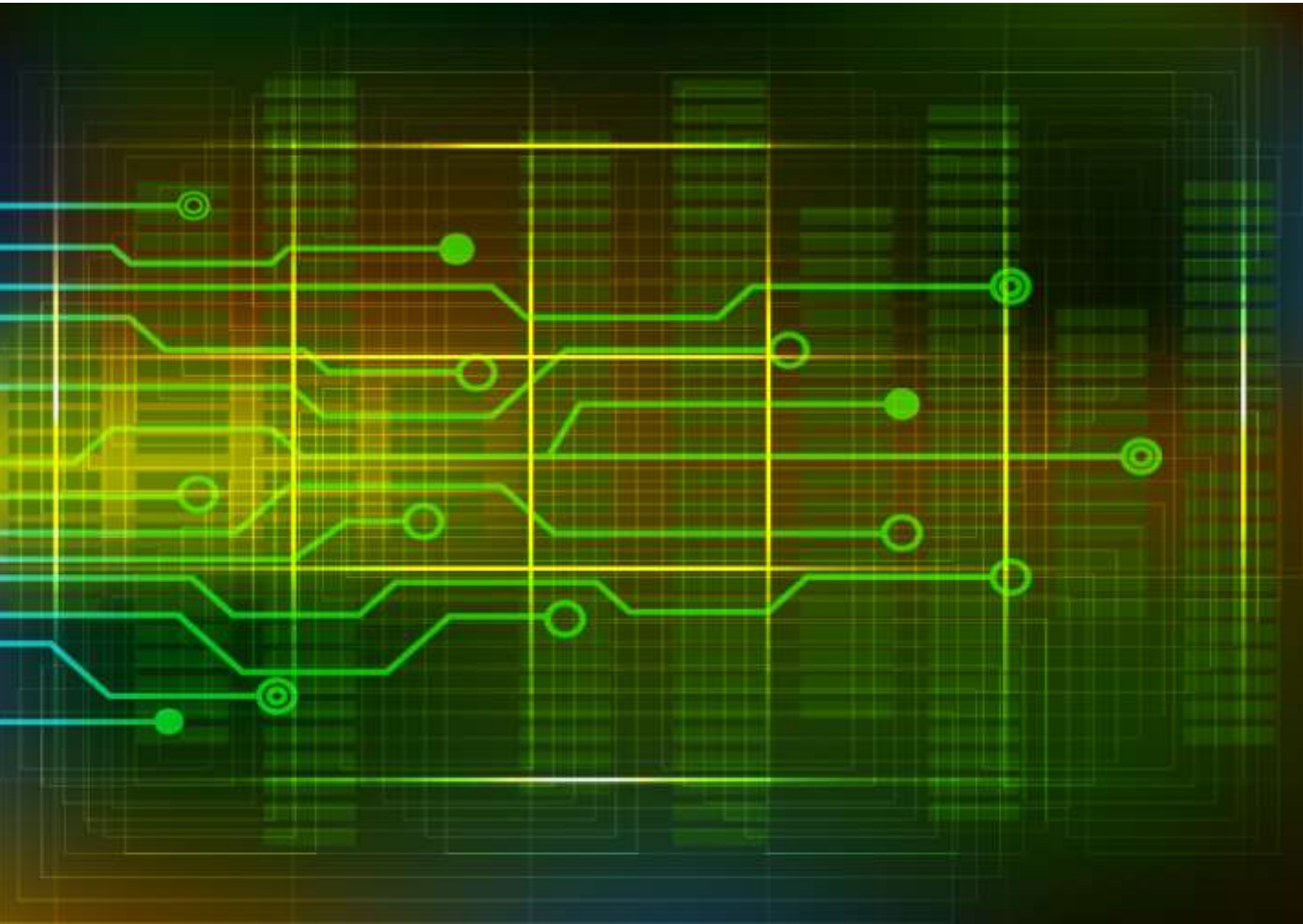


BENOIT BLANCHON  
CREATOR OF ARDUINOJSON



# Mastering ArduinoJson 6

Efficient JSON serialization for embedded C++



SECOND EDITION

# Contents

---

Contents	iv
<b>1 Introduction</b>	<b>1</b>
1.1 About this book . . . . .	2
1.1.1 Overview . . . . .	2
1.1.2 Code samples . . . . .	2
1.1.3 What's new in the second edition . . . . .	3
1.2 Introduction to JSON . . . . .	4
1.2.1 What is JSON? . . . . .	4
1.2.2 What is serialization? . . . . .	5
1.2.3 What can you do with JSON? . . . . .	5
1.2.4 History of JSON . . . . .	7
1.2.5 Why is JSON so popular? . . . . .	8
1.2.6 The JSON syntax . . . . .	9
1.2.7 Binary data in JSON . . . . .	12
1.2.8 Comments in JSON . . . . .	13
1.3 Introduction to ArduinoJson . . . . .	14
1.3.1 What ArduinoJson is . . . . .	14
1.3.2 What ArduinoJson is not . . . . .	14
1.3.3 What makes ArduinoJson different? . . . . .	15
1.3.4 Does size really matter? . . . . .	17
1.3.5 What are the alternatives to ArduinoJson? . . . . .	18
1.3.6 How to install ArduinoJson . . . . .	20
1.3.7 The examples . . . . .	25
1.4 Summary . . . . .	27
<b>2 The missing C++ course</b>	<b>28</b>
2.1 Why a C++ course? . . . . .	29
2.2 Stack, heap, and globals . . . . .	31
2.2.1 Globals . . . . .	31
2.2.2 Heap . . . . .	33

