



BENOIT BLANCHON

CREATOR OF ARDUINOJSON

Mastering ArduinoJson 6

Efficient JSON serialization for embedded C++



ArduinoJson

SECOND EDITION

Mastering ArduinoJson 6 - Second Edition

Copyright © 2018-2020 Benoît BLANCHON

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review.

Product and company names mentioned herein may be the trademarks of their respective owners.

While every precaution has been taken in the preparation of this book, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Published by Benoît Blanchon, Antony, FRANCE.

Logo design by bcendet.

Cover design by Iulia Ghimisli.

Revision date: June 30, 2020

<https://arduinojson.org>

To the early users of ArduinoJson, who pushed me in the right direction.

Contents

Contents	iv
1 Introduction	1
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
2 The missing C++ course	28
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
3	Deserialize with ArduinoJson	62
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	Changing 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
4	Serializing with ArduinoJson	105
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	Keys and values	128
4.6.3	Copy only occurs when adding values	128
4.6.4	ArduinoJson Assistant's "extra bytes"	129

4.6.5	Why copying Flash strings?	129
4.6.6	serialized()	131
4.7	Summary	132
5	Advanced Techniques	133
5.1	Introduction	134
5.2	Filtering the input	135
5.3	Deserializing in chunks	139
5.4	JSON streaming	144
5.5	Automatic capacity	147
5.6	Fixing memory leaks	149
5.7	Using external RAM	151
5.8	Logging	154
5.9	Buffering	157
5.10	Custom readers and writers	160
5.11	MessagePack	165
5.12	Summary	168
6	Inside ArduinoJson	170
6.1	Why JsonDocument?	171
6.1.1	Memory representation	171
6.1.2	Dynamic memory	172
6.1.3	Memory pool	173
6.1.4	Strengths and weaknesses	174
6.2	Inside JsonDocument	175
6.2.1	Differences with JsonVariant	175
6.2.2	Fixed capacity	175
6.2.3	Implementation of the allocator	176
6.2.4	Implementation of JsonDocument	177
6.3	Inside StaticJsonDocument	179
6.3.1	Capacity	179
6.3.2	Stack memory	179
6.3.3	Limitation	180
6.3.4	Other usages	181
6.3.5	Implementation	181
6.4	Inside DynamicJsonDocument	182
6.4.1	Capacity	182
6.4.2	Shrinking a DynamicJsonDocument	182
6.4.3	Automatic capacity	183
6.4.4	Heap memory	184

6.4.5	Allocator	184
6.4.6	Implementation	185
6.4.7	Comparison with StaticJsonDocument	186
6.4.8	How to choose?	186
6.5	Inside JsonVariant	187
6.5.1	Supported types	187
6.5.2	Reference semantics	187
6.5.3	Creating a JsonVariant	188
6.5.4	Implementation	189
6.5.5	Two kinds of null	190
6.5.6	The unsigned long trick	191
6.5.7	Integer overflow	191
6.5.8	ArduinoJson's configuration	192
6.5.9	Iterating through a JsonVariant	193
6.5.10	The or operator	194
6.5.11	The subscript operator	195
6.5.12	Member functions	196
6.5.13	Comparison operators	199
6.5.14	Const reference	200
6.6	Inside JsonObject	201
6.6.1	Reference semantics	201
6.6.2	Null object	201
6.6.3	Create an object	202
6.6.4	Implementation	202
6.6.5	Subscript operator	203
6.6.6	Member functions	204
6.6.7	Const reference	207
6.7	Inside JSONArray	209
6.7.1	Member functions	209
6.7.2	copyArray()	213
6.8	Inside the parser	215
6.8.1	Invoking the parser	215
6.8.2	Two modes	216
6.8.3	Pitfalls	216
6.8.4	Nesting limit	217
6.8.5	Quotes	218
6.8.6	Escape sequences	219
6.8.7	Comments	220
6.8.8	NaN and Infinity	220
6.8.9	Stream	220

6.8.10 Filtering	221
6.9 Inside the serializer	222
6.9.1 Invoking the serializer	222
6.9.2 Measuring the length	223
6.9.3 Escape sequences	224
6.9.4 Float to string	224
6.9.5 NaN and Infinity	225
6.10 Miscellaneous	226
6.10.1 The version macro	226
6.10.2 The private namespace	226
6.10.3 The ArduinoJson namespace	227
6.10.4 ArduinoJson.h and ArduinoJson.hpp	227
6.10.5 The single header	228
6.10.6 Code coverage	228
6.10.7 Fuzzing	228
6.10.8 Portability	229
6.10.9 Online compiler	230
6.10.10 License	231
6.11 Summary	232
7 Troubleshooting	233
7.1 Program crashes	234
7.1.1 Undefined Behaviors	234
7.1.2 A bug in ArduinoJson?	234
7.1.3 Null string	235
7.1.4 Use after free	235
7.1.5 Return of stack variable address	237
7.1.6 Buffer overflow	238
7.1.7 Stack overflow	240
7.1.8 How to diagnose these bugs?	240
7.1.9 How to prevent these bugs?	243
7.2 Deserialization issues	245
7.2.1 IncompleteInput	245
7.2.2 InvalidInput	246
7.2.3 NoMemory	250
7.2.4 NotSupported	251
7.2.5 TooDeep	252
7.3 Serialization issues	254
7.3.1 The JSON document is incomplete	254
7.3.2 The JSON document contains garbage	254

7.3.3	Too much duplication	256
7.4	Common error messages	258
7.4.1	X was not declared in this scope	258
7.4.2	Invalid conversion from const char* to char*	258
7.4.3	Invalid conversion from const char* to int	259
7.4.4	No match for operator[]	260
7.4.5	Ambiguous overload for operator=	261
7.4.6	Call of overloaded function is ambiguous	262
7.4.7	The value is not usable in a constant expression	263
7.5	Asking for help	264
7.6	Summary	266
8	Case Studies	267
8.1	Configuration in SPIFFS	268
8.1.1	Presentation	268
8.1.2	The JSON document	268
8.1.3	The configuration class	269
8.1.4	load() and save() members	270
8.1.5	Saving an ApConfig into a JsonObject	271
8.1.6	Loading an ApConfig from a JsonObject	271
8.1.7	Copying strings safely	272
8.1.8	Saving a Config to a JSON object	273
8.1.9	Loading a Config from a JSON object	273
8.1.10	Saving the configuration to a file	274
8.1.11	Reading the configuration from a file	275
8.1.12	Choosing the JsonDocument	276
8.1.13	Conclusion	277
8.2	OpenWeatherMap on mkr1000	279
8.2.1	Presentation	279
8.2.2	OpenWeatherMap's API	279
8.2.3	The JSON response	280
8.2.4	Reducing the size of the document	281
8.2.5	The filter document	283
8.2.6	The code	284
8.2.7	Summary	285
8.3	Reddit on ESP8266	286
8.3.1	Presentation	286
8.3.2	Reddit's API	287
8.3.3	The response	288
8.3.4	The main loop	289

8.3.5	Escaped Unicode characters	290
8.3.6	Sending the request	290
8.3.7	Assembling the puzzle	291
8.3.8	Summary	292
8.4	JSON-RPC with Kodi	294
8.4.1	Presentation	294
8.4.2	JSON-RPC Request	295
8.4.3	JSON-RPC Response	295
8.4.4	A JSON-RPC framework	296
8.4.5	JsonRpcRequest	297
8.4.6	JsonRpcResponse	298
8.4.7	JsonRpcClient	299
8.4.8	Sending a notification to Kodi	300
8.4.9	Reading properties from Kodi	302
8.4.10	Summary	304
8.5	Recursive analyzer	306
8.5.1	Presentation	306
8.5.2	Read from the serial port	306
8.5.3	Flushing after an error	307
8.5.4	Testing the type of a JsonVariant	308
8.5.5	Printing values	309
8.5.6	Summary	311
9	Conclusion	312
Index		313

Chapter 1

Introduction

”

Fortunately, JavaScript has some extraordinarily good parts. In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy.

– Douglas Crockford, [JavaScript: The Good Parts](#)

1.1 About this book

Welcome to the wonderful world of embedded C++! Together, we'll learn how to write software that performs JSON serialization with very limited resources. We'll use the most popular Arduino library: ArduinoJson, a library that is easy to use but can be quite hard to master.

1.1.1 Overview

Let's see how this book is organized. Here is a summary of each chapter:

1. An introduction to JSON and ArduinoJson.
2. A quick C++ course. This chapter teaches the fundamentals that many Arduino users lack. It's called "The Missing C++ Course" because it covers what other Arduino books don't.
3. A step-by-step tutorial that teaches how to use ArduinoJson to deserialize a JSON document. We'll use GitHub's API as an example.
4. Another tutorial, but for serialization. This time, we'll use Adafruit IO as an example.
5. Some advanced techniques that didn't fit in the tutorials.
6. How ArduinoJson works under the hood.
7. A troubleshooting guide. If you don't know why your program crashes or why compilation fails, this chapter is for you.
8. Several concrete project examples with explanations. This chapter shows the best coding practices in various situations.

1.1.2 Code samples

This version of the book covers **ArduinoJson 6.15**; you can download the code samples from arduinojson.org/book/sketchbook6.zip

1.1.3 What's new in the second edition

I updated Mastering ArduinoJson with all the changes in the library since the previous edition. These changes include `DeserializationOption::Filter`, `garbageCollect()`, `shrinkToFit()`, `BasicJsonDocument<T>`, custom reader and writer classes, and many small details.

I added a new chapter to present advanced programming techniques. Some of them were already presented in the case studies from the previous edition, like “deserialization in chunks”; others are entirely new, like the custom allocator technique.

I removed the decorator classes from the troubleshooting chapter because I published them as part of the StreamUtils library. These classes are now well tested and extended with a complete family of decorators for streams.

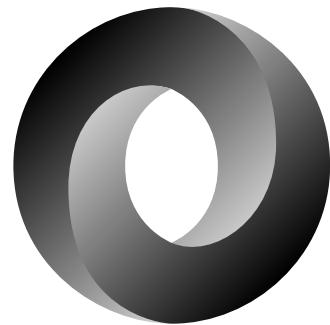
1.2 Introduction to JSON

1.2.1 What is JSON?

Simply put, JSON is a data format. More specifically, JSON is a way to represent complex data structures as text. The resulting string is called a JSON document and can then be sent via the network or saved to a file.

We'll see the JSON syntax in detail later in this chapter, but let's see an example first:

```
{"sensor": "gps", "time": 1351824120, "data": [ ↴ 48.756080, 2.302038]}
```



The text above is the JSON representation of an object composed of:

1. a string, named “sensor,” with the value “gps,”
2. an integer, named “time,” with the value 1351824120,
3. an array of float, named “data,” containing the two values 48.756080 and 2.302038.

JSON ignores spaces and line breaks; so the same object can be represented with the following JSON document:

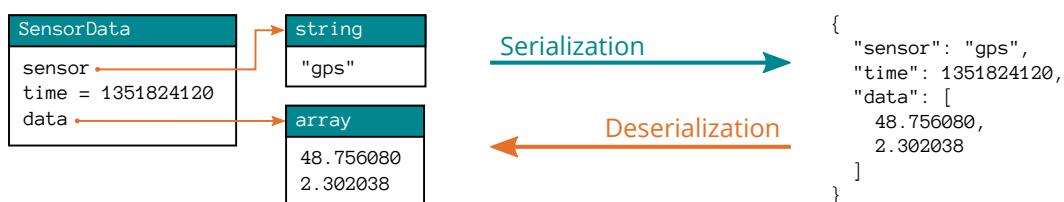
```
{  
  "sensor": "gps",  
  "time": 1351824120,  
  "data": [  
    48.756080,  
    2.302038  
  ]  
}
```

One says that the first JSON document is “minified” and the second is “prettified.”

1.2.2 What is serialization?

In computer science, *serialization* is the process of converting a data structure, into a *series* of bytes which can then be stored or transmitted. Conversely, *deserialization* is the process of converting a *series* of bytes to a data structure.

In the context of JSON, serialization is the creation of a JSON document from an object in memory, and deserialization is the reverse operation.



1.2.3 What can you do with JSON?

There are two reasons why you create a JSON document: either you want to save it, or you want to transmit it.

In the first case, you use JSON as a file format to save your data on disk. For example, in the last chapter, we'll see how we can use JSON to store the configuration of an application.

In the second case, you use JSON as a protocol between a client and a server, or between peers. Nowadays, most web services have an API based on JSON. An API (Application Programming Interface) is a way to interact with the web service from a computer program.

Here are a few examples of companies that provide a JSON-based API:

- Weather forecast
 - AccuWeather (accuweather.com)
 - Dark Sky (darksky.net), formerly known as `forecast.io`
 - OpenWeatherMap (openweathermap.org), we'll see an example in the case studies
- Internet of Thing (IoT)
 - Adafruit IO (io.adafruit.com), we'll see an example in the fourth chapter

- Google Cloud IoT (cloud.google.com/iot)
- ThingSpeak (thingspeak.com)
- Temboo (temboo.com)
- Xively (xively.com)
- Cloud providers
 - Amazon Web Services (aws.amazon.com)
 - Google Cloud Platform (cloud.google.com)
 - Microsoft Azure (azure.microsoft.com)
- Code hosting
 - Bitbucket (bitbucket.org)
 - GitHub (github.com), we'll see an example in the third chapter.
 - GitLab (gitlab.com)
- Home automation
 - Domoticz (domoticz.com)
 - Home Assistant (home-assistant.io)
 - ImperiHome (imperihome.com)
 - Jeedom (jeedom.com)
 - openHAB (openhab.org)
- Task automation
 - Automate.io (automate.io)
 - IFTTT (ifttt.com)
 - Integromat (integromat.com)
 - Microsoft Power Automate (flow.microsoft.com)
 - Workato (workato.com)
 - Zapier (zapier.com)
- Music services and online radios
 - Deezer (developers.deezer.com)

- Last.fm (last.fm)
- MusicBrainz (musicbrainz.org)
- Radio Browser (radio-browser.info)
- Spotify (developer.spotify.com)

This list is not exhaustive; you can find many more examples. If you wonder whether a specific web service has a JSON API, search for the following terms in the developer documentation: “API,” “HTTP API,” “REST API,” or “webhook.”



Choose wisely

Think twice before writing an application that depends on a third-party service. We see APIs come and go very frequently. If the vendor stops or changes its API, you need to rewrite most of your code.

In the first edition of this book, I used two APIs that are already discontinued: Yahoo! and Weather Underground. I had to rewrite an entire chapter because of that. Let this be a lesson.

1.2.4 History of JSON

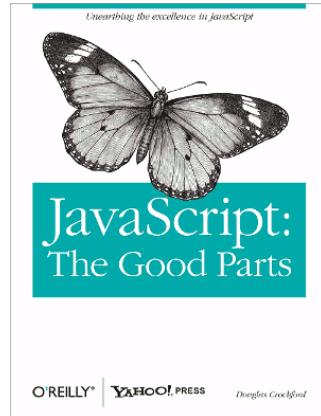
The acronym JSON stands for “JavaScript Object Notation.” As the name suggests, it is a syntax to create an object in the JavaScript language. As JSON is a subset of JavaScript, any JSON document is a valid JavaScript expression.

Here is how you can create the same object in JavaScript:

```
var result = {  
    "sensor": "gps",  
    "time": 1351824120,  
    "data": [  
        48.756080,  
        2.302038  
    ]  
};
```

As curious as it sounds, the JSON notation was “discovered” as a hidden gem in the JavaScript language itself. This discovery is attributed to Douglas Crockford and became popular in 2008 with his book “JavaScript, the Good Parts” (O'Reilly Media).

Before JSON, the go-to serialization format was XML. XML is more powerful than JSON, but the files are bigger and not human-friendly. That's why JSON was initially advertised as *The Fat-Free Alternative to XML*.



1.2.5 Why is JSON so popular?

A big part of the success of JSON can be attributed to the frustration caused by XML. XML is very powerful, but it is overkill for most projects. It is very verbose because you need to repeat the opening and closing tags. The angle brackets make XML very unpleasant to the eye.

Serializing and deserializing XML is quite complicated because tags not only have children but also attributes, and special characters must be encoded in a nontrivial way (e.g., > becomes >), and the CDATA sections must be handled differently.

On the other hand, JSON is less powerful but sufficient for the majority of projects. Its syntax is simpler, minimalistic, and much more pleasant to the eye. JSON has a set of predefined types that cannot be extended.

To give you an idea, here is the same object, written in XML:

```
<result>
  <sensor>gps</sensor>
  <time>1351824120</time>
  <data>
    <value>48.756080</value>
    <value>2.302038</value>
  </data>
</result>
```

Which one do you prefer? I certainly prefer the JSON version.

1.2.6 The JSON syntax

The format specification can be found on json.org, here is just a brief recap.

JSON documents are composed of the following values:

1. Booleans
2. Numbers
3. Strings
4. Arrays
5. Objects

Booleans

A *boolean* is a value that can be either `true` or `false`. It must not be surrounded by quotation marks; otherwise, it would be a string.

Numbers

A *number* can be an integer or a floating-point value.

Examples:

- 42
- 3.14159
- 3e8

The JSON specification uses the word *number* to refer to both integers and floating-point values; however, they are different types in ArduinoJson.



JSON vs. JavaScript

Unlike JavaScript, JSON supports neither hexadecimal (`0x1A`) nor octal (`0755`) notations. ArduinoJson doesn't support them either.

Although the JSON specification disallows `Nan` and `Infinity`. ArduinoJson supports them but this feature is disabled by default.

Strings

A *string* is a sequence of characters (i.e., some text) enclosed in double-quotes.

Example:

- "hi!"
- "hello world"
- "one\ntwo\nthree"
- "C:\\\"



JSON vs. JavaScript

In JSON, the strings are surrounded by double quotes. JSON is more restrictive than JavaScript which also supports single quotes. ArduinoJson supports both.

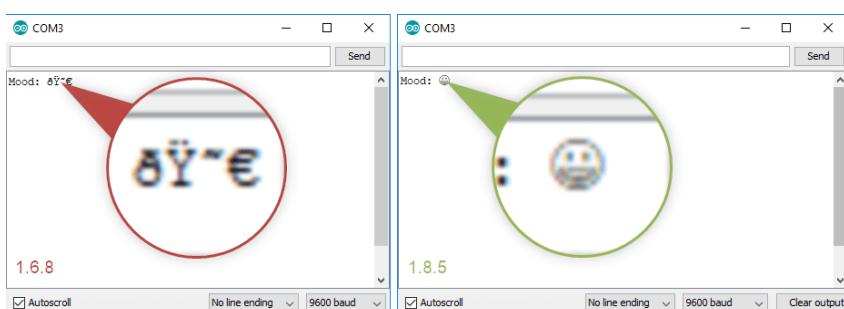
As in most programming languages, JSON requires special characters to be escaped by a backslash (\), for example, "\n" is a new line.

JSON has a notation to specify an extended character using a Unicode escape sequence, for example, "\uD83D", but very few projects use this syntax. Instead, most projects use UTF-8 to encode their JSON documents, so translating to UTF-16 escape sequence is not necessary. ArduinoJson can decode this notation but it's an opt-in feature. We'll see an example in the case studies.



Arduino IDE and UTF-8

The Arduino Serial Monitor supports UTF-8 since version 1.8.2. So, if you see gibberish in the Serial Monitor, make sure the IDE is up-to-date.



Arrays

An *array* is a list of values. In this book, we call “element” a value in an array.

Example:

```
["hi!", 42, true]
```

Syntax:

- An array is delimited by square brackets ([and])
- Elements are separated by commas (,)

The order of the elements matters; for example, [1, 2] is not the same as [2, 1].

Objects

An object is a collection of named values. In this book, we use the word “key” to refer to the name associated with a value.

Example:

```
{"key1": "value1", "key2": "value2"}
```

Syntax:

- An object is surrounded by braces ({ and })
- Key-value pairs are separated by commas (,)
- A colon (:) separates a value from its key
- Keys are surrounded by double quotes ("")

In a single object, each key should be unique. The specification doesn’t explicitly forbid it, but most implementations keep only the last value, ignoring all the previous duplicates.

The order of the values doesn’t matter; for example {**"a"**:1, **"b"**:2} is the same as {**"b"**:2, **"a"**:1}.



JSON vs. JavaScript

JSON requires that keys are surrounded by double-quotes. JSON is more restrictive than JavaScript which allows keys to have single quotes and even to have no quotes at all. ArduinoJson supports keys with single quotes and without quotes.

Misc

Like JavaScript, JSON accepts `null` as a value.

Unlike JavaScript, JSON doesn't allow `undefined` as a value.

1.2.7 Binary data in JSON

There are a few things that JSON is notoriously bad at, the most important being its inability to transmit raw (meaning unmodified) binary data. Indeed, to send binary data in JSON, you must either use an array of integer or encode the data in a string, most likely with base64.

Base64 is a way to encode any sequence of bytes to a sequence of printable characters. There are 64 symbols allowed, hence the name base64. As there are only 64 symbols, only 6 bits of information are sent per symbol; so, when you encode 3 bytes (24 bits), you get 4 characters. In other words, base64 produces an overhead of roughly 33%.

As an example, the title of the book encoded in base64 is:

```
TWFzdGVyaW5nIEFyZHVPbm9Kc29u
```



Binary JSON?

Several alternative data formats claim to be the “binary version of JSON,” the most famous are BSON, CBOR, and MessagePack. All these formats solve the problem of storing binary data in JSON documents.

ArduinoJson supports MessagePack, but it doesn't currently support binary values.

1.2.8 Comments in JSON

Unlike JavaScript, the JSON specification doesn't allow inserting comments in the document. As the developer of a JSON parser, I can confirm that this was a good decision: comments make everything more complicated.

However, comments are very handy for configuration files, that's why some implementations support them. Here is an example of a JSON document with comments:

```
{  
  /* WiFi configuration */  
  "wifi": {  
    "ssid": "TheBatCave",  
    "pass": "i'mbatman!" // <- not secure enough!  
  }  
}
```

ArduinoJson supports comments but it's an optional feature that you must explicitly enable.

1.3 Introduction to ArduinoJson

1.3.1 What ArduinoJson is

ArduinoJson is a library to serialize and deserialize JSON documents. It is designed to work in a deeply embedded environment, i.e., on devices with very limited power.

Because it is open-source and has a very permissive license, you can use it freely in any project, including closed-source and commercial projects.

You can use ArduinoJson outside of the Arduino IDE; all you need is a C++ compiler. Here are some of the many alternative platforms supported by the library:

- Atmel Studio (atmel.com)
- Atollic TrueSTUDIO (atollic.com)
- Energia (energia.nu)
- IAR Embedded Workbench (iar.com)
- MPLAB (microchip.com)
- Particle (particle.io)
- PlatformIO (platformio.org)
- Keil µVision (keil.com)



Then, of course, you can use ArduinoJson on a computer program (whether it's on Linux, Windows, or macOS) because it supports all major compilers.

1.3.2 What ArduinoJson is not

Now that we know that ArduinoJson is, let's see what it is not.

ArduinoJson is not a generic container for the state of your application. I understand it's very tempting to use the flexible JSON object model to store everything, as you'd do in JavaScript, but it's not the purpose of ArduinoJson. After all, we're writing C++, not JavaScript.

For example, let's say that your application has a configuration composed of a hostname and a port. If you need a global variable to store this configuration, don't use a `JsonDocument` (a type from `ArduinoJson`); instead, use a structure:

```
struct AppConfig {  
    char hostname[32];  
    short port;  
};  
  
AppConfig config;
```

Why? Because storing this information in a structure is very efficient in terms of memory usage, program size, and execution speed.

Should you use `ArduinoJson` to store the same data in memory, you would have to pay for every bit of flexibility offered by the JSON model, even if you don't use them.

What if you need to load and save this configuration to a file? Simple! Just create a `temporary JsonDocument`. Don't worry; we'll walk through a complete example in [the case studies](#).

Also, as we'll see in the chapter "[Inside ArduinoJson](#)," the library is very good at managing memory for short periods, but cannot deal with long-lived objects. It is a compromise made to improve the performance of the library.

`ArduinoJson` is quite forgiving: it doesn't require the input to be fully JSON-compliant. For example, it supports comments in the input, allows single quotes around strings, and even supports keys without any quotes. For this reason, you cannot use `ArduinoJson` as a JSON validator.

1.3.3 What makes `ArduinoJson` different?

First, `ArduinoJson` supports both serialization and deserialization. It has an intuitive API to set and get values from objects and arrays:

```
// get a value from an object  
float temp = doc["temperature"];  
  
// replace value  
doc["temperature"] = readTemperature();
```

The main strength of ArduinoJson resides in its memory management strategy. It uses a fixed-size memory pool and a monotonic allocator to perform very fast allocations. This technique avoids the overhead caused by dynamic memory allocations and reduces the heap fragmentation. You'll learn more about this topic in the chapter "[Inside ArduinoJson](#)".

This fixed-allocation strategy makes it suitable for real-time applications where execution time must be predictable. Indeed, when you use a `StaticJsonDocument`, the serialization and the deserialization run in bounded time.

ArduinoJson is a header-only library, meaning that all the code of the library fits in a single `.h` file. This feature greatly simplifies the integration in your projects: download one file, add one `#include`, and you're done! In fact, you can even use the library with web compilers like [wandbox.org](#); go to the ArduinoJson website, you'll find links to online demos.

ArduinoJson is self-contained: it doesn't depend on any library. In particular, it doesn't depend on Arduino, that's why you can use it in any C++ project. This feature allows compiling and running your unit tests on a computer, before compiling for the actual target.

It can deserialize directly from an input stream and can serialize directly to an output stream. This feature makes it very convenient to use with serial ports and network connections. We'll see many examples in this book.

When reading a JSON document from an input stream, ArduinoJson stops reading as soon as the document ends (e.g., at the closing brace). This unique feature allows reading JSON documents one after the other; for example, it allows reading line-delimited JSON streams. We'll see how to do that in the "[Advanced Techniques](#)" chapter.

Even if it's not dependent on Arduino, it plays well with the native Arduino types. It can also use the corresponding types from the C++ Standard Library (STL). The following table shows how the types relate:

Concept	Arduino type	STL type
Output stream	<code>Print</code>	<code>std::ostream</code>
Input stream	<code>Stream</code>	<code>std::istream</code>
String in RAM	<code>String</code>	<code>std::string</code>
String in Flash	<code>_FlashStringHelper</code>	

In addition to JSON, ArduinoJson supports MessagePack with a few limitations. Currently, binary values are not supported, for example. Switching from one format to the

other is as simple as changing a function name. We'll talk about that in the "[Advanced Techniques](#)" chapter.

Since version 6.15, ArduinoJson can filter the input to keep only the values you are interested in. This feature is handy when a web service returns a gigantic document, but you are only interested in a few fields. We'll see all the details in the "[Advanced Techniques](#)" chapter.

A great deal of effort has been put into reducing the code size. Indeed, microcontrollers usually have a limited amount of memory to store the executable, so it's essential to keep it for *your* program, not for the libraries.

1.3.4 Does size really matter?

Let's take a concrete example to show how important are program size and memory usage. Suppose we have an Arduino UNO, it has 32KB of flash memory to store the program, and 2KB of RAM to store the variables.

Now, let's compile the WebClient example provided with the Ethernet library. This program is very minimalistic: all it does is perform a predefined HTTP request and display the result. Here is what you can see in the Arduino output panel:

```
Sketch uses 16858 bytes (54%) of program storage space. Maximum is 30720
→ bytes.
Global variables use 952 bytes (46%) of dynamic memory, leaving 1096 bytes
→ for local variables. Maximum is 2048 bytes.
```

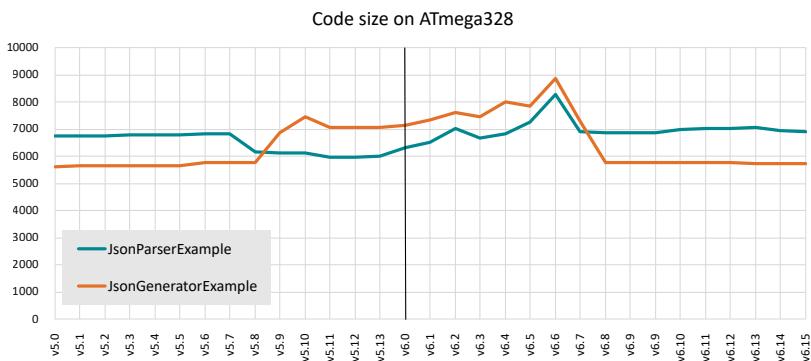
Yep. That is right. The skeleton already takes 54% of the Flash memory and 46% of the RAM. From this baseline, each new line of code increases these numbers until you need to purchase a bigger microcontroller.

Now, if we include ArduinoJson in this program and parse the JSON document contained in the HTTP response, we get something along those lines:

```
Sketch uses 19518 bytes (63%) of program storage space. Maximum is 30720
→ bytes.
Global variables use 1012 bytes (49%) of dynamic memory, leaving 1036 bytes
→ for local variables. Maximum is 2048 bytes.
```

ArduinoJson added only 2660 bytes of Flash and 60 bytes of RAM to the program, which is very small when you consider all the features that it supports. The library represents only 9% of the size of the program but enables a wide range of applications.

The following graph shows the evolution of the size of the two examples provided with ArduinoJson: `JsonGeneratorExample.ino` and `JsonParserExample.ino`.



As you can see, the code size has been kept under control despite adding many features. What does it mean for you? It means that you can safely upgrade to newer versions of ArduinoJson without being afraid that the code will become too big for your target.

1.3.5 What are the alternatives to ArduinoJson?

For a simple JSON document, you don't need a library; you can simply use the standard C functions `sprintf()` and `sscanf()`. However, as soon as there are nested objects and arrays with variable size, you need a library. Here are four alternatives for Arduino.

Arduino_JSON

Arduino_JSON is the "official" JSON library provided by the Arduino organization. Only one revision was published in March 2019, and there was no activity since then.

It offers a simple and convenient syntax but lacks many features:

- No support for comments
- No prettified output
- No error status

- No support for stream
- No filtering
- Only usable on Arduino
- No support for MessagePack
- No unit tests

Despite having significantly fewer features, its code is almost twice as big, consumes about 10% more RAM, and runs about 10% slower than ArduinoJson.

jsmn

jsmn (pronounced “jasmine”) is a JSON tokenizer written in C.

jsmn doesn’t deserialize but instead detects the location of elements in the input. As an input, jsmn takes a string; from that, it generates a list of tokens (object, array, string...), each token having a `start` and `end` positions which are indexes in the input string.

ArduinoJson versions 1 and 2 were built on top of jsmn.

aJson

aJson is a full-featured JSON library for Arduino.

It supports serialization and deserialization. You can parse a JSON document, modify it and serialize it back to JSON, a feature that only came with version 4 of ArduinoJson.

Its main drawback is that it relies on dynamic memory allocation, which makes it unusable in devices with limited memory. Indeed, dynamic allocations tend to create segments of unusable memory. We’ll talk about this phenomenon, called “heap fragmentation,” in [the next chapter](#).

aJson was created in 2010 and was the dominant JSON library for Arduino until 2016. In 2014, I created ArduinoJson because aJson was not able to work reliably on my Arduino Duemilanove.

json-streaming-parser

`json-streaming-parser` is “a library for parsing potentially huge JSON streams on devices with scarce memory.”

It only supports deserialization but can read a JSON input that is bigger than the RAM of the device. `ArduinoJson` cannot do that, but we’ll see some workarounds in the case studies.

`json-streaming-parser` is very different from `ArduinoJson`. Instead of deserializing the JSON document into a data structure, it reads an input stream one piece at a time and invokes a user-defined callback when an object, an array, or a literal is found.

If the terms “SAX” and “DOM” make sense to you, then `json-streaming-parser` is a SAX parser, whereas `ArduinoJson` is a DOM parser.

I often recommend this library when `ArduinoJson` is not suitable.

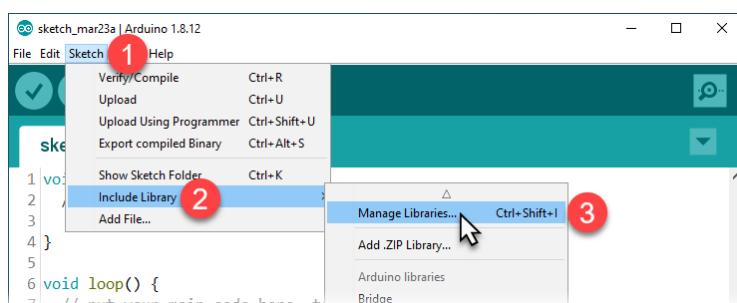
1.3.6 How to install ArduinoJson

There are several ways to install `ArduinoJson`, depending on your situation.

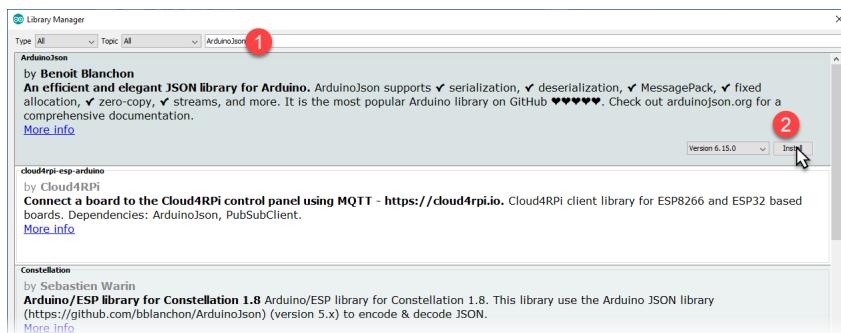
Using the Arduino Library Manager

If you use the Arduino IDE version 1.6 or newer, you can install `ArduinoJson` directly from the IDE, thanks to the “Library Manager.” The Arduino Library Manager lists all installed libraries; it allows installing new ones and updating the ones that are already installed.

To open the Library Manager, open the Arduino IDE and click on “Sketch,” “Include Library,” then “Manage Libraries...”.



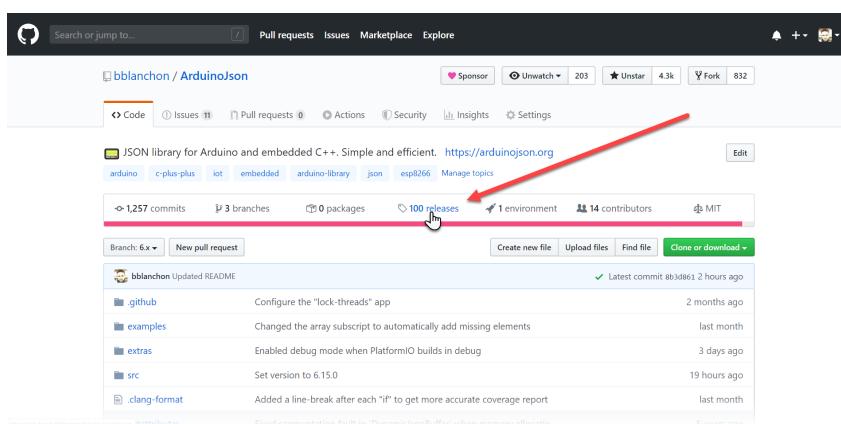
To install the library, enter “ArduinoJson” in the search box, then scroll to find ArduinoJson and click install.



