currently if an object contains duplicated named member, comparing equality with any object is always `false`.

Create/Modify Values

There are several ways to create values. After a DOM tree is created and/or modified, it can be saved as JSON again using `Writer`.

Change Value Type

When creating a Value or Document by default constructor, its type is Null. To change its type, call `SetXXX()` or assignment operator, for example:

```
Document d; // Null
d.SetObject();

Value v;     // Null
v.SetInt(10);
v = 10;      // Shortcut, same as above
```

Overloaded Constructors

There are also overloaded constructors for several types:

```
Value b(true);    // calls Value(bool)
Value i(-123);    // calls Value(int)
Value u(123u);    // calls Value(unsigned)
Value d(1.5);     // calls Value(double)
```

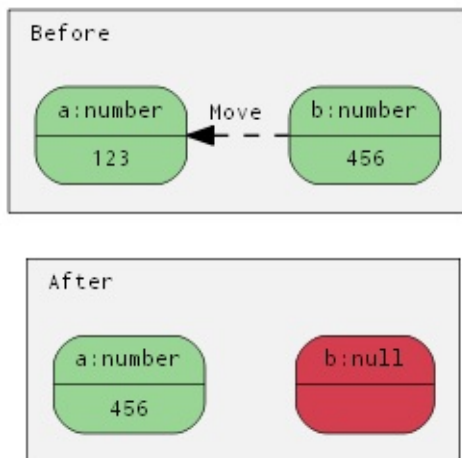
To create empty object or array, you may use `SetObject()` / `SetArray()` after default constructor, or using the `Value(Type)` in one shot:

```
Value o(kObjectType);
Value a(kArrayType);
```

Move Semantics

A very special decision during design of RapidJSON is that, assignment of value does not copy the source value to destination value. Instead, the value from source is moved to the destination. For example,

```
Value a(123);
Value b(456);
b = a;          // a becomes a Null value, b becomes number 123.
```

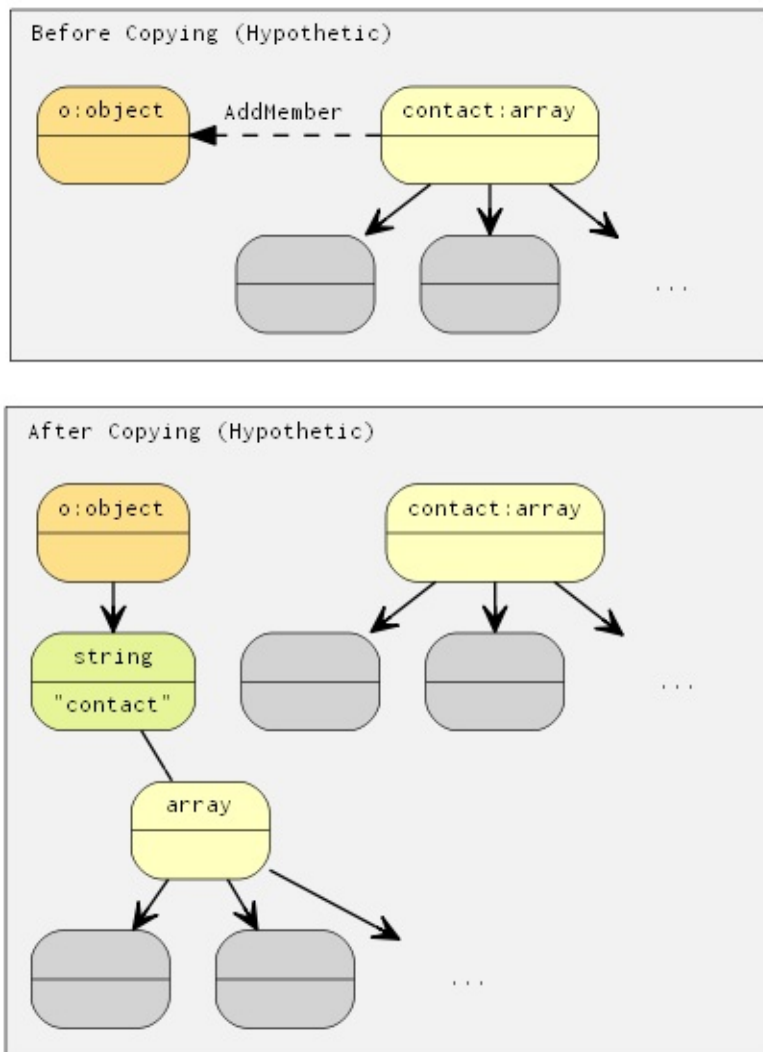


Why? What is the advantage of this semantics?

The simple answer is performance. For fixed size JSON types (Number, True, False, Null), copying them is fast and easy. However, For variable size JSON types (String, Array, Object), copying them will incur a lot of overheads. And these overheads are often unnoticed. Especially when we need to create temporary object, copy it to another variable, and then destruct it.

For example, if normal *copy* semantics was used:

```
Value o(kObjectType);
{
    Value contacts(kArrayType);
    // adding elements to contacts array.
    // ...
    o.AddMember("contacts", contacts); // deep clone contacts (may be with lots of allocations)
    // destruct contacts.
}
```



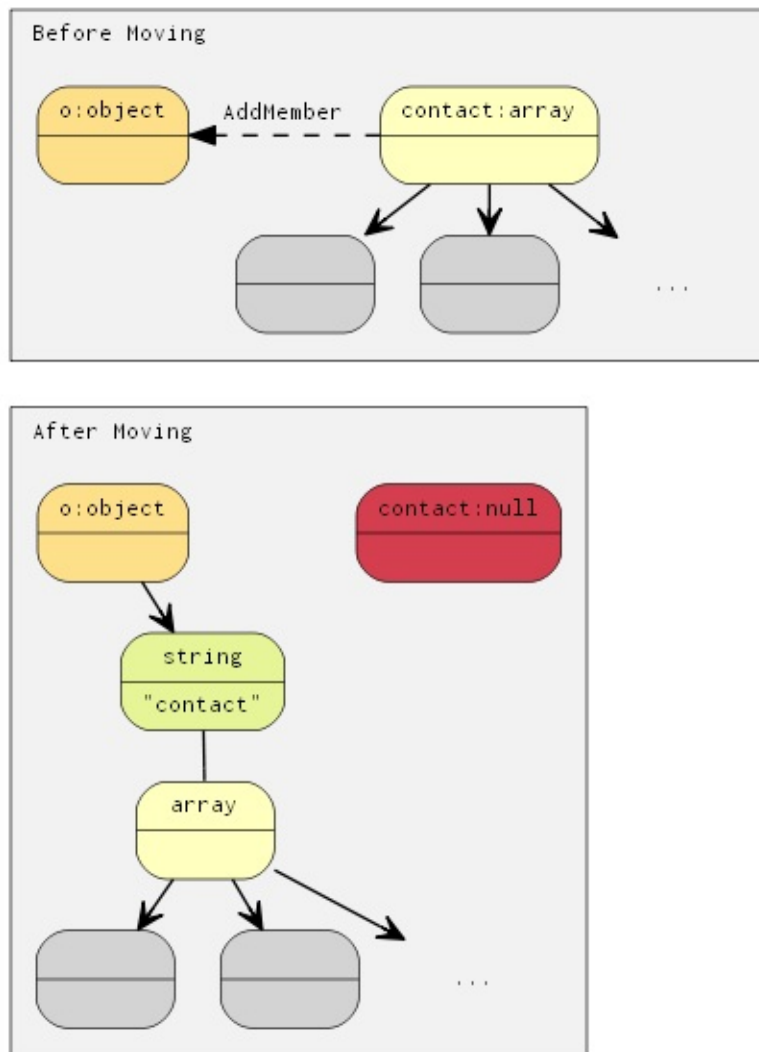
The object `o` needs to allocate a buffer of same size as `contacts`, makes a deep clone of it, and then finally `contacts` is destructed. This will incur a lot of unnecessary allocations/deallocations and memory copying.

There are solutions to prevent actual copying these data, such as reference counting and garbage collection(GC).

To make RapidJSON simple and fast, we chose to use *move* semantics for assignment. It is similar to `std::auto_ptr` which transfer ownership during assignment. Move is much faster and simpler, it just destructs the original value, `memcpy()` the source to destination, and finally sets the source as Null type.

So, with move semantics, the above example becomes:

```
Value o(kObjectType);
{
    Value contacts(kArrayType);
    // adding elements to contacts array.
    o.AddMember("contacts", contacts); // just memcpy() of contacts itself to the value of new member (16 bytes)
    // contacts became Null here. Its destruction is trivial.
}
```

This is called move assignment operator in C++11. As RapidJSON supports C++03, it adopts move semantics using assignment operator, and all other modifying function like `AddMember()`, `PushBack()`.

Move semantics and temporary values

Sometimes, it is convenient to construct a Value in place, before passing it to one of the "moving" functions, like

`PushBack()` OR `AddMember()`. As temporary objects can't be converted to proper Value references, the convenience function `Move()` is available:

```
Value a(kArrayType);
Document::AllocatorType& allocator = document.GetAllocator();
// a.PushBack(Value(42), allocator); // will not compile
a.PushBack(Value().SetInt(42), allocator); // fluent API
a.PushBack(Value(42).Move(), allocator); // same as above
```

Create String

RapidJSON provide two strategies for storing string.

1. copy-string: allocates a buffer, and then copy the source data into it.
2. const-string: simply store a pointer of string.

Copy-string is always safe because it owns a copy of the data. Const-string can be used for storing string literal, and in-situ parsing which we will mentioned in Document section.

To make memory allocation customizable, RapidJSON requires user to pass an instance of allocator, whenever an operation may require allocation. This design is needed to prevent storing a allocator (or Document) pointer per Value.

Therefore, when we assign a copy-string, we call this overloaded `SetString()` with allocator:

```
Document document;
Value author;
char buffer[10];
int len = sprintf(buffer, "%s %s", "Milo", "Yip"); // dynamically created string.
author.SetString(buffer, len, document.GetAllocator());
memset(buffer, 0, sizeof(buffer));
// author.GetString() still contains "Milo Yip" after buffer is destroyed
```

In this example, we get the allocator from a `Document` instance. This is a common idiom when using RapidJSON. But you may use other instances of allocator.

Besides, the above `SetString()` requires length. This can handle null characters within a string. There is another `SetString()` overloaded function without the length parameter. And it assumes the input is null-terminated and calls a `strlen()`-like function to obtain the length.

Finally, for string literal or string with safe life-cycle can use const-string version of `SetString()`, which lacks allocator parameter. For string literals (or constant character arrays), simply passing the literal as parameter is safe and efficient:

```
Value s;
s.SetString("rapidjson"); // can contain null character, length derived at compile time
s = "rapidjson";          // shortcut, same as above
```

For character pointer, the RapidJSON requires to mark it as safe before using it without copying. This can be achieved by using the `StringRef` function:

```
const char * cstr = getenv("USER");
size_t cstr_len = ...; // in case length is available
Value s;
// s.SetString(cstr); // will not compile
s.SetString(StringRef(cstr)); // ok, assume safe lifetime, null-terminated
s = StringRef(cstr); // shortcut, same as above
s.SetString(StringRef(cstr, cstr_len)); // faster, can contain null character
s = StringRef(cstr, cstr_len); // shortcut, same as above
```

Modify Array

Value with array type provides similar APIs as `std::vector`.

- `Clear()`
- `Reserve(SizeType, Allocator&)`
- `Value& PushBack(Value&, Allocator&)`
- `template <typename T> GenericValue& PushBack(T, Allocator&)`
- `Value& PopBack()`
- `ValueIterator Erase(ConstValueIterator pos)`
- `ValueIterator Erase(ConstValueIterator first, ConstValueIterator last)`

Note that, `Reserve(...)` and `PushBack(...)` may allocate memory for the array elements, therefore require an allocator.

Here is an example of `PushBack()` :

```
Value a(kArrayType);
Document::AllocatorType& allocator = document.GetAllocator();

for (int i = 5; i <= 10; i++)
    a.PushBack(i, allocator);    // allocator is needed for potential realloc().

// Fluent interface
a.PushBack("Lua", allocator).PushBack("Mio", allocator);
```

Differs from STL, `PushBack()` / `PopBack()` returns the array reference itself. This is called *fluent interface*.