2.2.3	Stack . . . . .	34
2.3	Pointers . . . . .	36
2.3.1	What is a pointer? . . . . .	36
2.3.2	Dereferencing a pointer . . . . .	36
2.3.3	Pointers and arrays . . . . .	37
2.3.4	Taking the address of a variable . . . . .	38
2.3.5	Pointer to class and struct . . . . .	38
2.3.6	Pointer to constant . . . . .	39
2.3.7	The null pointer . . . . .	41
2.3.8	Why use pointers? . . . . .	42
2.4	Memory management . . . . .	43
2.4.1	malloc() and free() . . . . .	43
2.4.2	new and delete . . . . .	43
2.4.3	Smart pointers . . . . .	44
2.4.4	RAII . . . . .	46
2.5	References . . . . .	47
2.5.1	What is a reference? . . . . .	47
2.5.2	Differences with pointers . . . . .	47
2.5.3	Reference to constant . . . . .	48
2.5.4	Rules of references . . . . .	49
2.5.5	Common problems . . . . .	49
2.5.6	Usage for references . . . . .	50
2.6	Strings . . . . .	51
2.6.1	How are the strings stored? . . . . .	51
2.6.2	String literals in RAM . . . . .	51
2.6.3	String literals in Flash . . . . .	52
2.6.4	Pointer to the “globals” section . . . . .	53
2.6.5	Mutable string in “globals” . . . . .	54
2.6.6	A copy in the stack . . . . .	55
2.6.7	A copy in the heap . . . . .	56
2.6.8	A word about the String class . . . . .	57
2.6.9	Pass strings to functions . . . . .	58
2.7	Summary . . . . .	60
<b>3</b>	<b>Deserialize with ArduinoJson</b>	<b>62</b>
3.1	The example of this chapter . . . . .	63
3.2	Deserializing an object . . . . .	64
3.2.1	The JSON document . . . . .	64
3.2.2	Placing the JSON document in memory . . . . .	64
3.2.3	Introducing JsonDocument . . . . .	65

3.2.4	How to specify the capacity? . . . . .	65
3.2.5	How to determine the capacity? . . . . .	66
3.2.6	StaticJsonDocument or DynamicJsonDocument? . . . . .	67
3.2.7	Deserializing the JSON document . . . . .	67
3.3	Extracting values from an object . . . . .	69
3.3.1	Extracting values . . . . .	69
3.3.2	Explicit casts . . . . .	69
3.3.3	When values are missing . . . . .	70
3.3.4	Change the default value . . . . .	71
3.4	Inspecting an unknown object . . . . .	72
3.4.1	Getting a reference to the object . . . . .	72
3.4.2	Enumerating the keys . . . . .	73
3.4.3	Detecting the type of value . . . . .	73
3.4.4	Variant types and C++ types . . . . .	74
3.4.5	Testing if a key exists in an object . . . . .	75
3.5	Deserializing an array . . . . .	76
3.5.1	The JSON document . . . . .	76
3.5.2	Parsing the array . . . . .	76
3.5.3	The ArduinoJson Assistant . . . . .	78
3.6	Extracting values from an array . . . . .	79
3.6.1	Retrieving elements by index . . . . .	79
3.6.2	Alternative syntaxes . . . . .	79
3.6.3	When complex values are missing . . . . .	80
3.7	Inspecting an unknown array . . . . .	82
3.7.1	Getting a reference to the array . . . . .	82
3.7.2	Capacity of JsonDocument for an unknown input . . . . .	82
3.7.3	Number of elements in an array . . . . .	83
3.7.4	Iteration . . . . .	83
3.7.5	Detecting the type of an element . . . . .	84
3.8	The zero-copy mode . . . . .	86
3.8.1	Definition . . . . .	86
3.8.2	An example . . . . .	86
3.8.3	Input buffer must stay in memory . . . . .	88
3.9	Reading from read-only memory . . . . .	90
3.9.1	The example . . . . .	90
3.9.2	Duplication is required . . . . .	90
3.9.3	Practice . . . . .	91
3.9.4	Other types of read-only input . . . . .	92
3.10	Reading from a stream . . . . .	94
3.10.1	Reading from a file . . . . .	94

3.10.2	Reading from an HTTP response . . . . .	95
3.11	Summary . . . . .	103
<b>4</b>	<b>Serializing with ArduinoJson</b>	<b>105</b>
4.1	The example of this chapter . . . . .	106
4.2	Creating an object . . . . .	107
4.2.1	The example . . . . .	107
4.2.2	Allocating the JsonDocument . . . . .	107
4.2.3	Adding members . . . . .	108
4.2.4	Alternative syntax . . . . .	108
4.2.5	Creating an empty object . . . . .	109
4.2.6	Removing members . . . . .	109
4.2.7	Replacing members . . . . .	109
4.3	Creating an array . . . . .	111
4.3.1	The example . . . . .	111
4.3.2	Allocating the JsonDocument . . . . .	111
4.3.3	Adding elements . . . . .	112
4.3.4	Adding nested objects . . . . .	112
4.3.5	Creating an empty array . . . . .	113
4.3.6	Replacing elements . . . . .	113
4.3.7	Removing elements . . . . .	114
4.3.8	Adding null . . . . .	114
4.3.9	Adding pre-formatted JSON . . . . .	114
4.4	Writing to memory . . . . .	116
4.4.1	Minified JSON . . . . .	116
4.4.2	Specifying (or not) the size of the output buffer . . . . .	116
4.4.3	Prettified JSON . . . . .	117
4.4.4	Measuring the length . . . . .	118
4.4.5	Writing to a String . . . . .	119
4.4.6	Casting a JsonVariant to a String . . . . .	119
4.5	Writing to a stream . . . . .	121
4.5.1	What's an output stream? . . . . .	121
4.5.2	Writing to the serial port . . . . .	122
4.5.3	Writing to a file . . . . .	123
4.5.4	Writing to a TCP connection . . . . .	123
4.6	Duplication of strings . . . . .	127
4.6.1	An example . . . . .	127
4.6.2	Copy only occurs when adding values . . . . .	128
4.6.3	Why copying Flash strings? . . . . .	128
4.6.4	serialized() . . . . .	129

4.7	Summary . . . . .	130
<b>5</b>	<b>Advanced Techniques</b>	<b>131</b>
5.1	Introduction . . . . .	132
5.2	Filtering the input . . . . .	133
5.2.1	Motivation . . . . .	133
5.2.2	Principle . . . . .	133
5.2.3	Implementation . . . . .	133
5.3	Deserializing in chunks . . . . .	137
5.3.1	Motivation . . . . .	137
5.3.2	Principle . . . . .	137
5.3.3	Implementation . . . . .	137
5.4	JSON streaming . . . . .	142
5.4.1	Motivation . . . . .	142
5.4.2	Principle . . . . .	142
5.4.3	Implementation . . . . .	143
5.5	Automatic capacity . . . . .	145
5.5.1	Motivation . . . . .	145
5.5.2	Principle . . . . .	145
5.5.3	Implementation . . . . .	146
5.6	Fixing memory leaks . . . . .	147
5.6.1	Motivation . . . . .	147
5.6.2	Principle . . . . .	147
5.6.3	Implementation . . . . .	148
5.7	Using external RAM . . . . .	149
5.7.1	Motivation . . . . .	149
5.7.2	Principle . . . . .	149
5.7.3	Implementation . . . . .	150
5.8	Logging . . . . .	152
5.8.1	Motivation . . . . .	152
5.8.2	Principle . . . . .	152
5.8.3	Implementation . . . . .	153
5.9	Buffering . . . . .	155
5.9.1	Motivation . . . . .	155
5.9.2	Principle . . . . .	155
5.9.3	Implementation . . . . .	156
5.10	Custom readers and writers . . . . .	158
5.10.1	Motivation . . . . .	158
5.10.2	Principle . . . . .	159
5.10.3	Implementation . . . . .	160