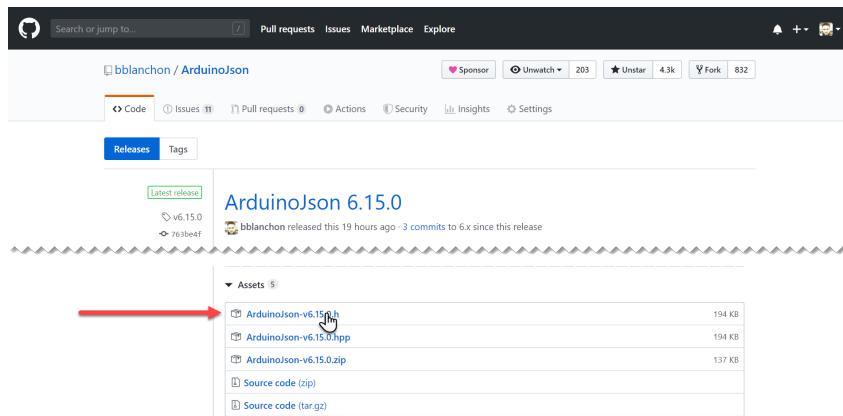
Using the “single header” distribution

If you don't use the Arduino IDE, the simplest way to install ArduinoJson is to put the entire source code of the library in your project folder. Don't worry; it's just one file!

Go to the [ArduinoJson GitHub page](#), then click on “Releases.”



Choose the latest release and scroll to find the “Assets” section.



Click on the `.h` file to download ArduinoJson as a single file.

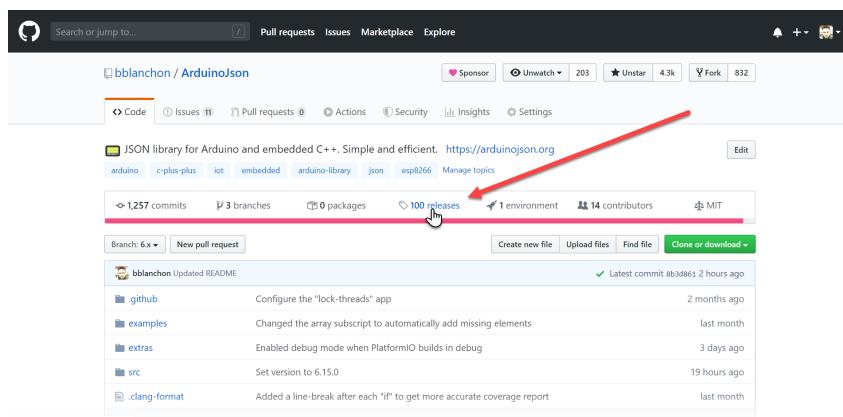
Save this file in your project folder, alongside your source code.

As you can see, there is also a `.hpp` file. This header file is identical to the `.h` file, except that it keeps everything inside the `ArduinoJson` namespace.

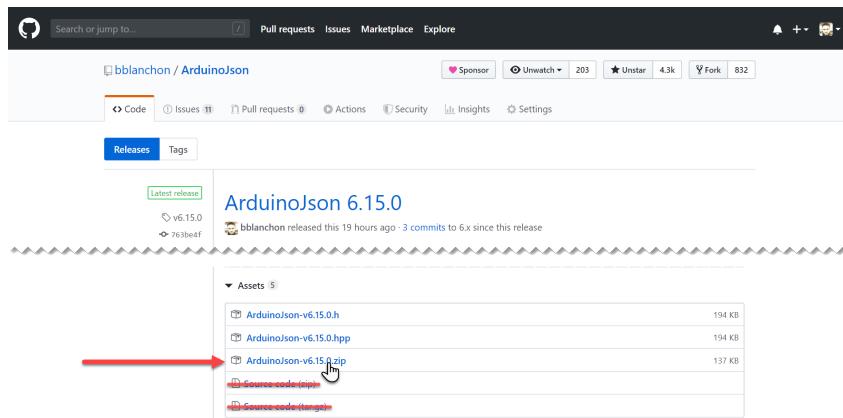
Using the Zip file

If you use an old version of the Arduino IDE (or an alternative), you may not be able to use the Library Manager. In this case, you need to do the job of the Library Manager yourself by downloading and unpacking the library into the right folder.

Go to the [ArduinoJson GitHub page](#), then click on “Releases.”



Choose the latest release and scroll to find the “Assets” section.

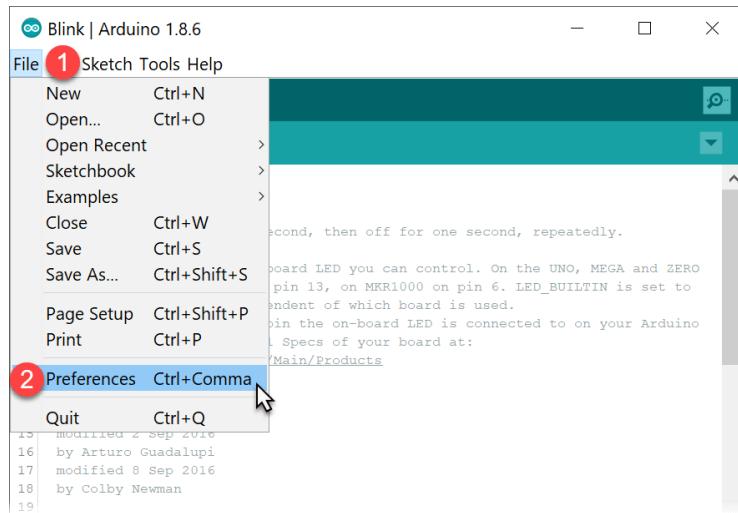


Click on the `ArduinoJson-vX.X.X.zip` file to download the package for Arduino. Don't use the link "Source code (zip)" as it includes unit tests and a few other things you don't need to use the library.

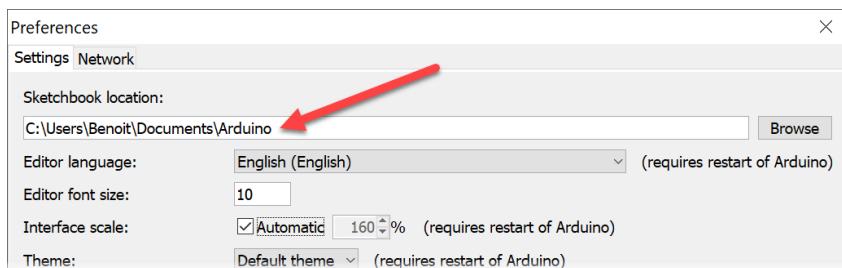
Using your favorite file archiver, unpack the zip file into the Arduino's libraries folder, which is:

```
<Arduino Sketchbook folder>/libraries/ArduinoJson
```

The “Arduino Sketchbook folder” is configured in the Arduino IDE; it is in the “Preferences” window, accessible via the “File” menu.



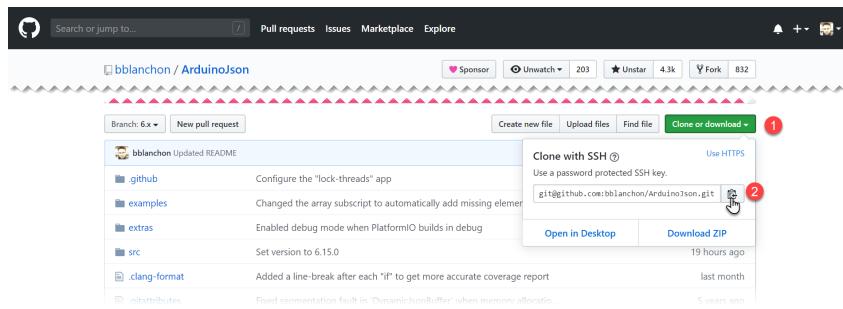
The setting is named "Sketchbook location."



Cloning the Git repository

Finally, you can check out the entire ArduinoJson source code using Git. Using this technique only makes sense if you plan to modify the source code of ArduinoJson, for example, if you want to make a Pull Request.

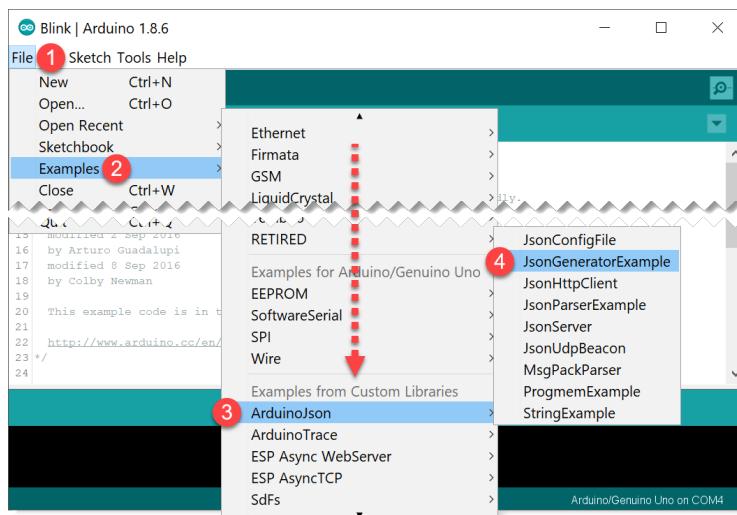
To find the URL of the ArduinoJson repository, go to GitHub and click on “Clone or download.”



If you use the Arduino IDE, perform the Git Clone in the “libraries” as above. If you don’t use the Arduino IDE, then you probably know what you’re doing, so you don’t need my help ;-).

1.3.7 The examples

If you use the Arduino IDE, you can quickly open the examples from the “File” / “Examples” menu.



If you don’t use the Arduino IDE, or if you installed ArduinoJson as a single-header, you can see the examples at arduinojson.org/example.

Here are the ten examples provided with ArduinoJson:

1. `JsonGeneratorExample.ino` shows how to serialize a JSON document and write the result to the serial port.
2. `JsonParserExample.ino` shows how to deserialize a JSON document and print the result to the Serial port.
3. `JsonFilterExample.ino` shows how to filter a large document to get only the parts you want.
4. `JsonConfigFile.ino` shows how to save a JSON document to an SD card.
5. `JsonHttpClient.ino` shows how to perform an HTTP request and parse the JSON document in the response.
6. `JsonServer.ino` shows how to implement an HTTP server that returns the status of analog and digital inputs in a JSON response.
7. `JsonUdpBeacon.ino` shows how to send UDP packets with a JSON payload.
8. `MsgPackParser.ino` shows how to deserialize a MessagePack document.
9. `ProgmemExample.ino` shows how to use Flash strings with ArduinoJson.
10. `StringExample.ino` shows how to use the `String` class with ArduinoJson.

1.4 Summary

In this chapter, we saw what JSON is and what we can do with it. Then we talked about ArduinoJson and saw how to install it.

Here are the key points to remember:

- JSON is a text data format.
- JSON is almost a subset of JavaScript but is more restrictive.
- JSON is a bad choice to transmit raw data.
- ArduinoJson works everywhere, not just on Arduino.
- ArduinoJson is a serialization library, not a container library.
- ArduinoJson uses a fixed-size memory pool.
- ArduinoJson supports MessagePack as well.

In the next chapter, we'll learn a bit of C++ to make sure you have everything you need to use ArduinoJson correctly.

Chapter 2

The missing C++ course

”

Within C++, there is a much smaller and cleaner language struggling to get out.

– Bjarne Stroustrup, [The Design and Evolution of C++](#)

2.1 Why a C++ course?

A common source of struggle among ArduinoJson users is the lack of understanding of some of the C++ fundamentals. That's why, before looking at ArduinoJson in detail, we're going to learn the elements of C++ that are necessary to understand the rest of the book.

There are a lot of beginner's guides to Arduino, they introduce you to the C++ syntax, but omit to explain how it works behind the scenes. This chapter is an attempt to fill this gap.

This course doesn't try to teach you the syntax of C++; there are plenty of excellent resources for that. Instead, it covers what is always left behind:

- How is memory managed?
- What's a pointer?
- What's a reference?
- How are the strings implemented?

Because it would take far too long to cover everything, this chapter focuses on what's important to know before using ArduinoJson. I chose the topics after observing common patterns among ArduinoJson users, especially the ones who come from managed languages like Python, Java, JavaScript, or C#.

In this book, I assume that you already know:

1. How to program in another object-oriented language like Java or C#. The following concepts should be familiar to you:
 - class and instance
 - constructor and destructor
 - scope: global, function, or block
2. How to do basic stuff with Arduino:
 - compile and upload



- Serial Monitor
 - `setup()` / `loop()`
3. How to write simple C / C++ programs:

- `#include`
- `class` / `struct`
- `void, int, float, String`
- functions

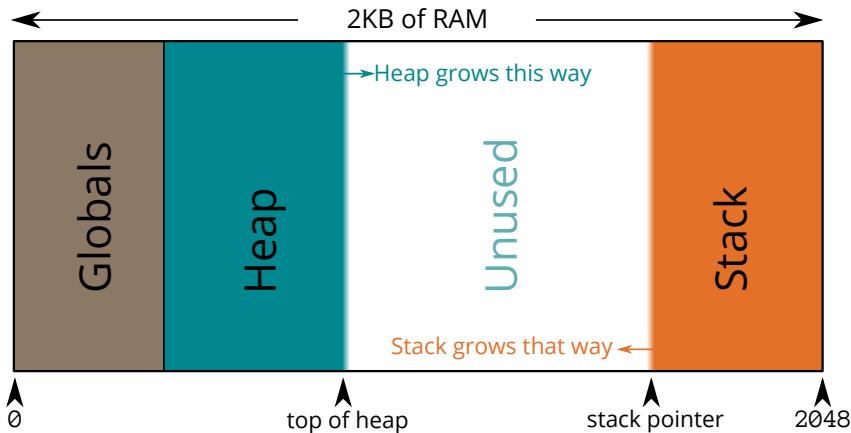
2.2 Stack, heap, and globals

In this section, we'll talk about the RAM of the microcontroller and how the program uses it. The goal here is not to be 100% accurate but to give you the right mental model to understand how C++ deals with memory management.

We'll use the microcontroller Atmel ATmega328 as an example. It is the chip that powers many original Arduino boards (UNO, Duemilanove...), and it is simple and easy to understand.

To see how the RAM works in C++, imagine a huge array that includes all the bytes in memory. The element at index 0 is the first byte of the RAM and so on. For the ATmega328, it would be an array of 2048 elements because it has 2KB of RAM. In fact, it's possible to declare such an array in C++; we'll see that in the section dedicated to pointers.

The compiler and the runtime libraries slice this huge array into three areas that they use for different kinds of data.



The three areas are: “globals,” “heap,” and “stack.” There is also a zone with free unused memory between the heap and the stack.

2.2.1 Globals

The “globals” area contains the global variables:

1. the variables declared out of function or class scope,
2. the variables in a function scope but which are declared `static`,
3. the static members of classes,
4. the string literals.

The size of this area is constant; it remains the same during the execution of the program. All the variables in here are always present in memory; they'd better be very useful because that's memory you cannot use for something else.

Here is a program that declares a global variable:

```
int i = 42; // a global variable

int main() {
    return i;
}
```

You should only use a global variable when it must be available at any given time of the execution, like the serial port. You should use a local variable if the program only needs it for short periods. For example, a variable that is only used during the initialization of the program should be a local variable of the `setup()` function.

String literals are in the “globals” areas, so you need to avoid having many strings in a program (for logging, for example) because it significantly reduces the RAM available for the actual program. Here is an example:

```
// the following string occupies 12 bytes in the "globals" area
const char* myString = "hello world";
```

To prevent strings literals from eating the whole RAM, you can ask the compiler to put them in the Flash memory (the non-volatile memory that holds the program) using the `PROGMEM` and `F()` macros. However, you need to call special functions to be able to use these strings because they are not regular strings. We will see that later when we talk about strings.

2.2.2 Heap

The “heap” contains the variables that are dynamically allocated. Unlike the “globals” area, its size varies during the execution of the program. The heap is mostly used for variables whose size is unknown at compile time or for long-lived variables.

To create a variable in the heap, a program needs to allocate it explicitly. If you’re used to C# or Java, it is similar to variables instantiated via a call to new.

In C++, it is the job of the program to release the memory. Unlike C# or Java, there is no garbage collector to manage the heap. Instead, the program must call a function to release the memory.

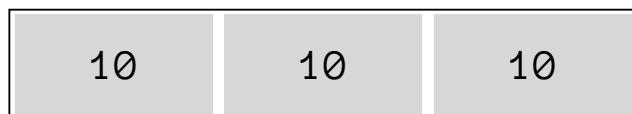
Here is a program that performs allocation and deallocation in the heap:

```
int main() {
    void *p = malloc(42);
    free(p);
}
```

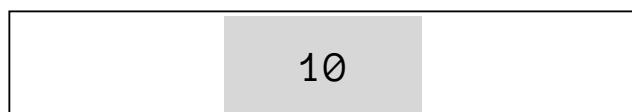
I insist on the fact that it is the role of the *program* and not the role of the *programmer* to manage the heap. Indeed, it is a common misunderstanding that, in C++, memory must be managed manually by the programmer; in reality, the language does that for us, as we’ll see in the [section dedicated to memory management](#).

Fragmentation

The problem with the heap is that releasing a block leaves a hole of unused memory. For example, let’s say you have 30 bytes of memory and you allocated three blocks of 10 bytes:



Now, imagine that the first and third blocks are released.



There are now 20 bytes of free memory. However, it is impossible to allocate a block of 20 bytes, since the free memory is made of several blocks. This phenomenon, called “heap fragmentation,” is the bane of embedded systems because it wastes the precious RAM.

For more information, see my article [What is Heap Fragmentation?](#).



Heap is optional

If your microcontroller has a very small amount of RAM (less than 16KB), it's best not to use the heap at all. Not only does it reduces RAM usage, but it also reduces the size of the program because the memory management functions can be removed.



When the heap is forbidden

Deeply embedded system with high safety requirements usually forbids the use of dynamic memory allocation.

Indeed, in a program that only uses fixed memory allocation, the success or failure of a function only depends on the current state of the program. However, as soon as dynamic memory allocation is involved, the success or failure of a function depends on the current state of the heap, so it depends on prior execution. It becomes impossible to verify the safety of the code formally.

2.2.3 Stack

The “stack” contains:

1. the local variables (i.e., the one declared in a function scope),
2. the function parameters,
3. the return addresses of function calls.

The stack size changes continuously during the execution of the program. Allocating a variable in the stack is almost instantaneous as it only requires changing the value of a register, the stack pointer. In most architectures, the stack pointer moves backward: it starts at the end of the memory and is decremented when an allocation is made.

When a program declares a local variable in a function, the compiler emits the instruction to decrease the stack pointer by the size of the variable.

When a program calls a function, the compiler emits the instructions to copy the parameters and the current position in the program (the instruction pointer) to the stack. This last value allows jumping back to the site of invocation when the function returns.

Here is a program that declares a variable in the stack:

```
int main() {
    int i = 42; // a local variable
    return i;
}
```



Stack size

While it is not the case for the ATmega328, many architectures limit the stack size. For example, the ESP8266 core for Arduino restricts the stack to 4KB, although it can be adjusted by changing the configuration.

You might wonder what happens when the stack pointer crosses the top of the heap. Well, sooner or later, the memory of the stack gets overwritten; therefore, the return addresses are wrong, and the program pointer jumps to an incorrect location, causing the program to crash.



The code is good but crashes anyway?

If your program crashes in a very unpredictable way, you likely have a stack corruption. To fix that, you need to reduce the memory consumption of your program or upgrade to a bigger microcontroller.

2.3 Pointers

Novice C and C++ programmers are always afraid of pointers, but there is no reason to be. On the contrary, pointers are very simple.

2.3.1 What is a pointer?

As I said in the previous section, the RAM is simply a huge array of bytes. We can read any byte by using an index in the array; this index would *point* to a specific byte.

To picture what a *pointer* is, just think about this index. A pointer is a variable that stores an address in memory. It is nothing more than an integer whose value is the index in our huge array.

To be exact, the value of the pointer doesn't exactly match the index because the beginning of the RAM is not at address `0`. For example, in the ATmega328, the RAM starts at address `0x100` or `256`. Therefore, if you want to read the 42nd byte of the RAM, you must use a pointer whose value is `0x100 + 42`.

Apart from this constant offset, the metaphor of a pointer being an index is perfectly valid.



What is there between `0` and `0x100`?

You may be wondering what is at the beginning of the address space (between `0` and `0x100`). These are “magic” addresses that map to the registers and the devices of the microcontroller. Internal Arduino functions, like `digitalWrite()`, use these addresses.

2.3.2 Dereferencing a pointer

Let's see how we can use a pointer to read a value in memory. Imagine that the RAM has the following content:

address	value
<code>0x100</code>	<code>42</code>
<code>0x101</code>	<code>43</code>
<code>0x102</code>	<code>44</code>
...	...

We can create a program that sets a pointer to `0x100` and uses it to read `42`:

```
// Create a pointer to the byte at address 0x100
byte* myPointer = 0x100;

// Print "42", the value of the byte at address 0x100
Serial.println(*myPointer);
```

As you can see in the declaration of `myPointer`, we use a star (*) to declare a pointer. At the left of the star, we need to specify the type of value pointed by the pointer; it's also the type of value that we can read from this pointer.

Then, you can see that we also use the star to read the value pointed by the pointer; we call that “dereferencing a pointer.” If we want to print the value of the pointer, i.e., the address, we need to remove the star and cast the pointer to an integer:

```
// Create a pointer to the byte at address 0x100
byte* myPointer = 0x100;

// Print "0x100"
Serial.println((int)myPointer, HEX);
```

2.3.3 Pointers and arrays

There is an alternative way to dereference a pointer with array syntax. We can write the same program this way:

```
// Create a pointer to the byte at address 0x100
byte* myPointer = 0x100;

// Print "42", the value of the byte at address 0x100
Serial.println(myPointer[0]);
```

Here, the `0` means we want to read the first value at the specified address. If we use `1` instead of `0`, it means we want to read the following value in memory. In our example, that would be the address `0x101`, where the value `43` is stored.

The computation of the address depends on the type of the pointer. If we had used a different type than `byte`, for example, `short` whose size is two bytes, the `myPointer[1]` would have read the value at address `0x102`.

By now, you should start to see that arrays and pointers are very similar in C. In fact, they are equivalent most of the time; you can use an array as a pointer and a pointer as an array.

2.3.4 Taking the address of a variable

Up till now, we used a hard-coded value for the pointer, but we can also take the address of an existing variable. Here is a program that stores the address of an integer in a pointer and use the pointer to modify the integer:

```
// Create an integer
int i = 666;

// Create a pointer pointing to the variable i
int* p = &i;

// Modify the variable i via the pointer
*p = 42;
```

As you see, we used the unary operator `&` to get the address of a variable. We used the operator `*` to dereference the pointer, but this time, to modify the value pointed by `p`.

2.3.5 Pointer to class and struct

In C++, object classes are declared with `class` or `struct`. The two keywords only differ by the default accessibility. The members of a `class` are private by default, whereas the members of a `struct` are public by default.



C++ vs. C#

If you come from C#, you may be confused because C++ uses the keywords `class` and `struct` differently.

In C#, a `class` is a reference type, and it is allocated in the (managed) heap. A `struct` is a value type, and it is allocated in the stack (except if it's a member of a `class` or if the value is "boxed").

In C++, `class` and `struct` are identical, only the default accessibility changes. It is the calling program that decides if the variable goes in the heap or the stack.

Like Java and C#, you access the members of an object using the operator `.`, unless you use a pointer. If you have a pointer to the object, you need to replace the `.` with a `->`.

Here is a program that uses both operators:

```
// Declare a structure
struct Point {
    int x;
    int y;
};

// Create an instance in the stack
Point center;

// Set the member directly
center.x = 0;

// Get a pointer to the instance
Point* p = &center;

// Modify the member via the pointer
p->x = 0;
```

2.3.6 Pointer to constant

When you lend a disk to a friend, you hope she will give it back in the same condition. The same can be true with variables. Your program may want to share a variable with

the agreement that it will not be modified. For this purpose, C++ offers the keyword `const` that allows marking pointers as a “pointer to constant,” meaning that the variable cannot be modified.

Here is an example

```
// Same as above
Point center;

const Point* p = &center;

// error: assignment of member 'Point::x' in read-only object
p->x = 2;
```

As you see, the compiler issues an error as soon as you try to modify a variable via a pointer-to-const.

This feature takes all its sense when a function takes a pointer as an argument. For example, compare the two following functions:

```
void translate(Point* p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}

void print(const Point* p) {
    Serial.print(p->x);
    Serial.print(", ");
    Serial.print(p->y);
}
```

`translate()` needs to modify the `Point`, so it must receive a non-`const` pointer to the structure. `print()`, however, only needs to read the information within `Point`, so a pointer-to-const suffices.

What does it mean for you? It means that you can always call `print()`, but you can only call `translate()` if you are allowed to, i.e., if someone gave you a non-`const` pointer to `Point`. For example, the function `print()` cannot call `translate()`. That makes sense, right?

Constness is one of my favorite features of C++, but to understand its full potential, you need to practice and play with it. For beginners, the difficulty is that the constness (or, more precisely, the non-constness) is contagious. If you want to mark a function parameter as pointer-to-const, you first need to make sure that all functions that it calls also take pointer-to-const.

The easiest way to use constness in your programs is to consider pointer-to-const to be the default and only switch to a non-const pointer when needed. That way, you are sure that all functions that do not modify an object take a pointer-to-const.

2.3.7 The null pointer

Zero is a special value to mean an empty pointer, just as you'd use `null` in Java or C#, or `None` in Python.

Just like an integer, a pointer whose value is zero evaluates to a false expression, whereas any other value evaluates to true. The following program leverages this feature:

```
if (p) {  
    // Pointer is not null :-)  
} else {  
    // Pointer is null :-(  
}
```

You can use `0` to create a null pointer; however, there is a global constant for that purpose: `nullptr`. The intent is more explicit with a `nullptr`, and it has its own type (`nullptr_t`) so that you won't accidentally call a function overload taking an integer.

The program above can also be written using `nullptr`:

```
if (p != nullptr) {  
    // Pointer is not null :-)  
} else {  
    // Pointer is null :-(  
}
```

2.3.8 Why use pointers?

C programmers use pointers for the following tasks:

1. To get access to a specific location in memory (as we did above).
2. To track the result of dynamic allocation (more in the next section).
3. To pass a parameter to a function when copying is expensive (e.g., a big struct).
4. To keep a dependency on an object that we don't own.
5. To iterate over an array.
6. To pass a string to a function (more on that later).

C++ programmers prefer references to pointers; we'll see that in [the dedicated section](#).

2.4 Memory management

In this section, we'll see how to allocate and release memory in the heap. As you'll see, C++ doesn't impose to manage the memory manually; in fact, it's quite the opposite.

2.4.1 `malloc()` and `free()`

The simplest way to allocate a bunch of bytes in the heap is to use the `malloc()` and `free()` functions inherited from C.

```
void* p = malloc(42);
free(p);
```

The first line allocates 42 bytes in the heap. If there is not enough space left in the heap, `malloc()` returns `nullptr`. The second line releases the memory at the specified address.

This is the way C programmers manage their memory, but C++ programmers don't like `malloc()` and `free()` because they don't call constructors and destructors.

2.4.2 `new` and `delete`

The C++ versions of `malloc()` and `free()` are the operators `new` and `delete`. Behind the scenes, these operators are likely to call `malloc()` and `free()`, but they also call constructors and destructors.

Here is a program that uses these operators:

```
// Declare a class
struct MyStruct {
    MyStruct() {} // constructor
    ~MyStruct() {} // destructor
    void myFunction() {} // member function
};

// Instantiate the class in the heap
```

```
MyStruct* str = new MyStruct();  
  
// Call a member function  
str->myFunction();  
  
// Destruct the instance  
delete str;
```

This feature is very similar to the `new` keyword in Java and C#, except that there is no garbage collector. If you forget to call `delete`, the memory cannot be reused for other purposes; we call that a “memory leak.”

Calling `new` and `delete` is the canonical way of allocating objects in the heap; however, seasoned C++ programmers prefer avoiding this technique as it’s likely to cause a memory leak. Indeed, it’s very difficult to make sure the program calls `delete` in every situation. For example, if a function has multiple return statements, we must ensure that every path calls `delete`. If exceptions can be thrown, we must ensure that a `catch` block will call the destructor.



No `finally` in C++

One could be tempted to use a `finally` clause to call `delete`, as we do in C#, Java or Python, but there is no such clause in C++, only `try` and `catch` are available.

This is a conscious design decision from the creator of C++. He prefers to encourage programmers to use a superior technique called “RAII”; more on that later.

2.4.3 Smart pointers

To make sure `delete` is always called, C++ programmers use a “smart pointer” class: a class whose destructor will call the `delete` operator.

Indeed, unlike garbage-collected languages where objects are destructed in a non-deterministic way, in C++, a local object is destructed as soon as it goes out of scope. Therefore, if we use a local object as a smart pointer, the `delete` operator is guaranteed to be called.

Here is an example:

```
// Same structure as before
class MyStruct {
    MyStruct() {} // constructor
    ~MyStruct() {} // destructor
    myFunction() {} // member function
};

// Declare a smart pointer for MyStruct
class MyStructPtr {
public:
    // Constructor: simply save the pointer
    MyStructPtr(MyStruct *p) : _p(p) {}

    // Destructor: call the delete operator
    ~MyStruct() {
        delete _p;
    }

    // Replace operator -> to return the actual pointer
    MyStruct* operator->() {
        return _p;
    }

private:
    // a pointer to the MyStruct instance
    MyStruct *_p;
};

// Create the instance of MyStruct and
// capture the pointer in a smart pointer
MyStructPtr p(new MyStruct());

// Call the member function as if we were using a raw pointer
p->myFunction();

// The destructor of MyStructPtr will call delete for us
```

As you see, C++ allows overloading built-in operators, like `->`, so that the smart pointer

keeps the same semantics as the raw pointer.

MyStructPtr is just an introduction to the concept of smart pointer; there are many other things to implement to get a complete smart pointer class.



unique_ptr and shared_ptr

C++ usually offers two smart pointer classes. `std::unique_ptr` implements what we've just seen and handles every tricky detail. `std::shared_ptr` adds reference counting, which gives a programming experience very similar to Java or C#.

Unfortunately, these classes are usually not available on Arduino, only a few cores (notably the ESP8266) have them.

2.4.4 RAI

With the smart pointer, we saw an implementation of a more general concept called “RAII.”

RAII is an acronym for Resource Acquisition Is Initialization. It's an idiom (i.e., a design pattern) that requires that every time you acquire a resource, you must create an object whose destructor will release the resource. Here, a resource can be either a memory block, a file, a mutex, etc.



The String class

Arduino's [String class](#) is an example of RAII with a memory resource. Indeed, the constructor copies the string to the heap, and the destructor releases the memory.



If you must remember only one thing from this book

RAII is the most fundamental C++ idiom. Never release a resource manually, always use a destructor to do it for you. It's the only way you can write code without memory leaks.

2.5 References

2.5.1 What is a reference?

In C++, a “reference” is an alias (i.e., another name) for a variable.

A reference must be initialized to be attached to a variable and cannot be detached. Here is an example:

```
// Same Point struct as before
Point center;

// Create a reference to center
Point& r = center;

// Modify center via the reference
r.x = 0;

// Now center.x == 0
```

As you can see, we use `&` to declare a reference, just like we used `*` to declare a pointer.

2.5.2 Differences with pointers

The example above could be rewritten with a pointer:

```
// Still the same Point struct
Point center;

// Create a pointer pointing to center
Point* p = &center;

// Modify center via the pointer
p->x = 0;

// Now center.x == 0
```

Pointers and references are very similar, except that references keep the value syntax. With a reference, you keep using `.` as if you were dealing with the actual variable, whereas, with a pointer, you need to use `->`.

Nevertheless, once compiled, pointers and references are identical. They translate to the same assembler code; only the syntax differs.

2.5.3 Reference to constant

Just like we added `const` in front of a pointer declaration to make a pointer-to-const, we can use `const` to declare a reference-to-const.

Like pointer-to-const, a reference-to-const only permits read access to a variable, which is very useful when your program needs to pass a variable to a function with the guarantee that it will not be modified. Here is an example:

```
// Again, the same Point struct
Point center;

// Create a reference-to-const to center
const Point& r = center;

// error: assignment of member 'Point::x' in read-only object
r.x = 0;
```

That's not all; there is an extra feature with reference-to-const. If you try to create a reference to a temporary object, for example, a `Point` returned by a function, the compiler emits an error because the reference would inevitably point to a destructed object. Instead, if you use a reference-to-const, the compiler extends the lifetime of the temporary, so that the reference-to-const points to an existing object. Here is an example:

```
Point getCenter() {
    Point center;
    center.x = 0;
    center.y = 0;
    return center;
}
```

```
// error: invalid initialization of non-const reference of type 'Point&'  
→   from  
// an rvalue of type 'Point'  
Point& center = getCenter();  
  
// OK, the reference-to-const extends the lifetime of the temporary  
const Point& center = getCenter();
```

Why is this important? This feature allows functions that take an argument of type reference-to-const to accept temporary values without the need to create a copy. Here is an example:

```
int time = 42;  
Serial.println(String("Result: ") + time);
```

Indeed, `String::operator+` returns a temporary `String`, but since `Serial::println()` takes a reference-to-const, it can read the temporary `String` without making a copy.

2.5.4 Rules of references

Compared to pointers, references provide additional compile-time safety:

1. A reference must be assigned when created. Therefore, a reference cannot be in an uninitialized state.
2. A reference cannot be reassigned to another variable.
3. The compiler emits an error when you create a (non-const) reference to a temporary.

2.5.5 Common problems

Despite all these properties, the references expose the same weakness as pointers.

You may think that, by definition, a reference cannot be null, but it's wrong:

```
// Create a null pointer  
int *p = nullptr;
```

```
// Create a reference to the value pointed by p
int& r = *p;
```

Admittedly, it is a horrible example, but it shows that you can put anything in a reference; the compile-time checks are not bullet-proof.