If you want to add a non-constant string or a string without sufficient lifetime (see [Create String](#)) to the array, you need to create a string Value by using the copy-string API. To avoid the need for an intermediate variable, you can use a [temporary value](#) in place:

```
// in-place Value parameter
contact.PushBack(Value("copy", document.GetAllocator()).Move(), // copy string
                 document.GetAllocator());

// explicit parameters
Value val("key", document.GetAllocator()); // copy string
contact.PushBack(val, document.GetAllocator());
```

Modify Object

Object is a collection of key-value pairs (members). Each key must be a string value. To modify an object, either add or remove members. The following APIs are for adding members:

- `Value& AddMember(Value&, Value&, Allocator& allocator)`
- `Value& AddMember(StringRefType, Value&, Allocator&)`
- `template <typename T> Value& AddMember(StringRefType, T value, Allocator&)`

Here is an example.

```
Value contact(kObject);
contact.AddMember("name", "Milo", document.GetAllocator());
contact.AddMember("married", true, document.GetAllocator());
```

The name parameter with `StringRefType` is similar to the interface of `SetString` function for string values. These overloads are used to avoid the need for copying the `name` string, as constant key names are very common in JSON objects.

If you need to create a name from a non-constant string or a string without sufficient lifetime (see [Create String](#)), you need to create a string Value by using the copy-string API. To avoid the need for an intermediate variable, you can use a [temporary value](#) in place:

```
// in-place Value parameter
contact.AddMember(Value("copy", document.GetAllocator()).Move(), // copy string
                  Value().Move(),                                // null value
                  document.GetAllocator());

// explicit parameters
Value key("key", document.GetAllocator()); // copy string name
Value val(42);                             // some value
contact.AddMember(key, val, document.GetAllocator());
```

For removing members, there are several choices:

- `bool RemoveMember(const Ch* name)` : Remove a member by search its name (linear time complexity).
- `bool RemoveMember(const Value& name)` : same as above but `name` is a `Value`.
- `MemberIterator RemoveMember(MemberIterator)` : Remove a member by iterator (*constant* time complexity).
- `MemberIterator EraseMember(MemberIterator)` : similar to the above but it preserves order of members (linear time complexity).
- `MemberIterator EraseMember(MemberIterator first, MemberIterator last)` : remove a range of members, preserves order (linear time complexity).

`MemberIterator RemoveMember(MemberIterator)` uses a "move-last" trick to achieve constant time complexity. Basically the member at iterator is destructed, and then the last element is moved to that position. So the order of the remaining members are changed.

Deep Copy Value

If we really need to copy a DOM tree, we can use two APIs for deep copy: constructor with allocator, and `CopyFrom()`.

```
Document d;
Document::AllocatorType& a = d.GetAllocator();
Value v1("foo");
// Value v2(v1); // not allowed

Value v2(v1, a);           // make a copy
assert(v1.IsString());      // v1 untouched
d.SetArray().PushBack(v1, a).PushBack(v2, a);
assert(v1.IsNull() && v2.IsNull()); // both moved to d

v2.CopyFrom(d, a);          // copy whole document to v2
assert(d.IsArray() && d.Size() == 2); // d untouched
v1.SetObject().AddMember("array", v2, a);
d.PushBack(v1, a);
```

Swap Values

`Swap()` is also provided.

```
Value a(123);
Value b("Hello");
a.Swap(b);
assert(a.IsString());
assert(b.IsInt());
```

Swapping two DOM trees is fast (constant time), despite the complexity of the trees.

What's next

This tutorial shows the basics of DOM tree query and manipulation. There are several important concepts in RapidJSON:

1. [Streams](#) are channels for reading/writing JSON, which can be a in-memory string, or file stream, etc. User can also create their streams.
2. [Encoding](#) defines which character encoding is used in streams and memory. RapidJSON also provide Unicode conversion/validation internally.
3. [DOM](#)'s basics are already covered in this tutorial. Uncover more advanced features such as *in situ* parsing, other

parsing options and advanced usages.

4. [SAX](#) is the foundation of parsing/generating facility in RapidJSON. Learn how to use `Reader / Writer` to implement even faster applications. Also try `PrettyWriter` to format the JSON.
5. [Performance](#) shows some in-house and third-party benchmarks.
6. [Internals](#) describes some internal designs and techniques of RapidJSON.

You may also refer to the [FAQ](#), API documentation, examples and unit tests.

Stream

In RapidJSON, `rapidjson::Stream` is a concept for reading/writing JSON. Here we first show how to use streams provided. And then see how to create a custom stream.

[TOC]

Memory Streams

Memory streams store JSON in memory.

StringStream (Input)

`StringStream` is the most basic input stream. It represents a complete, read-only JSON stored in memory. It is defined in `rapidjson/rapidjson.h`.

```
#include "rapidjson/document.h" // will include "rapidjson/rapidjson.h"

using namespace rapidjson;

// ...
const char json[] = "[1, 2, 3, 4]";
StringStream s(json);

Document d;
d.ParseStream(s);
```

Since this is very common usage, `Document::Parse(const char*)` is provided to do exactly the same as above:

```
// ...
const char json[] = "[1, 2, 3, 4]";
Document d;
d.Parse(json);
```

Note that, `StringStream` is a typedef of `GenericStringStream<UTF8<> >`, user may use another encodings to represent the character set of the stream.

StringBuffer (Output)

`StringBuffer` is a simple output stream. It allocates a memory buffer for writing the whole JSON. Use `GetString()` to obtain the buffer.

```
#include "rapidjson/stringbuffer.h"

StringBuffer buffer;
Writer<StringBuffer> writer(buffer);
d.Accept(writer);

const char* output = buffer.GetString();
```

When the buffer is full, it will increase the capacity automatically. The default capacity is 256 characters (256 bytes for

UTF8, 512 bytes for UTF16, etc.). User can provide an allocator and a initial capacity.

```
StringBuffer buffer1(0, 1024); // Use its allocator, initial size = 1024
StringBuffer buffer2(allocator, 1024);
```

By default, `StringBuffer` will instantiate an internal allocator.

Similarly, `StringBuffer` is a typedef of `GenericStringBuffer<UTF8<> >`.

File Streams

When parsing a JSON from file, you may read the whole JSON into memory and use `StringStream` above.

However, if the JSON is big, or memory is limited, you can use `FileReadStream`. It only read a part of JSON from file into buffer, and then let the part be parsed. If it runs out of characters in the buffer, it will read the next part from file.

FileReadStream (Input)

`FileReadStream` reads the file via a `FILE` pointer. And user need to provide a buffer.

```
#include "rapidjson/filereadstream.h"
#include <cstdio>

using namespace rapidjson;

FILE* fp = fopen("big.json", "rb"); // non-Windows use "r"

char readBuffer[65536];
FileReadStream is(fp, readBuffer, sizeof(readBuffer));

Document d;
d.ParseStream(is);

fclose(fp);
```

Different from string streams, `FileReadStream` is byte stream. It does not handle encodings. If the file is not UTF-8, the byte stream can be wrapped in a `EncodedInputStream`. It will be discussed very soon.

Apart from reading file, user can also use `FileReadStream` to read `stdin`.

FileWriteStream (Output)

`FileWriteStream` is buffered output stream. Its usage is very similar to `FileReadStream`.

```
#include "rapidjson/filewritestream.h"
#include <cstdio>

using namespace rapidjson;

Document d;
d.Parse(json);
// ...

FILE* fp = fopen("output.json", "wb"); // non-Windows use "w"

char writeBuffer[65536];
FileWriteStream os(fp, writeBuffer, sizeof(writeBuffer));
```

```

Writer<FileWriteStream> writer(os);
d.Accept(writer);

fclose(fp);

```

It can also directs the output to `stdout` .

Encoded Streams

Encoded streams do not contain JSON itself, but they wrap byte streams to provide basic encoding/decoding function.

As mentioned above, UTF-8 byte streams can be read directly. However, UTF-16 and UTF-32 have endian issue. To handle endian correctly, it needs to convert bytes into characters (e.g. `wchar_t` for UTF-16) while reading, and characters into bytes while writing.

Besides, it also need to handle [byte order mark \(BOM\)](#). When reading from a byte stream, it is needed to detect or just consume the BOM if exists. When writing to a byte stream, it can optionally write BOM.

If the encoding of stream is known in compile-time, you may use `EncodedInputStream` and `EncodedOutputStream` . If the stream can be UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE JSON, and it is only known in runtime, you may use `AutoUTFInputStream` and `AutoUTFOutputStream` . These streams are defined in `rapidjson/encodedstream.h` .

Note that, these encoded streams can be applied to streams other than file. For example, you may have a file in memory, or a custom byte stream, be wrapped in encoded streams.

EncodedInputStream

`EncodedInputStream` has two template parameters. The first one is a `Encoding` class, such as `UTF8` , `UTF16LE` , defined in `rapidjson/encodings.h` . The second one is the class of stream to be wrapped.

```

#include "rapidjson/document.h"
#include "rapidjson/filereadstream.h" // FileReadStream
#include "rapidjson/encodedstream.h" // EncodedInputStream
#include <cstdio>

using namespace rapidjson;

FILE* fp = fopen("utf16le.json", "rb"); // non-Windows use "r"

char readBuffer[256];
FileReadStream bis(fp, readBuffer, sizeof(readBuffer));

EncodedInputStream<UTF16LE<>, FileReadStream> eis(bis); // wraps bis into eis

Document d; // Document is GenericDocument<UTF8<> >
d.ParseStream<0, UTF16LE<> >(eis); // Parses UTF-16LE file into UTF-8 in memory

fclose(fp);

```

EncodedOutputStream

`EncodedOutputStream` is similar but it has a `bool putBOM` parameter in the constructor, controlling whether to write BOM into output byte stream.

```

#include "rapidjson/filewritestream.h" // FileWriteStream

```



```
#include "rapidjson/encodedstream.h" // EncodedOutputStream
#include <cstdio>

Document d; // Document is GenericDocument<UTF8<> >
// ...

FILE* fp = fopen("output_utf32le.json", "wb"); // non-Windows use "w"

char writeBuffer[256];
FileWriteStream bos(fp, writeBuffer, sizeof(writeBuffer));

typedef EncodedOutputStream<UTF32LE<>, FileWriteStream> OutputStream;
OutputStream eos(bos, true); // Write BOM

Writer<OutputStream, UTF32LE<>, UTF8<>> writer(eos);
d.Accept(writer); // This generates UTF32-LE file from UTF-8 in memory

fclose(fp);
```

AutoUTFInputStream

Sometimes an application may want to handle all supported JSON encoding. `AutoUTFInputStream` will detection encoding by BOM first. If BOM is unavailable, it will use characteristics of valid JSON to make detection. If neither method success, it falls back to the UTF type provided in constructor.

Since the characters (code units) may be 8-bit, 16-bit or 32-bit. `AutoUTFInputStream` requires a character type which can hold at least 32-bit. We may use `unsigned`, as in the template parameter:

```
#include "rapidjson/document.h"
#include "rapidjson/filereadstream.h" // FileReadStream
#include "rapidjson/encodedstream.h" // AutoUTFInputStream
#include <cstdio>

using namespace rapidjson;

FILE* fp = fopen("any.json", "rb"); // non-Windows use "r"

char readBuffer[256];
FileReadStream bis(fp, readBuffer, sizeof(readBuffer));

AutoUTFInputStream<unsigned, FileReadStream> eis(bis); // wraps bis into eis

Document d; // Document is GenericDocument<UTF8<> >
d.ParseStream<0, AutoUTF<unsigned> >(eis); // This parses any UTF file into UTF-8 in memory

fclose(fp);
```

When specifying the encoding of stream, uses `AutoUTF<CharType>` as in `ParseStream()` above.

You can obtain the type of UTF via `UTFType GetType()`. And check whether a BOM is found by `HasBOM()`

AutoUTFOutputStream

Similarly, to choose encoding for output during runtime, we can use `AutoUTFOutputStream`. This class is not automatic *per se*. You need to specify the UTF type and whether to write BOM in runtime.

```
using namespace rapidjson;

void WriteJSONFile(FILE* fp, UTFType type, bool putBOM, const Document& d) {
    char writeBuffer[256];
    FileWriteStream bos(fp, writeBuffer, sizeof(writeBuffer));
```

```
typedef AutoUTFOutputStream<unsigned, FileWriteStream> OutputStream;
OutputStream eos(bos, type, putBOM);

Writer<OutputStream, UTF8<>, AutoUTF<> > writer;
d.Accept(writer);
}
```

`AutoUTFInputStream` and `AutoUTFOutputStream` is more convenient than `EncodedInputStream` and `EncodedOutputStream`. They just incur a little bit runtime overheads.

Custom Stream

In addition to memory/file streams, user can create their own stream classes which fits RapidJSON's API. For example, you may create network stream, stream from compressed file, etc.