5.11	MessagePack . . . . .	163
5.11.1	Motivation . . . . .	163
5.11.2	Principle . . . . .	163
5.11.3	Implementation . . . . .	164
5.12	Summary . . . . .	166
<b>6</b>	<b>Inside ArduinoJson . . . . .</b>	<b>168</b>
6.1	Why JsonDocument? . . . . .	169
6.1.1	Memory representation . . . . .	169
6.1.2	Dynamic memory . . . . .	170
6.1.3	Memory pool . . . . .	171
6.1.4	Strengths and weaknesses . . . . .	172
6.2	Inside JsonDocument . . . . .	173
6.2.1	Differences with JsonVariant . . . . .	173
6.2.2	Fixed capacity . . . . .	173
6.2.3	Implementation of the allocator . . . . .	174
6.2.4	Implementation of JsonDocument . . . . .	175
6.3	Inside StaticJsonDocument . . . . .	177
6.3.1	Capacity . . . . .	177
6.3.2	Stack memory . . . . .	177
6.3.3	Limitation . . . . .	178
6.3.4	Other usages . . . . .	179
6.3.5	Implementation . . . . .	179
6.4	Inside DynamicJsonDocument . . . . .	180
6.4.1	Capacity . . . . .	180
6.4.2	Shrinking a DynamicJsonDocument . . . . .	180
6.4.3	Automatic capacity . . . . .	181
6.4.4	Heap memory . . . . .	182
6.4.5	Allocator . . . . .	182
6.4.6	Implementation . . . . .	183
6.4.7	Comparison with StaticJsonDocument . . . . .	184
6.4.8	How to choose? . . . . .	184
6.5	Inside JsonVariant . . . . .	185
6.5.1	Supported types . . . . .	185
6.5.2	Reference semantics . . . . .	185
6.5.3	Creating a JsonVariant . . . . .	186
6.5.4	Implementation . . . . .	187
6.5.5	Two kinds of null . . . . .	188
6.5.6	The unsigned long trick . . . . .	189
6.5.7	Integer overflow . . . . .	189

6.5.8	ArduinoJson's configuration . . . . .	190
6.5.9	Iterating through a JsonVariant . . . . .	191
6.5.10	The or operator . . . . .	192
6.5.11	The subscript operator . . . . .	193
6.5.12	Member functions . . . . .	194
6.5.13	Comparison operators . . . . .	197
6.5.14	Const reference . . . . .	198
6.6	Inside JsonObject . . . . .	199
6.6.1	Reference semantics . . . . .	199
6.6.2	Null object . . . . .	199
6.6.3	Create an object . . . . .	200
6.6.4	Implementation . . . . .	200
6.6.5	Subscript operator . . . . .	201
6.6.6	Member functions . . . . .	202
6.6.7	Const reference . . . . .	205
6.7	Inside JsonArray . . . . .	207
6.7.1	Member functions . . . . .	207
6.7.2	copyArray() . . . . .	211
6.8	Inside the parser . . . . .	213
6.8.1	Invoking the parser . . . . .	213
6.8.2	Two modes . . . . .	214
6.8.3	Pitfalls . . . . .	214
6.8.4	Nesting limit . . . . .	215
6.8.5	Quotes . . . . .	216
6.8.6	Escape sequences . . . . .	217
6.8.7	Comments . . . . .	218
6.8.8	NaN and Infinity . . . . .	218
6.8.9	Stream . . . . .	218
6.8.10	Filtering . . . . .	219
6.9	Inside the serializer . . . . .	220
6.9.1	Invoking the serializer . . . . .	220
6.9.2	Measuring the length . . . . .	221
6.9.3	Escape sequences . . . . .	222
6.9.4	Float to string . . . . .	222
6.9.5	NaN and Infinity . . . . .	223
6.10	Miscellaneous . . . . .	224
6.10.1	The version macro . . . . .	224
6.10.2	The private namespace . . . . .	224
6.10.3	The ArduinoJson namespace . . . . .	225
6.10.4	ArduinoJson.h and ArduinoJson.hpp . . . . .	225



6.10.5	The single header . . . . .	226
6.10.6	Code coverage . . . . .	226
6.10.7	Fuzzing . . . . .	226
6.10.8	Portability . . . . .	227
6.10.9	Online compiler . . . . .	228
6.10.10	License . . . . .	229
6.11	Summary . . . . .	230
<b>7</b>	<b>Troubleshooting</b>	<b>231</b>
7.1	Program crashes . . . . .	232
7.1.1	Undefined Behaviors . . . . .	232
7.1.2	A bug in ArduinoJson? . . . . .	232
7.1.3	Null string . . . . .	233
7.1.4	Use after free . . . . .	233
7.1.5	Return of stack variable address . . . . .	235
7.1.6	Buffer overflow . . . . .	236
7.1.7	Stack overflow . . . . .	238
7.1.8	How to diagnose these bugs? . . . . .	238
7.1.9	How to prevent these bugs? . . . . .	241
7.2	Deserialization issues . . . . .	243
7.2.1	IncompleteInput . . . . .	243
7.2.2	InvalidInput . . . . .	244
7.2.3	NoMemory . . . . .	248
7.2.4	NotSupported . . . . .	249
7.2.5	TooDeep . . . . .	250
7.3	Serialization issues . . . . .	252
7.3.1	The JSON document is incomplete . . . . .	252
7.3.2	The JSON document contains garbage . . . . .	252
7.3.3	Too much duplication . . . . .	254
7.4	Common error messages . . . . .	256
7.4.1	x was not declared in this scope . . . . .	256
7.4.2	Invalid conversion from const char* to char* . . . . .	256
7.4.3	Invalid conversion from const char* to int . . . . .	257
7.4.4	No match for operator[] . . . . .	258
7.4.5	Ambiguous overload for operator= . . . . .	259
7.4.6	Call of overloaded function is ambiguous . . . . .	260
7.4.7	The value is not usable in a constant expression . . . . .	261
7.5	Asking for help . . . . .	262
7.6	Summary . . . . .	264