As with pointers, the main danger is a dangling reference: a reference to a destructed variable. This problem happens when you create a reference to a temporary variable. Once the variable is destructed, the reference points to an invalid location. The compiler can detect some of these, but it's more an exception than a rule.

2.5.6 Usage for references

C++ programmers tend to use references where C programmers use pointers:

1. Passing parameters to function when copying is expensive.
2. Keeping a dependency on an object that we do not own.

2.6 Strings

2.6.1 How are the strings stored?

There are several ways to declare a string in C++, but the memory representation is always the same. In every case, a string is a sequence of characters terminated by a zero. The zero marks the end of the string and is called the “terminator.”

For example, the string `"hello"` is translated to the following sequence of byte:

'h'	'e'	'l'	'l'	'o'	0
-----	-----	-----	-----	-----	---

As you see, a string of 5 characters occupies 6 bytes in memory.



Outside of Arduino

We saw the most common way to encode strings in C++, but there are other ways. For example, in the Qt framework, strings are encoded in UTF-16, using two bytes per character, like in C# and Java.

2.6.2 String literals in RAM

Here are three ways to declare strings in C++:

```
const char* s = "hello";
char s[] = "hello";
String s = "hello";
```

We'll see how these forms differ next, but before, let's focus on the common part: the string literal `"hello"`. The three expressions above cause the same 6 bytes to be added to the “globals” area of the RAM. As we saw, this area should be as small as possible because it limits the remaining RAM for the rest of the program.

The compiler is smart enough to detect that a program uses the same string several times. In this case, instead of storing several copies of the string, it only stores one. This feature is called “String Interning”; we'll see it in action in a moment.

2.6.3 String literals in Flash

To reduce the size of the “globals” section, we can instruct the compiler to keep the strings within the Flash memory, alongside the program.

Here is how we can modify the code above to keep a string in “program memory”:

```
const char s[] PROGMEM = "hello";
```

As you see, I made two changes:

1. I used the `PROGMEM` attribute to tell the compiler that this variable is in “program memory.”
2. I marked the array as `const` because the program memory is read-only.

As the string literal `"hello"` is now stored in the Flash memory, the pointer `s` is not a regular pointer because it refers to an address in the program space, not in the memory space.

The functions designed for RAM strings do not support Flash strings. To use Flash strings, a program needs to call dedicated functions. For instance, instead of calling `strcpy()` to copy a string, it needs to call `strcpy_P()`.

Fortunately, many functions in Arduino (most notably in the classes `String` and `Serial`) call the right functions when you pass a Flash string. However, they need a way to detect if the pointer refers to a location in RAM or Flash. To differentiate between the two address spaces, we must mark the string with a special type: `_FlashStringHelper`. To do that, we simply need to cast the pointer to `const _FlashStringHelper*`:

```
const char s[] PROGMEM = "hello";
Serial.print((const _FlashStringHelper*)s);
```



Don't forget to cast!

If you call such function and forget to cast the pointer to `const _FlashStringHelper*`, the function will believe that the string is in RAM, and the program will probably crash instantly.

There is a short-hand syntax to declare literal in Flash memory and cast it appropriately: the `F()` macro. This macro is convenient because you can use it “in place,” like this:

```
Serial.print(F("hello"));
```



No string interning with F()

Unfortunately, `F()` prevents the string interning from happening. So, if you call this macro multiple times with the same string, the compiled executable will contain several copies of the same string.

Many functions need to copy the entire string to RAM before using it. That is especially true for the `String` class and `ArduinoJson`. Therefore, make sure that only a few copies are in RAM at any given time; otherwise, you would end up using more RAM than with regular strings.



When to use Flash string?

As you see, Flash strings have several caveats: it's easy to shoot yourself in the foot. Moreover, they cause significant overhead in code size and speed. I recommend using them only for long strings that are rarely used, like log messages.

2.6.4 Pointer to the “globals” section

Let's have a closer look at the first syntax introduced in this section:

```
const char* s = "hello";
```

As we saw, this statement creates an array of 6 bytes in the “globals” section of the RAM. It also creates a pointer named `s`, pointing to the beginning of the array. The pointer is marked as `const` as the compiler assumes that all strings in the “globals” section are `const`.

If you remove the `const` keyword, the compiler emits a warning because you are not supposed to modify the content of a string literal. Whether it is possible or not to modify the string depends on the platform. On an ATmega328, it will work because there is

no memory protection. However, on other platforms, such as a PC or an ESP8266, the program will cause an exception.

Remember the “string interning” feature we talked about earlier? Here is an example to see it in action:

```
const char* s1 = "hello";
const char* s2 = "hello";
if (s1 == s2) {
    // s1 and s2 point the same memory location
    // proving that there is only one copy of "hello"
} else {
    // this will never happen, even with optimizations disabled
}
```

2.6.5 Mutable string in “globals”

Let's see the second syntax:

```
char s[] = "hello";
```

When written in the global scope (i.e., out of any function), this expression allocates an array of 6 bytes initially filled with the `"hello"` string. Writing at this location is allowed.

If you need to allocate a bigger array, you can specify the size in the brackets:

```
char s[32] = "hello";
```

This way, you reserve space for a larger string, in case your program needs it.

By the way, we can see that we defeat the “string interning” with this syntax:

```
char s1[] = "hello";
char s2[] = "hello";
if (s1 == s2) {
    // this will never happen, even with optimizations enabled
} else {
```

```
// s1 and s2 point to different memory locations  
// proving that there are two copies of "hello"  
}
```

Does that surprise you? Maybe you expected the operator == to compare the content of the string? Unfortunately, it doesn't; instead, it compares the pointers, i.e., the addresses of the strings.



Comparing strings

If you need to compare the content of the strings, you need to call the function strcmp() which returns 0 if the strings match:

```
if (strcmp(s1, s2) == 0) {  
    // strings pointed by s1 and s2 match  
} else {  
    // strings pointed by s1 and s2 differ  
}
```

2.6.6 A copy in the stack

If we use the same syntax in a function scope, the behavior is different:

```
void myFunction() {  
    char s[] = "hello";  
    // ...  
}
```

When the program enters `myFunction()`, it allocates 6 bytes in the stack, and it fills them with the content of the string. The string "hello" is still present in the "globals" section, but a copy is made in the stack.



Code smell

This syntax causes the same string to be present twice in memory. It only makes sense if you want to make a copy of the string, which is rare.



Prefer uninitialized arrays

If you need an array of char in the stack, don't initialize it:

```
void myFunction() {  
    char s[32];  
    // ...  
}
```

2.6.7 A copy in the heap

We just saw how to get a copy of a string in the stack, now let's see how to copy in the heap. The third syntax presented was:

```
String s = "hello";
```

Thanks to implicit constructor call, this expression is identical to:

```
String s("hello");
```

As before, this expression creates a byte array in the “globals” section. Then, it constructs a `String` object and passes a pointer to the string to the constructor of `String`. The constructor makes a dynamic memory allocation (using `malloc()`) and copies the content of the string.



Code smell

This syntax causes the same string to be present twice in memory.

Unfortunately, many Arduino libraries force you to use `String`, causing useless copies. That is why `ArduinoJson` never imposes to use a `String` when a `char*` can do the job.



From Flash to heap

You can combine the `F()` macro with the `String` constructor, and you get a string in RAM that doesn't bloat the "globals" section.

```
String s = F("hello");
// or
String s(F("hello"));
```

This is a good usage of the `String` class, but there are still dynamic allocation and duplication behind the scenes.

2.6.8 A word about the `String` class

I rarely use the `String` class in my programs. Indeed, `String` relies on the two things I try to avoid in embedded code:

1. Dynamic memory allocation
2. Duplication

Most of the time, an instance of `String` can be replaced by a `char[]`, and most string manipulations by a call to `sprintf()`.

Here is an example:

```
int answer = 42;

// BAD: using String
String s = String("Answer is ") + answer;

// GOOD: using char[]
char s[16];
sprintf(s, "Answer is %d", answer);
```

`sprintf()` is part of the C Standard Library; I invite you to open [your favorite C book](#) for more information.

We can improve this program by moving the format string to the Flash memory:

```
sprintf_P(s, F("Answer is %d"), answer);
```

`sprintf_P()` combines several interesting properties: it's versatile, easy to use, easy to read, and memory efficient. I call it "the Holy Grail."

For more information, see my article [How to format strings without the String class](#).



Everything you know is wrong

If you come from Java or C#, using a `String` object surely feels more natural. Here, you are in a whole different world, so you need to reconsider everything you take for granted. Yes, prefer fixed-size strings to variable-size strings because they are faster, smaller, and more reliable.

2.6.9 Pass strings to functions

By default, when a program calls a function, it passes the arguments by value; in other words, the caller gives a copy of each argument to the callee. Let's see why it is especially important with strings.

```
void show(String s) {
    Serial.println(s);
}
String s = "hello world";
show(s); // pass by value
```

As expected, the program above prints "hello world" to the serial port. However, it is suboptimal because it makes an unnecessary copy of the string when it calls `show()`. When the microcontroller executes `show()`, the string "hello world" is present twice in memory. It might be acceptable for small strings, but it's dramatic for long strings (e.g., JSON documents) because they consume a lot of memory and take a long time to copy.

The solution? Change the signature of the function to take a reference, or better, a reference-to-const since we don't need to modify the `String` in the function.

```
void show(const String& s) {
    Serial.println(s);
}
String s = "hello world";
show(s); // pass by reference
```

The program above works like the previous one, except that it doesn't pass a copy of the string to `show()`; instead, it just passes a reference, which is just a fancy syntax to pass the address of the object. As a result, this upgraded program runs faster and uses less memory.

What about non-object strings? As we saw, a non-object string is either a pointer-to-char or an array-of-char. If it's a pointer-to-char, then there is no harm in passing the pointer by value because doing so doesn't copy the string. Concerning arrays, C (and therefore C++) always pass them by address, so the string isn't copied either. As we saw, arrays and pointers are often interchangeable.

```
void show(const char* s) {
    Serial.println(s);
}
const char* s1 = "hello world";
show(s1); // pass pointer by copy
char s2[] = "hello world";
show(s2); // pass array by address
```

The program above demonstrates how a function taking a `const char*` happily accepts a `char[]`, and the opposite is also true; this is a curiosity of the C language.

By the way, do you know that you can call the same function if you have a `String` object? You simply need to call `String::c_str()`, as below.

```
void show(const char* s) {
    Serial.println(s);
}
String s = "hello world";
show(s.c_str()); // pass a pointer to the internal array
```

2.7 Summary

That's the end of our C++ course. There is still a lot to cover, but it's not the goal of this book. I intentionally simplified some aspects to make this book accessible to beginners, so don't fool yourself into believing that you know C++ in depth.

However, this chapter should be enough to learn what remains by yourself. By the way, I started a blog called "C++ for Arduino," where I shared some C++ recipes for Arduino programmers. Please have a look at cpp4arduino.com.

Here are the key points to remember from this chapter:

- The three areas:
 - The "globals" area contains the global variables and the strings.
 - The "stack" contains the local variables, the arguments passed to functions, and the return addresses.
 - The "heap" contains the dynamically allocated variables.
- Pointers and references:
 - A pointer is a variable that contains an address.
 - A null pointer is a pointer that contains the address zero. We use this value to mean that a pointer is empty.
 - To dereference a pointer, you use the operators `*` and `->`.
 - Arrays and pointers are often interchangeable.
 - A reference is similar to a pointer except that a reference has value syntax.
 - You can use the keyword `const` to declare a read-only pointer or a read-only reference.
 - You should declare pointers and references as `const` by default, and remove the constness only if needed.
- Memory management:
 - `malloc()` allocates memory in the heap, and `free()` releases it.
 - Failing to call `free()` causes a memory leak.

- The C++ operators `new` and `delete` are similar to `malloc()` and `free()`, except that they call constructors and destructors.
 - You don't have to call `free()` or `delete` explicitly; instead, you should use a smart-pointer to do that.
 - Smart-pointer is an instance of RAII, the most fundamental idiom in C++.
 - Heap fragmentation reduces the actual capacity of the RAM.
- Strings:
 - There are several ways to declare a variable that contains (or points to) a string.
 - Depending on the syntax, the same string may be present several times in the RAM.
 - You can move constant strings to the Flash memory, but there are many caveats.
 - `String` objects are attractive but inefficient, so you should use `char` arrays instead.

In the next chapter, we'll use `ArduinoJson` to deserialize a JSON document.

Chapter 3

Deserialize with ArduinoJson

”

It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!

– Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship

3.1 The example of this chapter

Now that you're familiar with JSON and C++, we're going to learn how to use ArduinoJson. This chapter explains everything there is to know about deserialization. As we've seen, deserialization is the process of converting a sequence of bytes into a memory representation. In our case, it means converting a JSON document to a hierarchy of C++ structures and arrays.

In this chapter, we'll use a JSON response from GitHub's API as an example. As you already know, GitHub is a hosting service for source code; what you may not know, however, is that GitHub provides a very powerful API that allows you to interact with the platform.

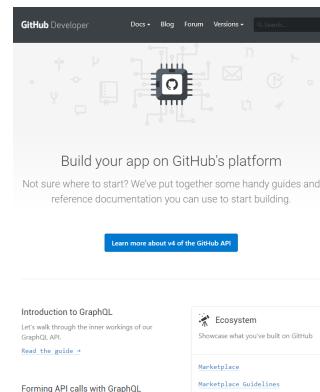
There are many things we could do with GitHub's API, but in this chapter, we'll only focus on a small part. We'll get your ten most popular repositories and display their names, numbers of stars, and numbers of opened issues.

There are several versions of GitHub's API; we'll use the latest one: the GraphQL API (or v4). We'll use this one because it allows us to make only one request and because it returns a considerably smaller response.

If you want to run the example, you'll need a user account on GitHub and a personal access token. Don't worry; we'll see that later.

Because GitHub only allows secure connections, we need a microcontroller that supports HTTPS. We'll use the ESP8266 with the `ESP8266HTTPClient` as an example. If you want to use `ArduinoJson` with `EthernetClient`, `WiFiClient`, or `WiFiClientSecure`, check out the case studies in the last chapter.

Now that you know where we are going, we'll back up a few steps and start with a basic example. Then, we'll progressively learn new things so that, by the end of the chapter, we'll finally be able to interact with GitHub.



3.2 Deserializing an object

We'll begin this tutorial with the simplest situation: a JSON document in memory. More precisely, our JSON document resides in the stack in a writable location. This fact is going to matter, as we will see later.

3.2.1 The JSON document

Our example is the repository information for ArduinoJson:

```
{  
  "name": "ArduinoJson",  
  "stargazers": {  
    "totalCount": 4241  
  },  
  "issues": {  
    "totalCount": 12  
  }  
}
```

As you see, it's a JSON object that contains two nested objects. It contains the name of the repository, the number of stars, and the number of open issues.

3.2.2 Placing the JSON document in memory

In our C++ program, this JSON document translates to:

```
char input[] = "{\"name\":\"ArduinoJson\", \"stargazers\":{\"  
  \"totalCount\":4241}, \"issues\":{\"totalCount\":12}}"
```

In the previous chapter, we saw that this code creates a duplication of the string in the stack. We know it's a code smell in production code, but it's a good example for learning. This unusual construction create a *writable* (i.e., not read-only) input string, which is important for our first contact with ArduinoJson.

3.2.3 Introducing JsonDocument

As we saw in the introduction, one of the unique features of ArduinoJson is its fixed memory allocation strategy.

Here is how it works:

1. First, you create a `JsonDocument` to reserve a specified amount of memory.
2. Then, you deserialize the JSON document.
3. Finally, you destroy the `JsonDocument`, which releases the reserved memory.

The memory of the `JsonDocument` can be either in the stack or in the heap. The location depends on the derived class you choose. If you use a `StaticJsonDocument`, it will be in the stack; if you use a `DynamicJsonDocument`, it will be in the heap.

A `JsonDocument` is responsible for reserving and releasing the memory used by ArduinoJson. It is an instance of the RAII idiom that we saw in the previous chapter.



StaticJsonDocument in the heap

I often say that a `StaticJsonDocument` resides in the stack, but it's possible to have it in the heap, for example, if a `StaticJsonDocument` is a member of an object in the heap.

It's also possible to allocate a `StaticJsonDocument` with `new`, but I strongly advise against it because you would lose the RAII feature.

3.2.4 How to specify the capacity?

When you create a `JsonDocument`, you must specify its capacity in bytes.

In the case of `DynamicJsonDocument`, you set the capacity via a constructor argument:

```
DynamicJsonDocument doc(capacity);
```

As it's a parameter of the constructor, you can use a regular variable, whose value can change at run-time.

In the case of a `StaticJsonDocument`, you set the capacity via a template parameter:

```
StaticJsonDocument<capacity> doc;
```

As it's a template parameter, you cannot use a variable. Instead, you must use a constant expression, which means that the value must be computed at compile-time. As we said in the previous chapter, the compiler manages the stack, so it needs to know the size of each variable when it compiles the program.

3.2.5 How to determine the capacity?

Now comes a tricky question for every new user of ArduinoJson: what should be the capacity of my `JsonDocument`?

To answer this question, you need to know what ArduinoJson stores in the `JsonDocument`. ArduinoJson needs to store a data structure that mirrors the hierarchy of objects in the JSON document. In other words, the `JsonDocument` contains objects which relate to one another the same way they do in the JSON document.

Therefore, the capacity of the `JsonDocument` highly depends on the complexity of the JSON document. If it's just one object with few members, like our example, a few dozens of bytes are enough. If it's a massive JSON document, like OpenWeatherMap's response, up to a hundred kilobytes are needed.

ArduinoJson provides macros for computing precisely the capacity of the `JsonDocument`. The macro to compute the size of an object is `JSON_OBJECT_SIZE()`. It takes one argument: the number of members in the object.

Here is how to compute the capacity for our sample document:

```
// Enough space for:  
// + 1 object with 3 members  
// + 2 objects with 1 member  
const int capacity = JSON_OBJECT_SIZE(3) + 2 * JSON_OBJECT_SIZE(1);
```

On an ESP8266, a 32-bit processor, this expression evaluates to 80 bytes. The result would be significantly smaller on an 8-bit processor; for example, it would be 40 bytes on an ATmega328.



A read-only input requires a higher capacity

In this part of the tutorial, we consider the case of a writeable input because it simplifies the computation of the capacity. However, if the input is read-only (for example, a `const char*` instead of `char[]`), you must increase the capacity.

We'll talk about that later, in the section "[Reading from read-only memory](#)".

3.2.6 StaticJsonDocument or DynamicJsonDocument?

Since our `JsonDocument` is small, we can keep it in the stack. By using the stack, we reduce the size of the executable and improve the performance, because we avoid the overhead due to the management of the heap.

Here is our program so far:

```
const int capacity = JSON_OBJECT_SIZE(3) + 2*JSON_OBJECT_SIZE(1);  
StaticJsonDocument<capacity> doc;
```

Of course, if the `JsonDocument` were bigger, it would make sense to move it to the heap. We'll do that later.



Don't forget `const`!

If you forget to write `const`, the compiler produces the following error:

```
error: the value of 'capacity' is not usable in a constant  
→ expression
```

Indeed, a template parameter is evaluated at compile-time, so it must be a constant expression. By definition, a constant expression is computed at compile-time, as opposed to a variable which is computed at run-time.

3.2.7 Deserializing the JSON document

Now that the `JsonDocument` is ready, we can parse the input with `deserializeJson()`:

```
DeserializationError err = deserializeJson(doc, input);
```

`deserializeJson()` returns a `DeserializationError` that tells whether the operation was successful. It can have one of the following values:

- `DeserializationError::Ok`: the deserialization was successful.
- `DeserializationError::IncompleteInput`: the input was valid but ended prematurely, or it was empty; this code often means that a timeout occurred.
- `DeserializationError::InvalidInput`: the input was not a valid JSON document.
- `DeserializationError::NoMemory`: the `JsonDocument` was too small
- `DeserializationError::NotSupported`: the input document was valid, but it contains features that are not supported by the library.
- `DeserializationError::TooDeep`: the input was valid, but it contained too many nesting levels; we'll talk about that later in the book.

I listed all the error codes above so that you can understand how the library works; however, I don't recommend using them directly in your code.

First, `DeserializationError` converts implicitly to `bool`, so you don't have to write `if (err != DeserializationError::Ok)`, you can simply write `if (err)`.

Second, `DeserializationError` has a `c_str()` member function that returns a string representation of the error.

Thanks to these two features of `DeserializationError`, you can simply write:

```
if (err) {  
    Serial.print(F("deserializeJson() failed with code "));  
    Serial.println(err.c_str());  
}
```

In the “[Troubleshooting](#)” chapter, we'll look at each error code and see what can cause the error.

3.3 Extracting values from an object

In the previous section, we created a `JsonDocument` and called `deserializeJson()`, so now, the `JsonDocument` contains a memory representation of the JSON input. Let's see how we can extract the values.

3.3.1 Extracting values

There are multiple ways to extract the values from a `JsonDocument`; let's start with the simplest:

```
const char* name    = doc["name"];
long      stars   = doc["stargazers"]["totalCount"];
int       issues  = doc["issues"]["totalCount"];
```

This syntax leverages two C++ features:

1. Operator overloading: the subscript operator (`[]`) has been customized to mimic a JavaScript object.
2. Implicit casts: the result of the subscript operator is implicitly converted to the type of the variable.

3.3.2 Explicit casts

Not everyone likes implicit casts, mainly because it messes with overload resolution, with template parameter type deduction, and with the `auto` keyword. That's why ArduinoJson offers an alternative syntax with explicit type conversion.



The `auto` keyword

The `auto` keyword is a feature of C++11. In this context, it allows inferring the type of the variable from the type of expression on the right. It is the equivalent of `var` in C#.

Here is the same code adapted for this school of thought:

```
auto name    = doc["name"].as<char*>();
auto stars   = doc["stargazers"]["totalCount"].as<long>();
auto issues  = doc["issues"]["totalCount"].as<int>();
```

**as<char*>() or as<const char*>()**

We could have used `as<const char*>()` instead of `as<char*>()`, it's just shorter this way. These two functions are identical; they both return a `const char*`.

**Implicit or explicit?**

We saw two different syntaxes to do the same thing. They are all equivalent and lead to the same executable.

I prefer the implicit version because it allows using the “or” operator, as we'll see. I use the explicit version only to solve an ambiguity.

3.3.3 When values are missing

We saw how to extract values from an object, but we didn't do error checking. Let's see what happens when a value is missing.

When you try to extract a value that is not present in the document, ArduinoJson returns a default value. This value depends on the requested type:

Requested type	Default value
<code>const char*</code>	<code>nullptr</code>
<code>float, double</code>	<code>0.0</code>
<code>int, long...</code>	<code>0</code>
<code>String</code>	<code>""</code>
<code>JSONArray</code>	a null object
<code>JsonObject</code>	a null object

The two last lines (`JSONArray` and `JsonObject`) happen when you extract a nested array or object, we'll see that in a later section.



No exceptions

ArduinoJson never throws exceptions. Exceptions are an excellent C++ feature, but they produce large executables, which is unacceptable for microcontrollers.

3.3.4 Changing the default value

Sometimes, the default value from the table above is not what you want. In this situation, you can use the operator `|` to change the default value. I call it the “or” operator because it provides a replacement when the value is missing or incompatible.

Here is an example:

```
// Get the port or use 80 if it's not specified
short tcpPort = config["port"] | 80;
```

This feature is handy to specify default configuration values, like in the snippet above, but it is even more useful to prevent a null string from propagating.

Here is an example:

```
// Copy the hostname or use "arduinojson.org" if it's not specified
char hostname[32];
strncpy(hostname, config["hostname"] | "arduinojson.org", 32);
```

`strncpy()`, a function that copies a source string to a destination string, crashes if the source is null. Without the operator `|`, we would have to use the following code:

```
char hostname[32];
const char* configHostname = config["hostname"];
if (configHostname != nullptr)
    strncpy(hostname, configHostname, 32);
else
    strcpy(hostname, "arduinojson.org");
```

We'll see a complete example that uses this syntax in the [case studies](#).

3.4 Inspecting an unknown object

In the previous section, we extracted the values from an object that we knew in advance. Indeed, we knew that the JSON object had three members: a string named “name,” a nested object named “stargazers,” and another nested object named “issues.” In this section, we’ll see how to inspect an *unknown* object.

3.4.1 Getting a reference to the object

So far, we have a `JsonDocument` that contains a memory representation of the input. A `JsonDocument` is a very generic container: it can hold an object, an array, or any other value allowed by JSON. Because it’s a generic container, `JsonDocument` only offers methods that apply unambiguously to objects, arrays, and any other supported type.

For example, we saw that we could call the subscript operator (`[]`), and the `JsonDocument` happily returned the associated value. However, the `JsonDocument` cannot enumerate the members of the object because it doesn’t know, at compile time, if it should behave like an object, or like an array.

To remove the ambiguity, we must get the object within the `JsonDocument`. We do that by calling the member function `as<JsonObject>()`, like so:

```
// Get a reference to the root object
JsonObject obj = doc.as<JsonObject>();
```

And now, we’re ready to enumerate the members of the object!



JsonObject has reference semantics

`JsonObject` is not a copy of the object in the document; on the contrary, it’s a reference to the object in the `JsonDocument`. When you modify the `JsonObject`, you also modify the `JsonDocument`.

In a sense, we can say that `JsonObject` is a smart pointer. It wraps a pointer with a class that is easy to use. Unlike the other smart pointers we talked about [in the previous chapter](#), `JsonObject` doesn’t release the memory for the object when it goes out of scope because that’s the role of the `JsonDocument`.

3.4.2 Enumerating the keys

Now that we have a `JsonObject`, we can look at all the keys and their associated values. In ArduinoJson, a key-to-value association, or a key-value pair, is represented by the type `JsonPair`.

We can enumerate all pairs with a simple for loop:

```
// Loop through all the key-value pairs in obj
for (JsonPair p : obj) {
    p.key() // is a JsonString
    p.value() // is a JsonVariant
}
```

Notice these three points about this code:

1. I explicitly used a `JsonPair` to emphasize the type, but you can use `auto`.
2. The value associated with the key is a `JsonVariant`, a type that can represent any JSON type.
3. You can convert the `JsonString` to a `const char*` with `JsonString::c_str()`.



When C++11 is not available

The code above leverages a C++11 feature called “range-based for loop”. If you cannot enable C++11 on your compiler, you must use the following syntax:

```
for (JsonObject::iterator it=obj.begin(); it!=obj.end(); ++it) {
    it->key() // is a JsonString
    it->value() // is a JsonVariant
}
```

3.4.3 Detecting the type of value

Like `JsonObject`, `JsonVariant` is a reference to a value stored in the `JsonDocument`. However, `JsonVariant` can refer to any JSON value: string, integer, array, object... A `JsonVariant` is returned when you call the subscript operator, like `obj["text"]` (we'll see that this statement is not entirely correct, but for now, we can say it's a `JsonVariant`).

To know the actual type of the value in a `JsonVariant`, you need to call `JsonVariant::is<T>()`, where `T` is the type you want to test.

For example, the following snippet checks if the value is a string:

```
// Is it a string?  
if (p.value().is<char*>()) {  
    // Yes!  
    // We can get the value via implicit cast:  
    const char* s = p.value();  
    // Or, via explicit method call:  
    auto s = p.value().as<char*>();  
}
```

If you use this with our sample document, you'll see that only the member "name" contains a string. The two others are objects, as `is<JsonObject>()` would confirm.

3.4.4 Variant types and C++ types

There are a limited number of types that a variant can use: boolean, integer, float, string, array, object. However, different C++ types can store the same JSON type; for example, a JSON integer could be a `short`, an `int` or a `long` in the C++ code.

The following table shows all the C++ types you can use as a parameter for `JsonVariant::is<T>()` and `JsonVariant::as<T>()`.

Variant type	Matching C++ types
Boolean	<code>bool</code>
Integer	<code>int</code> , <code>long</code> , <code>short</code> , <code>char</code> (all signed and unsigned)
Float	<code>float</code> , <code>double</code>
String	<code>char*</code> , <code>const char*</code>
Array	<code>JsonArray</code>
Object	<code>JsonObject</code>



More on arduinojson.org

The complete list of types that you can use as a parameter for `JsonVariant::is<T>()` can be found in the [API Reference](#).

3.4.5 Testing if a key exists in an object

If you have an object and want to know whether a key is present or not, you can call `JsonObject::containsKey()`.

Here is an example:

```
// Is there a value named "text" in the object?  
if (obj.containsKey("text")) {  
    // Yes!  
}
```

However, I don't recommend using this function because you can avoid it most of the time.

Here is an example where we can avoid `containsKey()`:

```
// Is there a value named "error" in the object?  
if (obj.containsKey("error")) {  
    // Get the text of the error  
    const char* error = obj["error"];  
    // ...  
}
```

The code above is not horrible, but it can be simplified and optimized if we just remove the call to `containsKey()`:

```
// Get the text of the error  
const char* error = obj["error"];  
  
// Is there an error after all?  
if (error != nullptr) {  
    // ...  
}
```

This code is faster and smaller because it only looks for the key "error" once (whereas the previous code did it twice).

3.5 Deserializing an array

3.5.1 The JSON document

We've seen how to parse a JSON object from GitHub's response; it's time to move up a notch by parsing an array of objects. Indeed, our goal is to display the top 10 of your repositories, so there will be up to 10 objects in the response. In this section, we'll suppose that there are only two repositories, but you and I know that it will be more in the end.

Here is the new sample JSON document:

```
[  
  {  
    "name": "ArduinoJson",  
    "stargazers": {  
      "totalCount": 4241  
    },  
    "issues": {  
      "totalCount": 12  
    }  
  },  
  {  
    "name": "pdfium-binaries",  
    "stargazers": {  
      "totalCount": 116  
    },  
    "issues": {  
      "totalCount": 9  
    }  
  }]
```

3.5.2 Parsing the array

Let's deserialize this array. You should now be familiar with the process:

1. Put the JSON document in memory.
2. Compute the size with `JSON_ARRAY_SIZE()`.
3. Allocate the `JsonDocument`.
4. Call `deserializeJson()`.

Let's do it:

```
// Put the JSON input in memory (shortened)
char input[] = "[{\\"name\\":\\"ArduinoJson\\",\\"stargazers\\":...}];

// Compute the required size
const int capacity = JSON_ARRAY_SIZE(2)
    + 2*JSON_OBJECT_SIZE(3)
    + 4*JSON_OBJECT_SIZE(1);

// Allocate the JsonDocument
StaticJsonDocument<capacity> doc;

// Parse the JSON input
DeserializationError err = deserializeJson(doc, input);

// Parse succeeded?
if (err) {
    Serial.print(F("deserializeJson() returned "));
    Serial.println(err.c_str());
    return;
}
```

As said earlier, an hard-coded input like that would never happen in production code, but it's a good step for your learning process.

You can see that the expression for computing the capacity of the `JsonDocument` is quite complicated:

- There is one array of two elements: `JSON_ARRAY_SIZE(2)`
- In this array, there are two objects with three members: `2*JSON_OBJECT_SIZE(3)`
- In each object, there are two objects with one member: `4*JSON_OBJECT_SIZE(1)`

3.5.3 The ArduinoJson Assistant

For complex JSON documents, the expression to compute the capacity of the `JsonDocument` becomes impossible to write by hand. I did it above so that you understand the process, but in practice, we use a tool to do that.

This tool is the “ArduinoJson Assistant.” You can use it online at arduinojson.org/assistant.

The screenshot shows the ArduinoJson Assistant interface. On the left, under the "Input" section, there is a code editor containing a JSON document. The JSON structure includes arrays and objects, such as "issues" and "stargazers". Below the code editor, it says "Input length: 270". On the right, under the "Memory pool size" section, there is a table titled "Expression" showing the calculated memory usage. The expression is `JSON_ARRAY_SIZE(2) + 4*JSON_OBJECT_SIZE(1) + 2*JSON_OBJECT_SIZE(3)`. Below this, there is a table titled "Additional bytes for strings duplication" with rows for AVR, ESP32, ESP8266, SAMD 21, and STM32, each with a calculated size. A note at the bottom states: "⚠️ Sizes can be significantly larger on a computer."

Platform	Size
AVR	96+118 = 214
ESP32	
ESP8266	
SAMD 21	192+118 = 310
STM32	

You just need to paste your JSON document in the box on the left, and the Assistant displays the expression in the box on the right. Don't worry; the Assistant respects your privacy: it computes the expression locally in the browser; it doesn't send your JSON document to a web service.

3.6 Extracting values from an array

3.6.1 Retrieving elements by index

The process of extracting the values from an array is very similar to the one for objects. The only difference is that arrays are indexed by an integer, whereas objects are indexed by a string.

To get access to the repository information, we need to get the `JsonObject` from the `JsonDocument`, except that, this time, we'll pass an integer to the subscript operator (`[]`).

```
// Get the first element of the array
JsonObject repo0 = doc[0];

// Extract the values from the object
const char* name    = repo0["name"];
long      stars   = repo0["stargazers"]["totalCount"];
int       issues  = repo0["issues"]["totalCount"];
```

Of course, we could have inlined the `repo0` variable (i.e., write `doc[0]["name"]` each time), but it would cost an extra lookup for each access to the object.

3.6.2 Alternative syntaxes

It may not be obvious, but the program above uses implicit casts. Indeed, the subscript operator (`[]`) returns a `JsonVariant` that is implicitly converted to a `JsonObject`.

Again, some programmers don't like implicit casts, that is why ArduinoJson offers an alternative syntax with `as<T>()`. For example:

```
auto repo0 = arr[0].as<JsonObject>();
```

All of this should sound very familiar because it's similar to what we've seen for objects.

3.6.3 When complex values are missing

When we learned how to extract values from an object, we saw that, if a member is missing, a default value is returned (for example, 0 for an int). Similarly, ArduinoJson returns a default value when you use an index that is out of the range of an array.

Let's see what happens in our case:

```
// Get an object out of array's range
JsonObject repo666 = arr[666];
```

The index 666 doesn't exist in the array, so a special value is returned: a null `JsonObject`. Remember that `JsonObject` is a reference to an object stored in the `JsonDocument`. In this case, there is no object in the `JsonDocument`, so the `JsonObject` points to nothing: it's a null reference.

You can test if a reference is null by calling `isNull()`:

```
// Does the object exists?
if (repo666.isNull()) {
    // Of course not!
}
```

A null `JsonObject` evaluates to `false`, so you can easily verify that it's not null:

```
// Does the object exists?
if (repo0) {
    // Yes!!
}
```

A null `JsonObject` looks like an empty object, except that you cannot modify it. You can safely call any function of a null `JsonObject`, it simply ignores the call and returns a default value. Here is an example:

```
// Get a member of a null JsonObject
int stars = repo666["stargazers"]["totalCount"];
// stars == 0
```

The same principles apply to null `JsonArray`, `JsonVariant`, and `JsonDocument`.