RapidJSON combines different types using templates. A class containing all required interface can be a stream. The Stream interface is defined in comments of `rapidjson/rapidjson.h`:

```
concept Stream {
    typename Ch;    //!< Character type of the stream.

    //!< Read the current character from stream without moving the read cursor.
    Ch Peek() const;

    //!< Read the current character from stream and moving the read cursor to next character.
    Ch Take();

    //!< Get the current read cursor.
    //!< \return Number of characters read from start.
    size_t Tell();

    //!< Begin writing operation at the current read pointer.
    //!< \return The begin writer pointer.
    Ch* PutBegin();

    //!< Write a character.
    void Put(Ch c);

    //!< Flush the buffer.
    void Flush();

    //!< End the writing operation.
    //!< \param begin The begin write pointer returned by PutBegin().
    //!< \return Number of characters written.
    size_t PutEnd(Ch* begin);
}
```

For input stream, they must implement `Peek()`, `Take()` and `Tell()`. For output stream, they must implement `Put()` and `Flush()`. There are two special interface, `PutBegin()` and `PutEnd()`, which are only for *in situ* parsing. Normal streams do not implement them. However, if the interface is not needed for a particular stream, it is still need to a dummy implementation, otherwise will generate compilation error.

Example: istream wrapper

The following example is a wrapper of `std::istream`, which only implements 3 functions.

```
class IStreamWrapper {
public:
    typedef char Ch;

    IStreamWrapper(std::istream& is) : is_(is) {
```

```

    }

    Ch Peek() const { // 1
        int c = is_.peek();
        return c == std::char_traits<char>::eof() ? '\0' : (Ch)c;
    }

    Ch Take() { // 2
        int c = is_.get();
        return c == std::char_traits<char>::eof() ? '\0' : (Ch)c;
    }

    size_t Tell() const { return (size_t)is_.tellg(); } // 3

    Ch* PutBegin() { assert(false); return 0; }
    void Put(Ch) { assert(false); }
    void Flush() { assert(false); }
    size_t PutEnd(Ch*) { assert(false); return 0; }

private:
    IStreamWrapper(const IStreamWrapper&);
    IStreamWrapper& operator=(const IStreamWrapper&);

    std::istream& is_;
};

```

User can use it to wrap instances of `std::stringstream` , `std::ifstream` .

```

const char* json = "[1,2,3,4]";
std::stringstream ss(json);
IStreamWrapper is(ss);

Document d;
d.Parse(is);

```

Note that, this implementation may not be as efficient as RapidJSON's memory or file streams, due to internal overheads of the standard library.

Example: ostream wrapper

The following example is a wrapper of `std::ostream` , which only implements 2 functions.

```

class OStreamWrapper {
public:
    typedef char Ch;

    OStreamWrapper(std::ostream& os) : os_(os) {
    }

    Ch Peek() const { assert(false); return '\0'; }
    Ch Take() { assert(false); return '\0'; }
    size_t Tell() const { }

    Ch* PutBegin() { assert(false); return 0; }
    void Put(Ch c) { os_.put(c); } // 1
    void Flush() { os_.flush(); } // 2
    size_t PutEnd(Ch*) { assert(false); return 0; }

private:
    OStreamWrapper(const OStreamWrapper&);
    OStreamWrapper& operator=(const OStreamWrapper&);

    std::ostream& os_;
};

```

User can use it to wrap instances of `std::stringstream`, `std::ofstream`.

```
Document d;  
// ...  
  
std::stringstream ss;  
OStreamWrapper os(ss);  
  
Writer<OStreamWrapper> writer(os);  
d.Accept(writer);
```

Note that, this implementation may not be as efficient as RapidJSON's memory or file streams, due to internal overheads of the standard library.

Summary

This section describes stream classes available in RapidJSON. Memory streams are simple. File stream can reduce the memory required during JSON parsing and generation, if the JSON is stored in file system. Encoded streams converts between byte streams and character streams. Finally, user may create custom streams using a simple interface.

Encoding

According to [ECMA-404](#),

(in Introduction) JSON text is a sequence of Unicode code points.

The earlier [RFC4627](#) stated that,

(in §3) JSON text SHALL be encoded in Unicode. The default encoding is UTF-8.

(in §6) JSON may be represented using UTF-8, UTF-16, or UTF-32. When JSON is written in UTF-8, JSON is 8bit compatible. When JSON is written in UTF-16 or UTF-32, the binary content-transfer-encoding must be used.

RapidJSON supports various encodings. It can also validate the encodings of JSON, and transcoding JSON among encodings. All these features are implemented internally, without the need for external libraries (e.g. [ICU](#)).

[TOC]

Unicode

From [Unicode's official website](#):

Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language.

Those unique numbers are called code points, which is in the range `0x0` to `0x10FFFF`.

Unicode Transformation Format

There are various encodings for storing Unicode code points. These are called Unicode Transformation Format (UTF). RapidJSON supports the most commonly used UTFs, including

- UTF-8: 8-bit variable-width encoding. It maps a code point to 1–4 bytes.
- UTF-16: 16-bit variable-width encoding. It maps a code point to 1–2 16-bit code units (i.e., 2–4 bytes).
- UTF-32: 32-bit fixed-width encoding. It directly maps a code point to a single 32-bit code unit (i.e. 4 bytes).

For UTF-16 and UTF-32, the byte order (endianness) does matter. Within computer memory, they are often stored in the computer's endianness. However, when it is stored in file or transferred over network, we need to state the byte order of the byte sequence, either little-endian (LE) or big-endian (BE).

RapidJSON provide these encodings via the structs in `rapidjson/encodings.h`:

```
namespace rapidjson {

template<typename CharType = char>
struct UTF8;

template<typename CharType = wchar_t>
struct UTF16;

template<typename CharType = wchar_t>
struct UTF16LE;

template<typename CharType = wchar_t>
```

```

struct UTF16BE;

template<typename CharType = unsigned>
struct UTF32;

template<typename CharType = unsigned>
struct UTF32LE;

template<typename CharType = unsigned>
struct UTF32BE;

} // namespace rapidjson

```

For processing text in memory, we normally use `UTF8` , `UTF16` or `UTF32` . For processing text via I/O, we may use `UTF8` , `UTF16LE` , `UTF16BE` , `UTF32LE` or `UTF32BE` .

When using the DOM-style API, the `Encoding` template parameter in `GenericValue<Encoding>` and `GenericDocument<Encoding>` indicates the encoding to be used to represent JSON string in memory. So normally we will use `UTF8` , `UTF16` or `UTF32` for this template parameter. The choice depends on operating systems and other libraries that the application is using. For example, Windows API represents Unicode characters in UTF-16, while most Linux distributions and applications prefer UTF-8.

Example of UTF-16 DOM declaration:

```

typedef GenericDocument<UTF16<> > WDocument;
typedef GenericValue<UTF16<> > WValue;

```

For a detail example, please check the example in [DOM's Encoding](#) section.

Character Type

As shown in the declaration, each encoding has a `CharType` template parameter. Actually, it may be a little bit confusing, but each `CharType` stores a code unit, not a character (code point). As mentioned in previous section, a code point may be encoded to 1–4 code units for UTF-8.

For `UTF16(LE|BE)` , `UTF32(LE|BE)` , the `CharType` must be integer type of at least 2 and 4 bytes respectively.

Note that C++11 introduces `char16_t` and `char32_t` , which can be used for `UTF16` and `UTF32` respectively.

AutoUTF

Previous encodings are statically bound in compile-time. In other words, user must know exactly which encodings will be used in the memory or streams. However, sometimes we may need to read/write files of different encodings. The encoding needed to be decided in runtime.

`AutoUTF` is an encoding designed for this purpose. It chooses which encoding to be used according to the input or output stream. Currently, it should be used with `EncodedInputStream` and `EncodedOutputStream` .

ASCII

Although the JSON standards did not mention about [ASCII](#), sometimes we would like to write 7-bit ASCII JSON for applications that cannot handle UTF-8. Since any JSON can represent unicode characters in escaped sequence `\uxxxx` , JSON can always be encoded in ASCII.

Here is an example for writing a UTF-8 DOM into ASCII:

```
using namespace rapidjson;
Document d; // UTF8<>
// ...
StringBuffer buffer;
Writer<StringBuffer, Document::EncodingType, ASCII<> > writer(buffer);
d.Accept(writer);
std::cout << buffer.GetString();
```

ASCII can be used in input stream. If the input stream contains bytes with values above 127, it will cause

```
kParseErrorStringInvalidEncoding error.
```

ASCII *cannot* be used in memory (encoding of `Document` or target encoding of `Reader`), as it cannot represent Unicode code points.

Validation & Transcoding

When RapidJSON parses a JSON, it can validate the input JSON, whether it is a valid sequence of a specified encoding. This option can be turned on by adding `kParseValidateEncodingFlag` in `parseFlags` template parameter.

If the input encoding and output encoding is different, `Reader` and `Writer` will automatically transcode (convert) the text. In this case, `kParseValidateEncodingFlag` is not necessary, as it must decode the input sequence. And if the sequence was unable to be decoded, it must be invalid.

Transcoder

Although the encoding functions in RapidJSON are designed for JSON parsing/generation, user may abuse them for transcoding of non-JSON strings.

Here is an example for transcoding a string from UTF-8 to UTF-16:

```
#include "rapidjson/encodings.h"

using namespace rapidjson;

const char* s = "..."; // UTF-8 string
StringStream source(s);
GenericStringBuffer<UTF16<> > target;

bool hasError = false;
while (source.Peek() != '\0')
    if (!Transcoder::Transcode<UTF8<>, UTF16<> >(source, target)) {
        hasError = true;
        break;
    }

if (!hasError) {
    const wchar_t* t = target.GetString();
    // ...
}
```

You may also use `AutoUTF` and the associated streams for setting source/target encoding in runtime.

DOM

Document Object Model(DOM) is an in-memory representation of JSON for query and manipulation. The basic usage of DOM is described in [Tutorial](#). This section will describe some details and more advanced usages.

[TOC]

Template

In the tutorial, `Value` and `Document` was used. Similarly to `std::string`, these are actually `typedef` of template classes:

```
namespace rapidjson {

template <typename Encoding, typename Allocator = MemoryPoolAllocator<> >
class GenericValue {
    // ...
};

template <typename Encoding, typename Allocator = MemoryPoolAllocator<> >
class GenericDocument : public GenericValue<Encoding, Allocator> {
    // ...
};

typedef GenericValue<UTF8<> > Value;
typedef GenericDocument<UTF8<> > Document;

} // namespace rapidjson
```

User can customize these template parameters.

Encoding

The `Encoding` parameter specifies the encoding of JSON String value in memory. Possible options are `UTF8`, `UTF16`, `UTF32`. Note that, these 3 types are also template class. `UTF8<>` is `UTF8<char>`, which means using `char` to store the characters. You may refer to [Encoding](#) for details.

Suppose a Windows application would query localization strings stored in JSON files. Unicode-enabled functions in Windows use UTF-16 (wide character) encoding. No matter what encoding was used in JSON files, we can store the strings in UTF-16 in memory.

```
using namespace rapidjson;

typedef GenericDocument<UTF16<> > WDocument;
typedef GenericValue<UTF16<> > WValue;

FILE* fp = fopen("localization.json", "rb"); // non-Windows use "r"

char readBuffer[256];
FileReadStream bis(fp, readBuffer, sizeof(readBuffer));

AutoUTFInputStream<unsigned, FileReadStream> eis(bis); // wraps bis into eis

WDocument d;
d.ParseStream<0, AutoUTF<unsigned> >(eis);

const WValue locale(L"ja"); // Japanese

MessageBoxW(hwnd, d[locale].GetString(), L"Test", MB_OK);
```


Allocator

The `Allocator` defines which allocator class is used when allocating/deallocating memory for `Document` / `Value`. `Document` owns, or references to an `Allocator` instance. On the other hand, `Value` does not do so, in order to reduce memory consumption.

The default allocator used in `GenericDocument` is `MemoryPoolAllocator`. This allocator actually allocate memory sequentially, and cannot deallocate one by one. This is very suitable when parsing a JSON into a DOM tree.

Another allocator is `CrtAllocator`, of which CRT is short for C RunTime library. This allocator simply calls the standard `malloc()` / `realloc()` / `free()`. When there is a lot of add and remove operations, this allocator may be preferred. But this allocator is far less efficient than `MemoryPoolAllocator`.