<b>8</b>	<b>Case Studies</b>	<b>265</b>
8.1	Configuration in SPIFFS	266
8.1.1	Presentation	266
8.1.2	The JSON document	266
8.1.3	The configuration class	267
8.1.4	load() and save() members	268
8.1.5	Saving an ApConfig into a JsonObject	269
8.1.6	Loading an ApConfig from a JsonObject	269
8.1.7	Copying strings safely	270
8.1.8	Saving a Config to a JSON object	271
8.1.9	Loading a Config from a JSON object	271
8.1.10	Saving the configuration to a file	272
8.1.11	Reading the configuration from a file	273
8.1.12	Choosing the JsonDocument	274
8.1.13	Conclusion	275
8.2	OpenWeatherMap on mkr1000	277
8.2.1	Presentation	277
8.2.2	OpenWeatherMap's API	277
8.2.3	The JSON response	278
8.2.4	Reducing the size of the document	279
8.2.5	The filter document	281
8.2.6	The code	282
8.2.7	Summary	283
8.3	Reddit on ESP8266	284
8.3.1	Presentation	284
8.3.2	Reddit's API	285
8.3.3	The response	286
8.3.4	The main loop	287
8.3.5	Escaped Unicode characters	288
8.3.6	Sending the request	288
8.3.7	Assembling the puzzle	289
8.3.8	Summary	290
8.4	JSON-RPC with Kodi	292
8.4.1	Presentation	292
8.4.2	JSON-RPC Request	293
8.4.3	JSON-RPC Response	293
8.4.4	A JSON-RPC framework	294
8.4.5	JsonRpcRequest	295
8.4.6	JsonRpcResponse	296
8.4.7	JsonRpcClient	297

8.4.8	Sending a notification to Kodi . . . . .	298
8.4.9	Reading properties from Kodi . . . . .	300
8.4.10	Summary . . . . .	302
8.5	Recursive analyzer . . . . .	304
8.5.1	Presentation . . . . .	304
8.5.2	Read from the serial port . . . . .	304
8.5.3	Flushing after an error . . . . .	305
8.5.4	Testing the type of a JsonVariant . . . . .	306
8.5.5	Printing values . . . . .	307
8.5.6	Summary . . . . .	309
<b>9</b>	<b>Conclusion</b>	<b>310</b>
	<b>Index</b>	<b>311</b>

# Chapter 4

## Serializing with ArduinoJson

---

”

*Any fool can write code that a computer can understand.  
Good programmers write code that humans can understand.*

– Martin Fowler, Refactoring: Improving the Design of Existing Code

## 4.1 The example of this chapter

Reading a JSON document is only half of the story; we'll now see how to write a JSON document with ArduinoJson.

In the previous chapter, we played with GitHub's API. We'll use a very different example for this chapter: pushing data to Adafruit IO.

Adafruit IO is a cloud storage service for IoT data. They have a free plan with the following restrictions:

- 30 data points per minute
- 30 days of data storage
- 5 feeds

If you need more, it's just \$10 a month. The service is very easy to use. All you need is an Adafruit account (yes, you can use the account from the Adafruit shop).

As we did in the previous chapter, we'll start with a simple JSON document and add complexity step by step.

Since Adafruit IO doesn't impose a secure connection, we can use a less powerful microcontroller than in the previous chapter; we'll use an Arduino UNO with an Ethernet Shield.



## 4.2 Creating an object

### 4.2.1 The example

Here is the JSON object we want to create:

```
{
  "value": 42,
  "lat": 48.748010,
  "lon": 2.293491
}
```

It's a flat object, meaning that it has no nested object or array, and it contains the following members:

1. "value" is an integer that we want to save in Adafruit IO.
2. "lat" is the latitude coordinate where the value was measured.
3. "lon" is the longitude coordinate where the value was measured.

Adafruit IO supports other optional members (like the elevation coordinate and the time of measurement), but the three members above are sufficient for our example.

### 4.2.2 Allocating the JsonDocument

As for the deserialization, we start by creating a `JsonDocument` to hold the memory representation of the object. The previous chapter introduces the `JsonDocument`; please go back and read ["Introducing JsonDocument"](#) if needed, as we won't repeat here.

As we saw, a `JsonDocument` has a fixed capacity that we must specify when we create it. Here, we have one object with no nested values, so the capacity is `JSON_OBJECT_SIZE(3)`. Remember that you can use the [ArduinoJson Assistant](#) to compute the required capacity.

For our example, we can use a `StaticJsonDocument` because the document is so small that it can easily fit in the stack.

Here is the code:

```
const int capacity = JSON_OBJECT_SIZE(3);  
StaticJsonDocument<capacity> doc;
```

The `JsonDocument` is currently empty and `JsonDocument::isNull()` returns `true`. If we serialize it now, the output is `"null"`.

### 4.2.3 Adding members

An empty `JsonDocument` automatically becomes an object when we add members to it. We do that with the subscript operator (`[]`), just like we did in the previous chapter:

```
doc["value"] = 42;  
doc["lat"] = 48.748010;  
doc["lon"] = 2.293491;
```

The memory usage is now `JSON_OBJECT_SIZE(3)`, so the `JsonDocument` is full. When the `JsonDocument` is full, you cannot add more members, so don't forget to increase the capacity if you need.

### 4.2.4 Alternative syntax

With the syntax presented above, it's not possible to tell whether the insertion succeeded. Let's see another syntax:

```
doc["value"].set(42);  
doc["lat"].set(48.748010);  
doc["lon"].set(2.293491);
```

The compiler generates the same executable as with the previous syntax, except that you can tell if the insertion succeeded. Indeed, `JsonVariant::set()` returns `true` for success or `false` if the `JsonDocument` is full.

To be honest, I never check if insertion succeeds in my programs. The reason is simple: the JSON document is roughly the same for each iteration; if it works once, it always works. There is no reason to bloat the code for a situation that cannot happen.

### 4.2.5 Creating an empty object

We just saw that the `JsonDocument` becomes an object as soon as you insert a member, but what if you don't have any member to add? What if you want to create an empty object?

When you need an empty object, you cannot rely on the implicit conversion anymore. Instead, you must convert the `JsonDocument` to a `JsonObject` explicitly with `JsonDocument::to<JsonObject>()`:

```
// Convert the document to an object
JsonObject obj = doc.to<JsonObject>();
```

This function clears the `JsonDocument`, so all existing references become invalid. Then, it creates an empty object at the root of the document and returns a reference to this object.