The null object design pattern

What we just saw is an implementation of the null object design pattern. In short, this pattern saves you from constantly checking that a result is not null. Instead of returning `nullptr` when the value is missing, a placeholder is returned: the “null object.” This object has no behavior, and all its methods fail.

If ArduinoJson didn't implement this pattern, we would not be able to write the following statement:

```
int stars = arr[0]["stargazers"]["totalCount"];
```

3.7 Inspecting an unknown array

In the previous section, our example was very straightforward because we knew that the JSON array had precisely two elements, and we knew the content of these elements. In this section, we'll see what tools are available when you don't know the content of the array.

3.7.1 Getting a reference to the array

Do you remember what we did when we wanted to enumerate the key-value pairs of an object? Right! We began by calling `JsonDocument::as<JsonObject>()` to get a reference to the root object.

Similarly, if we want to enumerate all the elements of an array, the first thing we have to do it to get a reference to it:

```
// Get a reference to the root array
JsonArray arr = doc.as<JsonArray>();
```

Again, `JsonArray` is a reference to an array stored in the `JsonDocument`; it's not a copy of the array. When you apply changes to the `JsonArray`, the changes are reflected on the `JsonDocument`.

3.7.2 Capacity of JsonDocument for an unknown input

If you know absolutely nothing about the input (which is strange), you need to determine a memory budget allowed for parsing the input. For example, you could decide that 10KB of heap memory is the maximum you accept to spend on JSON parsing.

This constraint looks terrible at first, especially if you're a desktop or server application developer; but, once you think about it, it makes complete sense. Indeed, your program will run in a loop on dedicated hardware. Since the hardware doesn't change, the amount of memory is always the same. Having an elastic capacity would just produce a larger and slower program with no additional value; it would also increase the heap fragmentation, which we must avoid at all costs.

However, most of the time, you know a lot about your JSON document. Indeed, there are usually a few possible variations in the input. For example, an array could have between zero and four elements, or an object could have an optional member. In that case, use the [ArduinoJson Assistant](#) to compute the size of each variant, and pick the biggest.

3.7.3 Number of elements in an array

The first thing you want to know about an array is the number of elements it contains. This is the role of `JsonArray::size()`:

```
// Get the number of elements in the array
int count = arr.size();
```

As the name may be confusing, let me clarify: `JsonArray::size()` returns the number of elements, not the memory consumption. If you want to know how many bytes of memory are used, call `JsonDocument::memoryUsage()`:

```
// How many bytes are used in the document
int memoryUsed = doc.memoryUsage();
```

Note that there is also a `JsonObject::size()` that returns the number of key-value pairs in an object, but it's rarely useful.

3.7.4 Iteration

Now that you have the size of the array, you probably want to write the following code:

```
// BAD EXAMPLE, see below
for (int i=0; i<arr.size(); i++) {
    JsonObject repo = arr[i];
    const char* name = repo["name"];
    // etc.
}
```

The code above works but is terribly slow. Indeed, ArduinoJson stores arrays as linked lists, so accessing an element at a random location costs $O(n)$; in other words, it takes n iterations to get to the n th element. Moreover, the value of `JSONArray::size()` is not cached, so it needs to walk the linked list too.

That's why it is **essential** to avoid `arr[i]` and `arr.size()` in a loop. Instead, you should use the iteration feature of `JSONArray`, like so:

```
// Walk the JSONArray efficiently
for (JsonObject repo : arr) {
    const char* name = repo["name"];
    // etc.
}
```

With this syntax, the internal linked list is walked only once, and it is as fast as it gets.

I used a `JsonObject` in the loop because I knew that the array contains objects. If it's not your case, you can use a `JsonVariant` instead.



When C++11 is not available

The code above leverages a C++11 feature called “range-based for loop.” If you cannot enable C++11 on your compiler, you must use the following syntax:

```
for (JSONArray::iterator it=arr.begin(); it!=arr.end(); ++it) {
    JsonObject repo = *it;
    const char* name = repo["name"];
    // etc.
}
```

3.7.5 Detecting the type of an element

We test the type of array elements the same way we did for object members: by using `JsonVariant::is<T>()`.

Here is an example:

```
// Is the first element an integer?  
if (arr[0].is<int>()) {  
    // Yes!  
    int value = arr[0];  
    // ...  
}  
  
// Same in a loop  
for (JsonVariant elem : arr) {  
    // Is the current element an object?  
    if (elem.is<JsonObject>()) {  
        JsonObject obj = elem;  
        // ...  
    }  
}
```

There is nothing new here, as it's exactly what we saw for object members.

3.8 The zero-copy mode

3.8.1 Definition

At the beginning of this chapter, we saw how to parse a JSON document from a *writable* source. Indeed, the `input` variable was a `char[]` in the stack, and therefore, it was writable. I told you that this fact would matter, and it's time to explain.

ArduinoJson behaves differently with writable inputs and read-only inputs.

When the argument passed to `deserializeJson()` is of type `char*` or `char[]`, ArduinoJson uses a mode called “zero-copy.” It has this name because the parser never makes any copy of the input; instead, it stores pointers pointing inside the input buffer.

In the zero-copy mode, when a program requests the content of a string member, ArduinoJson returns a pointer to the beginning of the string in the input buffer. To make it possible, ArduinoJson inserts null-terminators at the end of each string; it is the reason why this mode requires the input to be writable.



The jsmn library

As we said [at the beginning of the book](#), jsmn is a C library that detects the tokens in the input. The zero-copy mode is very similar to the behavior of jsmn.

This information should not be a surprise because the first version of ArduinoJson was just a C++ wrapper on top of jsmn.

3.8.2 An example

To illustrate how the zero-copy mode works, let's have a look at a concrete example. Suppose we have a JSON document that is just an array containing two strings:

```
["hip", "hop"]
```

And let's say that the variable is a `char[]` at address `0x200` in memory:

```
char input[] = "[\"hip\", \"hop\"]";
// We assume: &input == 0x200
```

After parsing the input, when the program requests the value of the first element, ArduinoJson returns a pointer whose address is `0x202` which is the location of the string in the input buffer:

```
deserializeJson(doc, input);

const char* hip = doc[0];
const char* hop = doc[1];
// Now: hip == 0x202 && hop == 0x208
```

We naturally expect `hip` to be `"hip"` and not `"hip\"", \"hop\""]`; that's why ArduinoJson adds a null-terminator after the first `p`. Similarly, we expect `hop` to be `"hop"` and not `"hop\""]`, so a second null-terminator is added.

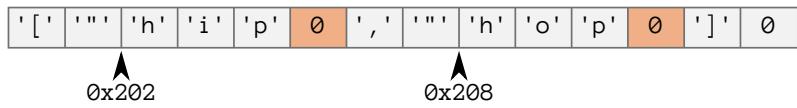
The picture below summarizes this process.

Input buffer before parsing



`deserializeJson()`

Input buffer after parsing



Adding null-terminators is not the only thing the parser modifies in the input buffer. It also replaces escaped character sequences, like `\n` by their corresponding ASCII characters.

I hope this explanation gives you a clear understanding of what the zero-copy mode is and why the input is modified. It is a bit of a simplified view, but the actual code is very similar.

3.8.3 Input buffer must stay in memory

As we saw, in the zero-copy mode, ArduinoJson returns pointers to the input buffer. So, for a pointer to be valid, the input buffer must be in memory at the moment the pointer is dereferenced.

If a program dereferences the pointer after the destruction of the input buffer, it is very likely to crash instantly, but it could also work for a while and crash later, or it could have nasty side effects. In the C++ jargon, this is what we call an “Undefined Behavior”; we’ll talk about that in [“Troubleshooting”](#).

Here is an example:

```
// Declare a pointer
const char *hip;

// New scope
{
    // Declare the input in the scope
    char input[] = "[\"hip\", \"hop\"]";

    // Parse input
    deserializeJson(doc, input);
    JSONArray arr = doc.as<JSONArray>();

    // Save a pointer
    hip = arr[0];
}

// input is destructed now

// Dereference the pointer
Serial.println(hip); // <- Undefined behavior
```



Common cause of bugs

Dereferencing a pointer to a destructed variable is a common cause of bugs.

To use a `JsonArray` or a `JsonObject`, you must keep the `JsonDocument` alive. In addition, when using the zero-copy mode, you must also keep the input buffer in memory.

3.9 Reading from read-only memory

3.9.1 The example

We saw how ArduinoJson behaves with a writable input, and how the zero-copy mode works. It's time to see what happens when the input is read-only.

Let's go back to our previous example except that, this time, we change its type from `char[]` to `const char*`:

```
const char* input = "[\"hip\", \"hop\"]";
```

As we saw in the [C++ course](#), this statement creates a sequence of bytes in the “globals” area of the RAM. This memory is supposed to be read-only; that's why we need to add the `const` keyword.

Previously, we had the whole string duplicated in the stack, but it's not the case anymore. Instead, the stack only contains the pointer `input` pointing to the beginning of the string in the “globals” area.

3.9.2 Duplication is required

As we saw in the previous section, in the zero-copy mode, ArduinoJson stores pointers pointing inside the input buffer. We saw that it has to replace some characters of the input with null-terminators.

With a read-only input, ArduinoJson cannot do that anymore, so it needs to make copies of `"hip"` and `"hop"`. Where do you think the copies would go? In the `JsonDocument`, of course!

In this mode, the `JsonDocument` holds a copy of each string, so we need to increase its capacity. Let's do the computation for our example:

1. We still need to store an object with two elements, that's `JSON_ARRAY_SIZE(2)`.
2. We have to make a copy of the string `"hip"`, that's 4 bytes including the null-terminator.
3. We also need to copy the string `"hop"`, that's 4 bytes too.

The required capacity is:

```
const int capacity = JSON_ARRAY_SIZE(2) + 8;
```

In practice, you should not use the exact length of the strings. It's safer to add a bit of slack, in case the input changes. My advice is to add 10% to the longest possible string, which gives a reasonable margin.



Use the ArduinoJson Assistant

The ArduinoJson assistant also computes the number of bytes required for the duplication of strings. It shows this value under "Additional bytes for string duplication."

The screenshot shows the ArduinoJson Assistant web application. On the left, the 'Input' section contains a JSON array: `[\"hip\", \"hop\"]`. A red arrow points from this input area to the 'Additional bytes for strings duplication' field in the 'Memory pool size' section on the right. The 'Memory pool size' section also displays the expression `JSON_ARRAY_SIZE(2)` and the calculated value `8`. Below this, a table lists memory requirements for different platforms: AVR (Size 24), ESP32 (Size 40), ESP8266 (Size 40), SAMD 21 (Size 40), and STM32 (Size 40). A note at the bottom states: `⚠ Sizes can be significantly larger on a computer.`.

3.9.3 Practice

Apart from the capacity of the `JsonDocument`, we don't need to change anything to the program.

Here is the complete hip-hop example with a read-only input:

```
// A read-only input
const char* input = "[\"hip\", \"hop\"]";
```

```
// Allocate the JsonDocument
const int capacity = JSON_ARRAY_SIZE(2) + 8;
StaticJsonDocument<capacity> doc;

// Parse the JSON input.
deserializeJson(doc, input);

// Extract the two strings.
const char* hip = doc[0];
const char* hop = doc[1];

// How much memory is used?
int memoryUsed = doc.memoryUsage();
```

I added a call to `JsonDocument::memoryUsage()`, which returns the current memory usage. Do not confuse it with the *capacity*, which is the maximum size.

If you compile this program on an ESP8266, the variable `memoryUsed` will contain `40`, as the ArduinoJson Assistant predicted.

3.9.4 Other types of read-only input

`const char*` is not the sole read-only input that ArduinoJson supports. For example, you can also use a `String`:

```
// Put the JSON input in a String
String input = "[\"hip\", \"hop\"]";
```

It's also possible to use a Flash string, but there is one caveat. As we said [in the C++ course](#), ArduinoJson needs a way to figure out if the input string is in RAM or in Flash. To do that, it expects a Flash string to have the type `const __FlashStringHelper*`. If you declare a `char[] PROGMEM`, ArduinoJson will not consider it as Flash string, unless you cast it to `const __FlashStringHelper*`.

Alternatively, you can use the `F()` macro, which casts the pointer to the right type:

```
// Put the JSON input in the Flash
auto input = F("[\"hip\", \"hop\"]");
// (auto is deduced to const __FlashStringHelper*)
```

In the next section, we'll see another kind of read-only input: streams.

3.10 Reading from a stream

In the Arduino jargon, a stream is a volatile source of data, like the serial port or a TCP connection. As opposed to a memory buffer, which allows reading any bytes at any location (after all, that's what the acronym "RAM" means), a stream only allows reading one byte at a time and cannot rewind.

This concept is materialized by the `Stream` abstract class. Here are examples of classes derived from `Stream`:

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>	
SD	<code>File</code>	
SoftwareSerial	<code>SoftwareSerial</code>	
WiFi	<code>WiFiClient</code>	
Wire	<code>TwoWire</code>	<code>Wire</code>



`std::istream`

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

ArduinoJson can use both `Stream` and `std::istream`.

3.10.1 Reading from a file

As an example, we'll create a program that reads a JSON file stored on an SD card. We suppose that this file contains the array [we used as an example earlier](#).

The program will just read the file and print the information for each repository.

Here is the relevant part of the code:

```
// Open file
File file = SD.open("repos.txt");

// Parse directly from file
deserializeJson(doc, file);

// Loop through all the elements of the array
for (JsonObject repo : doc.as<JsonArray>()) {
    // Print the name, the number of stars, and the number of issues
    Serial.println(repo["name"].as<char*>());
    Serial.println(repo["stargazers"]["totalCount"].as<int>());
    Serial.println(repo["issues"]["totalCount"].as<int>());
}
```

A few things to note:

1. I used the .txt extension instead of .json because the FAT file-system is limited to three characters for the file extension.
2. I used the ArduinoJson Assistant to compute the capacity (not shown above because it's not the focus of this snippet).
3. I called `JsonVariant::as<T>()` to pick the right overload of `Serial.println()`.

You can find the complete source code for this example in the folder `ReadFromSdCard` of the zip file.

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

3.10.2 Reading from an HTTP response

Now is the time to parse the real data coming from **GitHub's API!**

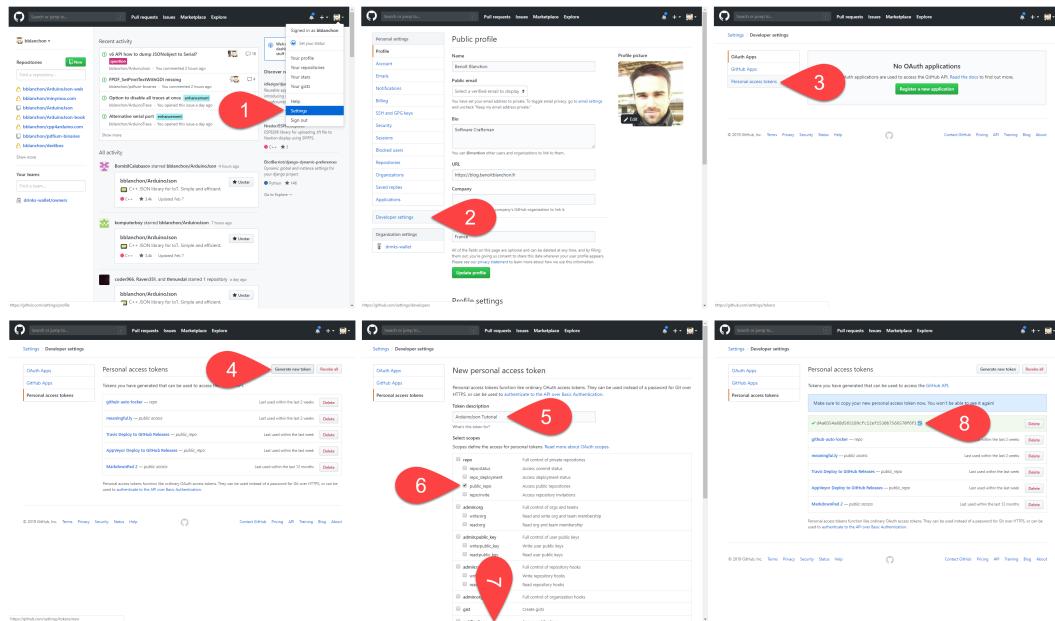
As I said, we need a microcontroller that supports HTTPS, so we'll use an ESP8266 with the library "ESP8266HTTPClient." Don't worry if you don't have a compatible board; we'll see other configurations [in the case studies](#).

Access token

Before using this API, you need a GitHub account and a “personal access token.” This token grants access to the GitHub API from your program; we might also call it an “API key.” To create it, open GitHub in your browser and follow these steps:

1. Go to your personal settings.
2. Go in “Developer settings.”
3. Go in “Personal access token.”
4. Click on “Generate a new token.”
5. Enter a name, like “ArduinoJson tutorial.”
6. Check the scopes (i.e., the permissions); we only need “public_repo.”
7. Click on “Generate token.”
8. GitHub shows the token, **copy it now** because it disappears after a few seconds.

You can see each step in the picture below:



GitHub won't show the token again, so don't waste any second and write it in the source code:

```
#define GITHUB_TOKEN "d4a0354a68d565189cf12ef1530b7566570f6f1"
```

With this token, our program can authenticate with GitHub's API. All we need to do is to add the following HTTP header to each request:

```
Authorization: bearer d4a0354a68d565189cf12ef1530b7566570f6f1
```

The request

To interact with the new GraphQL API, we need to send a POST request (as opposed to the more common GET request) to the URL <https://api.github.com/graphql>.

The body of the POST request is a JSON object that contains one string named "query." This string contains a GraphQL query. For example, if we want to get the name of the authenticated user, we need to send the following JSON document in the body of the request:

```
{
  "query": "{viewer{name}}"
}
```

The GraphQL syntax and the details of GitHub's API are obviously out of the scope of this book, so I'll simply say that the GraphQL query allows you to select the information you want within the universe of information that the API exposes.

In our case, we want to retrieve the names, numbers of stars, and numbers of opened issues of your ten most popular repositories. Here is the corresponding GraphQL query:

```
{
  viewer {
    name
    repositories(ownerAffiliations: OWNER,
      orderBy: {
        direction: DESC,
```

```
        field: STARGAZERS
    },
    first: 10) {
nodes {
    name
    stargazers {
        totalCount
    }
    issues(states: OPEN) {
        totalCount
    }
}
}
}
```

To find the correct query, I used the [GraphQL API Explorer](#). With this tool, you can test GraphQL queries in your browser. You'll find it in GitHub's API documentation.

We'll reduce this query to a single line to save some space and bandwidth; then, we'll put it in the "query" string in the JSON object. Since we haven't talked about JSON serialization yet, we'll hard-code the string in the program.

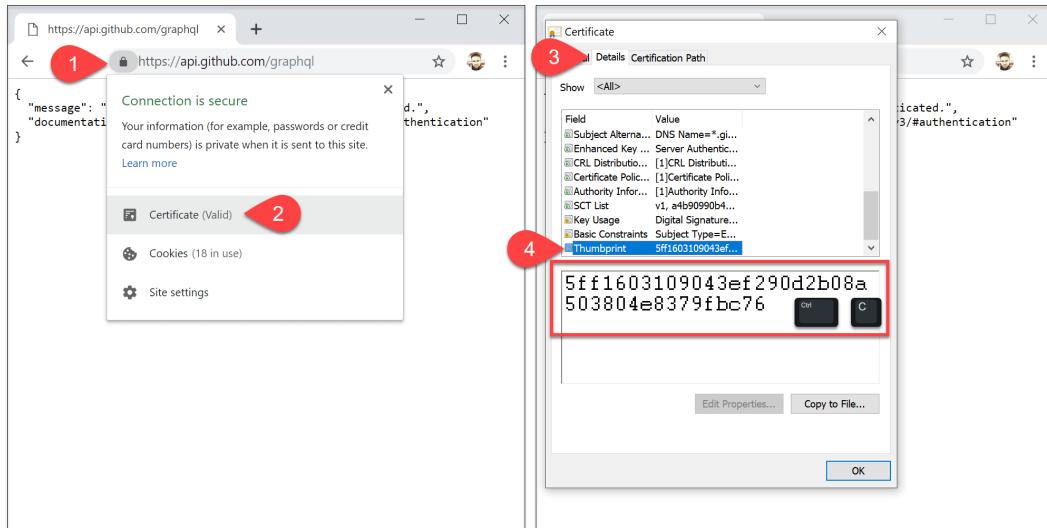
To summarize, here is how we will send the request:

```
HTTPClient http;
http.begin("https://api.github.com/graphql", GITHUB_FINGERPRINT);
http.addHeader("Authorization", "bearer " GITHUB_TOKEN));
http.POST("{\"query\":\"{viewer{name, repositories(ownerAffiliations:...\"});
```

The certificate fingerprint

In the code above, `GITHUB_FINGERPRINT` is the fingerprint of the SSL certificate of `api.github.com`. It allows verifying the authenticity of the server before sending our credentials. The fingerprint changes when the certificate changes, so you may need to update the value in the source code.

To get the current fingerprint, open `api.github.com` in your browser, click on the padlock, then click on “Certificate.” From there, you should find the fingerprint. The picture below shows where to find it with Chrome on Windows:



Maintenance nightmare

Fingerprint validation is mandatory with `ESP8266HTTPClient`.

It's good from a security perspective because you are sure to send your credentials to the right entity. However, it's terrible from a maintenance perspective because you need to update the code every time the certificate changes (every year in this case).

In the case studies, we'll see how to perform an HTTPS request without validating the certificate.

The response

After sending the request, we must get a reference to the Stream:

```
// Get a reference to the stream in HTTPClient
Stream& response = http.getStream();
```

As you see, we call `getStream()` to get the internal stream. Unfortunately, when we do that, we bypass the part of `ESP8266HTTPClient` that handles chunked transfer

encoding. To make sure GitHub doesn't return a chunked response, we must set the protocol to HTTP 1.0:

```
// Downgrade to HTTP 1.0 to prevent chunked transfer encoding
http.useHTTP10(true);
```

Because the version of the protocol is part of the request, we must call `useHTTP10()` before calling `POST()`.

Now that we have the stream, we can pass it to `deserializeJson()`:

```
// Allocate the JsonDocument in the heap
DynamicJsonDocument doc(2048);

// Deserialize the JSON document in the response
deserializeJson(doc, response);
```

Here, we used a `DynamicJsonDocument` because it is too big for the stack. As usual, I used the ArduinoJson Assistant to compute the capacity.

The body contains the JSON document that we want to deserialize. It's a little more complicated than what we saw earlier. Indeed, the JSON array is not at the root but under `data.viewer.repositories.nodes`, as you can see below:

```
{
  "data": {
    "viewer": {
      "name": "Benoît Blanchon",
      "repositories": {
        "nodes": [
          {
            "name": "ArduinoJson",
            "stargazers": {
              "totalCount": 3420
            },
            "issues": {
              "totalCount": 21
            }
          },
        ],
      }
    }
  }
},
```

```
{  
    "name": "WpfBindingErrors",  
    "stargazers": {  
        "totalCount": 48  
    },  
    "issues": {  
        "totalCount": 3  
    }  
}  
]  
}  
}  
}
```

So, compared to what we saw earlier, the only difference is that we'll have to walk several objects before getting the reference to the array. The following line will do:

```
JSONArray repos = doc["data"]["viewer"]["repositories"]["nodes"];
```

The code

I think we have all the pieces, let's assemble this puzzle:

```
// Send the request  
HTTPClient http;  
http.begin("https://api.github.com/graphql", GITHUB_FINGERPRINT);  
http.useHTTP10(true);  
http.addHeader("Authorization", "bearer " GITHUB_TOKEN));  
http.POST("{\"query\":\"{viewer{name,repositories(ownerAffiliations:...\"});  
  
// Get a reference to the stream in HTTPClient  
Stream& response = http.getStream();  
  
// Allocate the JsonDocument in the heap  
DynamicJsonDocument doc(2048);
```

```
// Deserialize the JSON document in the response
deserializeJson(doc, response);

// Get a reference to the array
JsonArray repos = doc["data"]["viewer"]["repositories"]["nodes"];

// Print the values
for (JsonObject repo : repos) {
    Serial.print(" - ");
    Serial.print(repo["name"].as<char *>());
    Serial.print(", stars: ");
    Serial.print(repo["stargazers"]["totalCount"].as<long>());
    Serial.print(", issues: ");
    Serial.println(repo["issues"]["totalCount"].as<int>());
}

// Disconnect
http.end();
```

If all works well, this program should print something like so:

- ArduinoJson, stars: 4241, issues: 12
- pdfium-binaries, stars: 116, issues: 9
- ArduinoTrace, stars: 67, issues: 5
- WpfBindingErrors, stars: 58, issues: 3
- ArduinoStreamUtils, stars: 28, issues: 1
- dllhelper, stars: 22, issues: 0
- SublimeText-HighlightBuildErrors, stars: 12, issues: 4
- HighSpeedMvvm, stars: 12, issues: 0
- cpp4arduino, stars: 11, issues: 1
- BuckOperator, stars: 10, issues: 0

You can find the complete source code of this example in the GitHub folder in the zip file provided with the book. Compared to what is shown above, the source code handles the connection to the WiFi network, check errors, and uses Flash strings when possible.

3.11 Summary

In this chapter, we learned how to deserialize a JSON input with ArduinoJson. Here are the key points to remember:

- `JsonDocument`:
 - `JsonDocument` stores the memory representation of the document.
 - `StaticJsonDocument` is a `JsonDocument` that resides in the stack.
 - `DynamicJsonDocument` is a `JsonDocument` that resides in the heap.
 - `JsonDocument` has a fixed capacity that you set at construction.
 - You can use the [ArduinoJson Assistant](#) to compute the capacity.
- `JsonArray` and `JsonObject`:
 - You can extract the value directly from the `JsonDocument` as long as there is no ambiguity.
 - To solve an ambiguity, you must call `as<JsonArray>()` or `as<JsonObject>()`.
 - `JsonArray` and `JsonObject` are references, not copies.
 - The `JsonDocument` must remain in memory; otherwise, the `JsonArray` or the `JsonObject` contains a dangling pointer.
- `JsonVariant`:
 - `JsonVariant` is also a reference and supports several types: object, array, integer, float, and boolean.
 - `JsonVariant` differs from `JsonDocument` because it doesn't own the memory; it just points to it.
 - `JsonVariant` supports implicit conversion, but you can also call `as<T>()`.
- The two modes:
 - The parser has two modes: zero-copy and classic.
 - It uses the zero-copy mode when the input is a `char*`.
 - It uses the classic mode with all other types.

- The zero-copy mode allows having smaller a `JsonDocument` because it stores pointers to the strings in the input buffer.

In the next chapter, we'll see how to serialize a JSON document with ArduinoJson.

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 so:

```
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 to 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 so:

```
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("[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	HardwareSerial	Serial, Serial1...
ESP	BluetoothSerial	SerialBT
	File	
	WiFiClient	
	WiFiClientSecure	
Ethernet	EthernetClient	
	EthernetUDP	
GSM	GSMClient	
LiquidCrystal	LiquidCrystal	
SD	File	
SoftwareSerial	SoftwareSerial	
WiFi	WiFiClient	
Wire	TwoWire	Wire



`std::ostream`

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

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

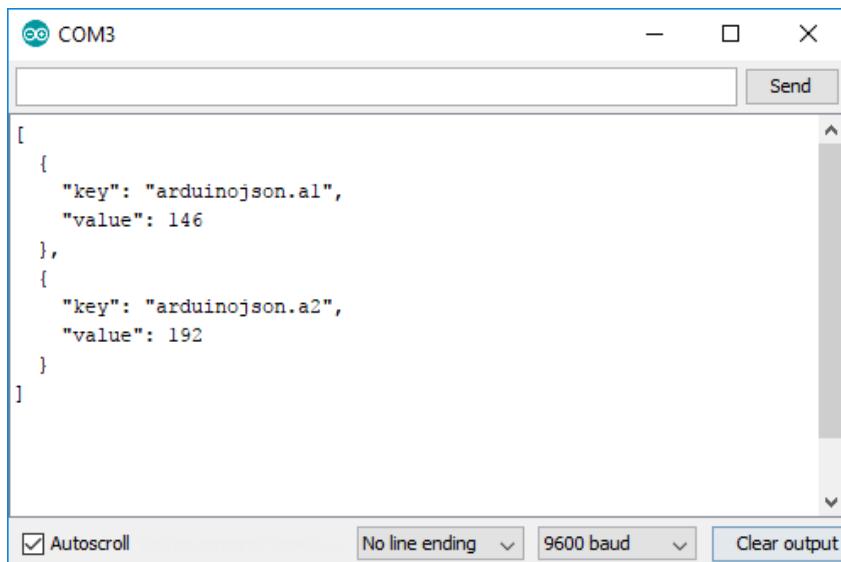
4.5.2 Writing to the serial port

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");
```

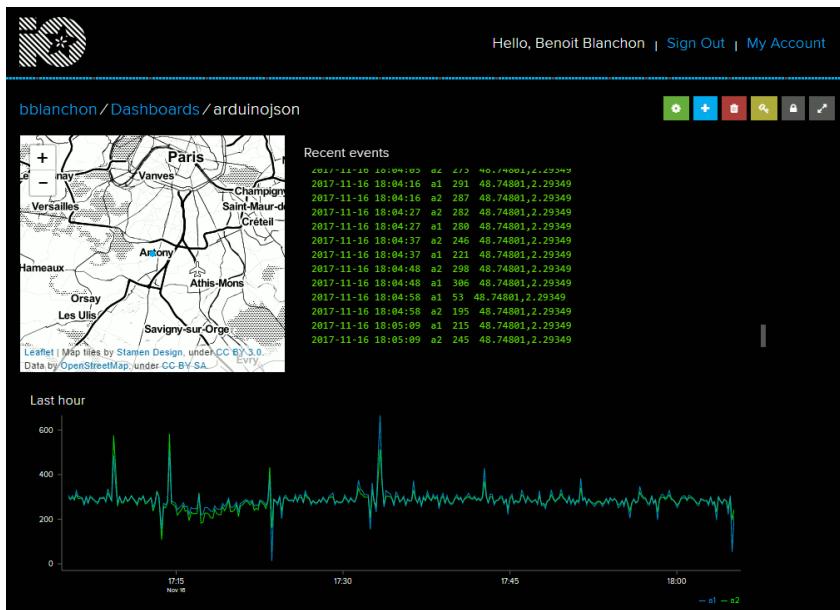
```
// Send the HTTP headers
client.println("Host: io.adafruit.com");
client.println("Connection: close");
client.print("Content-Length: ");
client.println(measureJson(doc));
client.println("Content-Type: application/json");
client.println("X-AIO-Key: " IO_KEY);

// 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()); // 16
```

with the following:

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

// Print the memory usage
Serial.println(doc.memoryUsage()); // 30
```

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. On an ESP8266, it would be `32` and `46`.

4.6.2 Keys and values

The duplication rules apply equally to keys and values. In practice, we mostly use string literals for keys, so they are rarely duplicated. String values, however, often originate from variables and then entail string duplication.

Here is a typical example:

```
String identifier = getIdentifier();
doc["id"] = identifier; // "id" is stored by pointer
                      // identifier is copied
```

Again, the duplication occurs for any type of string except `const char*`.