Parsing

`Document` provides several functions for parsing. In below, (1) is the fundamental function, while the others are helpers which call (1).

```
using namespace rapidjson;

// (1) Fundamental
template <unsigned parseFlags, typename SourceEncoding, typename InputStream>
GenericDocument& GenericDocument::ParseStream(InputStream& is);

// (2) Using the same Encoding for stream
template <unsigned parseFlags, typename InputStream>
GenericDocument& GenericDocument::ParseStream(InputStream& is);

// (3) Using default parse flags
template <typename InputStream>
GenericDocument& GenericDocument::ParseStream(InputStream& is);

// (4) In situ parsing
template <unsigned parseFlags>
GenericDocument& GenericDocument::ParseInsitu(Ch* str);

// (5) In situ parsing, using default parse flags
GenericDocument& GenericDocument::ParseInsitu(Ch* str);

// (6) Normal parsing of a string
template <unsigned parseFlags, typename SourceEncoding>
GenericDocument& GenericDocument::Parse(const Ch* str);

// (7) Normal parsing of a string, using same Encoding of Document
template <unsigned parseFlags>
GenericDocument& GenericDocument::Parse(const Ch* str);

// (8) Normal parsing of a string, using default parse flags
GenericDocument& GenericDocument::Parse(const Ch* str);
```

The examples of [tutorial](#) uses (8) for normal parsing of string. The examples of [stream](#) uses the first three. *In situ* parsing will be described soon.

The `parseFlags` are combination of the following bit-flags:

Parse flags	Meaning
<code>kParseNoFlags</code>	No flag is set.
<code>kParseDefaultFlags</code>	Default parse flags. It is equal to macro <code>RAPIDJSON_PARSE_DEFAULT_FLAGS</code> , which is

<code>kParseDefaultFlags</code>	defined as <code>kParseNoFlags</code> .
<code>kParseInsituFlag</code>	In-situ(destructive) parsing.
<code>kParseValidateEncodingFlag</code>	Validate encoding of JSON strings.
<code>kParseIterativeFlag</code>	Iterative(constant complexity in terms of function call stack size) parsing.
<code>kParseStopWhenDoneFlag</code>	After parsing a complete JSON root from stream, stop further processing the rest of stream. When this flag is used, parser will not generate <code>kParseErrorDocumentRootNotSingular</code> error. Using this flag for parsing multiple JSONs in the same stream.
<code>kParseFullPrecisionFlag</code>	Parse number in full precision (slower). If this flag is not set, the normal precision (faster) is used. Normal precision has maximum 3 ULP error.

By using a non-type template parameter, instead of a function parameter, C++ compiler can generate code which is optimized for specified combinations, improving speed, and reducing code size (if only using a single specialization). The downside is the flags needed to be determined in compile-time.

The `SourceEncoding` parameter defines what encoding is in the stream. This can be differed to the `Encoding` of the `Document` . See [Transcoding and Validation](#) section for details.

And the `InputStream` is type of input stream.

Parse Error

When the parse processing succeeded, the `Document` contains the parse results. When there is an error, the original DOM is *unchanged*. And the error state of parsing can be obtained by `bool HasParseError()` , `ParseErrorCode GetParseError()` and `size_t GetParseOffset()` .

Parse Error Code	Description
<code>kParseErrorNone</code>	No error.
<code>kParseErrorDocumentEmpty</code>	The document is empty.
<code>kParseErrorDocumentRootNotSingular</code>	The document root must not follow by other values.
<code>kParseErrorValueInvalid</code>	Invalid value.
<code>kParseErrorObjectMissName</code>	Missing a name for object member.
<code>kParseErrorObjectMissColon</code>	Missing a colon after a name of object member.
<code>kParseErrorObjectMissCommaOrCurlyBracket</code>	Missing a comma or <code>}</code> after an object member.
<code>kParseErrorArrayMissCommaOrSquareBracket</code>	Missing a comma or <code>]</code> after an array element.
<code>kParseErrorStringUnicodeEscapeInvalidHex</code>	Incorrect hex digit after <code>\\u</code> escape in string.
<code>kParseErrorStringUnicodeSurrogateInvalid</code>	The surrogate pair in string is invalid.
<code>kParseErrorStringEscapeInvalid</code>	Invalid escape character in string.
<code>kParseErrorStringMissQuotationMark</code>	Missing a closing quotation mark in string.
<code>kParseErrorStringInvalidEncoding</code>	Invalid encoding in string.
<code>kParseErrorNumberTooBig</code>	Number too big to be stored in <code>double</code> .
<code>kParseErrorNumberMissFraction</code>	Miss fraction part in number.
<code>kParseErrorNumberMissExponent</code>	Miss exponent in number.

of line number.

To get an error message, RapidJSON provided a English messages in `rapidjson/error/en.h` . User can customize it for other locales, or use a custom localization system.

Here shows an example of parse error handling.

```
#include "rapidjson/document.h"
#include "rapidjson/error/en.h"

// ...
Document d;
if (d.Parse(json).HasParseError()) {
    fprintf(stderr, "\nError(offset %u): %s\n",
        (unsigned)reader.GetErrorOffset(),
        GetParseError_En(reader.GetParseErrorCode()));
    // ...
}
```

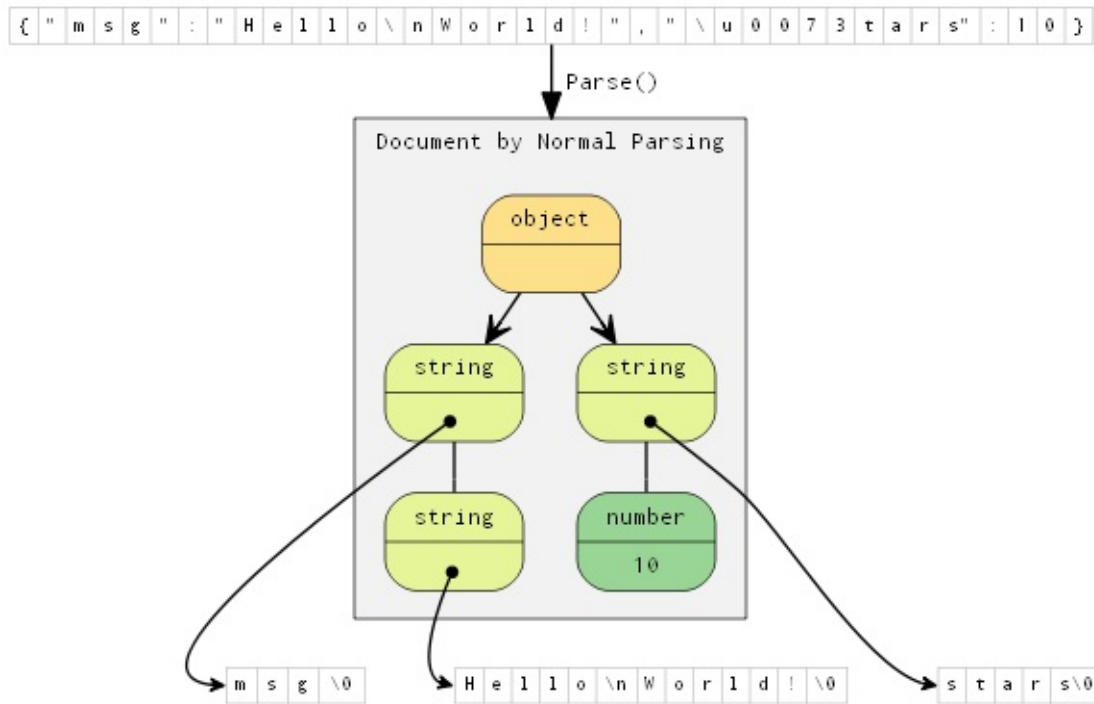
In Situ Parsing

From [Wikipedia](#):

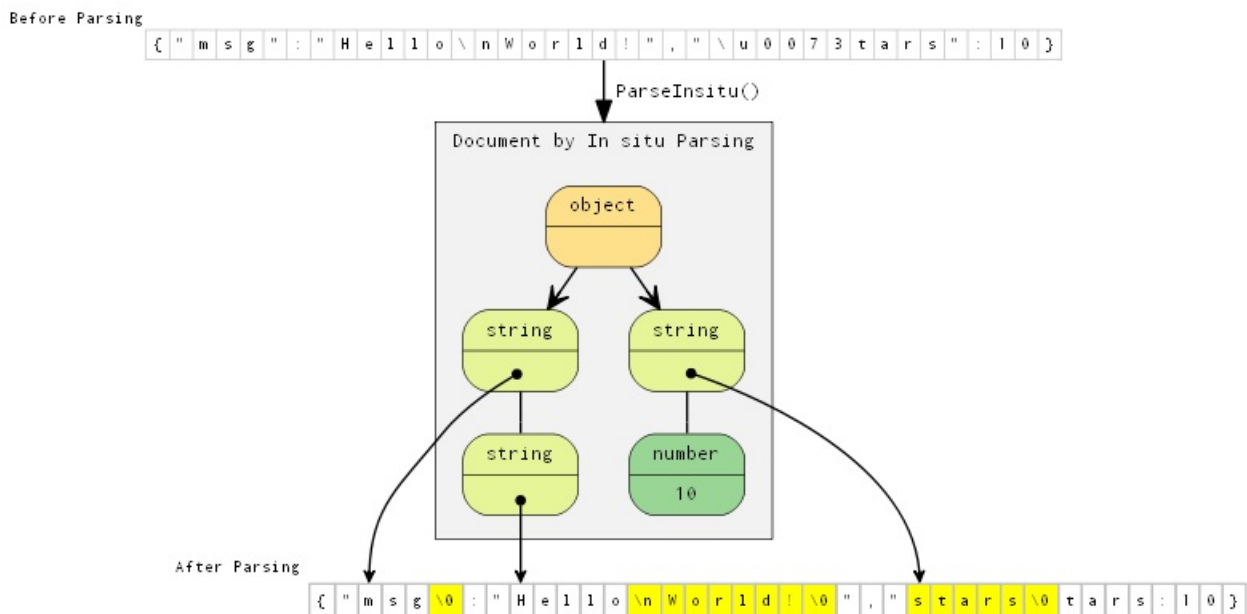
In situ ... is a Latin phrase that translates literally to "on site" or "in position". It means "locally", "on site", "on the premises" or "in place" to describe an event where it takes place, and is used in many different contexts. ... (In computer science) An algorithm is said to be an *in situ* algorithm, or in-place algorithm, if the extra amount of memory required to execute the algorithm is $O(1)$, that is, does not exceed a constant no matter how large the input. For example, heapsort is an *in situ* sorting algorithm.

In normal parsing process, a large overhead is to decode JSON strings and copy them to other buffers. *In situ* parsing decodes those JSON string at the place where it is stored. It is possible in JSON because the length of decoded string is always shorter than or equal to the one in JSON. In this context, decoding a JSON string means to process the escapes, such as `"\n"` , `"\u1234"` , etc., and add a null terminator (`'\0'`) at the end of string.

The following diagrams compare normal and *in situ* parsing. The JSON string values contain pointers to the decoded string.



In normal parsing, the decoded string are copied to freshly allocated buffers. `"\n"` (2 characters) is decoded as `"\n"` (1 character). `"\u0073"` (6 characters) is decoded as `"s"` (1 character).



In situ parsing just modified the original JSON. Updated characters are highlighted in the diagram. If the JSON string does not contain escape character, such as `"msg"`, the parsing process merely replace the closing double quotation mark with a null character.

Since *in situ* parsing modify the input, the parsing API needs `char*` instead of `const char*`.

```
// Read whole file into a buffer
FILE* fp = fopen("test.json", "r");
fseek(fp, 0, SEEK_END);
size_t filesize = (size_t)ftell(fp);
fseek(fp, 0, SEEK_SET);
char* buffer = (char*)malloc(filesize + 1);
size_t readLength = fread(buffer, 1, filesize, fp);
```

```

buffer[readLength] = '\0';
fclose(fp);

// In situ parsing the buffer into d, buffer will also be modified
Document d;
d.ParseIn situ(buffer);

// Query/manipulate the DOM here...

free(buffer);
// Note: At this point, d may have dangling pointers pointed to the deallocated buffer.

```

The JSON strings are marked as const-string. But they may not be really "constant". The life cycle of it depends on the JSON buffer.

In situ parsing minimizes allocation overheads and memory copying. Generally this improves cache coherence, which is an important factor of performance in modern computer.

There are some limitations of *in situ* parsing:

1. The whole JSON is in memory.
2. The source encoding in stream and target encoding in document must be the same.
3. The buffer need to be retained until the document is no longer used.
4. If the DOM need to be used for long period after parsing, and there are few JSON strings in the DOM, retaining the buffer may be a memory waste.

In situ parsing is mostly suitable for short-term JSON that only need to be processed once, and then be released from memory. In practice, these situation is very common, for example, deserializing JSON to C++ objects, processing web requests represented in JSON, etc.

Transcoding and Validation

RapidJSON supports conversion between Unicode formats (officially termed UCS Transformation Format) internally. During DOM parsing, the source encoding of the stream can be different from the encoding of the DOM. For example, the source stream contains a UTF-8 JSON, while the DOM is using UTF-16 encoding. There is an example code in [EncodedInputStream](#).