At this point, the `JsonDocument` is not empty anymore and `JsonDocument::isNull()` returns `false`. If we serialize this document, the output is `"{}"`.

### 4.2.6 Removing members

It's possible to erase a member from an object by calling `JsonObject::remove(key)`. However, for reasons that will become clear in [chapter 6](#), this function doesn't release the memory in the `JsonDocument`.

The `remove()` function is a frequent cause of bugs because it creates a memory leak. Indeed, if you add and remove members in a loop, the `JsonDocument` grows, but memory is never released.

### 4.2.7 Replacing members

It's possible to replace a member in the object, for example:

```
obj["value"] = 42;
obj["value"] = 43;
```



Most of the time, replacing a member doesn't require a new allocation in the `JsonDocument`. However, it can cause a memory leak if the old value has associated memory, for example, if the old value is a string, an array, or an object.



### Memory leaks

Replacing and removing values produce a memory leak inside the `JsonDocument`.

In practice, this problem only happens in programs that use a `JsonDocument` to store the state of the application, which is not the purpose of `ArduinoJson`. Let's be clear; the sole purpose of `ArduinoJson` is to serialize and deserialize JSON documents.

Be careful not to fall into this common anti-pattern and make sure you read the [case studies](#) to see how `ArduinoJson` should be used.

## 4.3 Creating an array

### 4.3.1 The example

Now that we know how to create an object, let's see how we can create an array. Our new example will be an array that contains two objects.

```
[
  {
    "key": "a1",
    "value": 12
  },
  {
    "key": "a2",
    "value": 34
  }
]
```

The values `12` and `34` are just placeholder; in reality, we'll use the result from `analogRead()`.

### 4.3.2 Allocating the `JsonDocument`

As usual, we start by computing the capacity of the `JsonDocument`:

- There is one array with two elements: `JSON_ARRAY_SIZE(2)`
- There are two objects with two members: `2*JSON_OBJECT_SIZE(2)`

Here is the code:

```
const int capacity = JSON_ARRAY_SIZE(2) + 2*JSON_OBJECT_SIZE(2);
StaticJsonDocument<capacity> doc;
```

### 4.3.3 Adding elements

In the previous section, we saw that an empty `JsonDocument` automatically becomes an object as soon as we insert the first member. Well, this statement was only partially right: it becomes an object as soon as we use it as an object.

Indeed, if we treat an empty `JsonDocument` as an array, it automatically becomes an array. For example if we call `JsonDocument::add()`:

```
doc.add(1);  
doc.add(2);
```

After these two lines, our `JsonDocument` contains `[1,2]`.

Alternatively, we can create the same array like that:

```
doc[0] = 1;  
doc[1] = 2;
```

However, this second syntax is a little slower because it requires walking the list of members. Don't use this syntax if you need to add many elements to the array.

Now that this topic is clear, let's rewind a little because that's not the JSON array we want to create; instead of two integers, we want two nested objects.

### 4.3.4 Adding nested objects

To add the nested objects to the array, we call `JsonArray::createNestedObject()`. This function creates a nested object, appends it the array, and returns a reference.

Here is how to create our sample document:

```
JsonObject obj1 = doc.createNestedObject();  
obj1["key"] = "a1";  
obj1["value"] = analogRead(A1);  
  
JsonObject obj2 = doc.createNestedObject();  
obj2["key"] = "a2";  
obj2["value"] = analogRead(A2);
```

Alternatively, we can create the same document like that:

```
doc[0]["key"] = "a1";  
doc[0]["value"] = analogRead(A1);  
  
doc[1]["key"] = "a2";  
doc[1]["value"] = analogRead(A2);
```

As I already said, this syntax runs slower, so only use it for small arrays.

### 4.3.5 Creating an empty array

We saw that the `JsonDocument` becomes an array as soon as we add elements, but this doesn't allow creating an empty array. If we want to create an empty array, we need to convert the `JsonDocument` explicitly with `JsonDocument::to<JsonArray>()`:

```
// Convert the JsonDocument to an array  
JsonArray arr = doc.to<JsonArray>();
```

Now the `JsonDocument` contains `[]`.

As we already saw, `JsonDocument::to<T>()` clears the `JsonDocument`, so it also invalidates all previously acquired references.

### 4.3.6 Replacing elements

As for objects, it's possible to replace elements in arrays using `JsonArray::operator[]`:

```
arr[0] = 666;  
arr[1] = 667;
```

Most of the time, replacing the value doesn't require a new allocation in the `JsonDocument`. However, if there was memory held by the previous value, for example, a `JsonObject`, this memory is not released. It's a limitation of `ArduinoJson`'s memory allocator, as we'll see later in this book.

### 4.3.7 Removing elements

As for objects, you can remove an element from the array, with `JsonArray::remove()`:

```
arr.remove(0);
```

Again, `remove()` doesn't release the memory from the `JsonDocument`, so you should never call this function in a loop.

### 4.3.8 Adding null

To conclude this section, let's see how we can insert special values in the JSON document.

The first special value is `null`, which is a legal token in a JSON. There are several ways to add a `null` in a `JsonDocument`; here they are:

```
// Use a nullptr (requires C++11)
arr.add(nullptr);

// Use a null char-pointer
arr.add((char*)0);

// Use a null JsonArray, JsonObject, or JsonVariant
arr.add(JsonVariant());
```

### 4.3.9 Adding pre-formatted JSON

The other special value is a JSON string that is already formatted and that `ArduinoJson` should not treat as a regular string.

You can do that by wrapping the string with a call to `serialized()`:

```
// adds "[1,2]"
arr.add(serialized("[1,2]"));
```

```
// adds [1,2]
arr.add(serialized("[1,2]"));
```

The program above produces the following JSON document:

```
[
  "[1,2]",
  [1,2]
]
```

This feature is useful when a part of the document never changes, and you want to optimize the code. It's also useful to insert something you cannot generate with the library.

## 4.4 Writing to memory

We saw how to construct an array. Now, it's time to serialize it into a JSON document. There are several ways to do that. We'll start with a JSON document in memory.