4.6.3 Copy only occurs when adding values

In the example above, ArduinoJson copied the `String` because it needed to add it 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")];
```

4.6.4 ArduinoJson Assistant's “extra bytes”

As we saw in the previous chapter, the Assistant shows the number of bytes required to duplicate *all* the strings of the document.

The screenshot shows the ArduinoJson Assistant interface. On the left, the 'Input' section contains a code editor with the JSON array `["hip","hop"]` and a note below it: 'Input length: 13'. On the right, the 'Memory pool size' section has a table for 'Platform' and 'Size'. A red arrow points from the input code editor to the 'Additional bytes for strings duplication' field in the 'Memory pool size' section, which is highlighted with a red border. The table data is as follows:

Platform	Size
AVR	16+8 = 24
ESP32	32+8 = 40
ESP8266	
SAMD 21	
STM32	

A yellow warning box at the bottom states: '⚠ Sizes can be significantly larger on a computer.'

In practice, this value much higher than necessary because you don't duplicate *all* the strings but only some. So, in the case of serialization, you should estimate the required value based on your program and use the Assistant's “extra bytes” only as an upper limit.

As you probably noticed, the Assistant generates a “serializing program” that assumes all the strings are constant, and therefore, entirely ignores this problem. So remember to increase the capacity as soon as you change the type of the strings.

4.6.5 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.

4.6.6 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.

Chapter 5

Advanced Techniques

”

“Early optimization is the root of all evils,” Knuth said, but on the other hand, “belated pessimization is the leaf of no good.”

– Andrei Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied

5.1 Introduction

In the previous chapters, we learned how to serialize and deserialize JSON documents with ArduinoJson. Now, we'll focus on the various techniques that you could use in your projects.

In this chapter, I'll introduce some advanced techniques that you can use in your project. These techniques cover serialization, deserialization, or both. Most of them are applicable to every projects, others are only usable with a specific hardware.

Unlike previous chapter, there won't be a complete example here. Instead, I'll only show small snippets to demonstrate how to use a techniques. In the "Case Studies" chapter, however, we'll use some of theses techniques in actual projects.

5.2 Filtering the input

Motivation

With the GitHub example, we saw how things happen in the ideal case. Indeed, the GraphQL syntax allowed us to tailor the perfect query so that the response contains just what we want and nothing more.

In most cases, however, web services return a response that contains way too much information. For example, OpenWeatherMap, which we'll explore in the [case studies](#), returns JSON documents that contain hundreds of fields, even when we're only interested in one or two. With such services, you cannot deserialize the response entirely; otherwise, the `JsonDocument` would be so large that it wouldn't fit in the RAM.

In this section, we'll see how we can reduce the size of the document by removing the parts that are not relevant to our application.

Principle

Reading a large document that contains a lot of uninteresting values is, unfortunately, very common. To solve this problem, `ArduinoJson` offers a filtering option.

To use this feature, you must create a second `JsonDocument` that serves as a pattern to filter the input. This document must contain the value `true` for each member that you want to keep. For arrays, create only one element; it will serve as a template for all elements of the array.

Once the filter is ready, wrap it in a `DeserializationOption::Filter` and pass it to `deserializeJson()`. The parser will ignore every field that is not present in the filter, saving a lot of memory.

Note that this feature is not available for `MessagePack` at the moment.

Implementation

Let's see an example. Imagine we still get the same response from GitHub, but this time, we only want the *name* of the repository and not the rest.

Here is the input:

```
{  
  "data": {  
    "viewer": {  
      "name": "Benoît Blanchon",  
      "repositories": {  
        "nodes": [  
          {  
            "name": "ArduinoJson",  
            "stargazers": {  
              "totalCount": 3420  
            },  
            "issues": {  
              "totalCount": 21  
            }  
          },  
          {  
            "name": "WpfBindingErrors",  
            "stargazers": {  
              "totalCount": 48  
            },  
            "issues": {  
              "totalCount": 3  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

To exclude every field except the name, we must use the following filter:

```
{  
  "data": {  
    "viewer": {  
      "repositories": {  
        "nodes": [  
          {  
            "name": "ArduinoJson",  
            "stargazers": {  
              "totalCount": 3420  
            },  
            "issues": {  
              "totalCount": 21  
            }  
          },  
          {  
            "name": "WpfBindingErrors",  
            "stargazers": {  
              "totalCount": 48  
            },  
            "issues": {  
              "totalCount": 3  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

```
        "name": true,
    }
]
}
}
}
```

To create this filter, I replaced the first instance of `name` with the value `true`, and I removed everything else. When the pattern contains an array, `deserializeJson()` uses the first element to filter all the elements from the input array, and it ignores the others.

Here is how this filter translates into code:

```
// Create the filter document
StaticJsonDocument<128> filter;
filter["data"]["viewer"]["repositories"]["nodes"][0]["name"] = true;

// Deserialize the response and apply filter
deserializeJson(doc, response, DeserializationOption::Filter(filter));
```

As you can see, we use a `StaticJsonDocument` for the filter because it's quite small and easily fits in the stack. Then we wrap this document with `DeserializationOption::Filter` and pass it to `deserializeJson()`.

There is nothing else to change; the rest of the code remains the same.

Now, if you call `serializeJsonPretty()` to see how the document looks like, you would get:

```
{
  "data": {
    "viewer": {
      "repositories": {
        "nodes": [
          {
            "name": "ArduinoJson",
          },
        ],
      },
    },
  },
}
```

```
{  
    "name": "WpfBindingErrors",  
}  
]  
}  
}  
}  
}
```

As you can see, only the `"name"` member is saved.

We'll use this technique in the [OpenWeatherMap case study](#), in the last chapter.

5.3 Deserializing in chunks

Motivation

The filtering technique we saw in the previous section is the most straightforward way to reduce memory usage. However, it still requires the final document to fit in memory. When the input is very large, however, even the filtered document may be too large. In that case, we cannot deserialize the complete document, at least not in one shot.

Principle

Since the complete document cannot fit in memory, our only option is to deserialize the input chunk by chunk. Instead of calling `deserializeJson()` once, we'll call it repeatedly, once for each part we are interested in.

Unfortunately, this technique doesn't work with all inputs; it is only applicable when these two conditions are fulfilled:

1. The input is a stream.
2. The document contains an array with many objects.

Luckily, this scenario is very common: often, a JSON document is large because it contains an array with many elements. A typical example is a response from a weather forecast service like OpenWeatherMap, as we'll see [in the case studies](#).

Here is how this technique works. Before calling `deserializeJson()`, we move the reading cursor to the beginning of the array. Then, we repeatedly call `deserializeJson()` for each object in the array. Of course, we skip the comma (,) between each object, and we stop the loop when we reach the closing bracket (]).

Implementation

As an example, we'll take the same JSON document as before. This time, however, we'll suppose that the array contains hundreds of records instead of ten. As a reminder, here it is (only two elements are shown):

```
{  
  "data": {  
    "viewer": {  
      "name": "Benoît Blanchon",  
      "repositories": {  
        "nodes": [  
          {  
            "name": "ArduinoJson",  
            "stargazers": {  
              "totalCount": 3420  
            },  
            "issues": {  
              "totalCount": 21  
            }  
          },  
          {  
            "name": "WpfBindingErrors",  
            "stargazers": {  
              "totalCount": 48  
            },  
            "issues": {  
              "totalCount": 3  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

Jumping into the array

Instead of calling `deserializeJson()` once for the whole document, we'll call it once for each element of the `"nodes"` array.

Before calling `deserializeJson()`, we must position the reading cursor to the first element of the array. To do that, we call `Stream::find()`, a function that consumes the

input stream until it finds the specified pattern. In our case, we are looking for the beginning of the `"nodes"`, so we invoke it like so:

```
// Jump to the first element of the "nodes" array
response.find("nodes:[");
```

Luckily, GitHub returns minified JSON documents, so we don't have to worry about spaces. If it were not the case, we would have to call `Stream::find()` two times: once with `"nodes"` and once with `[`.

When `Stream::find()` returns, the next character to read is the opening brace `{` of the first element. We can now pass the stream to `deserializeJson()`, it will consume the stream until it reaches the end of the object.

To do that, we must allocate a `JsonDocument`. Since this document only contains one element, it is fairly small, so we can use a `StaticJsonDocument`.

```
StaticJsonDocument doc;

// Deserialize one element of the node array
deserializeJson(doc, response);

// Print the content
Serial.print(" - ");
Serial.print(doc["name"].as<char *>());
Serial.print(", stars: ");
Serial.print(doc["stargazers"]["totalCount"].as<long>());
Serial.print(", issues: ");
Serial.println(doc["issues"]["totalCount"].as<int>());
```

Note that we use the information in the `JsonDocument` immediately because we'll soon destroy it to make room for the next element.

Jumping to the next element

When `deserializeJson()` returns, the next character in the stream is the comma `(,`). Before calling `deserializeJson()` again, we need to skip this character. The easiest way

to do that, is to call `Stream::find()` again:

```
// Jump to the next element
response.find(",");
```

Now we can call `deserializeJson()` again and repeat the operation for all elements. We must repeat until we reach the closing bracket `[]` that marks the end of the array. To detect this character, we can use `Stream::findUntil()`, which reads the stream until it finds the specified pattern or a terminator. In our case, the pattern is the comma, and the terminator is the bracket. Here is how we can write the loop:

```
// Repeat for each element of the array
do {
} while (response.findUntil(", ", "]"));
```

As you can see, we use the boolean returned by `Stream::findUntil()` as the stop condition. Indeed, this function returns `true` when it finds the pattern or `false` if it reaches the terminator first.

Complete code

Here is the complete code to implement “deserialization in chunks”.

```
// Jump to the first element of the "nodes" array
response.find("nodes:[");

// Repeat for each element of the array
do {
    StaticJsonDocument doc;

    // Deserialize one element of the node array
    deserializeJson(doc, response);

    // Print the content
    Serial.print(" - ");
    Serial.print(doc["name"].as<char *>());
```

```
Serial.print(", stars: ");
Serial.print(doc["stargazers"]["totalCount"].as<long>());
Serial.print(", issues: ");
Serial.println(doc["issues"]["totalCount"].as<int>());
} while (response.findUntil(", ", "["));
```

As you can see, I omitted the error checking. If your program must be resilient to errors, you need to check the results of `deserializeJson()`, `Stream::find()`, and `Stream::findUntil()`.

We'll use this technique in the [Reddit case-study](#) in the last chapter.



Buggy runtimes

Several runtimes libraries (called "core" in Arduino jargon) have buggy signatures for `Stream::find()` and `Stream::findUntil()`. Indeed, these functions take `char*` arguments, where they should take `const char*` instead. If you pass a string literal to these functions, the compiler produces the following warning:

```
warning: ISO C++ forbids converting a string constant to 'char*'
→ [-Wwrite-strings]
```

You can safely ignore this warning. If you want to fix it, you must copy the literal in a variable:

```
char beginningOfNodes[] = "nodes:[";
response.find(beginningOfNodes);
```

5.4 JSON streaming

Motivation

Up till now, we covered the following scenarios:

- Sending a request
- Receiving a response
- Writing to a file
- Reading from a file

However, we didn't see how to transmit a JSON document repeatedly over the same connection. For example, we could use a serial or a Bluetooth connection for the following tasks:

- Signaling events
- Sending logs
- Sending instructions

In this section, we'll see how we can send and receive a stream of JSON objects.

Principle

In the previous section, we leveraged the fact that `deserializeJson()` stops reading when it reaches the end of the document. For example, when it reads an object, it stops as soon as it sees the final brace `}`.

We can use this feature to deserialize a continuous stream of JSON objects. For example, imaging that we transmit instructions to our device via the serial port. Each instruction is a JSON object that contain the detail of the job:

```
{"action": "analogWrite", "pin": 3, "value": 18}  
{"action": "analogWrite", "pin": 4, "value": 605}  
{"action": "digitalWrite", "pin": 13, "value": "low"}  
...
```

This technique, called “JSON streaming,” comes in several flavors depending on the way you separate the objects. The most common convention is to use a newline between each object, like in the example above. This convention is known as LDJSON (for “Line-delimited JSON”), but also NDJSON (for “Newline-delimited JSON”), and JSONLines.

Other conventions use different separators. We’ll only study line-separated JSON, but you could use ArduinoJson with the other formats as well.

Implementation

Reading a JSON stream

When reading from a stream, `deserializeJson()` waits for incoming data, but it times out if nothing comes. To avoid this timeout, you must wait until some data is ready, and only then call `deserializeJson()`.

The simplest way to wait for incoming data is to monitor the result of `Stream::available()`, which returns the number of bytes ready to be read. After waiting, you can call `deserializeJson()` and perform the requested action.

After performing the action, you can discard the `JsonDocument`, and you should be ready to accept the next message. Here is the complete loop:

```
void loop() {
    // Wait for incoming data in serial port
    while (Serial.available() > 2)
        delay(100);

    // Read next instruction
    StaticJsonDocument<128> doc;
    deserializeJson(doc);

    // Perform requested action
    performAction(doc);
}
```

When the program starts, it sometimes happens that the serial port buffer contains garbage. In that case, I recommend adding a flushing loop in the `setup()` function:

```
void setup() {  
    // ...  
  
    // Flush the content of the serial port buffer  
    while (Serial.available() > 0)  
        Serial.read();  
}
```

We'll use this technique in the [“Recursive Analyzer”](#) case study.

Writing a JSON stream

As you saw, reading a JSON stream is fairly easy. Well. Writing is even simpler. All you have to do is to call `serializeJson()` and then call `Stream::println()` to add a line-break. Here is an example with the serial port:

```
// Send next object  
serializeJson(doc, Serial);  
  
// Send a line break  
Serial.println();
```

`Stream::println()` adds two characters (CR and LF), that's why I used the condition `Serial.available() > 2` in the waiting loop.

5.5 Automatic capacity

Motivation

As you know, `DynamicJsonDocument` uses a fixed-size memory pool. In most cases, you know the shape of the document, so you can easily compute the required capacity. Sometimes, however, you cannot predict what the document will look like, so you cannot compute its size.

In this section, we'll see how we can write code that supports any JSON document as long as it fits in memory. This technique works for both serialization and deserialization.

Principle

The technique consists in allocating a huge `DynamicJsonDocument` and reducing its capacity afterward. The capacity is still not elastic, but at least it adapts to the input document and to the available RAM.

As you know, the memory pool of a `DynamicJsonDocument` is a contiguous block of heap memory. Therefore, the largest possible memory pool has the size of the largest contiguous block of free memory.

To know the size of this block, you must call a function from the Arduino core. Unfortunately, the name of this function depends on the core that you use. Here are two common examples:

Core	Function
ESP8266	<code>ESP.getMaxFreeBlockSize()</code>
ESP32	<code>ESP.getMaxAllocHeap()</code>

These functions return the size of the largest block we can allocate with a single call to `malloc()`. If we pass the result to the constructor of `DynamicJsonDocument`, we'll allocate the largest possible memory pool.

After allocating the document, we can populate it as usual, either by inserting values or by calling `deserializeJson()`.

Once the document is complete, we can release all unused memory. We do that with `DynamicJsonDocument::shrinkToFit()`, a function that reduces the capacity of the memory pool to the minimum required by the document. This function calls `realloc()` to release all the unused memory from the memory pool.

Unfortunately, `DynamicJsonDocument::shrinkToFit()` doesn't release the memory leaked inside the pool. For example, if you removed a value from the document, the memory consumed by this value remains in the memory pool. We'll see a workaround in the next section.

Implementation

Here is how you can implement a function that deserializes from an input stream and returns an optimized document:

```
DynamicJsonDocument deserializeInput(Stream& input)
{
    // Allocate the largest possible document (platform dependent)
    DynamicJsonDocument doc(ESP.getMaxFreeBlockSize());

    // Read input
    deserializeJson(doc, input);

    // Release unused memory
    doc.shrinkToFit();

    return doc;
}
```

As you can see, this code applies the technique we just described; there is not much to say.

5.6 Fixing memory leaks

Motivation

As we saw in the introduction, the main strength of ArduinoJson is its memory allocator: a monotonic allocator offers the best speed with the smallest code.

By definition, a monotonic allocator cannot release memory. When you remove or replace a value in a `JsonDocument`, the memory reserved for this value is not released. If you do that repeatedly, the memory pool fills up until it cannot accept more value.

In practice, this memory leak only occurs when you use ArduinoJson to store the state of your application. Remember that ArduinoJson is a *serialization* library and not a generic container library. If you only use it to serialize and deserialize JSON documents, you won't have any leak.

Despite this limitation, many users still want to use `JsonDocument` as a long-lived storage. A typical example is storing the configuration of the application, or the current state of the outputs.

Luckily, there is a simple way to eliminate the memory leak.

Principle

When it copies a `JsonDocument`, ArduinoJson recursively duplicates each value one by one. We say that it performs a “deep copy”, as opposed to a “shallow copy” (that would only copy the root), or a *memberwise* copy (that would blindly copy the class members).

By doing a deep copy, it eliminates all values that are not attached to the root. In other words, the copy is free from any leaked value.

With this feature, we can clean up a `JsonDocument` by making a copy and replacing the original. Since it's a common pattern, ArduinoJson offers a function that does that for you: `JsonDocument::garbageCollect()`. As the name suggests, this function reclaims all lost bytes from the memory pool.

Because it has to duplicate the content of the document, `garbageCollect()` is quite slow and requires a lot of memory. Depending on your situation, you may want to run it after each modification of the `JsonDocument` or only when the memory pool reaches a certain level of occupation.

Implementation

All we have to do is call `garbageCollect()` after modifying the document. Since this function is quite slow, you should call it after making several modification.

```
// Update the configuration
doc["wifi"]["ssid"] = ssid; // leaks when "ssid" is a String
doc["wifi"]["pass"] = pass; // leaks when "pass" is a String

// Remove memory leaks (slow)
doc.garbageCollect();
```

When grouping the modifications is not possible, you can postpone the call to `garbageCollect()` until the memory pool reaches a critical level:

```
// Clean the document is memory pool is over 80% of its capacity
if (doc.memoryUsage() > doc.capacity() * 4 / 5) {
    doc.garbageCollect();
}
```

You can place this code in its own function so you can call it from anywhere, or you can place it in the `loop()` function.

5.7 Using external RAM

Motivation

Microcontrollers have a very small amount of memory, but sometimes, you can add an external RAM chip to increase the total capacity. For example, many ESP32 boards embed an external PSRAM connected to the SPI bus. This external chip adds up to 4MB to the internal 520kB of the ESP32.

Depending on the configuration, the external RAM may be used implicitly or explicitly.

- With the *implicit* mode, the standard `malloc()` uses the internal and the external RAM. This behavior is completely transparent to the application.
- With the *explicit* mode, the standard `malloc()` only uses the internal RAM. To use the external RAM, the program must call dedicated functions.

Because it's transparent, the *implicit* mode is simpler to use. Unfortunately, the external RAM is much slower than the internal RAM, so mixing the two might slow down your whole application.

When performance matters, it's better to use the *explicit* mode, which means we cannot use the standard functions. Therefore, classes like `DynamicJsonDocument`, which call the regular `malloc()` and `free()`, cannot use the external RAM.

Principle

In addition to `DynamicJsonDocument`, `ArduinoJson` supports `BasicJsonDocument<T>`, a generic implementation of `JsonDocument` with a customizable allocator. In fact, `DynamicJsonDocument` is an alias of `BasicJsonDocument<DefaultAllocator>`, where `DefaultAllocator` is an allocator class using the standard `malloc()` and `free()` functions.

I'm sure it will make more sense as soon as you see the code, so here are the definitions of `DefaultAllocator` and `DynamicJsonDocument`:

```
struct DefaultAllocator {  
    void* allocate(size_t size) {  
        return malloc(size);  
    }  
};
```

```
}

void deallocate(void* ptr) {
    free(ptr);
}

void* reallocate(void* ptr, size_t new_size) {
    return realloc(ptr, new_size);
}
};

using DynamicJsonDocument = BasicJsonDocument<DefaultAllocator>;
```

As you can see, the allocator class simply forwards the calls to the appropriate functions. Note that `reallocate()` is only used when you call `DynamicJsonDocument::shrinkToFit()`.

To use the external RAM instead of the internal one, we must create a new allocator class that calls the right functions.

Implementation

We'll only show how to implement this technique to use the external PSRAM provided with some ESP32. You should be able to apply the same principles with other chips.

According to the documentation of the ESP32, you must call the following functions instead of `malloc()`, `realloc()`, and `free()`:

```
void *heap_caps_malloc(size_t size, uint32_t caps);
void *heap_caps_realloc(void *ptr, size_t size, int caps);
void heap_caps_free(void *ptr);
```

These functions uses the “capabilities-based heap memory allocator”, hence the prefix `heap_caps_`. As you can see `heap_caps_malloc()` and `heap_caps_realloc()` support an extra `caps` parameter. This parameter allows us to specifies flags that describe the capabilities of the requested chunk of memory.

In our case, we want a chunk from the external RAM, so we'll use the flag `MALLOC_CAP_SPIRAM`. As you can see from the name, this flag identifies the "SPIRAM", i.e., the external RAM connected to the SPI bus.

Let's write the new allocator class:

```
struct SpiRamAllocator {
    void* allocate(size_t size) {
        return heap_caps_malloc(size, MALLOC_CAP_SPIRAM);
    }

    void deallocate(void* pointer) {
        heap_caps_free(pointer);
    }

    void* realloc(void* ptr, size_t new_size) {
        return heap_caps_realloc(ptr, new_size, MALLOC_CAP_SPIRAM);
    }
};
```

Nothing fancy here, we just created a class that forwards the three calls to the appropriate functions. Now, we can use this class like so:

```
// Create a JsonDocument in the external RAM
BasicJsonDocument<SpiRamAllocator> doc(16384);

// Use the document as usual
deserializeJson(doc, input);
```

In the first line, we instantiate the template class `BasicJsonDocument<T>` by injecting our custom allocator class. Of course, we can create an alias for this class, which makes the code more readable:

```
// Define a new type of JsonDocument that resides in external RAM
using SpiRamJsonDocument = BasicJsonDocument<SpiRamAllocator>

// Use the new type as usual
SpiRamJsonDocument doc(16384);
deserializeJson(doc, input);
```

5.8 Logging

Motivation

Consider a program that serializes a JSON document and sends it directly to its destination:

```
// Send a JSON document over the air
serializeJson(doc, wifiClient);
```

On the one hand, we like this kind of code because it minimizes the memory consumption. On the other hand, if anything goes wrong, we wish we had a copy of the document to check that it was serialized correctly.

Now, consider another program that deserializes a JSON document directly from its origin:

```
// Receive a JSON document and parse it on-the-fly
deserializeJson(doc, wifiClient);
```

Again, on the one hand, we know it is the best way to use the library. On the other hand, if parsing fails, we'd like to see what the document looked like, so we can understand why parsing failed.

In this section, we'll see how to print the document to the serial port, so we can easily debug the program.

Principle

The statement `serializeJson(doc, wifiClient)` is calling the function `serializeJson(const JsonDocument&, Print&)`. This function takes an instance of `Print`, an abstract class that represents the concept of “output stream” in Arduino. We are free to create our own implementation of `Print` and pass it to `serializeJson()`.

For example, we could create a `Print` class whose job is to log and to delegate the work to another implementation. In other words, this class would implement the abstract `write()` method from `Print`, print the content to the serial port, and forward the call to `WiFiClient`.

What I just described is a “decorator”, a design pattern that allows adding behavior to an object without modifying its implementation. In our case, it gives the logging ability to any instance of `Print`. This pattern is one of the original 23 patterns from [the Gang of Four](#).

We can apply the decorator pattern to `deserializeJson()` as well. Instead of `Print`, this function takes a reference to `Stream`, an abstract class representing the concept of “bidirectional stream” in Arduino.

We could easily write the two decorator classes, but we don't have too because they already exist in the [StreamUtils library](#). The first is `LoggingPrint`, and the second is `ReadLoggingStream`.



No input stream

We saw that `Print` represents an output stream, and `Stream` a bidirectional stream. However, Arduino doesn't define any class to represent the concept of input stream.

Implementation

Because we'll use StreamUtils, the first step is to install the library. Open the Arduino Library Manager, search for “StreamUtils”, and click install.

Then, of course, we must include the header to import the decorator classes in our program:

```
#include <StreamUtils.h>
```

To print the document sent by `serializeJson()`, we must decorate `wifiClient` with `LoggingPrint`. The constructor of `LoggingPrint` takes two references to `Print`. The first is the stream to decorate, and the second is where to write the log.

```
// Add logging to "wifiClient", print the log to "Serial"
LoggingPrint wifiClientWithLog(wifiClient, Serial);

// Send a JSON document to "wifiClient" and log at the same time
serializeJson(doc, wifiClientWithLog);
```

As you can see, we created an instance of `LoggingPrint` to decorate `wifiClient`, and then we passed this instance to `serializeJson()`.

We can now apply the same technique to `deserializeJson()`. We just need to replace `LoggingPrint` with `ReadLoggingStream`.

```
// Add logging to "wifiClient", print the log to "Serial"
ReadLoggingStream wifiClientWithLog(wifiClient, Serial);

// Deserialize from "wifiClient" and log at the same time
deserializeJson(doc, wifiClientWithLog);
```

In these two snippets, we used `LoggingPrint` to log what we sent to a stream, and then we use `ReadLoggingStream` to log what we received from a stream. But what if we need to do both at the same time? Well, in that case, we need to use the `LoggingStream` decorator, which is a combination of the two others.

```
// Add two-way logging to "wifiClient"
LoggingStream wifiClientWithLog(wifiClient, Serial);

// Deserialize from "wifiClient" and log at the same time
deserializeJson(doc, wifiClientWithLog);

// Send a JSON document to "wifiClient" and log at the same time
serializeJson(doc, wifiClientWithLog);
```

StreamUtils offers other kinds of decorators, as we'll see next.

5.9 Buffering

Motivation

When `serializeJson()` writes to a stream, it sends bytes one by one. Most of the time, it's not a problem because the stream contains an internal buffer. In some cases, however, sending bytes one by one at a time can hurt performance.

For example, some implementations of `WiFiClient` send a packet for each byte, which produces a terrible overhead. Some implementations of `File` write one byte to disk at a time, which is horribly slow.

Sure, we could serialize to memory and then send the entire document, but it would consume a lot of memory.

Similarly, `deserializeJson()` consumes a stream one byte at a time. This feature is crucial for deserializing in chunks and JSON streaming. Unfortunately, it sometimes hurt the performance with some implementations of `Stream`.

In this section, we'll learn how to add buffering to `ArduinoJson` and improve the reading and writing performance.

Principle

In the previous section, we saw how to use the “decorator” design pattern to add the logging capability to a stream. In short, a decorator adds a feature to a class without modifying the implementation.

Now, we'll use the same technique to add the buffering capability to a stream. Here too, we could implement the decorators ourselves, but the `StreamUtils` library already provides them.

As a reminder, the `Print` abstract class defines the interface for an *output* stream, and `Stream` defines the interface for a *bidirectional* stream. The decorator for the `Print` interface is `BufferingPrint`; the one for `Stream` is `ReadBufferingStream`.

Implementation

Since we're using the StreamUtils, make sure it's installed and include the header:

```
#include <StreamUtils.h>
```

To bufferize `serializeJson()`, we can decorate the `Print` with `BufferingPrint`. The constructor of `BufferingPrint` takes two parameters: the first is the stream to decorate, and the second is the size of the buffer. Here is how we can add a buffer of 64 bytes:

```
// Add buffering to "file"
BufferingPrint bufferedFile(file, 64);

// Send the JSON document by chunks of 64 bytes
serializeJson(doc, bufferedFile);

// Send the remaining bytes
bufferedFile.flush();
```

The destructor of `BufferingPrint` calls `flush()`, so you can remove the last line if you destroy the instance.

To bufferize `deserializeJson()`, we can decorate the `Stream` with `ReadBufferingStream`. As previously, the constructor of `ReadBufferingStream`: the stream to decorate and the size of the buffer.

```
// Add buffering to "file"
ReadBufferingStream bufferedFile(file, 64);

// Read the JSON document by chunks of 64 bytes
deserializeJson(doc, bufferedFile);
```

If you need to bufferize both the reading and the writing sides of a stream, you can use the decorator `BufferedStream`, which combines the two others.

```
// Add buffering to "file" in both direction: read and write
BufferingStream bufferedClient(wifiClient, 64);
```

```
// Receive the JSON document by chunks of 64 bytes
deserializeJson(doc, bufferedClient);

// Send the JSON document by chunks of 64 bytes
serializeJson(doc, bufferedClient);
bufferedClient.flush();
```

The StreamUtils library offers many other options; please check out the documentation.

5.10 Custom readers and writers

Motivation

In the previous chapters, we saw how to serialize to and from Arduino streams. It was easy because ArduinoJson natively supports the `Print` and `Stream` interfaces. Similarly, you can use the standard STL streams: `serializeJson()` supports `std::ostream`, and `deserializeJson()` supports `std::istream`.

What if you want to write to another type of stream? What if your class implements neither `Print` nor `std::ostream`? One possible solution would be to write an adapter class that implements `Print` and forwards the calls to your stream class. “Adapter” is another pattern of the classic [“Gang of Four” book](#).

The adapter is a valid solution, but passing via the virtual functions of `Print` adds a small overhead. Most of the time, this overhead is negligible, but if performance is crucial, it’s better to avoid virtual calls.

In this section, we’ll see how we can write an adapter class without virtual methods.



The hidden cost of virtual methods

By definition, a virtual method is an indirect call. Unlike a regular method, whose address is known at compile-time, the address of a virtual method is resolved at run-time. Each time it calls a virtual method, a program must lookup the address of the function.

This run-time dispatch is the mechanism that allows the two pieces of code (the caller and the implementation) to be independent, which is excellent from a design perspective. Unfortunately, the lookup affects the performance: a virtual call is slightly slower than a direct call. Also, since the call is resolved at run-time, the compiler cannot optimize it.

Most of the time, the performance overhead of virtual calls is negligible. However, when the body of the method is short, the overhead can become significant. Not only the lookup costs a few extra cycles, but we also miss the opportunity of a compiler optimization.

Hardcore C++ programmers avoid virtual calls as much as possible. Instead of the run-time polymorphism based on virtual methods, they prefer the compile-time polymorphism based on templates.

Principle

ArduinoJson provides several versions of `serializeJson()`. In the tutorial, we saw the overloads that write to a string and a stream. Now, we'll see another overload that supports a template argument:

```
template<typename Writer>
size_t serializeJson(const JsonDocument& doc, Writer& destination);
```

This template function requires that the `Writer` class implements two `write()` methods, as shown below:

```
struct CustomWriter {
    // Writes one byte, returns the number of bytes written (0 or 1)
    size_t write(uint8_t c);

    // Writes several bytes, returns the number of bytes written
    size_t write(const uint8_t *buffer, size_t length);
};
```

As you see, there are no virtual functions involved, so we won't pay the cost of the virtual dispatch.

Similarly, `deserializeJson()` supports a template overload:

```
template<typename Reader>
DeserializationError deserializeJson(JsonDocument& doc, Reader& input);
```

This template function also requires the `Reader` class to implement two methods:

```
struct CustomReader {
    // Reads one byte, or returns -1
    int read();

    // Reads several bytes, returns the number of bytes read.
    size_t readBytes(char* buffer, size_t length);
};
```

Implementation

Let's use the functions from `stdio.h` as an example. I'm assuming that you're familiar with the standard C functions. If not, I recommend reading the [K&R book](#), the best on this topic.

Custom writer

Suppose that we created a file with `fopen()` and that we want to write a JSON document to it. To write to the file from `serializeJson()`, we need to create an adapter class that calls the appropriate functions.

```
class FileWriter {
public:
    FileWriter(FILE *fp) : _fp(fp) {}

    size_t write(uint8_t c) {
        fputc(c, _fp);
        return 1;
    }

    size_t write(const uint8_t *buffer, size_t length) {
        return fwrite(buffer, 1, length, _fp);
    }

private:
    FILE *_fp;
};
```

As you can see, we save the file handle in the constructor so we can pass it to `fputc()` and `fwrite()`.

Here is a sample program that uses `FileWriter`:

```
// Create the file
FILE *fp = fopen("config.json", "wt");
```

```
// Create the adapter
FileWriter writer(fp);

// Write the JSON document
serializeJson(doc, writer);

// Close the file
fclose(fp);
```

We could push further and implement the RAII pattern. For example, we could make `FileWriter` call `fopen()` from its constructor and `fclose()` from its destructor. I let this as an exercise for the reader.

Custom reader

Now, let's see how we can read a JSON document from an existing file opened by `fopen()`. To read the file from `deserializeJson()`, we need to create another adapter that calls the file reading functions.

```
class FileReader {
public:
    FileReader(FILE *fp) : _fp(fp) {}

    int read() {
        return fgetc(_fp);
    }

    size_t readBytes(char* buffer, size_t length) {
        return fread(buffer, 1, length, _fp);
    }

private:
    FILE *_fp;
}
```

As you can see, this class is very similar to `FileWriter`. In the constructor, we save the file pointer so we can pass it to `fgetc()` and `fread()`.

Here is the sample program for `FileReader`:

```
// Open the file
FILE *fp = fopen("config.json", "rt");

// Create the adapter
FileReader reader(fp);

// Read the JSON document
deserializeJson(doc, reader);

// Close the file
fclose(fp);
```

If you need, you can merge `FileReader` and `FileWriter` to create a bidirectional file adapter. Again, this is left as an exercise.

5.11 MessagePack

Motivation

As a text format, JSON is easy to read for a human, but a little harder for a machine. Machines prefer binary formats: there are smaller, simpler, and more predictable. Unfortunately, binary formats often require a lot of “set-up” code.

Indeed, that’s what we love about JSON: no schema, no interface definition, no nothing! A JSON document is just a generic container for objects, arrays, and values. With JSON, there is nothing to set up. Create an empty document, add some values, and you’re done.

Once JSON became popular, people soon realized that we could adapt the concept of generic container to binary formats. And so were born the “binary JSON” formats that are CBOR, BSON, and MessagePack. They offer the same ease of use as JSON, but in a binary format, so with a small boost in performance.

In this section, we’ll see how we can use MessagePack with ArduinoJson.

Principle

ArduinoJson supports MessagePack with a few restrictions. It supports all value formats except binary and custom.

You can use most pieces of ArduinoJson (`JsonDocument`, `JsonArray`, `JsonObject`, etc.) indifferently with JSON or MessagePack. The only things you have to change are the serialization functions. You must substitute `serializeJson()` and `deserializeJson()` with their MessagePack counterparts:

JSON	MessagePack
<code>deserializeJson()</code>	<code>deserializeMsgPack()</code>
<code>serializeJson()</code>	<code>serializeMsgPack()</code>
<code>serializeJsonPretty()</code>	

That’s the only changes you’ll have to make.

Implementation

To demonstrate how we can serialize a MessagePack document, I'll adapt one of the examples of the serialization tutorial. Here was the JSON document:

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

To create a similar document in the MessagePack format, we must write the following code:

```
// Create and populate the JsonDocument as usual
StaticJsonDocument<200> doc;
doc[0]["key"] = "a1";
doc[0]["value"] = 12;
doc[1]["key"] = "a2";
doc[1]["value"] = 32;

// Generate a MessagePack document in memory
char buffer[64];
size_t length = serializeMsgPack(doc, buffer);
```

As you can see, I just replaced `serializeJson()` with `serializeMsgPack()`.

After running this piece of code, the buffer contains the following bytes:

```
92 82 A3 6B 65 79 A2 61 31 A5 76 61 6C 75 65 0C 82 A3 6B 65 79 A2 61 32 A5
↪ 76 61 6C 75 65 20
```

Let's see if we can make sense of this.

- 92 begins an array with two elements.
- 82 begins an object with two members.
- A3 begins a string with three characters.
- 6B 65 79 are the three characters of "key".
- A2 begins a string with two characters.
- 61 31 are the two characters of "a1".
- A5 begins a string with five characters.

- 76 61 6C 75 65 are the five characters of "value".
- 0C is the integer 12.
- 82 begins the second object, and the rest repeats the first part.

This MessagePack document is significantly smaller than the equivalent JSON document: 31 bytes vs. 49 bytes.

Similarly, you can deserialize a MessagePack document by calling `deserializeMsgPack()` instead of `deserializeJson()`:

```
// Deserialize a MessagePack document
deserializeMsgPack(doc, buffer);
```

As you see, there is not a lot to say about MessagePack. I personally don't encourage people to use this format. Sure, it reduces the size of the payload (we saw a reduction of 37% in our example), but the gain is too small to be a game-changer.

5.12 Summary

In this chapter, we saw a collection of techniques commonly used with ArduinoJson. Here are the things to remember.

- Filtering:
 - You can filter a large input to get only the relevant values.
 - Create a second `JsonDocument` that contains `true` for each value you want to keep.
 - For arrays, only the first element of the filter matters.
- Deserializing in chunks:
 - Applicable when the input contains a large array.
 - Use `Stream::find()` to jump to the first element.
 - Use `Stream::findUntil()` to jump to the next element.
- JSON streaming:
 - Known as LDJSON, NDJSON, and JSONLines.
 - Commonly used to report events or to send instructions.
 - `deserializeJson()` stops reading when the document ends.
 - Wait before calling `deserializeJson()` to avoid a time-out.
- Automatic capacity:
 - Call `getMaxFreeBlockSize()` (or equivalent) to get the largest block of free heap.
 - Call `DynamicJsonDocument::shrinkToFit()` to free the unused memory.
- Fixing leaks:
 - Call `JsonDocument::garbageCollect()`.
 - This function is slow and requires a lot of memory.
- Using external RAM:
 - Define an allocator class that implements `allocate()` and `deallocate()`.

- Use `BasicJsonDocument<Allocator>` instead of `DynamicJsonDocument`.
- Logging:
 - Use `LoggingPrint`, `ReadLoggingStream`, or `LoggingStream` from the `StreamUtils` library.
- Buffering:
 - Use `BufferingPrint`, `ReadBufferingStream`, or `BufferingStream` from the `StreamUtils` library.
- Custom readers and writers:
 - The writer class requires two `write()` functions.
 - The reader class requires `read()` and `readBytes()`.
 - No virtual call involved.
- MessagePack:
 - Binary values are not supported.
 - Replace `serializeJson()` with `serializeMsgPack()`.
 - Replace `deserializeJson()` with `deserializeMsgPack()`.