When writing a JSON from DOM to output stream, transcoding can also be used. An example is in [EncodedOutputStream](#).

During transcoding, the source string is decoded to into Unicode code points, and then the code points are encoded in the target format. During decoding, it will validate the byte sequence in the source string. If it is not a valid sequence, the parser will be stopped with `kParseErrorStringInvalidEncoding` error.

When the source encoding of stream is the same as encoding of DOM, by default, the parser will *not* validate the sequence. User may use `kParseValidateEncodingFlag` to force validation.

Techniques

Some techniques about using DOM API is discussed here.

DOM as SAX Event Publisher

In RapidJSON, stringifying a DOM with `Writer` may be look a little bit weird.

```
// ...
Writer<StringBuffer> writer(buffer);
d.Accept(writer);
```

Actually, `Value::Accept()` is responsible for publishing SAX events about the value to the handler. With this design, `Value` and `Writer` are decoupled. `Value` can generate SAX events, and `Writer` can handle those events.

User may create custom handlers for transforming the DOM into other formats. For example, a handler which converts the DOM into XML.

For more about SAX events and handler, please refer to [SAX](#).

User Buffer

Some applications may try to avoid memory allocations whenever possible.

`MemoryPoolAllocator` can support this by letting user to provide a buffer. The buffer can be on the program stack, or a "scratch buffer" which is statically allocated (a static/global array) for storing temporary data.

`MemoryPoolAllocator` will use the user buffer to satisfy allocations. When the user buffer is used up, it will allocate a chunk of memory from the base allocator (by default the `CrtAllocator`).

Here is an example of using stack memory. The first allocator is for storing values, while the second allocator is for storing temporary data during parsing.

```
typedef GenericDocument<UTF8<>, MemoryPoolAllocator<>, MemoryPoolAllocator<>> DocumentType;
char valueBuffer[4096];
char parseBuffer[1024];
MemoryPoolAllocator<> valueAllocator(valueBuffer, sizeof(valueBuffer));
MemoryPoolAllocator<> parseAllocator(parseBuffer, sizeof(parseBuffer));
DocumentType d(&valueAllocator, sizeof(parseBuffer), &parseAllocator);
d.Parse(json);
```

If the total size of allocation is less than 4096+1024 bytes during parsing, this code does not invoke any heap allocation (via `new` or `malloc()`) at all.

User can query the current memory consumption in bytes via `MemoryPoolAllocator::Size()`. And then user can determine a suitable size of user buffer.

SAX

The term "SAX" originated from [Simple API for XML](#). We borrowed this term for JSON parsing and generation.

In RapidJSON, `Reader` (typedef of `GenericReader<...>`) is the SAX-style parser for JSON, and `Writer` (typedef of `GenericWriter<...>`) is the SAX-style generator for JSON.

[TOC]

Reader

`Reader` parses a JSON from a stream. While it reads characters from the stream, it analyze the characters according to the syntax of JSON, and publish events to a handler.

For example, here is a JSON.

```
{
  "hello": "world",
  "t": true ,
  "f": false,
  "n": null,
  "i": 123,
  "pi": 3.1416,
  "a": [1, 2, 3, 4]
}
```

While a `Reader` parses this JSON, it publishes the following events to the handler sequentially:

```
StartObject()
Key("hello", 5, true)
String("world", 5, true)
Key("t", 1, true)
Bool(true)
Key("f", 1, true)
Bool(false)
Key("n", 1, true)
Null()
Key("i")
UInt(123)
Key("pi")
Double(3.1416)
Key("a")
StartArray()
UInt(1)
UInt(2)
UInt(3)
UInt(4)
EndArray(4)
EndObject(7)
```

These events can be easily matched with the JSON, except some event parameters need further explanation. Let's see the `simplereader` example which produces exactly the same output as above:

```
#include "rapidjson/reader.h"
#include <iostream>

using namespace rapidjson;
using namespace std;
```

```

struct MyHandler {
    bool Null() { cout << "Null()" << endl; return true; }
    bool Bool(bool b) { cout << "Bool(" << boolalpha << b << ")" << endl; return true; }
    bool Int(int i) { cout << "Int(" << i << ")" << endl; return true; }
    bool Uint(unsigned u) { cout << "Uint(" << u << ")" << endl; return true; }
    bool Int64(int64_t i) { cout << "Int64(" << i << ")" << endl; return true; }
    bool Uint64(uint64_t u) { cout << "Uint64(" << u << ")" << endl; return true; }
    bool Double(double d) { cout << "Double(" << d << ")" << endl; return true; }
    bool String(const char* str, SizeType length, bool copy) {
        cout << "String(" << str << ", " << length << ", " << boolalpha << copy << ")" << endl;
        return true;
    }
    bool StartObject() { cout << "StartObject()" << endl; return true; }
    bool Key(const char* str, SizeType length, bool copy) {
        cout << "Key(" << str << ", " << length << ", " << boolalpha << copy << ")" << endl;
        return true;
    }
    bool EndObject(SizeType memberCount) { cout << "EndObject(" << memberCount << ")" << endl; return true; }
    bool StartArray() { cout << "StartArray()" << endl; return true; }
    bool EndArray(SizeType elementCount) { cout << "EndArray(" << elementCount << ")" << endl; return true; }
};

void main() {
    const char json[] = " { \"hello\" : \"world\", \"t\" : true , \"f\" : false, \"n\": null, \"i\":123, \"pi\": 3.1416 } ";

    MyHandler handler;
    Reader reader;
    StringStream ss(json);
    reader.Parse(ss, handler);
}

```

Note that, RapidJSON uses template to statically bind the `Reader` type and the handler type, instead of using class with virtual functions. This paradigm can improve the performance by inlining functions.

Handler

As the previous example showed, user needs to implement a handler, which consumes the events (function calls) from `Reader`. The handler must contain the following member functions.

```

class Handler {
    bool Null();
    bool Bool(bool b);
    bool Int(int i);
    bool Uint(unsigned i);
    bool Int64(int64_t i);
    bool Uint64(uint64_t i);
    bool Double(double d);
    bool String(const Ch* str, SizeType length, bool copy);
    bool StartObject();
    bool Key(const Ch* str, SizeType length, bool copy);
    bool EndObject(SizeType memberCount);
    bool StartArray();
    bool EndArray(SizeType elementCount);
};

```

`Null()` is called when the `Reader` encounters a JSON null value.

`Bool(bool)` is called when the `Reader` encounters a JSON true or false value.

When the `Reader` encounters a JSON number, it chooses a suitable C++ type mapping. And then it calls *one* function out of `Int(int)`, `Uint(unsigned)`, `Int64(int64_t)`, `Uint64(uint64_t)` and `Double(double)`.

`String(const char* str, SizeType length, bool copy)` is called when the `Reader` encounters a string. The first parameter is

pointer to the string. The second parameter is the length of the string (excluding the null terminator). Note that RapidJSON supports null character `'\0'` inside a string. If such situation happens, `strlen(str) < length`. The last `copy` indicates whether the handler needs to make a copy of the string. For normal parsing, `copy = true`. Only when *insitu* parsing is used, `copy = false`. And beware that, the character type depends on the target encoding, which will be explained later.

When the `Reader` encounters the beginning of an object, it calls `StartObject()`. An object in JSON is a set of name-value pairs. If the object contains members it first calls `Key()` for the name of member, and then calls functions depending on the type of the value. These calls of name-value pairs repeats until calling `EndObject(SizeType memberCount)`. Note that the `memberCount` parameter is just an aid for the handler, user may not need this parameter.

Array is similar to object but simpler. At the beginning of an array, the `Reader` calls `BeginArray()`. If there is elements, it calls functions according to the types of element. Similarly, in the last call `EndArray(SizeType elementCount)`, the parameter `elementCount` is just an aid for the handler.

Every handler functions returns a `bool`. Normally it should returns `true`. If the handler encounters an error, it can return `false` to notify event publisher to stop further processing.

For example, when we parse a JSON with `Reader` and the handler detected that the JSON does not conform to the required schema, then the handler can return `false` and let the `Reader` stop further parsing. And the `Reader` will be in error state with error code `kParseErrorTermination`.

GenericReader

As mentioned before, `Reader` is a typedef of a template class `GenericReader`:

```
namespace rapidjson {

template <typename SourceEncoding, typename TargetEncoding, typename Allocator = MemoryPoolAllocator<> >
class GenericReader {
    // ...
};

typedef GenericReader<UTF8<>, UTF8<> > Reader;

} // namespace rapidjson
```

The `Reader` uses UTF-8 as both source and target encoding. The source encoding means the encoding in the JSON stream. The target encoding means the encoding of the `str` parameter in `String()` calls. For example, to parse a UTF-8 stream and outputs UTF-16 string events, you can define a reader by:

```
GenericReader<UTF8<>, UTF16<> > reader;
```

Note that, the default character type of `UTF16` is `wchar_t`. So this `reader` needs to call `String(const wchar_t*, SizeType, bool)` of the handler.

The third template parameter `Allocator` is the allocator type for internal data structure (actually a stack).

Parsing

The one and only one function of `Reader` is to parse JSON.

```
template <unsigned parseFlags, typename InputStream, typename Handler>
bool Parse(InputStream& is, Handler& handler);
```

```
// with parseFlags = kDefaultParseFlags
template <typename InputStream, typename Handler>
bool Parse(InputStream& is, Handler& handler);
```

If an error occurs during parsing, it will return `false`. User can also calls `bool HasParseError()`, `ParseErrorCode GetParseErrorCode()` and `size_t GetErrorOffset()` to obtain the error states. Actually `Document` uses these `Reader` functions to obtain parse errors. Please refer to [DOM](#) for details about parse error.

Writer

`Reader` converts (pares) JSON into events. `Writer` does exactly the opposite. It converts events into JSON.

`Writer` is very easy to use. If your application only need to converts some data into JSON, it may be a good choice to use `Writer` directly, instead of building a `Document` and then stringifying it with a `Writer`.

In `simplewriter` example, we do exactly the reverse of `simplereader`.

```
#include "rapidjson/writer.h"
#include "rapidjson/stringbuffer.h"
#include <iostream>

using namespace rapidjson;
using namespace std;

void main() {
    StringBuffer s;
    Writer<StringBuffer> writer(s);

    writer.StartObject();
    writer.Key("hello");
    writer.String("world");
    writer.Key("t");
    writer.Bool(true);
    writer.Key("f");
    writer.Bool(false);
    writer.Key("n");
    writer.Null();
    writer.Key("i");
    writer.Uint(123);
    writer.Key("pi");
    writer.Double(3.1416);
    writer.Key("a");
    writer.StartArray();
    for (unsigned i = 0; i < 4; i++)
        writer.Uint(i);
    writer.EndArray();
    writer.EndObject();

    cout << s.GetString() << endl;
}
```

```
{"hello":"world", "t":true, "f":false, "n":null, "i":123, "pi":3.1416, "a":[0,1,2,3]}
```

There are two `String()` and `Key()` overloads. One is the same as defined in handler concept with 3 parameters. It can handle string with null characters. Another one is the simpler version used in the above example.

Note that, the example code does not pass any parameters in `EndArray()` and `EndObject()`. An `SizeType` can be passed but it will be simply ignored by `Writer`.

You may doubt that, why not just using `sprintf()` or `std::stringstream` to build a JSON?

There are various reasons:

1. `Writer` must output a well-formed JSON. If there is incorrect event sequence (e.g. `Int()` just after `StartObject()`), it generates assertion fail in debug mode.
2. `Writer::String()` can handle string escaping (e.g. converting code point `U+000A` to `\n`) and Unicode transcoding.
3. `Writer` handles number output consistently.
4. `Writer` implements the event handler concept. It can be used to handle events from `Reader`, `Document` or other event publisher.
5. `Writer` can be optimized for different platforms.

Anyway, using `Writer` API is even simpler than generating a JSON by ad hoc methods.

Template

`Writer` has a minor design difference to `Reader`. `Writer` is a template class, not a typedef. There is no `GenericWriter`. The following is the declaration.

```
namespace rapidjson {

template<typename OutputStream, typename SourceEncoding = UTF8<>, typename TargetEncoding = UTF8<>, typename Allocator
class Writer {
public:
    Writer(OutputStream& os, Allocator* allocator = 0, size_t levelDepth = kDefaultLevelDepth)
    // ...
};

} // namespace rapidjson
```

The `OutputStream` template parameter is the type of output stream. It cannot be deduced and must be specified by user.

The `SourceEncoding` template parameter specifies the encoding to be used in `String(const Ch*, ...)`.

The `TargetEncoding` template parameter specifies the encoding in the output stream.