We could use a `String`, but as you know, I prefer avoiding dynamic memory allocation. Instead, we'd use a good old `char[]`:

```
// Declare a buffer to hold the result
char output[128];
```

### 4.4.1 Minified JSON

To produce a JSON document from a `JsonDocument`, we simply need to call `serializeJson()`:

```
// Produce a minified JSON document
serializeJson(doc, output);
```

Now, the string `output` contains:

```
[{"key": "a1", "value": 12}, {"key": "a2", "value": 34}]
```

As you see, there are neither space nor line breaks; it's a "minified" JSON document.

### 4.4.2 Specifying (or not) the size of the output buffer

If you're a C programmer, you may have been surprised that I didn't provide the size of the buffer to `serializeJson()`. Indeed, there is an overload of `serializeJson()` that takes a `char*` and a size:

```
serializeJson(doc, output, sizeof(output));
```

However, that's not the overload we called in the previous snippet. Instead, we called a template method that infers the size of the buffer from its type (in this case, `char[128]`).

Of course, this shorter syntax only works because `output` is an array. If it were a `char*` or a variable-length array, we would have had to specify the size.



### Variable-length array

A variable-length array, or VLA, is an array whose size is unknown at compile time. Here is an example:

```
void f(int n) {  
    char buf[n];  
    // ...  
}
```

C99 and C11 allow VLAs, but not C++. However, some compilers support VLAs as an extension.

This feature is often criticized in C++ circles, but Arduino users seem to love it. That's why ArduinoJson supports VLAs in all functions that accept a string.

## 4.4.3 Prettified JSON

The minified version is what you use to store or transmit a JSON document because the size is optimal. However, it's not very easy to read. Humans prefer “prettified” JSON documents with spaces and line breaks.

To produce a prettified document, you must use `serializeJsonPretty()` instead of `serializeJson()`:

```
// Produce a prettified JSON document  
serializeJsonPretty(doc, output);
```

Here is the content of `output`:

```
[  
  {
```



```
"key": "a1",  
"value": 12  
,  
{  
  "key": "a2",  
  "value": 34  
}  
]
```

Of course, you need to make sure that the output buffer is big enough; otherwise, the JSON document will be incomplete.

#### 4.4.4 Measuring the length

ArduinoJson allows computing the length of the JSON document before producing it. This information is useful for:

1. allocating an output buffer,
2. reserving the size on disk, or
3. setting the Content-Length header.

There are two methods, depending on the type of document you want to produce:

```
// Compute the length of the minified JSON document  
int len1 = measureJson(doc);  
  
// Compute the length of the prettified JSON document  
int len2 = measureJsonPretty(doc);
```

In both cases, the result doesn't count the null-terminator.

By the way, `serializeJson()` and `serializeJsonPretty()` return the number of bytes that they wrote. The results are the same as `measureJson()` and `measureJsonPretty()`, except if the output buffer is too small.

**Avoid prettified documents**

With the example above, the sizes are 73 and 110. In this case, the prettified version is only 50% bigger because the document is simple, but in most cases, the ratio is largely above 100%.

Remember, we're in an embedded environment: every byte counts, and so does every CPU cycle. Always prefer a minified version.

### 4.4.5 Writing to a String

The functions `serializeJson()` and `serializeJsonPretty()` have overloads taking a `String`:

```
String output = "JSON = ";  
serializeJson(doc, output);
```

The behavior is slightly different: the JSON document is appended to the `String`; it doesn't replace it. That means the above snippet sets the content of the output variable to:

```
JSON = [{"key": "a1", "value": 12}, {"key": "a2", "value": 34}]
```

This behavior seems inconsistent? That's because `ArduinoJson` treats `String` like a stream; more on that later.

### 4.4.6 Casting a JsonVariant to a String

You should remember from the chapter on deserialization that we must cast `JsonVariant` to the type we want to read.

It is also possible to cast a `JsonVariant` to a `String`. If the `JsonVariant` contains a string, the return value is a copy of the string. However, if the `JsonVariant` contains something else, the returned string is a serialization of the variant.

We could rewrite the previous example like this:

```
// Cast the JsonDocument to a string  
String output = "JSON = " + doc.as<String>();
```

This trick works with `JsonDocument` and `JsonVariant`, but not with `JsonArray` and `JsonObject` because they don't have an `as<T>()` function.

## 4.5 Writing to a stream

### 4.5.1 What's an output stream?

For now, every JSON document we produced remained in memory, but that's usually not what we want. In many situations, it's possible to send the JSON document directly to its destination (whether it's a file, a serial port, or a network connection) without any copy in RAM.

We saw in the previous chapter what an “input stream” is, and we saw that Arduino represents this concept with the `Stream` class. Similarly, there are “output streams,” which are sinks of bytes. We can write to an output stream, but we cannot read. In the Arduino land, an output stream is materialized by the `Print` class.

Here are examples of classes derived from `Print`:

Library	Class	Well known instances
Core	<code>HardwareSerial</code>	<code>Serial</code> , <code>Serial1</code> ...
ESP	<code>BluetoothSerial</code>	<code>SerialBT</code>
	<code>File</code>	
	<code>WiFiClient</code>	
	<code>WiFiClientSecure</code>	
Ethernet	<code>EthernetClient</code>	
	<code>EthernetUDP</code>	
GSM	<code>GSMClient</code>	
LiquidCrystal	<code>LiquidCrystal</code>	
SD	<code>File</code>	
SoftwareSerial	<code>SoftwareSerial</code>	
WiFi	<code>WiFiClient</code>	
Wire	<code>TwoWire</code>	<code>Wire</code>



#### `std::ostream`

In the C++ Standard Library, an output stream is represented by the `std::ostream` class.

ArduinoJson supports both `Print` and `std::ostream`.