In the next chapter, we'll open the hood and look at `ArduinoJson` from the inside.

Chapter 6

Inside ArduinoJson

”

If you're not at all interested in performance, shouldn't you be in the Python room down the hall?

– Scott Meyers, Effective Modern C++

6.1 Why JsonDocument?

On your first contact with ArduinoJson, I'm sure you had this thought: "What is this curious `JsonDocument`, and why do we need it?" I'll try to answer both questions in this section.

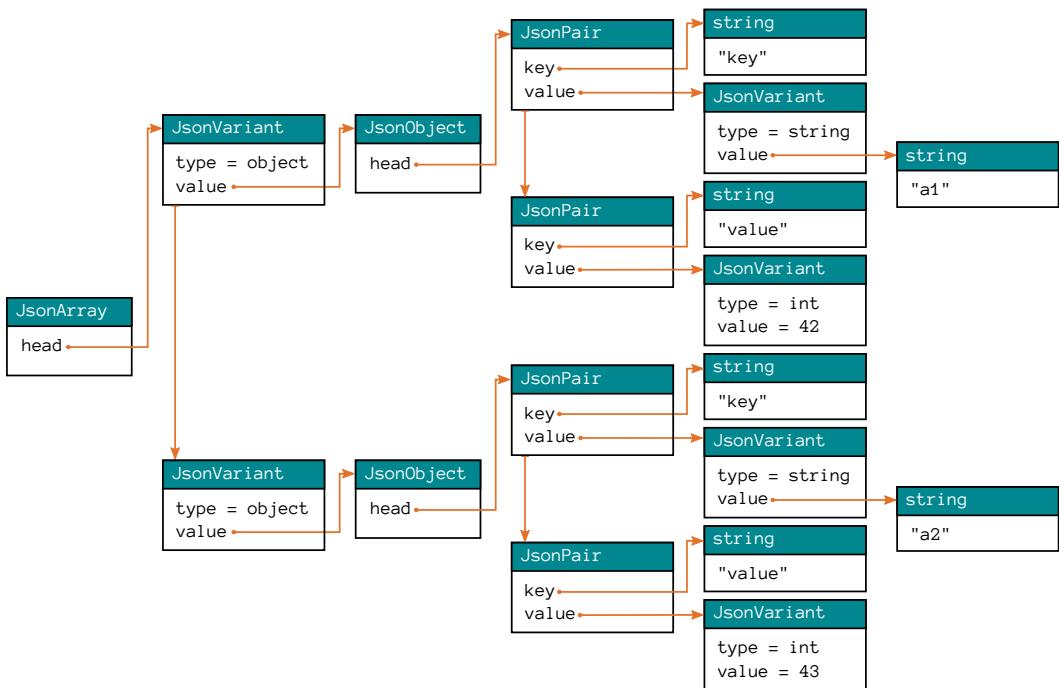
6.1.1 Memory representation

To illustrate this section, we'll take the simple JSON document from the previous chapter:

```
[  
  {  
    "key": "a1",  
    "value": 42  
  },  
  {  
    "key": "a2",  
    "value": 43  
  }  
]
```

It's an array of two elements. Both elements are objects. Each object has two values: a string named "key" and an integer named "value."

This document is fairly simple; however, its memory representation can be quite complex, as you can see in the diagram below.



In this diagram, every box represents an instance of an object, and every arrow represents a pointer.

A `JSONArray` is a linked list of `JsonVariant`; that's why it contains a pointer to the first variant, which is then linked to the second. The same logic is implemented in `JsonObject`.



Names in actual implementation

In this chapter, I'll use the names `JSONArray`, `JsonObject`, and `JsonVariant` to refer to the internal data structures. As you know, these classes are the smart-pointers, and the data structures that hold the information have different names. However, I prefer using the names you already know to reduce the cognitive load.

6.1.2 Dynamic memory

Our JSON document is very simple, yet it requires 19 objects in memory, as we see in the diagram.

If you were to implement your own JSON library, you would probably allocate each object using `new`. That would be the most natural technique in a program written in Java or C#. In fact, this is how most JSON libraries work, but this approach is not suitable for embedded programs.

Indeed, dynamic memory allocation has a cost:

1. Each allocation produces a small memory overhead, which is not negligible when the objects are small like ours.
2. Each allocation and each deallocation requires a significant housekeeping work from the microcontroller.
3. Repeated allocations produce “heap fragmentation,” as we saw in the C++ course.

Now, imagine that this operation is repeated nineteen times! And that’s a really simple example!

Memory management is what makes ArduinoJson different from other libraries: it does the same job with just one allocation and one deallocation.

6.1.3 Memory pool

ArduinoJson reduces the allocation/deallocation work to the minimum by using a “memory pool.” Instead of making 19 small allocations, it makes a big one to create the memory pool. Then it creates the 19 objects inside the pool. Allocations in the pool are very fast: just a few CPU cycles.

ArduinoJson also takes a radical choice concerning deallocations in the pool: it is simply not supported. Yes, you read it right. It’s not possible to release memory inside the pool. However, it’s possible to clear the memory pool, thereby freeing the memory of the 19 objects.

This pattern isn’t unique to ArduinoJson, and it even has a name: the “monotonic allocator.” Indeed, we say that the allocator is “monotonic” because the quantity of allocated memory always increases, just like a “monotonic function” in mathematics.

Since it’s not possible to delete an object inside the pool, none of the objects have a destructor. Therefore, the destruction of the memory pool is very fast.

In ArduinoJson 6, the memory pool is hidden in `JsonDocument`. The two derived classes that we used before, `StaticJsonDocument` and `DynamicJsonDocument`, differ in the way they allocate the memory pool. One allocates in place (in the stack most likely), whereas the other allocates in the heap.



Memory leaks

Because it's not possible to delete an object inside the pool, the functions `JsonArray::remove()` and `JsonObject::remove()` leak memory. Do not use them in a loop; otherwise, the `JsonDocument` would be full after a few iterations.

6.1.4 Strengths and weaknesses

The advantages of this implementation are:

1. allocations are super fast,
2. deallocations are even faster,
3. no fragmentation of the heap,
4. minimal code size.

The drawbacks are:

1. the memory pool is a new concept to learn,
2. objects in the pool cannot be freed,
3. you need to choose the capacity of the pool.



Can't we do better?

ArduinoJson 6.6 contained a full-blown memory allocator for the memory pool. Not only was it able to release blocks in the pool, but it was also able to move the blocks inside the pool to prevent fragmentation.

This feature was awesome, but the overhead in memory usage and code size was unacceptable for small microcontrollers. That's why I removed it from further versions of the library.

So, yes, we can do better, but not now.

6.2 Inside JsonDocument

In the previous section, we saw why a memory pool is necessary to use the memory of the microcontroller efficiently; it's time to see how this mechanism is implemented.

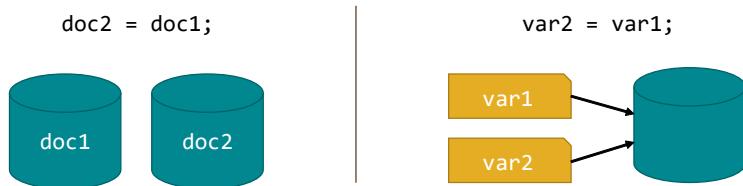
6.2.1 Differences with JsonVariant

We just saw that the `JsonDocument` contains a memory pool, but that's not all, it also contains a variant. A variant is a variable that can contain any value supported by the library (integer, float, string, array, object...). In a sense, we can say that the `JsonDocument` is a `JsonVariant` with a memory-pool, but let's review the differences to make sure you get this analogy right.

`JsonVariant`
+ `MemoryPool`
= `JsonDocument`

Like a `JsonVariant`, a `JsonDocument` can contain any type of value, but the `JsonDocument` **owns** the value, whereas a `JsonVariant` is just a reference.

Similarly, `JsonVariant` has reference semantics, whereas `JsonDocument` has value semantics. When you copy a `JsonVariant`, the new instance refers to the same value as the original. When you copy a `JsonDocument`, the new instance is a complete clone of the original. The picture below illustrates the difference:



6.2.2 Fixed capacity

A `JsonDocument` has a fixed capacity: it cannot grow at run-time. When full (i.e., when the memory usage reached the capacity), a `JsonDocument` rejects all further allocations, just like `malloc()` would do if the heap was full.

As a consequence, you need to determine the appropriate capacity at the time you create a `JsonDocument`. As said before, you can use the ArduinoJson Assistant on arduinojson.org to determine the right capacity for your project.

You can call `JsonDocument::capacity()` to get the capacity of the document in bytes. The value can be slightly higher than what you specified because ArduinoJson adds some padding to align the pointers. For example, the value is rounded to the next multiple of 4 when you compile for a 32-bit processor (like the ESP8266).

You can call `JsonDocument::memoryUsage()` to get the current memory usage in bytes. This value increases each time you add something in the document. Because the memory pool implements a monotonic allocator, this value doesn't decrease when you remove or replace something from the document. However, this value goes back to zero as soon as you call `JsonDocument::to<T>()` or `JsonDocument::clear()`.

6.2.3 Implementation of the allocator

The implementation of the allocator is surprisingly simple: it is just a pointer to the memory pool and an integer to keep track of the current usage. Allocating in the pool is just a matter of increasing the integer.

Here is a simplified definition:

```
class MemoryPool {
public:
    MemoryPool(char* buffer, size_t capacity)
        : _buffer(buffer), _size(0), _capacity(capacity) {}

    void *alloc(int n) {
        // Verify that there is enough space
        if (_size + n <= _capacity)
            return nullptr;

        // Take the address of the next unused byte
        void *p = &_pool[_size];

        // Increment size
        _size += n;

        return p;
    }

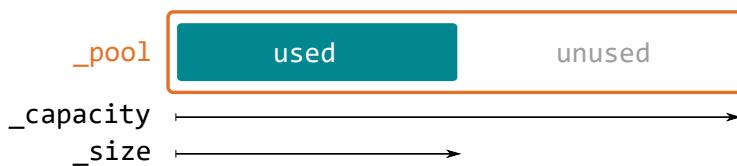
    void clear() {
```

```
    _size = 0;
}

size_t memoryUsage() const { return _size; }
size_t capacity() const { return _capacity; }

private:
    char* _pool;
    size_t _size, _capacity;
};
```

You can visualize the three member variables on the drawing below.



The actual implementation of the memory pool is very close to that, except that it allocates variants and string differently. It places strings at the beginning of the buffer, and variants at the end. This separation preserves the alignment of the structures without padding.

6.2.4 Implementation of JsonDocument

As we saw, `JsonDocument` is simply a `JsonVariant` with a memory pool. Here is a simplified definition:

```
class JsonDocument {
public:
    template<typename T>
    T as() const {
        return _variant.as<T>();
    }

    template<typename T>
    T to() {
```

```
_pool.clear();
return _variant.to<T<>();
}

// other functions shared with JsonVariant

protected:
JsonDocument(char* buffer, size_t capacity);

MemoryPool _pool;
JsonVariant _variant;
};
```

As you see, `JsonDocument` delegates all the work to `JsonVariant`. Additionally, `JsonDocument` clears the memory pool when relevant. For example, `JsonDocument::to<T>()` calls `JsonVariant::to<T>()` and also clears the memory pool. We'll see how `JsonVariant` works later in this chapter.

It's worth noting that there isn't any virtual function in neither `JsonDocument` nor `MemoryPool`. In fact, there isn't any virtual function in ArduinoJson version 6, which is very good for performance because the calls are direct, so the compiler inlines most of them.

6.3 Inside StaticJsonDocument

In the previous section, we saw the common features of `JsonDocument`; let's see the specific features of `StaticJsonDocument`.

6.3.1 Capacity

The capacity of the `StaticJsonDocument` must be set at compile-time with a template parameter. As we said, you cannot use a variable to specify the capacity. Instead, you need to use a constant expression (`const` or `constexpr` in C++11).

```
int capacity = 42;  
const int capacity = 42;
```



Compile-time vs.run-time

“Compile-time” refers to the work done by the compiler when it generates the binary executable for the program. “Run-time” refers to the work done by the program when it runs on the target platform.

Experienced C++ programmers try to do as much work as possible at compile-time, to improve the performance at run-time.

6.3.2 Stack memory

Because its size is known at compile-time, a `StaticJsonDocument` can be allocated in the stack, and it is the recommended usage.



As said in the C++ course, allocation in the stack is very fast because it doesn't require to look for available memory; the compiler does all the work in advance. Creating and destructing a `StaticJsonDocument` in the stack costs a handful of CPU cycles, just like a local variable.



Prefer stack

As the cost of allocation is virtually zero, seasoned C++ programmers try to put everything in the stack.

6.3.3 Limitation

Of course, the stack size cannot exceed the amount of physical memory installed on the board. On top of that, many platforms put a limit on the stack size. This restriction doesn't come from the hardware but from the runtime libraries, which Arduino calls the "core."

See the table below for a list of popular platforms and limits:

Platform	Available RAM	Default stack size
ATmega328	2KB	unlimited
ESP8266	96KB	4KB
ESP32	520KB	8KB
MSP430	<=10KB	unlimited
SAMD21	32KB	unlimited

As a general rule, platforms with a small amount of RAM put no limit to the stack size.

On platforms that limit the stack size, if the program allocates more memory than allowed, it usually crashes. Therefore, on these platforms, you cannot use a huge `StaticJsonDocument`; instead, you need to switch to heap memory. You could increase the stack size by changing the configuration of the "core," but it's easier to switch to `DynamicJsonDocument`.

On a computer program, the stack size is typically limited to 1MB.



Why limit the stack size?

Limiting the stack size allows detecting bugs, like infinite recursions. It also simplifies the implementation of the heap because it has a fixed address range.

6.3.4 Other usages

It's possible to allocate a `StaticJsonDocument` in the heap by using `new`, but I strongly recommend against it because you would lose the RAII feature. Instead, it's safer to use a `DynamicJsonDocument`.

It's possible to use a `StaticJsonDocument` as a member of an object. We'll see an example in the [case studies](#).

6.3.5 Implementation

The implementation of `StaticJsonDocument` is very straightforward: it declares a member array and passes it to the parent class.

Here is a simplified definition:

```
template<size_t capacity>
class StaticJsonDocument : public JsonDocument {
public:
    StaticJsonDocument() : JsonDocument(_buffer, capacity) {}

private:
    char _buffer[capacity];
};
```

6.4 Inside DynamicJsonDocument

In the previous section, we looked at `StaticJsonDocument`; now, let's look at the other implementation of `JsonDocument`: `DynamicJsonDocument`.

6.4.1 Capacity

Like `StaticJsonDocument`, `DynamicJsonDocument` has a fixed capacity: once it's full, you cannot make further allocations.

However, unlike `StaticJsonDocument`, the capacity of `DynamicJsonDocument` doesn't need to be a constant: it can be computed at run-time.

You must specify the capacity to the constructor of `DynamicJsonDocument`. The constructor allocates the memory pool, and the destructor releases it.



Elastic capacity

In ArduinoJson 5, the `DynamicJsonBuffer` was able to expand automatically. This feature was removed from version 6 because it turned out to be a bad idea. It produced heap fragmentation, the one thing it was supposed to prevent.

6.4.2 Shrinking a DynamicJsonDocument

As we saw in the previous chapter, you can reduce the capacity of a `DynamicJsonDocument` by calling `shrinkToFit()`. This function reallocates the memory pool to be just as big as required by the current content of the document. After calling it, the `DynamicJsonDocument` is full, so you cannot add any more values.

```
DynamicJsonDocument readConfig(File file) {
    DynamicJsonDocument doc(2048);
    deserializeJson(doc, file);
    doc.shrinkToFit();
    return doc;
}
```

This function initially creates a large `JsonDocument` to make sure that there is enough space to deserialize the file. However, before returning the `JsonDocument` to the caller, it calls `shrinkToFit()` to release the unused space. This way, we return an optimized `JsonDocument`.

Note that `shrinkToFit()` doesn't recover the memory leaked by `remove()` or by replacing a value. If you want to recover this space, you must copy the `DynamicJsonDocument` (see next section).

Currently, no function allows you to grow the capacity of a `DynamicJsonDocument`.

6.4.3 Automatic capacity

There is one situation where `ArduinoJson` chooses the capacity of the memory pool for you: when you create a `DynamicJsonDocument` from a `JsonArray`, a `JsonObject`, or a `JsonVariant`.

Imagine you have a big JSON configuration file, but you are only interested in a small part of it. Thanks to this feature, you can write the following function:

```
DynamicJsonDocument readNetworkConfig(File file) {
    DynamicJsonDocument doc(2048);
    deserializeJson(doc, file);
    return doc["network"];
}
```

Let's describe this function in detail:

1. We create a temporary `DynamicJsonDocument` named `doc`.
2. We deserialize the file into the temporary `DynamicJsonDocument`.
3. We extract the `JsonVariant` that we want with `doc["network"]`
4. We construct a new `DynamicJsonDocument` from the `JsonVariant`. This second document is much smaller than the original because its capacity matches the memory usage of the `JsonVariant`.
5. `doc` goes out of scope, so it releases its memory pool.
6. The function returns the small `DynamicJsonDocument`.

6.4.4 Heap memory

As we said, the main difference with the `StaticJsonDocument` is that the `DynamicJsonDocument` allocates its memory pool in the heap, whereas the `StaticJsonDocument` uses the stack (in most cases).



I often recommend avoiding the heap when possible, because of the overhead and of the fragmentation. However, since `DynamicJsonDocument` only performs one allocation, the performance overhead is very small. Also, if you use a constant capacity, a `DynamicJsonDocument` doesn't increase the fragmentation.

6.4.5 Allocator

`DynamicJsonDocument` doesn't call `malloc()` and `free()` directly; instead, it passes by the following “allocator class”:

```
struct DefaultAllocator {
    void* allocate(size_t size) {
        return malloc(size);
    }

    void deallocate(void* ptr) {
        free(ptr);
    }

    void* reallocate(void* ptr, size_t new_size) {
        return realloc(ptr, new_size);
    }
};
```

As you see, this class simply forwards the calls to the standard functions. Using an allocator class allows you to customize the allocation functions. For example, if you want to use another heap implementation, you can create your custom allocator class. We'll see how to use this class in the following section.

6.4.6 Implementation

`DynamicJsonDocument` is not a real class: it's a `typedef`, an alias for another class. The real class behind it is `BasicJsonDocument<T>`.

Here is a simplified definition:

```
template<typename TAllocator>
class BasicJsonDocument : public JsonDocument {
public:
    BasicJsonDocument(size_t capacity)
        : JsonDocument(_allocator.allocate(capacity), capacity) {}

    ~BasicJsonDocument() {
        _allocator.deallocate(_buffer);
    }

private:
    TAllocator _allocator;
};
```

Just as you expect, `BasicJsonDocument<T>` allocates the memory pool in the constructor and releases it in the destructor. In both cases, it uses the template parameter to allow you to customize the allocator class.

And now, here is the definition of `DynamicJsonDocument`:

```
typedef BasicJsonDocument<DefaultAllocator> DynamicJsonDocument;
```

This `typedef` syntax is a bit hard to read. In C++11, we can use `using` instead:

```
using DynamicJsonDocument = BasicJsonDocument<DefaultAllocator>;
```

However, since ArduinoJson 6 must remain compatible with C++98, it still uses the old syntax.

6.4.7 Comparison with StaticJsonDocument

To summarize, here is a chart that compares the features:

	StaticJsonDocument	DynamicJsonDocument
Location	stack	heap
Construction cost	near 0	one malloc()
Destruction cost	near 0	one free()
Capacity	fixed	fixed
Specified in	template parameter	constructor parameter
Code size	near 0	pulls malloc() and free()

6.4.8 How to choose?

Always start with a `StaticJsonDocument` as it's simpler and faster. Switch to a `DynamicJsonDocument` only when the `StaticJsonDocument` becomes too big.

Refer to [the table in the previous section](#) to know when a `StaticJsonDocument` is too big. As a general rule, switch to `DynamicJsonDocument` as soon as you need more than half of the stack capacity. For example, on the ESP8266, the stack is limited to 4KB, so the biggest `StaticJsonDocument` you should use is 2KB.

On platforms that put no limit on the stack size, always use a `StaticJsonDocument`. It doesn't make sense to use a `DynamicJsonDocument` there.

6.5 Inside JsonVariant

After `JsonDocument`, `JsonVariant` is the other cornerstone of ArduinoJson. Let's see how it works.

6.5.1 Supported types

ArduinoJson supports the following value types:

- `bool`
- `signed / unsigned char / short / int / long / long long`
- `float / double`
- `String / std::string`
- `JsonArray / JsonObject`
- `char* / const char* / char[]`
- `const __FlashStringHelper*`
- `SerializedValue` (the type returned by `serialized()`)

6.5.2 Reference semantics

I've said it several times, but it's very important to get this concept right: `JsonVariant` has reference semantics, not value semantics.

What does it mean? It means that you can have two `JsonVariants` that refer to the same value. If you change one, you change the other. Here is an example:

```
JsonVariant variant1 = doc.to<JsonVariant>();  
variant1.set("hello");  
  
JsonVariant variant2 = variant1;  
variant2.set("world");  
  
// now, variant1 points to "world"
```

As you see, I didn't use the assignment operator to change the value of the variant; instead, I used `JsonVariant::set()`. Indeed, the assignment operator changes the reference, not the value.

Why is it a reference and not a value? Because it allows manipulating values in the `JsonDocument`. If it were a copy of the value, we would not be able to update the value in the `JsonDocument`.

`JsonVariant` is a kind of smart-pointer: it wraps a pointer in an easy-to-use object. As we saw, it implements the “null object pattern”. A null `JsonVariant` behaves like an empty value, so you don't have to check that the pointer is not null.

In the actual implementation, `JsonVariant` contains a pointer to an internal class named `VariantData`. In many places, I use the name `JsonVariant`, when I should really say `VariantData` because this implementation detail doesn't matter.

6.5.3 Creating a `JsonVariant`

Because `JsonVariant` is a reference and not a value, you cannot instantiate it like a regular variable. For example, if you write:

```
JsonVariant myVariant; // uninitialized
```

`myVariant` is not a variant with no value, as you might expect; instead, it's an uninitialized reference, a reference that points to nothing, like a null pointer. To initialize a `JsonVariant` correctly, you need to get a reference from the `JsonDocument`:

```
// get the root value of the document
JsonVariant existingVariant = doc.as<JsonVariant>();

// clear the document and create a variant
JsonVariant newVariant = doc.to<JsonVariant>();
```

As we saw, it is safe to use an uninitialized variant: it just does nothing.

6.5.4 Implementation

JsonVariant is just a union of the types allowed in a JSON document, alongside an enum to select the type.

In C++, a union is a structure where every member overlap, i.e., they all share the same memory. The size of a union is, therefore, the size of its biggest member. A program needs to know which member of the union is valid before using it. JsonVariant uses an enum to remember which member of the union is the right one.

Here is a simplified definition of JsonVariant:

```
union VariantValue {
    const char* asStrubg;
    double asFloat;
    unsigned long asInteger;
    CollectionData asCollection;
};

enum VariantType {
    VALUE_IS_NULL,
    VALUE_IS_STRING,
    VALUE_IS_DOUBLE,
    VALUE_IS_LONG,
    VALUE_IS_BOOL,
    VALUE_IS_ARRAY,
    VALUE_IS_OBJECT,
    VALUE_IS_RAW
};

struct VariantData {
    VariantValue value;
    VariantType type;
};
```

As you see, a JsonVariant stores pointers to strings; that's why ArduinoJson duplicates the strings into the JsonDocument to make sure that the pointer remains valid. It uses the structure CollectionData to store the linked list used in arrays and objects.

The actual implementation of VariantData is a bit more complicated because it needs to remember whether the string is stored by copy or by pointer. This distinction is needed

to copy a `JsonDocument` to another without duplicating the static strings.

6.5.5 Two kinds of null

A `JsonVariant` can be null for two reasons:

1. either the reference is null,
2. or the value is null.

Here are some examples of the first kind:

```
// Uninitialized
JsonVariant var1;

// Points to a non-existent value
JsonVariant var2 = doc["dz1e6ez"];
```

And now, here are some examples of the second kind of null:

```
// The value is "null"
deserializeJson(doc, "{\"a\":null}");
JsonVariant var3 = doc["x"];

// Clear the value
var4.clear();
```

If you call `JsonVariant::set()` on a null *reference*, it doesn't do anything; if you call it on a null *value*, it changes the value.

As we already saw, you should use `JsonVariant::isNull()` to test if a `JsonVariant` is null. This function returns `true` in both cases.

There is an **undocumented** way to differentiate between the two kinds of null. `JsonVariant::isUndefined()` returns `true` if the reference is undefined, but `false` if the reference points to null. This function is used internally and has never been part of the public API because I think it adds complexity and doesn't solve anything. If you believe that this function should be part of the API, please contact me so we can check together if it really makes sense. For now, the conclusion is that treating undefined references and null references the same way simplifies your code.

6.5.6 The unsigned long trick

In the simplified definition above, I said that ArduinoJson stores integral values as `long`, but it is more complicated than that.

Consider this example:

```
StaticJsonDocument<64> doc;
arr.add(4294967295UL); // biggest unsigned long
```

You naturally expect `arr` to be serialized as:

```
[4294967295]
```

However, if it were stored in a `long`, it would be serialized as:

```
[-1]
```

Indeed, there is an overflow: `(long)4294967295UL` becomes `-1`.

To work around this problem, ArduinoJson uses different enum values for positive and negative values, and everything works as expected.

6.5.7 Integer overflow

Speaking of integer overflows, ArduinoJson also handles overflows in `JsonVariant::as<T>()`. If the value is out of the range of `T`, it returns `0` instead of letting the integer overflow.

Here is an example:

```
doc["value"] = 20000;

doc["value"].as<int8_t>();    // 0
doc["value"].as<uint8_t>();   // 0
doc["value"].as<int16_t>();   // 0
doc["value"].as<uint16_t>();  // 20000
doc["value"].as<int32_t>();   // 20000
```

```
doc["value"].as<uint32_t>(); // 20000
```

Of course, this feature also applies when you use the syntax with implicit conversion:

```
doc["value"] = 20000;

int8_t a = doc["value"]; // 0
uint8_t b = doc["value"]; // 0
int16_t c = doc["value"]; // 0
uint16_t d = doc["value"]; // 20000
int32_t e = doc["value"]; // 20000
uint32_t f = doc["value"]; // 20000
```

This feature protects your program against integer overflow attacks.

6.5.8 ArduinoJson's configuration

ArduinoJson has two compile-time settings that affect the definition of `JsonVariant`.

`ARDUINOJSON_USE_DOUBLE` determines whether to use `double` or `float`.

- When set to `1`, it uses `double`. Floating points values are stored with better precision (up to 9 digits), but `JsonVariant` is much bigger. This mode is the default when the target is a computer.
- When set to `0`, it uses `float`. The precision is lower (up to 6 digits), but `JsonVariant` is smaller. This mode is the default when the target is an embedded platform.

`ARDUINOJSON_USE_LONG_LONG` determines whether to use `long long` or `long`.

- When set to `1`, it uses `long long`. Integral values are stored in a 64-bit integer, but the `JsonVariant` is bigger. This mode is the default when the target is a computer.
- When set to `0`, it uses `long`. Integral values are stored in a 32-bit integer only, but `JsonVariant` is smaller. This mode is the default when the target is an embedded platform.

**The ArduinoJson Assistant assumes the defaults**

If you change the default value, it affects the result of `JSON_ARRAY_SIZE()` and `JSON_OBJECT_SIZE()`; therefore, the ArduinoJson Assistant will compute wrong results.

**Don't mix and match!**

A program composed of multiple compilation units (i.e., several .cpp files) should have the same configuration in all of them. If you use different configurations, the executable will end up with several copies of the library.

6.5.9 Iterating through a JsonVariant

We saw in a previous chapter how to enumerate all the values in a JSONArray and all the key-value pairs of a JsonObject, but doing the same with a `JsonVariant` is a bit more complicated.

As a `JsonVariant` can be a `JSONArray` or a `JsonObject`, we need to help the compiler and tell it which type we expect. We can do that with an implicit cast, or with a call to `as<T>()`.

Let's see two concrete examples: a nested object and a nested array.

An object in an array

Here is an example of an object nested in an array:

```
[  
  {  
    "hello": "world"  
  }  
]
```

You can enumerate the key-value pairs of the object, by casting the `JsonVariant` to a `JsonObject`:

```
for(JsonPair kvp : doc[0].as<JsonObject>()) {  
    const char* key = kvp.key;      // "hello"  
    JsonVariant value = kvp.value; // "world"  
}
```

An array in an object

Here is an example of an array nested in an object:

```
{  
    "results": [  
        1,  
        2  
    ]  
}
```

You can enumerate the elements of the array, by casting the `JsonVariant` to a `JsonArray`:

```
for(int result : doc["results"].as<JsonArray>()) {  
    // result = 1, then 2...  
}
```



What happens if the type is incompatible?

It is safe to do the conversion to `JsonArray` or `JsonObject`, even if the `JsonVariant` doesn't contain the expected type, there will simply be no iteration of the loop.

For example, if you call `as<JsonArray>()` on a `JsonVariant` that points to an object, the function returns a null `JsonArray`, which behaves like an empty array, so no iteration is made.

6.5.10 The or operator

As we saw in the chapter [Deserialize with ArduinoJson](#), you can use the `|` operator to provide a default value in case the value is missing or is incompatible. Here is an

example:

```
// Get the port or use 80 if it's not specified
short tcpPort = config["port"] | 80;
```

This operator doesn't use the implicit cast. Instead, it looks at the type of the value on the right side. If the variant is compatible, it returns the value; otherwise, it returns the default value. Here is a simplified implementation of this operator:

```
template<typename T>
T operator|(JsonVariant variant, T defaultValue) {
    return variant.is<T>() ? variant.as<T>() : defaultValue;
}
```

We saw that in the deserialization tutorial, but I think it's worth repeating: with this operator, you can easily stop the propagation of null strings and protect your program against undefined behavior (more on that in the next chapter). Here is an example:

```
// Copy the hostname or use "arduinojson.org" if it's not specified
char hostname[32];
strlcpy(hostname, config["hostname"] | "arduinojson.org", 32);
```

Indeed, `strlcpy()` doesn't allow `nullptr` as a second parameter; using the "or" operator protects us from this case.

6.5.11 The subscript operator

Operator `[]` is the most sophisticated part of the library.

At first, you might think that it returns a `JsonVariant`, but it's not that easy. It couldn't simply return a `JsonVariant` because you wouldn't be able to write expressions like:

```
doc["config"]["wifi"][0]["ssid"] = "TheBatCave";
```

Instead, operator `[]` returns a proxy class that overrides operator `=`. The actual class depends on the type of index you pass to `[]`. If you pass a string, it's `MemberProxy`. If you pass an integer, it's `ElementProxy`.

MemberProxy treats the variant as an object. ElementProxy treats the variant as an array. Both proxy classes convert the variant to the appropriated type (object or array) if the variant is null.

In the section dedicated to `JsonObject`, I'll talk about MemberProxy in greater detail.

6.5.12 Member functions

A detailed list of methods and overloads is available in the “API Reference” on arduinojson.org. Here is a summary:

`JsonVariant::add()`

This function reproduces `JsonArray::add()`. It only works if the value is an array, in which case it appends a value to the array.

```
doc["ports"].add(443);
// {"ports": [443]}
```

`JsonVariant::as<T>()`

This function converts the value to the type T, returns a default value (like 0, 0.0 or `nullptr`) if the type is incompatible.

Example:

```
// {"pi":3.14159}
auto pi = doc["pi"].as<float>();
```

`JsonVariant::createNestedArray()`

This function creates a nested array. Depending on the presence of a parameter, this function behaves as `JsonArray::createNestedArray()` or `JsonObject::createNestedArray()`.

Examples:

```
JsonArray ports = doc["config"].createNestedArray("ports");
// {"config":{"ports":[]}}  
  
JsonArray users = doc["users"].createNestedArray();
// {"users":[]}
```

JsonVariant::createNestedObject()

This function creates a nested object. Depending on the presence of a parameter, this function behaves as `JsonArray::createNestedObject()` or `JsonObject::createNestedObject()`.

Examples:

```
JsonObject ports = doc["network"].createNestedObject("wifi");
// {"network":{"wifi":{}}}  
  
JsonObject module = doc["modules"].createNestedObject();
// {"modules":[]}
```

JsonVariant::is<T>()

This function tells whether the variant has the type T.

Example:

```
// {"host":"arduinojson.org"}  
if (doc["host"].is<char*>()) {  
    // yes, "host" contains a string  
}
```

This function returns false if the `JsonVariant` is null.

JsonVariant::operator[]

This operator gets or sets the value at the specified index or key. Depending on the type of the argument, this function behaves like `JsonArray::operator[]` or `JsonObject::operator[]`. The details were explained earlier in this chapter.

Examples:

```
// {"ports": [443]}
int firstPort = doc["ports"][0];

// {"config": {"user": "bblanchon"}}
const char* username = doc["config"]["user"];

doc["ports"][0] = 80;
// {"ports": [80]}
```

JsonVariant::isNull()

This function tests if the `JsonVariant` is null.

Example:

```
// {"error": "File not found."}
if (!doc["error"].isNull()) {
    // Yes, there is a member called "error"
}
```

JsonVariant::set()

This function changes the value in the variant. It returns `true` on success, and `false` when there wasn't enough space to store the value.

```
doc["id"].set(42);
// {"id": 42}
```

JsonVariant::size()

Depending on the type of the value, this function behaves like `JsonArray::size()` or `JsonObject::size()`. In other words, it returns the number of elements of the array or object.

Example:

```
// {"ports": [80, 443]}\nint numberOfPorts = doc["ports"].size(); // 2
```

Other functions

Like `JsonObject`, `JsonVariant` supports `getMember` and `getOrAddMember()`. Please see the section dedicated to `JsonObject` for details.

Similarly, `JsonVariant` supports functions from `JsonArray`: `add()`, `getElement()`, `getOrAddElement()`.

6.5.13 Comparison operators

`JsonVariant` supports the following comparison operators: `==`, `!=`, `<=`, `<`, `>=`, and `>`.

You can compare a `JsonVariant` with another, like so:

```
if (doc["value"] < doc["max"]) ...
```

You can also compare a `JsonVariant` with a numeric value:

```
if (doc["voltage"] > 5) ...
```

Finally, you can use `==` to compare a `JsonVariant` with a string:

```
if (doc["status"] == "success") ...
```

6.5.14 Const reference

If you look closely at each method of `JsonVariant`, you'll see that most of them are `const`, even the ones that modify the value of the variant. Indeed, they are `const` methods because they do not modify the reference.