The last one, `Allocator` is the type of allocator, which is used for allocating internal data structure (a stack).

Besides, the constructor of `Writer` has a `levelDepth` parameter. This parameter affects the initial memory allocated for storing information per hierarchy level.

PrettyWriter

While the output of `Writer` is the most condensed JSON without white-spaces, suitable for network transfer or storage, it is not easily readable by human.

Therefore, RapidJSON provides a `PrettyWriter`, which adds indentation and line feeds in the output.

The usage of `PrettyWriter` is exactly the same as `Writer`, expect that `PrettyWriter` provides a `SetIndent(Ch indentChar, unsigned indentCharCount)` function. The default is 4 spaces.

Completeness and Reset

A `Writer` can only output a single JSON, which can be any JSON type at the root. Once the singular event for root (e.g. `String()`), or the last matching `EndObject()` or `EndArray()` event, is handled, the output JSON is well-formed and

complete. User can detect this state by calling `Writer::IsComplete()`.

When a JSON is complete, the `Writer` cannot accept any new events. Otherwise the output will be invalid (i.e. having more than one root). To reuse the `Writer` object, user can call `Writer::Reset(OutputStream& os)` to reset all internal states of the `Writer` with a new output stream.

Techniques

Parsing JSON to Custom Data Structure

`Document`'s parsing capability is completely based on `Reader`. Actually `Document` is a handler which receives events from a reader to build a DOM during parsing.

User may uses `Reader` to build other data structures directly. This eliminates building of DOM, thus reducing memory and improving performance.

In the following `messagereader` example, `ParseMessages()` parses a JSON which should be an object with key-string pairs.

```
#include "rapidjson/reader.h"
#include "rapidjson/error/en.h"
#include <iostream>
#include <string>
#include <map>

using namespace std;
using namespace rapidjson;

typedef map<string, string> MessageMap;

struct MessageHandler
: public BaseReaderHandler<UTF8<>, MessageHandler> {
    MessageHandler() : state_(kExpectObjectStart) {}
}

bool StartObject() {
    switch (state_) {
        case kExpectObjectStart:
            state_ = kExpectNameOrObjectEnd;
            return true;
        default:
            return false;
    }
}

bool String(const char* str, SizeType length, bool) {
    switch (state_) {
        case kExpectNameOrObjectEnd:
            name_ = string(str, length);
            state_ = kExpectValue;
            return true;
        case kExpectValue:
            messages_.insert(MessageMap::value_type(name_, string(str, length)));
            state_ = kExpectNameOrObjectEnd;
            return true;
        default:
            return false;
    }
}

bool EndObject(SizeType) { return state_ == kExpectNameOrObjectEnd; }

bool Default() { return false; } // All other events are invalid.

MessageMap messages_;
enum State {
    kExpectObjectStart,
```

```

        kExpectNameOrObjectEnd,
        kExpectValue,
    }state_;
    std::string name_;
};

void ParseMessages(const char* json, MessageMap& messages) {
    Reader reader;
    MessageHandler handler;
    stringstream ss(json);
    if (reader.Parse(ss, handler))
        messages.swap(handler.messages_); // Only change it if success.
    else {
        ParseErrorCode e = reader.GetParseErrorCode();
        size_t o = reader.GetErrorOffset();
        cout << "Error: " << GetParseError_En(e) << endl;;
        cout << " at offset " << o << " near '" << string(json).substr(o, 10) << "...'" << endl;
    }
}

int main() {
    MessageMap messages;

    const char* json1 = "{ \"greeting\" : \"Hello!\", \"farewell\" : \"bye-bye!\" }";
    cout << json1 << endl;
    ParseMessages(json1, messages);

    for (MessageMap::const_iterator itr = messages.begin(); itr != messages.end(); ++itr)
        cout << itr->first << ": " << itr->second << endl;

    cout << endl << "Parse a JSON with invalid schema." << endl;
    const char* json2 = "{ \"greeting\" : \"Hello!\", \"farewell\" : \"bye-bye!\", \"foo\" : {} }";
    cout << json2 << endl;
    ParseMessages(json2, messages);

    return 0;
}

```

```

{ "greeting" : "Hello!", "farewell" : "bye-bye!" }
farewell: bye-bye!
greeting: Hello!

Parse a JSON with invalid schema.
{ "greeting" : "Hello!", "farewell" : "bye-bye!", "foo" : {} }
Error: Terminate parsing due to Handler error.
at offset 59 near '}' }...'

```

The first JSON (`json1`) was successfully parsed into `MessageMap` . Since `MessageMap` is a `std::map` , the printing order are sorted by the key. This order is different from the JSON's order.

In the second JSON (`json2`), `foo` 's value is an empty object. As it is an object, `MessageHandler::StartObject()` will be called. However, at that moment `state_ = kExpectValue` , so that function returns `false` and cause the parsing process be terminated. The error code is `kParseErrorTermination` .

Filtering of JSON

As mentioned earlier, `Writer` can handle the events published by `Reader` . `condense` example simply set a `Writer` as handler of a `Reader` , so it can remove all white-spaces in JSON. `pretty` example uses the same relationship, but replacing `Writer` by `PrettyWriter` . So `pretty` can be used to reformat a JSON with indentation and line feed.

Actually, we can add intermediate layer(s) to filter the contents of JSON via these SAX-style API. For example, `capitalize` example capitalize all strings in a JSON.

```
#include "rapidjson/reader.h"
```

```

#include "rapidjson/writer.h"
#include "rapidjson/filereadstream.h"
#include "rapidjson/filewritestream.h"
#include "rapidjson/error/en.h"
#include <vector>
#include <cctype>

using namespace rapidjson;

template<typename OutputHandler>
struct CapitalizeFilter {
    CapitalizeFilter(OutputHandler& out) : out_(out), buffer_() {
    }

    bool Null() { return out_.Null(); }
    bool Bool(bool b) { return out_.Bool(b); }
    bool Int(int i) { return out_.Int(i); }
    bool Uint(unsigned u) { return out_.Uint(u); }
    bool Int64(int64_t i) { return out_.Int64(i); }
    bool Uint64(uint64_t u) { return out_.Uint64(u); }
    bool Double(double d) { return out_.Double(d); }
    bool String(const char* str, SizeType length, bool) {
        buffer_.clear();
        for (SizeType i = 0; i < length; i++)
            buffer_.push_back(std::toupper(str[i]));
        return out_.String(&buffer_.front(), length, true); // true = output handler need to copy the string
    }
    bool StartObject() { return out_.StartObject(); }
    bool Key(const char* str, SizeType length, bool copy) { return String(str, length, copy); }
    bool EndObject(SizeType memberCount) { return out_.EndObject(memberCount); }
    bool StartArray() { return out_.StartArray(); }
    bool EndArray(SizeType elementCount) { return out_.EndArray(elementCount); }

    OutputHandler& out_;
    std::vector<char> buffer_;
};

int main(int, char*[]) {
    // Prepare JSON reader and input stream.
    Reader reader;
    char readBuffer[65536];
    FileReadStream is(stdin, readBuffer, sizeof(readBuffer));

    // Prepare JSON writer and output stream.
    char writeBuffer[65536];
    FileWriteStream os(stdout, writeBuffer, sizeof(writeBuffer));
    Writer<FileWriteStream> writer(os);

    // JSON reader parse from the input stream and let writer generate the output.
    CapitalizeFilter<Writer<FileWriteStream>> filter(writer);
    if (!reader.Parse(is, filter)) {
        fprintf(stderr, "\nError(%u): %s\n", (unsigned)reader.GetErrorOffset(), GetParseError_En(reader.GetParseErrorCo
        return 1;
    }

    return 0;
}

```

Note that, it is incorrect to simply capitalize the JSON as a string. For example:

```
["Hello\nWorld"]
```

Simply capitalizing the whole JSON would contain incorrect escape character:

```
["HELLO\nWORLD"]
```

The correct result by `capitalize` :

```
[ "HELLO\nWORLD" ]
```

More complicated filters can be developed. However, since SAX-style API can only provide information about a single event at a time, user may need to book-keeping the contextual information (e.g. the path from root value, storage of other related values). Some processing may be easier to be implemented in DOM than SAX.

Performance

There is a [native JSON benchmark collection](#) which evaluates speed, memory usage and code size of various operations among 20 JSON libraries.

The old performance article for RapidJSON 0.1 is provided [here](#).

Additionally, you may refer to the following third-party benchmarks.

Third-party benchmarks

- [Basic benchmarks for miscellaneous C++ JSON parsers and generators](#) by Mateusz Loskot (Jun 2013)
 - [casablanca](#)
 - [json_spirit](#)
 - [jsoncpp](#)
 - [libjson](#)
 - [rapidjson](#)
 - [QJsonDocument](#)
- [JSON Parser Benchmarking](#) by Chad Austin (Jan 2013)
 - [sajson](#)
 - [rapidjson](#)
 - [vjson](#)
 - [YAJL](#)
 - [Jansson](#)

Internals

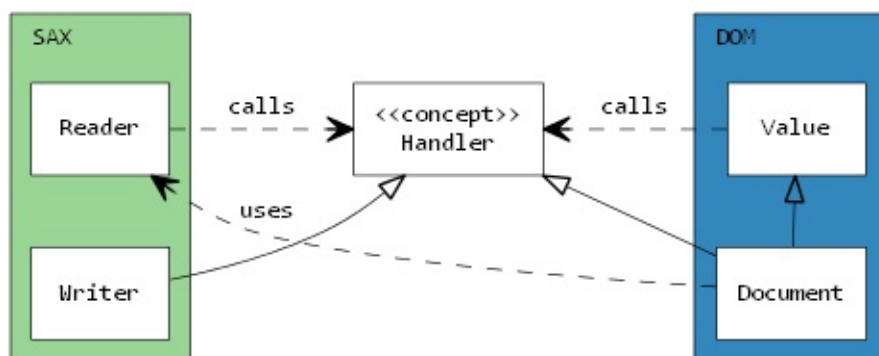
This section records some design and implementation details.

[TOC]

Architecture

SAX and DOM

The basic relationships of SAX and DOM is shown in the following UML diagram.



The core of the relationship is the `Handler` concept. From the SAX side, `Reader` parses a JSON from a stream and publish events to a `Handler`. `Writer` implements the `Handler` concept to handle the same set of events. From the DOM side, `Document` implements the `Handler` concept to build a DOM according to the events. `Value` supports a `Value::Accept(Handler&)` function, which traverses the DOM to publish events.

With this design, SAX is not dependent on DOM. Even `Reader` and `Writer` have no dependencies between them. This provides flexibility to chain event publisher and handlers. Besides, `Value` does not depends on SAX as well. So, in addition to stringify a DOM to JSON, user may also stringify it to a XML writer, or do anything else.

Utility Classes

Both SAX and DOM APIs depends on 3 additional concepts: `Allocator`, `Encoding` and `Stream`. Their inheritance hierarchy is shown as below.



Value

`Value` (actually a typedef of `GenericValue<UTF8<>>`) is the core of DOM API. This section describes the design of it.

Data Layout

`Value` is a [variant type](#). In RapidJSON's context, an instance of `Value` can contain 1 of 6 JSON value types. This is possible by using `union`. Each `Value` contains two members: `union Data data_` and a `unsigned flags_`. The `flags_` indicates the JSON type, and also additional information.

The following tables show the data layout of each type. The 32-bit/64-bit columns indicates the size of the field in bytes.