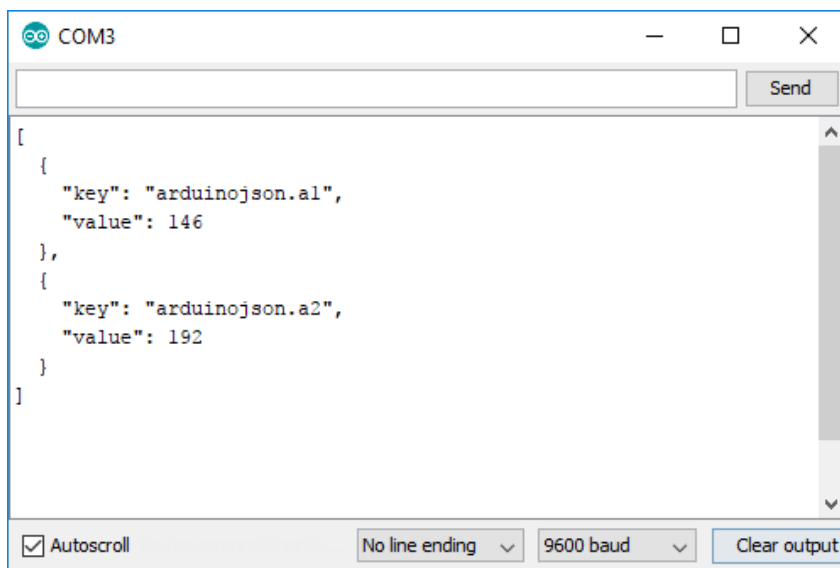
### 4.5.2 Writing to the serial port

The most famous implementation of `Print` is `HardwareSerial`, which is the class of `Serial`. To serialize a `JsonDocument` to the serial port of your Arduino, just pass `Serial` to `serializeJson()`:

```
// Print a minified JSON document to the serial port
serializeJson(doc, Serial);

// Same with a prettified document
serializeJsonPretty(doc, Serial);
```

You can see the result in the Arduino Serial Monitor, which is very handy for debugging.



There are also other serial port implementations that you can use this way, for example, `SoftwareSerial` and `TwoWire`.

### 4.5.3 Writing to a file

Similarly, we can use a `File` instance as the target of `serializeJson()` and `serializeJsonPretty()`. Here is an example with the SD library:

```
// Open file for writing
File file = SD.open("adafruit.txt", FILE_WRITE);

// Write a prettified JSON document to the file
serializeJsonPretty(doc, file);
```

You can find the complete source code for this example in the `WriteSdCard` folder of the zip file provided with the book.

You can apply the same technique to write a file on an ESP8266, as we'll see [in the case studies](#).

### 4.5.4 Writing to a TCP connection

We're now reaching our goal of sending our measurements to **Adafruit IO**.

As I said in the introduction, we'll suppose that our program runs on an Arduino UNO with an Ethernet shield. Because the Arduino UNO has only 2KB of RAM, we'll not use the heap at all.

#### Preparing the Adafruit IO account

If you want to run this program, you need an account on Adafruit IO (a free account is sufficient). Then, you need to copy your user name and your "AIO key" to the source code.

```
#define IO_USERNAME "bblanchon"
#define IO_KEY "baf4f21a32f6438eb82f83c3eed3f3b3"
```

We'll include the AIO key in an HTTP header, and it will authenticate our program on Adafruit's server:

```
X-AIO-Key: baf4f21a32f6438eb82f83c3eed3f3b3
```

Finally, to run this program, you need to create a “group” named “arduinojson” in your Adafruit IO account. In this group, you need to create two feeds: “a1” and “a2.”

### The request

To send our measured samples to Adafruit IO, we have to send a `POST` request to `http://io.adafruit.com/api/v2/bblanchon/groups/arduinojson/data`, and include the following JSON document in the body:

```
{
  "location": {
    "lat": 48.748010,
    "lon": 2.293491
  },
  "feeds": [
    {
      "key": "a1",
      "value": 42
    },
    {
      "key": "a2",
      "value": 43
    }
  ]
}
```

As you see, it’s a little more complex than our previous sample because the array is not at the root of the document. Instead, the array is nested in an object under the key “feeds”.

Let’s review the HTTP request before jumping to the code:

```
POST /api/v2/bblanchon/groups/arduinojson/data HTTP/1.1
Host: io.adafruit.com
Connection: close
Content-Length: 103
Content-Type: application/json
X-AIO-Key: baf4f21a32f6438eb82f83c3eed3f3b3
```

```
{"location":{"lat":48.748010,"lon":2.293491},"feeds":[{"key":"a1",...
```

### The code

OK, time for action! We'll open a TCP connection to `io.adafruit.com` using an `EthernetClient`, and we'll send the request. As far as `ArduinoJson` is concerned, there are very few changes compared to the previous examples because we can pass the `EthernetClient` as the target of `serializeJson()`. We'll call `measureJson()` to set the value of the `Content-Length` header.

Here is the code:

```
// Allocate JsonDocument
const int capacity = JSON_ARRAY_SIZE(2) + 4 * JSON_OBJECT_SIZE(2);
StaticJsonDocument<capacity> doc;

// Add the "location" object
JsonObject location = doc.createNestedObject("location");
location["lat"] = 48.748010;
location["lon"] = 2.293491;

// Add the "feeds" array
JsonArray feeds = doc.createNestedArray("feeds");
JsonObject feed1 = feeds.createNestedObject();
feed1["key"] = "a1";
feed1["value"] = analogRead(A1);
JsonObject feed2 = feeds.createNestedObject();
feed2["key"] = "a2";
feed2["value"] = analogRead(A2);

// Connect to the HTTP server
EthernetClient client;
client.connect("io.adafruit.com", 80);

// Send "POST /api/v2/bblanchon/groups/arduinojson/data HTTP/1.1"
client.println("POST /api/v2/" IO_USERNAME
              "/groups/arduinojson/data HTTP/1.1");
```



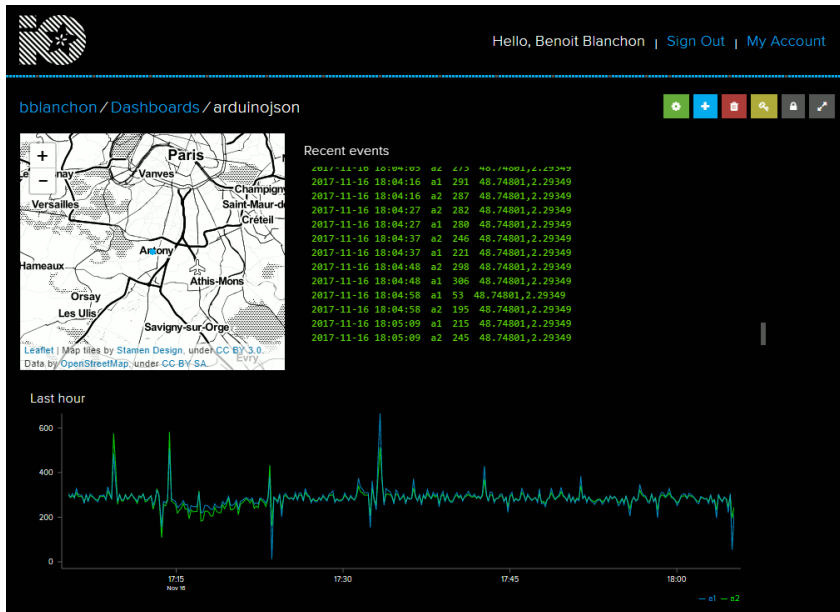
```
// Send the HTTP headers
client.println("Host: io.adafruit.com");
client.println("Connection: close");
client.print("Content-Length: ");
client.println(measureJson(doc));
client.println("Content-Type: application/json");
client.println("X-AIO-Key: " IO_KEY);

// Terminate headers with a blank line
client.println();

// Send JSON document in body
serializeJson(doc, client);
```

You can find the complete source code of this example in the `AdafruitIo` folder of the zip file. This code includes the necessary error checking that I removed from the manuscript for clarity.