If you need a constant reference, you must use `JsonVariantConst`: it is similar to `JsonVariant`, excepts that it cannot modify the value of the variant.

That's not all: there is also a tiny performance improvement if you use `JsonVariantConst`. Indeed, because `JsonVariant` needs to allocate memory, it contains a pointer to the memory pool. Here is a simplified definition:

```
class JsonVariant {  
    VariantData* _data;  
    MemoryPool* _pool;  
};
```

`JsonVariantConst`, however, never allocates memory, so it doesn't need this pointer. Here is a simplified definition:

```
class JsonVariantConst {  
    const ArrayData* _data;  
};
```

As you can see, `JsonVariantConst` is slightly smaller than `JsonVariant`, so copying it is slightly faster.

6.6 Inside JsonObject

In the previous section, we looked at `JsonVariant`, a versatile class that supports several types of value. Now, we'll look at a class that only supports one type: objects.

6.6.1 Reference semantics

We saw that `JsonVariant` is a reference to a value in a `JsonDocument`. Similarly, `JsonObject` is a reference to an object in the `JsonDocument`.

Like `JsonVariant`, `JsonObject` has reference semantics. As we saw, it means that when you assign a `JsonObject` to another, you change the reference, not the value. Here is an example:

```
JsonObject obj1 = doc.to<JsonObject>();
obj1["value"] = 1;
// "doc" contains {"value":1}

JsonObject obj2 = obj1;
obj2["value"] = 2;
// now, "doc" contains {"value":2}
```

In this snippet, `obj1` and `obj2` point to the same object; the object itself is located in the `JsonDocument`.

6.6.2 Null object

Like `JsonVariant`, `JsonObject` implements the null object pattern: when a `JsonObject` is null, it silently ignores all the calls.

For example, the following line declares a null `JsonObject`:

```
JsonObject obj1;
```

You can safely use `obj1` in your code: it will behave as an empty object.

If you need to know if a `JsonObject` is null, you can call `JsonObject::isNull()`

6.6.3 Create an object

Because `JsonObject` is a reference, you need `JsonDocument` to initialize a `JsonObject` correctly.

Here are several examples:

```
// Clear the document and create a object
JsonObject newObject = doc.to<JsonObject>();

// Get the root object of the document
JsonObject existingObject = doc.as<JsonObject>();

// Create a nested object in the document
JsonObject nestedObject = doc.createNestedObject("key");
```

6.6.4 Implementation

As you already know, `JsonObject` is a collection of key-value pairs; the key is a `JsonString`, and the value is a `JsonVariant`. The collection is implemented as a linked-list: each element is stored in a “node” that contains a pointer to the next node.

Here is a simplified definition of the node class:

```
class CollectionNode {
    JsonString key;
    JsonVariant value;
    CollectionNode* next;
};
```

The node is named `CollectionNode` and not `ObjectNode` because, to reduce the size of the code, ArduinoJson uses the same class for arrays as well.

In the previous section, we saw that the union in the `JsonVariant` has one member of type `CollectionData`, which serves when the variant contains an object.

Here is a simplified definition of `CollectionData`:

```
struct CollectionData {
    CollectionNode* head;
    CollectionNode* tail;
};
```

This structure implements the classic linked-list with `head` that points to the first node and `tail` that points to the last node.

6.6.5 Subscript operator

The implementation of the subscript operator (`[]`) is more complicated than it seems. Indeed, it needs to support two opposite use cases: reading a value from the object and writing a value to the object.

```
// Use case 1: reading
value = obj["key"];

// Use case 2: writing
obj["key"] = value;
```

Returning a `JsonVariant` would address the first use case, but it would not solve the second, because the assignment operator (`=`) would replace the reference, not the value.

We could change the definition of the assignment operator, but we would still have a problem: what should we return when the key doesn't exist in the object? On the one hand, returning a null object solves the first use case, but it fails the second because the null object ignores all calls. On the other hand, we could create a new variant and return a reference, but it would modify the object even if we just want to read a value; so it would fail the first use case.

Since it's not possible to return a `JsonVariant`, `JsonObject::operator[]` returns a `MemberProxy`, a proxy class that mostly behaves like a `JsonVariant`, except for the assignment operator and other modifying functions.

When used for reading, `MemberProxy` calls `JsonObject::getMember()`, which returns a null object when the key is missing. When used for writing `MemberProxy` calls `JsonObject::getOrAddMember()` which creates the key-value pair if needed.

MemberProxy is a template class that works for `JsonObject` and `JsonDocument`. It can also recursively work on a `MemberProxy`. This feature allows chaining the calls to the subscript operator to read or write nested values.

Here is an example that leverages the recursive feature:

```
doc["config"]["network"]["port"] = 2736;
```

This line calls `MemberProxy<MemberProxy<MemberProxy<JsonDocument>>>::operator=(int)` to create the following document:

```
{
  "config": {
    "network": {
      "port": 2736
    }
  }
}
```

6.6.6 Member functions

A detailed list of methods and overloads is available in the “API Reference” on arduinojson.org. We’ll just see a summary here.

`JsonObject::begin()` / `end()`

This couple of functions returns the iterators that allow enumerating all the key-value pairs in the object.

Example:

```
// Print all keys
for (JsonObject::iterator it=obj.begin(); it!=obj.end(); ++it) {
  Serial.println(it->key()->c_str());
}
```

C++11 allows us to simplify the code above to:

```
// Print all keys
for (JsonPair p : obj) {
    Serial.println(p.key()->c_str());
}
```

JsonObject::clear()

This function removes all the key-value pairs from the object.

Remember that, because ArduinoJson uses a monotonic allocator, this function cannot release the memory used by the removed key-value pairs. In other words, **this function creates a memory leak**. You can use it, but not in a loop.

Example:

```
obj.clear();
```

This function doesn't set the `JsonObject` to `null`, so the result of `JsonObject::isNull()` is unchanged. This behavior differs from `JsonDocument::clear()`.

JsonObject::createNestedArray()

This function creates a new array as a member of the object. It takes the key as a parameter and returns a `JsonArray` that points to the new array.

Example:

```
// Create {"ports": [80, 443]}
JsonArray arr = obj.createNestedArray("ports");
arr.add(80);
arr.add(443)
```

JsonObject::createNestedObject()

This function creates a new object as a member of the object. It takes the key as a parameter and returns a `JsonObject` that points to the new object.

Example:

```
// Create {"wifi":{"ssid":"TheBatCave","password":"s0_S3crEt!"}}
JsonObject wifi = obj.createNestedObject("wifi");
wifi["ssid"] = "TheBatCave";
wifi["password"] = "s0_S3crEt!"
```

JsonObject::getMember()

This function takes a key as a parameter. If the key is present in the object, it returns a `JsonVariant` that points to the corresponding value. If the key is not present, it returns a null object.

I recommend using the subscript operator (`[]`) instead of this function because it provides a more intuitive syntax. As we saw, the subscript operator calls this function behinds the scenes.

JsonObject::getOrAddMember()

This function takes a key as a parameter. Like `getMember()`, if the key is present in the object, it returns a `JsonVariant` that points to the corresponding value. Unlike `getMember()`, if the key is not present, it creates a new key-value pair.

I recommend using the subscript operator (`[]`) instead of this function because it provides a more intuitive syntax. As we saw, the subscript operator calls this function behinds the scenes.

JsonObject::isNull()

This function tells whether the `JsonObject` is null, or if it points to an object.

Example:

```
if (obj.isNull()) ...
```

JsonObject::remove()

This function removes the specified member from the object.

Remember that, because ArduinoJson uses a monotonic allocator, this function cannot release the memory used by the member. In other words, **this function creates a memory leak**. You can use it, but not in a loop.

Example:

```
obj.remove("password");
```

JsonObject::size()

This function returns the number of key-value pairs in the object.

Example:

```
size_t numberOfThings = obj.size();
```

6.6.7 Const reference

In the previous section, we saw that `JsonVariantConst` is a read-only version of `JsonVariant`. Similarly, `JsonObjectConst` is a read-only version of `JsonObject`.

As with `JsonVariantConst`, there is a small performance benefit to using `JsonObjectConst` because it doesn't contain a pointer to the memory pool. Here are the simplified definitions:

```
class JsonObject {  
    CollectionData* _data;  
    MemoryPool* _pool;
```

```
};

class JsonObjectConst {
    const CollectionData* _data;
};
```

We'll use `JsonObjectConst` in the case studies.

6.7 Inside JsonArray

`JsonArray` shares most characteristics of `JsonObject`:

- It has reference semantics.
- Internally, it contains a pointer to a `CollectionData` structure.
- It uses the same `CollectionNode`, except that it ignores the `key` member.
- The subscript operator returns a proxy class named `ElementProxy` (as opposed to `MemberProxy`).
- There is a read-only version called `JsonArrayConst`

I think you get the idea, no need to repeat.

6.7.1 Member functions

A detailed list of methods and overloads is available in the “API Reference” on arduinojson.org. Here is a summary.

[JsonArray::add\(\)](#)

This function appends an element to the array. It returns a boolean that tells whether the operation was successful.

Example:

```
// Create ["127.0.0.1", 80]
arr.add("127.0.0.1");
arr.add(80);
```

JsonArray::begin() / end()

This couple of functions returns the iterators that allow enumerating all the elements in the array.

Example:

```
// Print integers
for (JsonArray::iterator it=arr.begin(); it!=arr.end(); ++it) {
    Serial.println(it->as<int>());
}
```

C++11 allows us to simplify the code above to:

```
// Print integers
for (JsonVariant elem : arr) {
    Serial.println(elem.as<int>());
}
```

JsonArray::clear()

This function removes all the elements from the array.

Remember that, because ArduinoJson uses a monotonic allocator, this function cannot release the memory used by the elements. In other words, **this function creates a memory leak**. You can use it, but not in a loop.

Example:

```
arr.clear();
```

This function doesn't set the JsonArray to null, so the result of `JsonArray::isNull()` is unchanged. This behavior differs from `JsonDocument::clear()`.

JsonArray::createNestedArray()

This function creates a new array and appends it to the current array. It returns a `JsonArray` that points to the new array.

Example:

```
// Create [[1,2],[3,4]]  
  
JsonArray arr1 = arr.createNestedArray();  
arr1.add(1);  
arr1.add(2);  
  
JsonArray arr2 = arr.createNestedArray();  
arr2.add(3);  
arr2.add(4);
```

JsonArray::createNestedObject()

This function creates a new object and appends it to the array. It returns a `JsonObject` that points to the new object.

Example:

```
// Create [{"ip":"192.168.0.1","port":5689}]  
JsonObject module = arr.createNestedObject();  
module["ip"] = "192.168.0.1";  
module["port"] = 5689;
```

JsonArray::getElement()

This function returns the `JsonVariant` at the specified index.

I recommend using the subscript operator (`[]`) instead of this function because it offers a more intuitive syntax. `ElementProxy` calls this function behind the scenes.

JsonArray::getOrAddElement()

This function returns the `JsonVariant` at the specified index, and creates it if it doesn't exist.

I recommend using the subscript operator (`[]`) instead of this function because it offers a more intuitive syntax. `ElementProxy` calls this function behind the scenes.

JsonArray::isNull()

This function tells whether the `JsonArray` is null, or if it points to an array.

Example:

```
if (arr.isNull()) ...
```

JsonArray::remove()

This function removes the element at the specified index from the array.

Remember that, because ArduinoJson uses a monotonic allocator, this function cannot release the memory used by element. In other words, **this function creates a memory leak**. You can use it, but not in a loop.

Example:

```
// Remove the second element
arr.remove(1);
```

JsonArray::size()

This function returns the number of elements in the array.

Example:

```
size_t numberOfElements = arr.size();
```

6.7.2 copyArray()

We saw the member functions of `JSONArray`, but that's not all. There is also `copyArray()`, a free function that allows copying elements between a `JSONArray` and a C array.

Copy from a JSONArray to a C array

If you call `copyArray()` with a `JSONArray` as the first argument, it copies all the elements of the `JSONArray` to the C array.

```
int destination[3];

deserializeJson(doc, "[1,2,3]");
copyArray(doc.as<JSONArray>(), destination);
```

This function also supports two-dimensional arrays:

```
int destination[3][2];

deserializeJson(doc, "[[1,2],[3,4],[4,6]]");
copyArray(doc.as<JSONArray>(), destination);
```

Copy from a C array to a JSONArray

If you call `copyArray()` with a `JSONArray` as the second argument, it copies all the elements of the C array to the `JSONArray`.

Example:

```
int source[] = {1, 2, 3};

// Create [1,2,3]
copyArray(source, arr);
```

This function also supports two-dimensional arrays:

```
int source[][][3] = {{1, 2, 3}, {4, 5, 6}};  
  
// Create [[1,2,3],[4,5,6]]  
copyArray(source, arr);
```

6.8 Inside the parser

6.8.1 Invoking the parser

The parser is the piece of code that performs the text analysis. It is used during the deserialization process to convert the input into a `JsonDocument`.

With ArduinoJson, a program invokes the parser through `deserializeJson()`:

```
DeserializationError deserializeJson(JsonDocument&, TInput input);
```

`TInput`, the input type, can be one of the following:

- `const char*`
- `char*` (enables the “zero-copy” mode, see below)
- `const __FlashStringHelper*`
- `const String&`
- `Stream&`
- `const std::string&`
- `std::istream&`

When `TInput` is a pointer, you can pass an integer to limit the size of the input, for example:

```
deserializeJson(doc, input, sizeof(input));
```

As we'll see later, this function also supports two optional parameters.

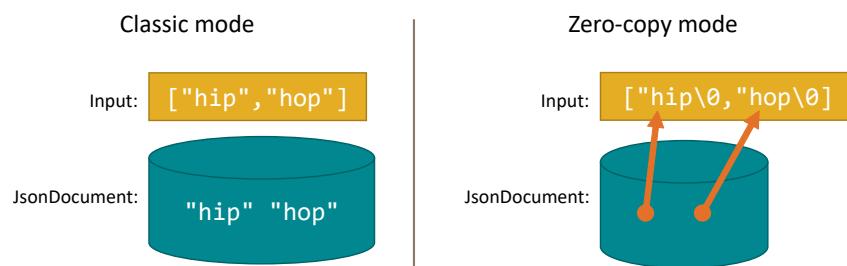
6.8.2 Two modes

As we said in the chapter [Deserialize with ArduinoJson](#), the parser has two modes:

- the “classic” mode, used by default,
- the “zero-copy” mode, used when the input is a `char*`.

The differences between these two modes have already been explained, but here is a summary:

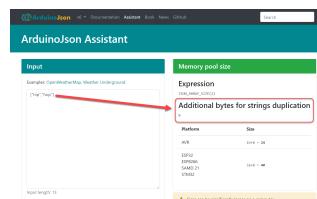
- the classic mode copies the strings from the input to the `JsonDocument`,
- the classic mode works with read-only inputs, like streams,
- the zero-copy mode stores pointers in the `JsonDocument`,
- the zero-copy mode modifies the input buffer to insert null terminators.



In a sense, the zero-copy mode uses the input buffer as an additional memory pool.

6.8.3 Pitfalls

With the classic mode, make sure that the capacity of the `JsonDocument` is large enough to store a copy of each string from the input. The ArduinoJson Assistant computes the total size of the strings under the name “Additional bytes for string duplication”; add this value to the capacity of the `JsonDocument`.



With the zero-copy mode, the `JsonDocument` stores pointers to the input buffer, so make sure you keep the input buffer in memory long enough.

6.8.4 Nesting limit

ArduinoJson's parser contains a recursive function. Each time there is a { or [in the input, the parser makes a new recursive call.

This recursion can be problematic if the input contains a deeply nested object or array. As each nesting level causes an additional recursion, it expands the stack. There is a risk of overflowing the stack, which would crash the program.

Here is an example:

```
[[[[[[[[[[[[[666]]]]]]]]]]]]]
```

The JSON document above contains 15 opening brackets. When the parser reads this document, it enters in 15 levels of recursions. If we suppose that each recursion adds 8 bytes to the stack (the size of the local variables, arguments, and return address), then this document adds up to 120 bytes to the stack. Imagine what we would get with more nesting levels...

This overflow is dangerous because a malicious user could send a specially crafted JSON document that makes your program crash. In the best-case scenario, it just causes a Denial of Service (DoS); but in the worst case, the attacker may be able to alter the variables of your program.

As it is a security risk, ArduinoJson puts a limit on the number of nesting levels allowed. This limit is 10 on an embedded platform and 50 on a computer. You can temporarily change the limit by passing an extra parameter of type `DeserializationOption::NestingLimit` to `deserializeJson()`.

The program below raises the nesting limit to 15:

```
const int size = 15*JSON_ARRAY_SIZE(1);
StaticJsonDocument<size> doc;

deserializeJson(doc, input, DeserializationOption::NestingLimit(15));

int value = doc[0][0][0][0][0][0][0][0][0][0][0][0][0][0]; // 666
```



ArduinoJson Assistant to the rescue!

When you paste your JSON input in the box, the ArduinoJson Assistant generates a sample program to parse the input. This program changes the nesting limit when necessary.

6.8.5 Quotes

The JSON specification states that strings are delimited by double quotes ("), but JavaScript allows single quotes (') too. In fact, JavaScript even allows object keys without quotes when there is no ambiguity.

Here is an example that is valid in JavaScript, but not in JSON:

```
{  
  hello: 'world'  
}
```

This example works because the key `hello` only contains alphabetic characters, but as soon as the key includes punctuations or spaces, we must use quotes.

Like JavaScript, ArduinoJson accepts:

- double quotes (")
 - single quotes (')
 - no quote for the key

6.8.6 Escape sequences

The JSON specification defines a list of escape sequences that allows including special characters (like line-breaks and tabs) in strings.

For example:

```
["hello\nworld"]
```

In this document, a line-break (`\n`) separates the words `hello` and `world`.

ArduinoJson handles the following escape sequences:

Escape sequence	Meaning
<code>\"</code>	Quote (")
<code>\/</code>	Forward slash
<code>\\\</code>	Backslash (\)
<code>\b</code>	Backspace
<code>\f</code>	Form feed
<code>\n</code>	Newline
<code>\r</code>	Carriage return
<code>\t</code>	Tabulation
<code>\uXXXX</code>	Unicode character*

When parsing a document, ArduinoJson replaces each escape sequence with its matching ASCII character. Similarly, when serializing, it replaces each character from the list with the matching escape sequence.

There is a star (*) on the Unicode escape sequence, because ArduinoJson supports it, but **this feature is disabled by default**. To enable this feature, you must define ARDUINOJSON_DECODE_UNICODE to 1. We'll see an example in [the case studies](#).

Note that the conversion works only in one way: ArduinoJson can decode the Unicode escape sequences to convert them to UTF-8 characters, but it cannot encode them back.

In practice, Unicode escape sequences are rarely used. Almost everyone encodes JSON documents in UTF-8, in which case escape sequences are not required. ArduinoJson disables the support by default because it reduces the size of the library significantly.

6.8.7 Comments

The JSON specification does not allow writing comments in a document, but JavaScript does. Here is an example that is valid JavaScript, but not in JSON:

```
{  
  /* a block comment */  
  "hello": "world" // a trailing comment  
}
```

ArduinoJson can read a document that contains comments; it simply ignores the comments. However, ArduinoJson is not able to write comments in a document. As a consequence, if you deserialize then serialize a document, you lose all the comments.

Like Unicode, this feature is optional and disabled by default. To enable it, you must define ARDUINOJSON_ENABLE_COMMENTS to 1.

6.8.8 NaN and Infinity

Similarly, JSON doesn't support `Nan` and `Infinity`, and by default, `deserializeJson()` returns `InvalidInput` when the input document contains them.

You can enable the support for these special values by setting `ARDUINOJSON_ENABLE_NAN` and `ARDUINOJSON_ENABLE_INFINITY` to 1.

6.8.9 Stream

A useful property of ArduinoJson's parser is that it tries to read as few bytes as possible. This behavior is helpful when parsing from a stream because it stops reading as soon as the JSON document ends.

For example:

```
{"hello": "world"}XXXXX
```

With this example, the parser stops reading at the closing brace `()`, giving you the opportunity to read the remaining of the stream as you want. For example, it allows sending JSON documents one after the other:

```
{"time":1582642357,"event":"temperature","value":9.8}  
{"time":1582642386,"event":"pressure","value":1004}  
{"time":1582642562,"event":"wind","value":22.8}
```

This technique, known as “JSON streaming”, was covered in the [“Advanced Techniques” chapter](#).

We also saw that this feature allows deserializing the input in chunks, and we’ll apply this technique in the [case studies](#).

Because ArduinoJson stops reading as soon as possible, it needs to consume the stream one byte at a time, which may affect the performance. If that’s a problem, remember that you can apply the buffering technique that we saw in the [“Advanced Techniques” chapter](#).

6.8.10 Filtering

Since ArduinoJson 6.15, `deserializeJson()` can filter the input document to keep only the values you’re interested in. Values excluded by the filter are simply ignored, which saves a lot of space in the `JsonDocument`.

To use this feature, you must create a second `JsonDocument` that will act as the filter. In this document, you must insert the value `true` as a placeholder for each value you want to keep. Then, you must wrap the filter document in `DeserializationOption::Filter` and pass it to `deserializeJson()`.

```
deserializeJson(doc, input, DeserializationOption::Filter(filter));
```

At the time of this writing, `deserializeMsgPack()` doesn’t support `DeserializationOption::Filter`.

We covered this feature in detail in the [“Advanced Techniques” chapter](#), and we’ll use it in the [OpenWeatherMap case study](#).

6.9 Inside the serializer

6.9.1 Invoking the serializer

The serializer is the piece of code that converts a `JsonDocument` to a textual representation. In ArduinoJson, there are three serializers:

- The minified JSON serializer, invoked by `serializeJson()`.
- The prettified JSON serializer, invoked by `serializeJsonPretty()`.
- The MessagePack serializer, invoked by `serializeMsgPack()`.

Since `serializeJson()`, `serializeJsonPretty()`, and `serializeMsgPack()` are very similar, we'll only study the first one.

Here is the signature:

```
size_t serializeJson(TSource, TOutput);
```

Source types

The type `TSource` can be:

- `const JsonDocument&`
- `JsonArray / JsonArrayConst`
- `JsonObject / JsonObjectConst`
- `JsonVariant / JsonVariantConst`

As you see, you don't have to serialize the complete `JsonDocument`. If you want, you can serialize only a part of it. For example, you can serialize only the "config" member like so:

```
serializeJson(doc["config"], Serial);
```

Output types

The type `TOutput` can be:

- `char[]`
- `char*`, in which case, you must specify the size
- `Print` or `std::ostream`
- `String` or `std::string`

If `TOutput` is a pointer, you must pass an extra argument to specify the size of the target buffer. For example:

```
serializeJson(doc, buffer, 256);
```

`serializeJson()` and all other variants return the number of bytes written.

6.9.2 Measuring the length

ArduinoJson provides three functions to measure the length of the serialized document:

1. `measureJson()` for a minified JSON document,
2. `measureJsonPretty()` for a prettified JSON document.
3. `measureMsgPack()` for a MessagePack document.

These functions are wrappers on top of `serializeJson()`, `serializeJsonPretty()`, and `serializeMsgPack()`. They pass a dummy implementation of `Print` that does nothing but counting the number of bytes written.



Performance

`measureJson()` (and two other variants) is a costly operation because it involves doing the complete serialization. Use it only if you must.

6.9.3 Escape sequences

The JSON serializer supports the same escape sequences as the parser, except the Unicode escape sequence.

When it encounters a special ASCII character, the JSON serializer replaces the character with the appropriate escape sequence. However, it doesn't produce the Unicode escape sequence (\uXXXX); instead, it leaves the UTF-8 characters unchanged.

6.9.4 Float to string

ArduinoJson implements float-to-string conversion using a unique algorithm optimized for:

1. low memory consumption,
2. small code size,
3. speed.

This algorithm has the following limitations:

1. it prints only nine digits after the decimal point,
2. it doesn't allow to change the number of digits,
3. it doesn't allow to change the exponentiation threshold.

I explained how this algorithm works in the article "Lightweight float to string conversion" in my blog. I invite you to check if you're interested.

After introducing this algorithm, I received several complaints of users that though there were too many digits after the decimal point. It was not a problem with the accuracy of the conversion, but these users wanted a value rounded to 2 decimal places, as it was in the previous version.

As stated above, we cannot specify the number of digits, but there is a very simple workaround: you can perform the rounding in your program.

```
// Rounds a number to 2 decimal places
double round2(double value) {
    return (int)(value * 100 + 0.5) / 100.0;
}
```

```
obj["pi"] = round2(3.14159); // 3.14
```

You can also use `serialized()` with a `String` to get the same result:

```
obj["pi"] = serialized(String(3.14159)); // 3.14
```

However, this version uses `String` and, therefore, dynamic memory allocation. By now, you should understand that it's the kind of thing I try to avoid.

6.9.5 NaN and Infinity

As we saw in the previous section, JSON doesn't support `Nan` and `Infinity`.

By default, `serializeJson()` writes `null` instead of `Nan` or `Infinity`, so that the output is conform to the JSON specification.

However, when `ARDUINOJSON_ENABLE_NAN` and `ARDUINOJSON_ENABLE_INFINITY` are set to 1, `serializeJson()` writes `Nan` and `Infinity`.

6.10 Miscellaneous

6.10.1 The version macro

ArduinoJson defines several macros that tell you which version of the library was included. You can use them to implement version-specific code, or most likely, to stop the build when the wrong version is installed.

Macro	Value
ARDUINOJSON_VERSION	"6.15.0"
ARDUINOJSON_VERSION_MAJOR	6
ARDUINOJSON_VERSION_MINOR	15
ARDUINOJSON_VERSION_REVISION	0

For example, if you want to stop the build if an older version of ArduinoJson is installed, you can write:

```
#if ARDUINOJSON_VERSION_MAJOR!=6 || ARDUINOJSON_VERSION_MINOR<15
#error ArduinoJson 6.15+ is required
#endif
```

6.10.2 The private namespace

Every single piece of the library is defined in a private namespace whose name is dynamic. The name includes the version number and the compilation options, which allows embedding several versions of the library in the same executable. For example, this feature is useful when you use a third-party library that depends on an older version of ArduinoJson. Of course, it's better to avoid embedding several versions of the library because it increases the size of the executable.

The actual name of this namespace is defined in the macro `ARDUINOJSON_NAMESPACE`. With ArduinoJson 6.15.0, with the default options, the name is `ArduinoJson6150_0000010`. You can recognize the version number in the first part of the name. The second part contains zeros and ones that correspond to each compilation option.

This namespace is an implementation detail, so in theory, you should never see it. Unfortunately, this name pops up frequently in error messages.

If ArduinoJson were written in C++11, it would use an “inline namespace” instead of this macro. Hopefully, in a future version, I’ll find a way to use an inline namespace without breaking the compatibility with old compilers.

6.10.3 The ArduinoJson namespace

As we just saw, ArduinoJson defines everything in a private namespace. To make the symbols public, it imports them in the ArduinoJson namespace with `typedef` and `using` statements.

In many cases, the symbol in the public namespace differs from the one in the public namespace. For example, `JsonObject` is `ObjectRef` in the private namespace. This technique allows me to drop the `Json` prefix and to use a richer vocabulary in the implementation.

6.10.4 ArduinoJson.h and ArduinoJson.hpp

Since using a namespace is not a common practice for Arduino libraries, the last line of `ArduinoJson.h` is a `using namespace` statement that brings all symbols in the global namespace.

ArduinoJson also provides another header, `ArduinoJson.hpp`, which doesn’t have this line. If you want to keep ArduinoJson in its namespace, include `ArduinoJson.hpp` instead of `ArduinoJson.h`.

```
#include <ArduinoJson.h>
#include <ArduinoJson.hpp>

void setup() {
    StaticJsonDocument<200> doc;
    ArduinoJson::StaticJsonDocument<200> doc;
    // ...
}
```

6.10.5 The single header

ArduinoJson is a header-only library: all the code is in the headers, there is no `.cpp` file. When you download the library via the Arduino Library Manager or by cloning the repository, you get the 129 header files that compose the library.

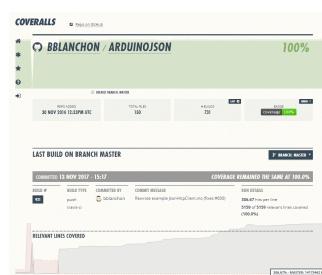
Alternatively, you can download a single file that aggregates all the 129 files. Having only one file simplifies the integration in a project because you can copy this file in your project folder. It also simplifies dependency management: everything is in the source of your project, so you don't have to worry about installed libraries.

You can download the single file distribution from the [Release page on GitHub](#). You can choose between `ArduinoJson.h` and `ArduinoJson.hpp` (see above).

6.10.6 Code coverage

The development of ArduinoJson followed the discipline of Test Driven Development (TDD), which imposes that one must write the tests before the production code. This technique allowed me to add many features to the library with relatively few bugs.

ArduinoJson is probably the only Arduino library to have a code coverage above 99%, meaning that almost every single line of code is verified by a unit test. The coverage status is monitored on [coveralls.io](#), and you can find the link on the GitHub page.



6.10.7 Fuzzing

As far as I know, only one vulnerability has been found in ArduinoJson. In 2015, a bug in the parser allowed an attacker to crash the program with a malicious JSON document. This vulnerability was filed under the reference [CVE-2015-4590](#). ArduinoJson 4.5 fixed this bug on the same day it was discovered.

A student found this bug with a technique called “fuzzing,” which consists in sending random inputs into the parser until the program crashes. After this incident, I added ArduinoJson to OSS-Fuzz, a project led by Google, that performs continuous fuzzing on open-source projects.

The screenshot shows the 'OSS-Fuzz build status' section of the CVE website. It lists several projects with their last build times and step details. For example, the 'fuchsia_fidl' project's last build was on 21/11/2017 at 06:06, starting with 'git clone' and ending with 'make'. Other projects listed include wget2, wireshark, arduinojson, and augeas.

Again, ArduinoJson is probably the only Arduino library to perform this kind of test.

6.10.8 Portability

ArduinoJson is very portable, meaning that the code can be compiled for a wide range of targets. Indeed, the library has very few dependencies because many parts, like the float-to-string conversion, are implemented from scratch. In particular, ArduinoJson doesn't depend on Arduino, so you can use it in any C++ project.

I've been able to compile ArduinoJson with all compilers that I had in hand, with one notable exception: Embarcadero C++ Builder.

Continuous Integration (CI) is a technique that consists in automating a build and test of the library each time the source code changes. ArduinoJson uses two web services for that: AppVeyor and Travis (see screen captures below).

The screenshot shows the Travis CI interface for the 'benoitblanchon / ArduinoJson' repository. The main summary indicates that the 'master' branch build has passed. Below this, the 'Build Jobs' section lists individual build steps with their status (e.g., #921.1, #921.2, etc.) and duration. The 'Build Artifacts' section shows compressed files available for download.

Here are the compilers used by the CI:

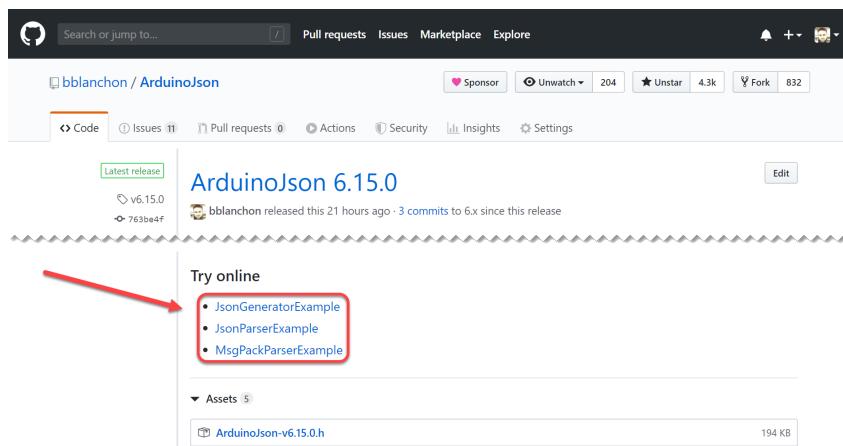
Compiler	Versions
Visual Studio	2010, 2012, 2013, 2015, 2017, 2019
GCC	4.4, 4.6, 4.7, 5, 6, 7, 8, 9
Clang	3.5, 3.6, 3.7, 3.8, 4, 5, 6, 7, 8

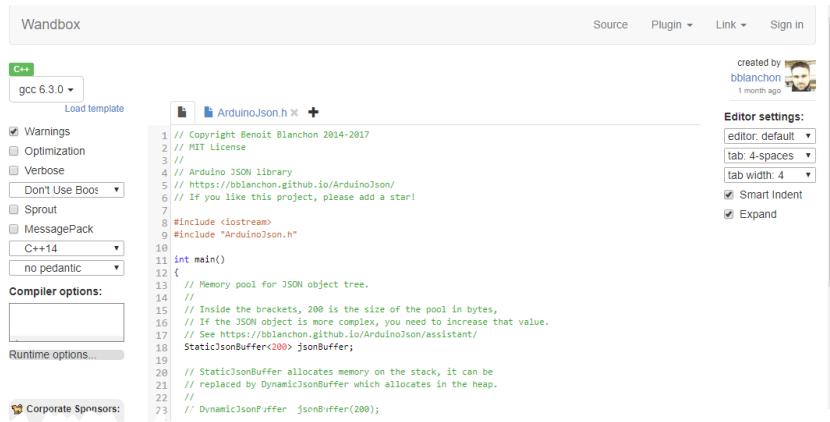
As this table attests, the portability of ArduinoJson is considered seriously. Needless to say that the CI forbids any error and any warnings.

6.10.9 Online compiler

Because ArduinoJson is self-contained (it has no dependency) and fits in a single header file, it is a perfect candidate for online compilers. You just need to copy the content of the header and paste it into the online compiler.

Every release of ArduinoJson is accompanied by links to examples on [wandbox.org](#) so that you can try the library online. That is very handy when you experiment with the library; for example, when you want to demonstrate some issue. I often use [wandbox.org](#) to answer questions on GitHub.