Null		32-bit	64-bit
(unused)		4	8
(unused)		4	4
(unused)		4	4
<code>unsigned flags_</code>	<code>kNullType kNullFlag</code>	4	4

Bool		32-bit	64-bit
(unused)		4	8
(unused)		4	4
(unused)		4	4
<code>unsigned flags_</code>	<code>kBoolType (either kTrueFlag or kFalseFlag)</code>	4	4

String		32-bit	64-bit
<code>Ch* str</code>	Pointer to the string (may own)	4	8
<code>SizeType length</code>	Length of string	4	4
(unused)		4	4
<code>unsigned flags_</code>	<code>kStringType kStringFlag ...</code>	4	4

Object		32-bit	64-bit
<code>Member* members</code>	Pointer to array of members (owned)	4	8
<code>SizeType size</code>	Number of members	4	4
<code>SizeType capacity</code>	Capacity of members	4	4
<code>unsigned flags_</code>	<code>kObjectType kObjectFlag</code>	4	4

Array		32-bit	64-bit
<code>Value* values</code>	Pointer to array of values (owned)	4	8
<code>SizeType size</code>	Number of values	4	4
<code>SizeType capacity</code>	Capacity of values	4	4
<code>unsigned flags_</code>	<code>kArrayType kArrayFlag</code>	4	4

Number (Int)		32-bit	64-bit
<code>int i</code>	32-bit signed integer	4	4

(zero padding)	0	4	4
(unused)		4	8
unsigned flags_	kNumberType kNumberFlag kIntFlag kInt64Flag ...	4	4

Number (UInt)		32-bit	64-bit
unsigned u	32-bit unsigned integer	4	4
(zero padding)	0	4	4
(unused)		4	8
unsigned flags_	kNumberType kNumberFlag kUIntFlag kUInt64Flag ...	4	4

Number (Int64)		32-bit	64-bit
int64_t i64	64-bit signed integer	8	8
(unused)		4	8
unsigned flags_	kNumberType kNumberFlag kInt64Flag ...	4	4

Number (UInt64)		32-bit	64-bit
uint64_t i64	64-bit unsigned integer	8	8
(unused)		4	8
unsigned flags_	kNumberType kNumberFlag kInt64Flag ...	4	4

Number (Double)		32-bit	64-bit
uint64_t i64	Double precision floating-point	8	8
(unused)		4	8
unsigned flags_	kNumberType kNumberFlag kDoubleFlag	4	4

Here are some notes:

- To reduce memory consumption for 64-bit architecture, `SizeType` is typedef as `unsigned` instead of `size_t`.
- Zero padding for 32-bit number may be placed after or before the actual type, according to the endianness. This makes possible for interpreting a 32-bit integer as a 64-bit integer, without any conversion.
- An `Int` is always an `Int64`, but the converse is not always true.

Flags

The 32-bit `flags_` contains both JSON type and other additional information. As shown in the above tables, each JSON type contains redundant `kXXType` and `kXXFlag`. This design is for optimizing the operation of testing bit-flags (`IsNumber()`) and obtaining a sequential number for each type (`GetType()`).

String has two optional flags. `kCopyFlag` means that the string owns a copy of the string. `kInlineStrFlag` means using [Short-String Optimization](#).

Number is a bit more complicated. For normal integer values, it can contains `kIntFlag`, `kUIntFlag`, `kInt64Flag` and/or `kUInt64Flag`, according to the range of the integer. For numbers with fraction, and integers larger than 64-bit range, they will be stored as `double` with `kDoubleFlag`.

Short-String Optimization

Kosta (@Kosta-Github) provided a very neat short-string optimization. The optimization idea is given as follow. Excluding the `flags_`, a `Value` has 12 or 16 bytes (32-bit or 64-bit) for storing actual data. Instead of storing a pointer to a string, it is possible to store short strings in these space internally. For encoding with 1-byte character type (e.g. `char`), it can store maximum 11 or 15 characters string inside the `Value` type.

ShortString (Ch=char)		32-bit	64-bit
<code>Ch str[MaxChars]</code>	String buffer	11	15
<code>Ch invLength</code>	MaxChars - Length	1	1
<code>unsigned flags_</code>	<code>kStringType kStringFlag ...</code>	4	4

A special technique is applied. Instead of storing the length of string directly, it stores (MaxChars - length). This make it possible to store 11 characters with trailing `\0`.

This optimization can reduce memory usage for copy-string. It can also improve cache-coherence thus improve runtime performance.

Allocator

`Allocator` is a concept in RapidJSON:

```
concept Allocator {
    static const bool kNeedFree;    //!< Whether this allocator needs to call Free().

    // Allocate a memory block.
    // \param size of the memory block in bytes.
    // \returns pointer to the memory block.
    void* Malloc(size_t size);

    // Resize a memory block.
    // \param originalPtr The pointer to current memory block. Null pointer is permitted.
    // \param originalSize The current size in bytes. (Design issue: since some allocator may not book-keep this, expli
    // \param newSize the new size in bytes.
    void* Realloc(void* originalPtr, size_t originalSize, size_t newSize);

    // Free a memory block.
    // \param pointer to the memory block. Null pointer is permitted.
    static void Free(void *ptr);
};
```

Note that `Malloc()` and `Realloc()` are member functions but `Free()` is static member function.

MemoryPoolAllocator

`MemoryPoolAllocator` is the default allocator for DOM. It allocate but do not free memory. This is suitable for building a DOM tree.

Internally, it allocates chunks of memory from the base allocator (by default `CrtAllocator`) and stores the chunks as a singly linked list. When user requests an allocation, it allocates memory from the following order:

1. User supplied buffer if it is available. (See [User Buffer section in DOM](#))
2. If user supplied buffer is full, use the current memory chunk.

3. If the current block is full, allocate a new block of memory.

Parsing Optimization

Skip Whitespaces with SIMD

When parsing JSON from a stream, the parser need to skip 4 whitespace characters:

1. Space (`U+0020`)
2. Character Tabulation (`U+000B`)
3. Line Feed (`U+000A`)
4. Carriage Return (`U+000D`)

A simple implementation will be simply:

```
void SkipWhitespace(InputStream& s) {
    while (s.Peek() == ' ' || s.Peek() == '\n' || s.Peek() == '\r' || s.Peek() == '\t')
        s.Take();
}
```

However, this requires 4 comparisons and a few branching for each character. This was found to be a hot spot.

To accelerate this process, SIMD was applied to compare 16 characters with 4 white spaces for each iteration. Currently RapidJSON only supports SSE2 and SSE4.2 instructions for this. And it is only activated for UTF-8 memory streams, including string stream or *in situ* parsing.

To enable this optimization, need to define `RAPIDJSON_SSE2` OR `RAPIDJSON_SSE42` before including `rapidjson.h`. Some compilers can detect the setting, as in `perftest.h`:

```
// __SSE2__ and __SSE4_2__ are recognized by gcc, clang, and the Intel compiler.
// We use -march=native with gmake to enable -msse2 and -msse4.2, if supported.
#ifdef __SSE4_2__
#   define RAPIDJSON_SSE42
#elif defined(__SSE2__)
#   define RAPIDJSON_SSE2
#endif
```

Note that, these are compile-time settings. Running the executable on a machine without such instruction set support will make it crash.

Local Stream Copy

During optimization, it is found that some compilers cannot localize some member data access of streams into local variables or registers. Experimental results show that for some stream types, making a copy of the stream and used it in inner-loop can improve performance. For example, the actual (non-SIMD) implementation of `SkipWhitespace()` is implemented as:

```
template<typename InputStream>
void SkipWhitespace(InputStream& is) {
    internal::StreamLocalCopy<InputStream> copy(is);
    InputStream& s(copy.s);

    while (s.Peek() == ' ' || s.Peek() == '\n' || s.Peek() == '\r' || s.Peek() == '\t')
```

```
s.Take();
}
```

Depending on the traits of stream, `StreamLocalCopy` will make (or not make) a copy of the stream object, use it locally and copy the states of stream back to the original stream.

Parsing to Double

Parsing string into `double` is difficult. The standard library function `strtod()` can do the job but it is slow. By default, the parsers use normal precision setting. This has maximum 3 ULP error and implemented in `internal::StrtodNormalPrecision()`.

When using `kParseFullPrecisionFlag`, the parsers calls `internal::StrtodFullPrecision()` instead, and this function actually implemented 3 versions of conversion methods.

1. [Fast-Path](#).
2. Custom DIY-FP implementation as in [double-conversion](#).
3. Big Integer Method as in (Clinger, William D. How to read floating point numbers accurately. Vol. 25. No. 6. ACM, 1990).

If the first conversion methods fail, it will try the second, and so on.

Generation Optimization

Integer-to-String conversion

The naive algorithm for integer-to-string conversion involves division per each decimal digit. We have implemented various implementations and evaluated them in [itoa-benchmark](#).

Although SSE2 version is the fastest but the difference is minor by comparing to the first running-up `branchlut`. And `branchlut` is pure C++ implementation so we adopt `branchlut` in RapidJSON.

Double-to-String conversion

Originally RapidJSON uses `snprintf(..., ..., "%g")` to achieve double-to-string conversion. This is not accurate as the default precision is 6. Later we also find that this is slow and there is an alternative.

Google's V8 [double-conversion](#) implemented a newer, fast algorithm called Grisu3 (Loitsch, Florian. "Printing floating-point numbers quickly and accurately with integers." ACM Sigplan Notices 45.6 (2010): 233-243.).

However, since it is not header-only so that we implemented a header-only version of Grisu2. This algorithm guarantees that the result is always accurate. And in most of cases it produces the shortest (optimal) string representation.

The header-only conversion function has been evaluated in [dtoa-benchmark](#).

Parser

Iterative Parser

The iterative parser is a recursive descent LL(1) parser implemented in a non-recursive manner.

Grammar

The grammar used for this parser is based on strict JSON syntax:

```
S -> array | object
array -> [ values ]
object -> { members }
values -> non-empty-values | ε
non-empty-values -> value addition-values
addition-values -> ε | , non-empty-values
members -> non-empty-members | ε
non-empty-members -> member addition-members
addition-members -> ε | , non-empty-members
member -> STRING : value
value -> STRING | NUMBER | NULL | BOOLEAN | object | array
```

Note that left factoring is applied to non-terminals `values` and `members` to make the grammar be LL(1).

Parsing Table

Based on the grammar, we can construct the FIRST and FOLLOW set.

The FIRST set of non-terminals is listed below:

NON-TERMINAL	FIRST
array	[
object	{
values	ε STRING NUMBER NULL BOOLEAN { [
addition-values	ε COMMA
members	ε STRING
addition-members	ε COMMA
member	STRING
value	STRING NUMBER NULL BOOLEAN { [
S	[{
non-empty-members	STRING
non-empty-values	STRING NUMBER NULL BOOLEAN { [

The FOLLOW set is listed below:

NON-TERMINAL	FOLLOW
S	\$
array	, \$ }]
object	, \$ }]
values]
non-empty-values]
addition-values]

members	}
non-empty-members	}
addition-members	}
member	, }
value	, }]

Finally the parsing table can be constructed from FIRST and FOLLOW set:

NON- TERMINAL	[{	,	:]	}	STRING	NUMBER	NULL
S	array	object							
array	[values]								
object		{ members }							
values	non- empty- values	non- empty- values			ε		non- empty- values	non- empty- values	non- empty- values
non-empty- values	value addition- values	value addition- values					value addition- values	value addition- values	value addition- values
addition- values			, non- empty- values			ε			
members						ε	non- empty- members		
non-empty- members							member addition- members		
addition- members			, non- empty- members			ε			
member							STRING : value		
value	array	object					STRING	NUMBER	NULL

There is a great [tool](#) for above grammar analysis.

Implementation

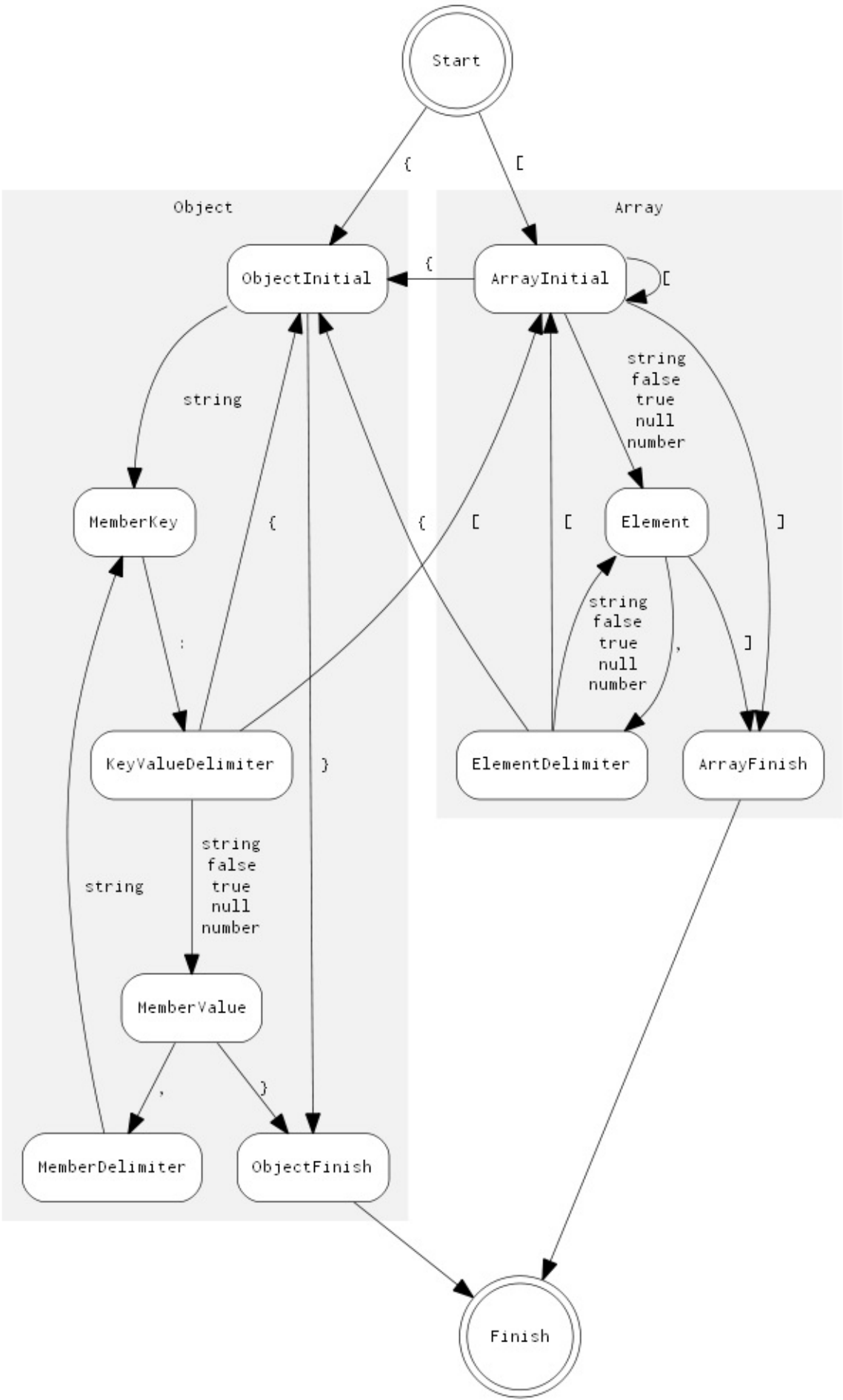
Based on the parsing table, a direct(or conventional) implementation that pushes the production body in reverse order while generating a production could work.