Below is a picture showing the results on the Adafruit IO dashboard.



## 4.6 Duplication of strings

Depending on the type, ArduinoJson stores strings either by pointer or by copy. If the string is a `const char*`, it stores a pointer; otherwise, it makes a copy. This feature reduces memory consumption when you use string literals.

String type	Storage
<code>const char*</code>	pointer
<code>char*</code>	copy
<code>String</code>	copy
<code>const __FlashStringHelper*</code>	copy

As usual, the copy lives in the `JsonDocument`, so you may need to increase its capacity depending on the type of string you use.

### 4.6.1 An example

Compare this program:

```
// Create the array ["value1","value2"]
doc.add("value1");
doc.add("value2");

// Print the memory usage
Serial.println(doc.memoryUsage());
```

with the following:

```
// Create the array ["value1","value2"]
doc.add(String("value1"));
doc.add(String("value2"));

// Print the memory usage
Serial.println(doc.memoryUsage());
```

They both produce the same JSON document, but the second one requires much more memory because ArduinoJson copies the strings. If you run these programs on an ATmega328, you'll see 16 for the first one and 30 for the second.

### 4.6.2 Copy only occurs when adding values

In the example above, ArduinoJson copied the Strings because it needed to add them to the JsonDocument. On the other hand, if you use a String to extract a value from a JsonDocument, it doesn't make a copy.

Here is an example:

```
// The following line produces a copy of "key"
doc[String("key")] = "value";

// The following line produces no copy
const char* value = doc[String("key")];
```

### 4.6.3 Why copying Flash strings?

I understand that it is disappointing that ArduinoJson copies Flash strings into the JsonDocument. Unfortunately, there are several situations where it needs to have the strings in RAM.

For example, if the user calls `JsonVariant::as<char*>()`, a pointer to the copy is returned:

```
// The value is originally in Flash memory
obj["hello"] = F("world");

// But the returned value is in RAM (in the JsonDocument)
const char* world = obj["hello"];
```

It is required for `JsonPair` too. If the string is a key in an object and the user iterates through the object, the `JsonPair` contains a pointer to the copy:

```
// The key is originally in Flash memory
obj[F("hello")] = "world";

for(JsonPair kvp : obj) {
    // But the key is actually stored in RAM (in the JsonDocument)
    const char* key = kvp.key().c_str();
}
```

However, retrieving a value using a Flash string as a key doesn't cause a copy:

```
// The Flash string is not copied in this case
const char* world = obj[F("hello")];
```



### Avoid Flash strings with ArduinoJson

Storing strings in Flash is a great way to reduce RAM usage, but remember that ArduinoJson copies them in the `JsonDocument`.

If you wrap all your strings with `F()`, you'll need a much bigger `JsonDocument`. Moreover, the program will waste much time copying the string; it will be much slower than with conventional strings.

I plan to avoid this duplication in a future revision of the library, but it's not on the roadmap yet.

#### 4.6.4 `serialized()`

We saw earlier in this chapter that the `serialized()` function marks a string as a JSON fragment that should not be treated as a regular string value.

`serialized()` supports all the string types (`char*`, `const char*`, `String`, and `const __FlashStringHelper*`) and duplicates them as expected.

## 4.7 Summary

In this chapter, we saw how to serialize a JSON document with ArduinoJson. Here are the key points to remember:

- Creating the document:
  - To add a member to an object, use the subscript operator (`[]`)
  - To append an element to an array, call `add()`
  - The first time you add a member to a `JsonDocument`, it automatically becomes an object.
  - The first time you append an element to a `JsonDocument`, it automatically becomes an array.
  - You can explicitly convert a `JsonDocument` with `JsonDocument::to<T>()`.
  - `JsonDocument::to<T>()` clears the `JsonDocument`, so it invalidates all previously acquired references.
  - `JsonDocument::to<T>()` return a reference to the root array or object.
  - To create a nested array or object, call `createNestedArray()` or `createNestedObject()`.
  - When you insert a string in a `JsonDocument`, it makes a copy, except if it's a `const char*`.
- Serializing the document:
  - To serialize a `JsonDocument`, call `serializeJson()` or `serializeJsonPretty()`.
  - To compute the length of the JSON document, call `measureJson()` or `measureJsonPretty()`.
  - `serializeJson()` appends to `String`, but it overrides the content of a `char*`.
  - You can pass an instance of `Print` (like `Serial`, `EthernetClient`, and `WiFiClient`) to `serializeJson()` to avoid a copy in the RAM.

In the next chapter, we'll see advanced techniques like filtering and logging.

## Continue reading...

---

That was a free chapter from “Mastering ArduinoJson”; the book contains seven chapters like this one. Here is what readers say:

This book is 100% worth it. Between solving my immediate problem in minutes, Chapter 2, and the various other issues this book made solving easy, **it is totally worth it**. I build software but I work in managed languages and for someone just getting started in C++ and embedded programming this book has been indispensable. — Nathan Burnett

I think the missing C++ course and the troubleshooting chapter **are worth the money by itself**. Very useful for C programming dinosaurs like myself. — Doug Petican

The short C++ section was a great refresher. The practical use of ArduinoJson in small embedded processors was just what I needed for my home automation work. **Certainly worth having!** Thank you for both the book and the library. — Douglas S. Basberg

For a really reasonable price, not only you'll learn new skills, but you'll also be one of the few people that **contribute to sustainable open-source software**. Yes, giving money for free software is a political act!

The e-book comes in three formats: PDF, epub and mobi. If you purchase the e-book, **you get access to newer versions for free**. A carefully edited paperback edition is also available.

Ready to jump in?

Go to [arduinojson.org/book](http://arduinojson.org/book) and use the coupon code THIRTY to get a **30% discount**.

*Thank you for your support!  
Benit*