In RapidJSON, several modifications(or adaptations to current design) are made to a direct implementation.

First, the parsing table is encoded in a state machine in RapidJSON. States are constructed by the head and body of production. State transitions are constructed by production rules. Besides, extra states are added for productions involved with `array` and `object`. In this way the generation of array values or object members would be a single state transition, rather than several pop/push operations in the direct implementation. This also makes the estimation of stack size more

easier.

The state diagram is shown as follows:



Second, the iterative parser also keeps track of array's value count and object's member count in its internal stack, which may be different from a conventional implementation.

FAQ

[TOC]

General

1. What is RapidJSON?

RapidJSON is a C++ library for parsing and generating JSON. You may check all [features](#) of it.

2. Why is RapidJSON named so?

It is inspired by [RapidXML](#), which is a fast XML DOM parser.

3. Is RapidJSON similar to RapidXML?

RapidJSON borrowed some designs of RapidXML, including *in situ* parsing, header-only library. But the two APIs are completely different. Also RapidJSON provide many features that are not in RapidXML.

4. Is RapidJSON free?

Yes, it is free under MIT license. It can be used in commercial applications. Please check the details in [license.txt](#).

5. Is RapidJSON small? What are its dependencies?

Yes. A simple executable which parses a JSON and prints its statistics is less than 30KB on Windows.

RapidJSON depends on C++ standard library only.

6. How to install RapidJSON?

Check [Installation section](#).

7. Can RapidJSON run on my platform?

RapidJSON has been tested in many combinations of operating systems, compilers and CPU architecture by the community. But we cannot ensure that it can be run on your particular platform. Building and running the unit test suite will give you the answer.

8. Does RapidJSON support C++03? C++11?

RapidJSON was firstly implemented for C++03. Later it added optional support of some C++11 features (e.g., move constructor, `noexcept`). RapidJSON shall be compatible with C++03 or C++11 compliant compilers.

9. Does RapidJSON really work in real applications?

Yes. It is deployed in both client and server real applications. A community member reported that RapidJSON in their system parses 50 million JSONs daily.

10. How RapidJSON is tested?

RapidJSON contains a unit test suite for automatic testing. [Travis](#)(for Linux) and [AppVeyor](#)(for Windows) will compile and run the unit test suite for all modifications. The test process also uses Valgrind (in Linux) to detect memory leaks.

11. Is RapidJSON well documented?

RapidJSON provides user guide and API documentationn.

12. Are there alternatives?

Yes, there are a lot alternatives. For example, [nativejson-benchmark](#) has a listing of open-source C/C++ JSON libraries. [json.org](#) also has a list.

JSON

1. What is JSON?

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It uses human readable text format. More details of JSON can be referred to [RFC7159](#) and [ECMA-404](#).

2. What are applications of JSON?

JSON are commonly used in web applications for transferring structured data. It is also used as a file format for data persistence.

3. Does RapidJSON conform to the JSON standard?

Yes. RapidJSON is fully compliance with [RFC7159](#) and [ECMA-404](#). It can handle corner cases, such as supporting null character and surrogate pairs in JSON strings.

4. Does RapidJSON support relaxed syntax?

Currently no. RapidJSON only support the strict standardized format. Support on related syntax is under discussion in this [issue](#).

DOM and SAX

1. What is DOM style API?

Document Object Model (DOM) is an in-memory representation of JSON for query and manipulation.

2. What is SAX style API?

SAX is an event-driven API for parsing and generation.

3. Should I choose DOM or SAX?

DOM is easy for query and manipulation. SAX is very fast and memory-saving but often more difficult to be applied.

4. What is *in situ* parsing?

in situ parsing decodes the JSON strings directly into the input JSON. This is an optimization which can reduce memory consumption and improve performance, but the input JSON will be modified. Check [in-situ parsing](#) for details.

5. When does parsing generate an error?

The parser generates an error when the input JSON contains invalid syntax, or a value can not be represented (a number is too big), or the handler of parsers terminate the parsing. Check [parse error](#) for details.

6. What error information is provided?

The error is stored in `ParseResult`, which includes the error code and offset (number of characters from the beginning of JSON). The error code can be translated into human-readable error message.

7. Why not just using `double` to represent JSON number?

Some applications use 64-bit unsigned/signed integers. And these integers cannot be converted into `double` without loss of precision. So the parsers detects whether a JSON number is convertible to different types of integers and/or `double`.

Document/Value (DOM)

1. What is move semantics? Why?

Instead of copy semantics, move semantics is used in `Value`. That means, when assigning a source value to a target value, the ownership of source value is moved to the target value.

Since moving is faster than copying, this design decision forces user to aware of the copying overhead.

2. How to copy a value?

There are two APIs: constructor with allocator, and `CopyFrom()`. See [Deep Copy Value](#) for an example.

3. Why do I need to provide the length of string?

Since C string is null-terminated, the length of string needs to be computed via `strlen()`, with linear runtime complexity. This incurs an unnecessary overhead of many operations, if the user already knows the length of string.

Also, RapidJSON can handle `\u0000` (null character) within a string. If a string contains null characters, `strlen()` cannot return the true length of it. In such case user must provide the length of string explicitly.

4. Why do I need to provide allocator parameter in many DOM manipulation API?

Since the APIs are member functions of `Value`, we do not want to save an allocator pointer in every `Value`.

5. Does it convert between numerical types?

When using `GetInt()`, `GetUint()`, ... conversion may occur. For integer-to-integer conversion, it only convert when it is safe (otherwise it will assert). However, when converting a 64-bit signed/unsigned integer to double, it will convert but be aware that it may lose precision. A number with fraction, or an integer larger than 64-bit, can only be obtained by `GetDouble()`.

Reader/Writer (SAX)

1. Why don't we just `printf` a JSON? Why do we need a `Writer`?

Most importantly, `Writer` will ensure the output JSON is well-formed. Calling SAX events incorrectly (e.g. `StartObject()` pairing with `EndArray()`) will assert. Besides, `Writer` will escapes strings (e.g., `\n`). Finally, the numeric output of `printf()` may not be a valid JSON number, especially in some locale with digit delimiters. And the number-to-string conversion in `Writer` is implemented with very fast algorithms, which outperforms than `printf()` or `iostream`.

2. Can I pause the parsing process and resume it later?

This is not directly supported in the current version due to performance consideration. However, if the execution environment supports multi-threading, user can parse a JSON in a separate thread, and pause it by blocking in the input stream.

Unicode

1. Does it support UTF-8, UTF-16 and other format?

Yes. It fully support UTF-8, UTF-16 (LE/BE), UTF-32 (LE/BE) and ASCII.

2. Can it validate the encoding?

Yes, just pass `kParseValidateEncodingFlag` to `Parse()`. If there is invalid encoding in the stream, it will generate `kParseErrorStringInvalidEncoding` error.

3. What is surrogate pair? Does RapidJSON support it?

JSON uses UTF-16 encoding when escaping unicode character, e.g. `\u5927` representing Chinese character "big". To handle characters other than those in basic multilingual plane (BMP), UTF-16 encodes those characters with two 16-bit values, which is called UTF-16 surrogate pair. For example, the Emoji character U+1F602 can be encoded as `\uD83D\uDE02` in JSON.

RapidJSON fully support parsing/generating UTF-16 surrogates.

4. Can it handle `\u0000` (null character) in JSON string?

Yes. RapidJSON fully support null character in JSON string. However, user need to be aware of it and using `GetStringLength()` and related APIs to obtain the true length of string.

5. Can I output `\uxxxx` for all non-ASCII character?

Yes, use `ASCII<>` as output encoding template parameter in `Writer` can enforce escaping those characters.

Stream

1. I have a big JSON file. Should I load the whole file to memory?

User can use `FileReadStream` to read the file chunk-by-chunk. But for *in situ* parsing, the whole file must be loaded.

2. Can I parse JSON while it is streamed from network?

Yes. User can implement a custom stream for this. Please refer to the implementation of `FileReadStream`.

3. I don't know what encoding will the JSON be. How to handle them?

You may use `AutoUTFInputStream` which detects the encoding of input stream automatically. However, it will incur some performance overhead.

4. What is BOM? How RapidJSON handle it?

[Byte order mark \(BOM\)](#) sometimes reside at the beginning of file/stream to indicate the UTF encoding type of it.

RapidJSON's `EncodedInputStream` can detect/consume BOM. `EncodedOutputStream` can optionally write a BOM. See [Encoded Streams](#) for example.

5. Why little/big endian is related?

little/big endian of stream is an issue for UTF-16 and UTF-32 streams, but not UTF-8 stream.

Performance

1. Is RapidJSON really fast?

Yes. It may be the fastest open source JSON library. There is a [benchmark](#) for evaluating performance of C/C++ JSON libraries.

2. Why is it fast?

Many design decisions of RapidJSON is aimed at time/space performance. These may reduce user-friendliness of APIs. Besides, it also employs low-level optimizations (intrinsics, SIMD) and special algorithms (custom double-to-string, string-to-double conversions).

3. What is SIMD? How it is applied in RapidJSON?

[SIMD](#) instructions can perform parallel computation in modern CPUs. RapidJSON support Intel's SSE2/SSE4.2 to accelerate whitespace skipping. This improves performance of parsing indent formatted JSON. Define `RAPIDJSON_SSE2` or `RAPIDJSON_SSE42` macro to enable this feature. However, running the executable on a machine without such instruction set support will make it crash.

4. Does it consume a lot of memory?

The design of RapidJSON aims at reducing memory footprint.

In the SAX API, `Reader` consumes memory portional to maximum depth of JSON tree, plus maximum length of JSON string.

In the DOM API, each `value` consumes exactly 16/24 bytes for 32/64-bit architecture respectively. RapidJSON also uses a special memory allocator to minimize overhead of allocations.

5. What is the purpose of being high performance?

Some applications need to process very large JSON files. Some server-side applications need to process huge amount of JSONs. Being high performance can improve both latency and throuput. In a broad sense, it will also save energy.

Gossip

1. Who are the developers of RapidJSON?

Milo Yip ([miloyip](#)) is the original author of RapidJSON. Many contributors from the world have improved RapidJSON. Philipp A. Hartmann ([pah](#)) has implemented a lot of improvements, setting up automatic testing and also involves in a lot of discussions for the community. Don Ding ([thebusytypist](#)) implemented the iterative parser. Andrii Senkovych ([jollyroger](#)) completed the CMake migration. Kosta ([Kosta-Github](#)) provided a very neat short-string optimization. Thank you for all other contributors and community members as well.

2. Why do you develop RapidJSON?

It was just a hobby project initially in 2011. Milo Yip is a game programmer and he just knew about JSON at that time and would like to apply JSON in future projects. As JSON seems very simple he would like to write a header-only and

fast library.

3. Why there is a long empty period of development?

It is basically due to personal issues, such as getting new family members. Also, Milo Yip has spent a lot of spare time on translating "Game Engine Architecture" by Jason Gregory into Chinese.

4. Why did the repository move from Google Code to GitHub?

This is the trend. And GitHub is much more powerful and convenient.