The screenshot shows the Wandbox IDE interface. On the left, there's a sidebar with compiler settings: C++, gcc 6.3.0, Warnings, Optimization, Verbose, Don't Use Boot, Sprout, MessagePack, C++14, no pedantic, Compiler options, Runtime options, and Corporate Sponsors. The main area displays the ArduinoJson.h code. The code includes a copyright notice, MIT license, and a main function. It also includes comments about memory pool sizes and links to GitHub. On the right, there are user settings for editor default, tab 4-spaces, tab width: 4, smart indent, and expand.

```
// Copyright Benoit Blanchon 2014-2017
// MIT License
//
// Arduino JSON library
// https://bblanchon.github.io/ArduinoJson/
// If you like this project, please add a star!
//
#include <iostream>
#include "ArduinoJson.h"

int main()
{
    // Memory pool for JSON object tree.
    //
    // Inside the brackets, 200 is the size of the pool in bytes,
    // If the JSON object is more complex, you need to increase that value.
    // See https://bblanchon.github.io/ArduinoJson/assistant/
    StaticJsonBuffer<200> jsonBuffer;
    //
    // StaticJsonBuffer allocates memory on the stack, it can be
    // replaced by DynamicJsonBuffer which allocates in the heap.
    //
    // DynamicJsonBuffer jsonBuffer(200);
}
```

6.10.10 License

ArduinoJson is released under the terms of the MIT license, which is very permissive.

- You can use ArduinoJson in closed-source projects.
- You can use ArduinoJson in commercial applications without redistributing royalties.
- You can modify ArduinoJson without publishing the source.

Your only obligation is to provide a copy of the license, including the name of the author, with every copy of your software, but I promise I won't sue if you forget :-)

6.11 Summary

In this chapter, we looked at the library from the inside to understand how it works.

Here are the key points to remember:

- `JsonDocument` = `JsonVariant` + memory pool
- The size of the memory pool is fixed and must be specified when constructing the `JsonDocument`
- `JsonDocument` has value semantics.
- `JsonArray`, `JsonObject`, and `JsonVariant` have reference semantics.
- `JsonArrayConst`, `JsonObjectConst`, and `JsonVariantConst` are read-only versions of `JsonArray`, `JsonObject`, and `JsonVariant`.
- To enumerate the values in a `JsonDocument` or a `JsonVariant`, you must first cast it into a `JsonArray` or `JsonObject`.
- You can use the operator `|` to change the default value and stop the propagation of null pointers.

In the next chapter, we'll look at the common issues that ArduinoJson users are facing. From compilation errors to program crashes, we'll see how to solve these issues.

Chapter 7

Troubleshooting

”

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

– Brian Kernighan, Unix for Beginners

7.1 Program crashes

Like any C++ code, your program may crash if it contains a bug. In this section, we'll see the most common reasons why your program may crash.

7.1.1 Undefined Behaviors

Some bugs are very simple to find and understand, like dereferencing a null pointer, but most of the time, it's complicated because the program may work for a while and then fails for an unknown reason.

In the C++ jargon, we call that an "Undefined Behavior" or "UB." The C++ specification and the C++ Standard Library contain many UBs. Here are some examples with the `std::string` class, which is similar to `String`:

```
// Construct a string from a nullptr
std::string s(nullptr); // <- UB!

// Copy a string with more char than available
std::string s("hello", 666); // <- UB!

// Compare a string with nullptr
if(s == nullptr) {} // <- UB!
```

Fortunately, the `String` class of Arduino is different, but it still has some vulnerabilities.

7.1.2 A bug in ArduinoJson?

If your program crashes, especially if it fails when calling a function of ArduinoJson, don't blame the library too quickly.

By design, ArduinoJson never makes the program crash. As we saw, it implements the [null object design pattern](#), which protects your program from dereferencing a null pointer when a memory allocation fails or when a value is missing.

However, if you feed ArduinoJson with dangling pointers, it may cause the program to crash. The same thing happens if the program uses objects that have been destroyed.



It's not a bug in the library

Many ArduinoJson users reported crashes that they attributed to the library, but a tiny fraction was actually due to a bug in the library.

7.1.3 Null string

A common cause of crash with ArduinoJson is dereferencing a null string. For example, it can happen with the code below:

```
// Compare the password with the string "secret"
if (strcmp(doc["password"], "secret") != 0) { // <- UB if null
    // Password matches
}
```

`strcmp()` is a function of the C Standard Library that compares two strings and returns `0` when they are equal. Unfortunately, the specification states that the behavior is undefined if one of the two strings is null.

Since the behavior is undefined, the code above could work on one platform but crash on another. In fact, this example works on an ATmega328 but crashes on ESP8266.

The simplest way to fix the program above is to replace `strcmp()` with the equal operator (`==`) of `JsonVariant`:

```
// Compare the password with the string "secret"
if (doc["password"] == "secret") {
    // Password matches
}
```

7.1.4 Use after free

Security professionals use the term “use-after-free” to designate a frequent error in C and C++ programs. It refers to a heap variable that is used after being freed. Such a bug is very likely to corrupt the memory of the heap, and ultimately to cause a crash, but

it can also have no incidence. It can also expose a security risk if an attacker manages to exploit the way the memory is corrupted.

Here is an obvious instance of use-after-free:

```
// Allocate memory in the heap
int* i = (int*)malloc(sizeof(int));

// Release the memory
free(i);

// Use-after-free!
*i = 666;
```

The program dereferences the pointer after freeing the memory. It's easy to see the bug here because `free()` is called explicitly, but it can be more subtle in a C++ program, where the destructor releases the memory.

Here is an example where the bug is more difficult to see:

```
// Returns the name of the digital pin: D0, D1...
const char* pinName(int id) {
    // Construct a local variable with the name
    String name = String("D") + id;

    // Return teh string as a const char*
    return name.c_str();

    // The local variable "name" is destructed when the function returns
}

// Use-after-free!
Serial.println(pinName(0));
```

Here is another example involving ArduinoJson:

```
// Returns the name of the pin as configured in the JSON document
const char* pinName(int id) {
    // Allocate a JsonDocument
```

```
DynamicJsonDocument doc;

// Deserialize the JSON configuration
deserializeJson(doc, json);

// Return the name of the pin
return doc[id];

// The local variable "doc" is destructed when the function returns
}

// Use-after-free!
Serial.println(pinName(0));
```

Here is a last one to show that we don't need a function:

```
// Set the name of the pin
doc["pin"] = (String("D") + pinId).c_str();
// The temporary String is destructed after the semicolon ;)

// Use-after-free!
serializeJson(doc, Serial);
```

7.1.5 Return of stack variable address

The example above used a `DynamicJsonDocument` because the topic was use-after-free, so it had to be in the heap. It's possible to make a similar mistake using a variable on the stack, but this vulnerability is called a "return-of-stack-variable-address." Just replace `DynamicJsonDocument` by `StaticJsonDocument`, and you have one. As with use-after-free, the program may or may not work, and may also be vulnerable to exploits.

Here is an obvious example of return-of-stack-variable-address:

```
// Returns the name of the digital pin: D0, D1...
const char* pinName(int id) {
    // Declare a local string
    char name[4];
```

```
// Format the name of the pin
sprintf(name, "D%d", id);

// Return the name of the pin
return name;

// The local variable "name" is destructed when the function returns
}

// Use destroyed variable "name"!
Serial.println(pinName(0));
```

Here is another involving ArduinoJson:

```
// Returns an object with the configuration
JsonObject readConfig() {
    // Allocate a JsonDocument
    StaticJsonDocument<256> doc;

    // Parse the configuration file
    deserializeJson(doc, config);

    return doc.as<JsonObject>();

    // The local variable "doc" is destructed when the function returns.
    // Remember that JsonObject points to an object in the JsonDocument
};

// Use destroyed variable "doc"!
Serial.println(readConfig()["pins"][0]);
```

7.1.6 Buffer overflow

A “buffer overflow” happens when an index goes beyond the last element of an array. In another language, this would cause an exception, but in C++, it doesn’t. A buffer overflow usually corrupts the stack and therefore is very likely to cause a crash.

Here is an obvious buffer overflow:

```
char name[4];
name[0] = 'h';
name[1] = 'e';
name[2] = 'l';
name[3] = 'l';
name[4] = 'o'; // 4 is out of range
```

Hackers love buffer overflows because they are ubiquitous and often allow them to modify the behavior of the program. `strcpy()`, `sprintf()`, and the like are traditional sources of buffer overflow.

Here is a program using ArduinoJson, that presents a serious risk:

```
// Parse a JSON input
deserializeJson(doc, input);

// Declare a string to hold an IP address
char ipAddress[16];

// Copy the content into the variable
strcpy(ipAddress, obj["ip"]);
```

Indeed, what would happen if the string `obj["ip"]` contains more than 15 characters? A buffer overflow! This bug is very dangerous if the JSON document comes from an untrusted source. An attacker could craft a special JSON document that would change the behavior of the program.

We can fix the code above by using `strlcpy()` instead of `strcpy()`. `strlcpy()`, as the name suggests, takes an additional parameter that specifies the *length* of the destination buffer.

```
// Copy the content into the variable
strlcpy(ipAddress, obj["ip"] | "", sizeof(ipAddress));
```

As you see, I also added the “or” operator (`|`) to avoid the UB if `obj["ip"]` returns null.



Don't use `strncpy()`

The C Standard Library contains two similar functions: `strlcpy()` and `strncpy()`.

They both copy strings, they both limit the size, but only `strlcpy()` adds the null-terminator. `strncpy()` is almost as dangerous as `strcpy()`, so make sure you use `strlcpy()`.

7.1.7 Stack overflow

A “stack overflow” happens when the stack exceeds its capacity; it can have two different effects:

1. On a platform that limits the stack size (like the ESP8266), an exception is raised.
2. On other platforms (like the ATmega328), the stack and the heap walk on each other's feet.

In the first case, the program is almost guaranteed to crash, which is good. In the second case, the stack and the heap are corrupted, so the program is likely to crash and to be vulnerable to exploits.

With ArduinoJson, it happens when you use a `StaticJsonDocument` that is too big. As a general rule, limit the size of a `StaticJsonDocument` to half the maximum, so that there is plenty of room for other variables and the call stack (function arguments and return addresses).

By the way, if none of this makes sense, make sure you read [the C++ course](#) at the beginning of the book.



Unpredictable program

You just changed one line of code, and suddenly the program behaves unpredictably? Does it look like the processor is not executing the code you wrote?

This is the sign of a stack overflow; you need to reduce the number and the size of variables in the stack.

7.1.8 How to diagnose these bugs?

When troubleshooting this kind of bug, you must begin by finding the line of code that causes the crash. Here are the four techniques that I use.

Technique 1: use a debugger

As the name suggests, a debugger is a program that allows finding bugs in another program. With a debugger, you can execute your program line by line, see the content of the memory, and see exactly where your program crashes.

I often use this technique when the target program can run on a computer, or when I'm working on professional embedded software. Still, I never used a debugger for Arduino-like projects. I know there are some tool out there that allows debugging Arduino programs; I simply don't want to invest too much time setting them up. In my opinion, the better you are at programming, the less time you spend in the debugger. I don't want to spend time learning how to set up a debugging tool when I could write unit tests instead.

Technique 2: tracing

The second technique doesn't require a debugger, only a serial port or a similar way to view the log. Tracing consists in logging the operations of the program to better understand why it fails. In practice, this technique involves adding a few *judiciously placed* `Serial.println()`, for example:

```
Serial.println("Copying the ip address...");  
strcpy(ipAddress, obj["ip"]);  
Serial.println("IP address copied.")
```

If you run this program and only see "Copying the ip address...", you know that something went wrong with `strcpy()`.

The annoying side of this technique is that you have to pass a distinctive string each time you call `Serial.println()`. That's why most people write "1," "2," "3," etc., but then it becomes a mess when you add new traces between existing ones.

Personally, I prefer using a macro to automatically set the string to something distinctive: the name of the file, the line number, and the name of the current function. Here is an example:

```
TRACE();  
strcpy(ipAddress, obj["ip"]);  
TRACE();
```

This program would print something like:

```
MyProject.ino:42: void setup()  
MyProject.ino:44: void setup()
```

I packaged this macro with a few other goodies in the [ArduinoTrace](#) library; you can download it from the Arduino Library Manager.

Technique 3: remove/replace lines

The previous technique only works when the program crashes instantly. Sometimes, however, the bug doesn't express itself instantly, and the program crashes a few lines or a few seconds later. In that case, the tracing technique doesn't help because it will point to the *effect* and not to the *cause*.

To troubleshoot this kind of bug, the best way I know is to remove suspicious lines or to replace them with something simpler. For example, if we suspect that there could be something wrong with our previous call to `strcpy()`, we could replace the variable with a constant:

```
strcpy(ipAddress, "192.168.1.1"); // <- obj["ip"]
```

If, after doing this simple substitution, the program works fine, then it means there was a problem with this line.

When using this technique, I strongly recommend that you use a source control system, such as Git, to keep track of all the changes you make.

Technique 4: git bisect

Unfortunately, it can be quite challenging to implement the previous technique when the bug is sneaky. Indeed, sometimes you try every possible substitution and yet, nothing improves.

When I'm in this situation, my technique is to roll back the code until I get to a stable version. Of course, to implement this technique, you must have a version control system in place, and you must regularly commit your files.

Git provides a command that helps with this task: `git bisect`. This command will perform a binary search to find the first faulty commit. To start the search, you must

specify which version you know to be “good”, and which one you know to be “bad.” Git will checkout commit between these points, and you’ll have to tell if it’s good or bad. It will repeat the operation until there is only one possible commit. I realize that this process seems cumbersome, but it’s not that complicated once you understand it.

Once I know which commit causes the error, I try to find the smallest possible change that triggers it, and that’s how I find the bug.

Note that this technique is not limited to finding bugs; I frequently use it to reduce the code size and to improve the performance.

7.1.9 How to prevent these bugs?

The ultimate solution is to write unit tests and run them under monitored conditions. There are two ways to do that:

1. You can run the executable in an instrumented environment. For example, Valgrind does precisely that.
2. You can build the executable in with a flag that enables instrumentation of the code. Clang and GCC offer `-fsanitize` for that.

However, you and I know you’re not going to write unit tests for an Arduino program, so we need to find another way of detecting these bugs.

I’m sorry to tell you that, but there is no magic solution, you need to learn how C++ works and recognize the bugs in the code. These bugs manifest in many ways, so it’s impossible to dress an exhaustive list. However, there are risky practices that are likely to cause troubles:

- functions returning a pointer (like `String::c_str()` or `JsonVariant::as<char*>()`)
- functions returning a reference
- a pointer living longer than its target
- manual memory management with `malloc()/free()` or `new/delete`

These risky practices should raise a red flag when you review the code, as they are likely to cause trouble. Unfortunately, you cannot eliminate all of them because there are many legitimate usages too.



Making sense of Exception in ESP8266 and ESP32

When an exception occurs in an ESP8266 or an ESP32, it's logged to the serial port. The log contains the type of the exception, the state of the registers, and a dump of the stack. There is an excellent tool to extract the call stack, which is very helpful to debug the program.

Visit: github.com/me-no-dev/EspExceptionDecoder

The screenshot shows two windows side-by-side. On the left is the Arduino IDE window titled 'sketch_dec07a | Arduino 1.8.5'. It displays a sketch with the following code:1 void setup() {
2 Serial.begin(115200);
3 while (!Serial);
4 char* p = 0;
5 p[0] = 666;
6 }
7
8 void loop() {
9 }A red arrow points from the line 'p[0] = 666;' in the sketch towards the right window. The right window is titled 'Exception Decoder' and shows the output of the decoder tool. The log includes:

```
epc1=0x40201c1b epc2=0x00000000 epc3=0x00000000 excvaddr=0x00000000  
depc=0x00000000  
  
ctx: cont  
sp: 3ffef160 end: 3ffef330 offset: 01a0  
>>>stack>>>  
3ffe300: 3ffffd00 00000000 3ffee2e4 40201c1b  
3ffe310: feeffefffe feeffefffe 3ffee300 40201ff8  
3ffe320: feeffefffe feeffefffe 3ffee310 40100114  
<<<stack<<<  
  
ets Jan  8 2013,rst cause:2, boot mode:(3,6)  
  
load 0x4010f000, len 1384, room 16  
tail 8  
  
Decoding 5 results  
0x40201c1b: setup at C:\Users\Benoit\AppData\Local\Temp\arduino_modified_sketch_17635  
0x40201c1b: setup at C:\Users\Benoit\AppData\Local\Temp\arduino_modified_sketch_17635  
0x40201ff8: loop_wrapper at C:\Users\Benoit\AppData\Local\Arduino15\packages\esp8266\br...  
0x40100114: cont_norm at C:\Users\Benoit\AppData\Local\Arduino15\packages\esp8266\br...
```

7.2 Deserialization issues

In the chapter [Deserialize with ArduinoJson](#), we saw that `deserializeJson()` returns a `DeserializationError` that can have one of the following values: `Ok`, `IncompleteInput`, `InvalidInput`, `NoMemory`, `NotSupportedException`, and `TooDeep`.

In this section, we'll see when these errors occur and how to fix them.

7.2.1 IncompleteInput

ArduinoJson returns `DeserializationError::IncompleteInput` when the beginning of the document was correct, but the end was missing.

Reason 1: Input buffer is too small

Typically, `deserializeJson()` returns `IncompleteInput` when the program uses a buffer that is too small to contain the entire document.

Here is an example:

```
File file = SD.open("myapp.cfg");

char json[32];
file.readBytes(json, sizeof(json));

deserializeJson(doc, json);
```

If the buffer is too small, the document gets truncated, and because the end is missing, `deserializeJson()` returns `IncompleteInput`.

One solution is to increase the size of the buffer. However, if this JSON document comes from a stream, the best solution is to let ArduinoJson read the stream directly.

Here is how we could fix the example above:

```
File file = SD.open("myapp.cfg");
deserializeJson(doc, file);
```

If you choose this option, remember that you need to increase the capacity of the `JsonDocument` because it will contain a copy of each string from the input stream.

Reason 2: Input stream timed out

If `deserializeJson()` reads from a stream and returns `IncompleteInput`, it often means that a timeout occurred before the end of the document. It can be because the transmission is unreliable or too slow.

To fix this problem, you can increase the timeout by calling `Stream::setTimeout()`:

```
client.setTimeout(10000);
deserializeJson(doc, client);
```

If the problem persists, it probably means that the transmission is unreliable. There is no easy fix for that: you need to work on the quality of the transmission; for example, you can reduce the speed to improve the error ratio.

Reason 3: Input is empty

`ArduinoJson` doesn't have a code dedicated to empty input, so it uses `IncompleteInput` for that.

7.2.2 InvalidInput

`deserializeJson()` returns `DeserializationError::InvalidInput` when the input contains something that is not allowed by the JSON format.

Reason 1: Wrong input format

`InvalidInput` can mean that the input is in a completely different format, like XML or MessagePack.

For example, it could be because you forgot to specify the `Accept` header in an HTTP request. To fix that, you can explicitly state that you want a response in the `application/json` format:

```
GET /example/ HTTP/1.0
Accept: application/json
```

Of course, that's just one example. Some HTTP APIs don't use the `Accept` header, but instead use a query parameter (e.g., `?format=json`), or the file extension (e.g., `/weather.json`).

Reason 2: Corrupted file

You can get the same error if you try to read a corrupted file from an SD card or similar. Indeed, my experience showed that SD cards are unreliable on Arduino.

Reason 3: Transmission errors

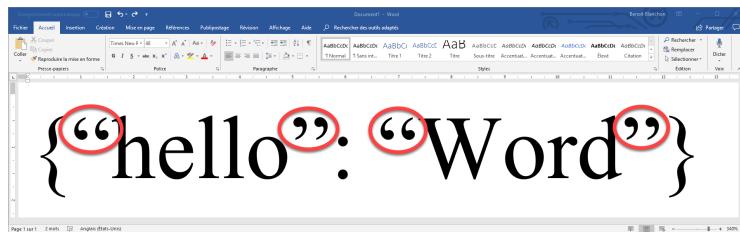
`InvalidInput` can be caused by one or several incorrect characters in the input. For example, it can happen if you use a serial connection between two boards. Indeed, the serial connection often inserts errors in the transmission.

If that happens to you, you can reduce the error ratio by reducing the transmission speed. However, the best solution is to add an error detection mechanism (for example, with a checksum) and retransmit buggy messages.

Reason 4: Wrong quotation marks

Sometimes, a JSON document that looks good to the human eye produces an `InvalidInput`. In that case, it's often a problem with the quotation marks: the document contains curly quotation marks ("..."), but the JSON format uses straight quotation marks ("...").

This problem occurs when you copy a JSON document from a website or a word processor (see image below). To fix this error, you need to replace all the buggy quotation marks with the right ones.



Reason 5: HTTP headers

The following program also produces `InvalidInput`:

```
// send HTTP request
client.println("GET /api/example/ HTTP/1.0");
client.println();

// read HTTP response
deserializeJson(doc, client);
```

The error occurs because the program doesn't skip the HTTP headers before parsing the body. To fix this program, you just need to call `Stream::find()`, as we did in [the tutorial](#):

```
// send HTTP request
client.println("GET /api/example/ HTTP/1.0");
client.println();

// skip headers
client.find("\r\n\r\n");

// read HTTP response
deserializeJson(doc, client);
```

Reason 6: Chunked Transfer Encoding

Very often, `InvalidInput` is caused by HTTP's "chunked transfer encoding," a feature that allows the server to send the response in multiple parts or "chunks." Here is a

response that uses chunked transfer encoding:

```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Transfer-Encoding: chunked

9
{"hello":
8
"world"}
0
```

Before each chunk, the server sends a line with the size of the chunk. To end the transmission, it sends an empty chunk. This is a problem for ArduinoJson because the chunk sizes can appear right in the middle of a JSON document.

To avoid this problem, you can use HTTP/1.0 instead of HTTP/1.1 because chunked transfer encoding is an addition of HTTP version 1.1.

Reason 7: Byte order mark

The byte order mark (BOM) is a hidden Unicode character that is sometimes added at the beginning of a document to identify the encoding.

Since ArduinoJson only supports 8-bit characters, the only possible BOM that we can encounter is the UTF-8 one. The UTF-8 BOM is composed of the three following bytes: EF BB BF. Usually, this character is invisible, but editors that don't support UTF-8 show it as `\u202a`.

ArduinoJson doesn't support byte order marks, so you need to skip the first three bytes before passing the input to `deserializeJson()`. For example, if you use a stream for input, you can write:

```
if (input.peek() == 0xEF) {
    // Skip the BOM
    input.read();
    input.read();
    input.read();
```

```
}
```

```
deserializeJson(doc, input);
```

If you're using a buffer as the input, you can write:

```
if (input[0] == 0xEF)
    // Skip the BOM
    deserializeJson(doc, input + 3);
else
    deserializeJson(doc, input);
```

As you see, this code uses pointer arithmetic to skip the first three characters.

Reason 8: Support for comments is disabled

`deserializeJson()` also returns `InvalidInput` if the input contains a comment and the support is disabled. Remember that JSON doesn't allow comments, but `ArduinoJson` supports them as an optional feature. To enable comments in `ArduinoJson`, you must define `ARDUINOJSON_ENABLE_COMMENTS` to 1:

```
#define ARDUINOJSON_ENABLE_COMMENTS 1
#include <ArduinoJson.h>
```

7.2.3 NoMemory

Reason 1: JsonDocument is too small

`ArduinoJson` returns `DeserializationError::NoMemory` when the `JsonDocument` is too small. In that case, you need to increase the capacity of the `JsonDocument`.

Remember that you can use the `ArduinoJson` Assistant to compute the right capacity for your project.

Reason 2: Heap memory is exhausted

If you are using a `DynamicJsonDocument` and are still getting a `NoMemory` error after increasing the capacity, it means that the allocation of the memory pool failed.

Remember that `DynamicJsonDocument`'s constructor calls `malloc()` to allocate its memory pool in the heap. If the requested size is too big, `malloc()`, fails and the `DynamicJsonDocument` ends up having no memory pool at all.

You can check that a `DynamicJsonDocument`'s memory pool was correctly allocated by calling `JsonDocument::capacity()`:

```
DynamicJsonDocument doc(1048576);
if (doc.capacity() == 0) {
    // allocation failed!
}
```

If this problem happens, you need to make some room in the heap and fight heap fragmentation. I recommend that you start by removing as many `Strings` as possible because they are heap killers.

If there is no way to make enough room in the heap, look at the size-reduction techniques presented in the previous chapter.

7.2.4 NotSupported

Reason 1: Unicode escape sequence

`deserializeJson()` returns `DeserializationError::NotSupported` when the input document contains Unicode escape sequences (`\uXXXX`).

ArduinoJson can decode Unicode escape sequences, but this feature is disabled by default. To enable this feature, you must define `ARDUINOJSON_DECODE_UNICODE` to `1` before including the library:

```
#define ARDUINOJSON_DECODE_UNICODE 1
#include <ArduinoJson.h>
```

This feature is disabled by default because it significantly increases the size of the parser, and very few users need it. Indeed, the JSON documents rarely contain Unicode escape sequences because they are usually encoded with UTF-8 (see next).



How to specify the charset in an HTTP request?

Most HTTP servers return response encoded with UTF-8, but sometimes you need to explicitly tell the server that you want UTF-8. There are two ways to do that. You can either add the `Accept-Charset` header in the request:

```
GET /example/ HTTP/1.0
Accept: application/json
Accept-Charset: UTF-8
```

Or you can specify the charset in the `Accept` header:

```
GET /example/ HTTP/1.0
Accept: application/json; charset=UTF-8
```

Reason 2: MessagePack

ArduinoJson supports almost all features of MessagePack, except “bin format” and “timestamp.” If `deserializeMsgPack()` returns `NotSupportedException`, your only option is to use another library.

7.2.5 TooDeep

As we saw in the previous chapter, ArduinoJson’s parser limits the depth of the input document to protect your program against potential attacks. If the depth (i.e., the nesting level) of the input document exceeds the limit, ArduinoJson returns `DeserializationError::TooDeep`.

To fix this error, you need to raise the nesting limit. You can do that with an extra parameter to `deserializeJson()`:

```
deserializeJson(doc, input, DeserializationOption::NestingLimit(20));
```

Alternatively, you can change the default nesting limit by defining the macro ARDUINOJSON_DEFAULT_NESTING_LIMIT:

```
#define ARDUINOJSON_DEFAULT_NESTING_LIMIT 20
#include <ArduinoJson.h>
```

7.3 Serialization issues

In this section, we'll see the most common problems you might get when serializing a JSON document.

7.3.1 The JSON document is incomplete

If the generated JSON document misses some parts, it's because the `JsonDocument` is too small.

For example, suppose you want to generate the following:

```
{  
  "firstname": "max",  
  "name": "power"  
}
```

If instead, you get the following output:

```
{  
  "firstname": "max"  
}
```

Then the only thing you need to do is increase the capacity of the `JsonDocument`.

As usual, use the `ArduinoJson Assistant` to compute the appropriate capacity, which may include additional bytes for the strings that need to be duplicated.

7.3.2 The JSON document contains garbage

If the generated JSON document includes a series of random characters, it's because the `JsonDocument` contains a dangling pointer.

Here is an example:

```
// Returns the name of a digital pin: D0, D1...
const char* pinName(int id) {
    String s = String("D") + id;
    return s.c_str();
}

StaticJsonDocument<128> doc;
doc["pin"] = pinName(0);
serializeJson(doc, Serial);
```

The author of this program expects the following output:

```
{"pin": "D0"}
```

Instead, she is very likely to get something like that instead:

```
{"pin": "sÙd4xaÜY9ËåQ¥º;"}
```

The JSON document contains garbage because `obj["pin"]` stores a pointer to a de-structured object. Indeed, the temporary `String` declared in `pinName()` dies as soon as the function exits. By the way, this is an instance of use-after-free, as we saw [earlier in this chapter](#).

There are many ways to fix this program; the simplest is to return a `String` from `pinName()`:

```
// Returns the name of a digital pin: D0, D1...
String pinName(int id) {
    return String("D") + id;
}
```

When we insert a `String`, the `JsonDocument` duplicates it, so the temporary string can safely be destructed. However, we now need to increase the capacity of the `JsonDocument` to be large enough to hold a copy of the string.

7.3.3 Too much duplication

We saw that ArduinoJson stores strings differently depending on their types. If the type of the string is `char*`, `String`, or `const __FlashStringHelper*`, ArduinoJson duplicates the string in the `JsonDocument` before saving the pointer. Only strings whose type is `const char*` avoid the duplication.

Usually, duplicating `String`s is what you expect because they are volatile by nature. However, duplicating a Flash string is probably not what you want, but ArduinoJson needs it. It's probably acceptable to make one or two copies of a Flash string, but it becomes problematic when the number increases.

Here is an example that fills the `JsonDocument` with a copy of the same string:

```
// Create an array with four nested objects
for (int i=0; i<4; i++) {
    JsonObject obj = doc.createNestedObject();

    obj[F("sensor_id")] = i;
}

serializeJson(doc, Serial);
```

This program produces the expected output:

```
[{"sensor_id":0}, {"sensor_id":1}, {"sensor_id":2}, {"sensor_id":3}]
```

However, the `JsonDocument` contains four copies of the string "sensor_id," which is a waste of memory. This duplication occurs because the program uses a Flash string, but it would be the same if it used a `String()`.

In this case, the solution is to use a regular string:

```
// Create an array with four nested objects
for (int i=0; i<4; i++) {
    JsonObject obj = doc.createNestedObject();

    obj[F("sensor_id")] = i;
    obj["sensor_id"] = i;
```

```
}
```

```
serializeJson(doc, Serial);
```

This program generates the same JSON document, except that the `JsonDocument` is much smaller because it stores pointers instead of copies of the string. As we saw in the C++ course, the string `"sensor_id"` resides in the “globals” area of the RAM; there is only one copy of the string and four pointers.



Don't overuse Flash strings

When used correctly, Flash strings are a good way to save RAM, but when misused, they waste RAM and ruin performance.

Only use Flash string for rarely used long strings, like log messages.

7.4 Common error messages

In this section, we'll see the most frequent compilation errors and how to fix them.

7.4.1 `x was not declared in this scope`

When the compiler doesn't find a symbol, it generates an error like:

```
error: 'StaticJsonDocument' was not declared in this scope  
error: 'DynamicJsonDocument' was not declared in this scope  
error: 'serializeJson' was not declared in this scope  
error: 'deserializeJson' was not declared in this scope
```

When this error occurs, it means that the program doesn't include `ArduinoJson.h`, or includes another version.

To solve this error:

- Check that your `.ino` file contains `#include <ArduinoJson.h>` at the top.
- If you have additional `.h` or `.cpp` files, they may also need to include `ArduinoJson`.
- Check that the correct version of `ArduinoJson` is installed; it should be version `6.x`.
- Check that there is no other `ArduinoJson.h` on your computer. This file could be in your project folder, or in an older installation of the Arduino IDE. Make a complete search on your hard drive if needed.

7.4.2 `Invalid conversion from const char* to char*`

`ArduinoJson` returns keys and values as `const char*`. If you try to put these values into a `char*`, the compiler will issue an error (or a warning) like the following:

```
error: invalid conversion from 'ArduinoJson... {aka const char*}' to 'char*'
```

This error occurs with any of the following expression:

```
char* sensor = root["sensor"];
char* sensor = root["sensor"].as<char*>();

// in a function whose return type is char*
return root["sensor"].as<char*>();
```

To fix this error, replace `char*` with `const char*`:

```
const char* sensor = root["sensor"];
const char* sensor = root["sensor"].as<char*>();

// change the return type of the function to const char*
return root["sensor"].as<char*>();
```

7.4.3 Invalid conversion from `const char*` to `int`

Let's say you have to deserialize the following JSON document:

```
{
  "modules": [
    {
      "name": "kitchen",
      "ip": "192.168.1.23"
    },
    {
      "name": "garage",
      "ip": "192.168.1.35"
    }
  ]
}
```

If you write the following program:

```
deserializeJson(doc, input);
JSONArray modules = doc["modules"];
```

```
const char* ip = modules["kitchen"]["ip"];
```

You'll get the following compilation error:

```
invalid conversion from 'const char*' to 'size_t {aka unsigned int}'...
```

`modules` is an array of objects; like any array, it expects an integer argument to the subscript operator (`[]`), not a string.

To fix this error, you must pass an integer to `JsonArray::operator[]`:

```
int id = modules["kitchen"]["id"];
int id = modules[0]["id"];
```

Now, if you need to find the module named “kitchen,” you need to loop and check each module one by one:

```
for (JsonObject module : modules) {
    if (module["name"] == "kitchen") {
        const char* ip = module["ip"];
        // ...
    }
}
```

7.4.4 No match for operator[]

This error occurs when you index a `JsonObject` with an integer instead of a string.

For example, it happens with the following code:

```
JsonObject root = doc.as<JsonObject>();

int key = 0;
const char* value = obj[key];
```

The compiler generates an error similar to the following:

```
no match for 'operator[]' (operand types are 'JsonObject' and 'int')
```

Indeed, a `JsonObject` can only be indexed by a string, like so:

```
int key = 0;
const char* key = "key";
const char* value = obj[key];
```

If you want to access the members of the `JsonObject` one by one, consider iterating over the key-value pairs:

```
for (JsonPair kv : obj) {
    Serial.println(kv.key().c_str());
    Serial.println(kv.value().as<char*>());
}
```

7.4.5 Ambiguous overload for operator=

Most of the time you can rely on implicit casts, but there is one notable exception: when you convert a `JsonVariant` to a `String`. For example:

```
String ssid = network["ssid"];
ssid = network["ssid"];
```

The first line compiles but the second fails with the following error:

```
ambiguous overload for 'operator=' (operand types are 'String' and 'Ardui...')
```

The solution is to remove the ambiguity by explicitly casting the `JsonVariant` to a `String`:

```
ssid = network["ssid"];
ssid = network["ssid"].as<String>();
```

7.4.6 Call of overloaded function is ambiguous

When the compiler says “ambiguous,” it’s usually a problem with the implicit casts. For example, the following call is ambiguous:

```
Serial.println(doc["version"]);
```

If you try to compile this line, the compiler will say:

```
call of overloaded 'println(ArduinoJson...)' is ambiguous
```

Indeed, `Print::println()` has several overloads, and the compiler cannot decide which is the right one. The following overloads are all equally viable:

- `Print::println(const char*)`
- `Print::println(const String&)`
- `Print::println(int)`
- `Print::println(float)`

There are two possible solutions depending on what you’re trying to do. If you know the type of value you want to print, then you need to call `JsonVariant::as<T>()`:

```
Serial.println(doc["version"].as<int>());
```

However, if you want to print any type, you need to call `serializeJson()`:

```
serializeJson(doc["version"], Serial); // print the value
Serial.println(); // line break
```

7.4.7 The value is not usable in a constant expression

```
int capacity = JSON_OBJECT_SIZE(2);
StaticJsonDocument<capacity> doc;
```

This program produces the following compilation error:

```
error: the value of 'capacity' is not usable in a constant expression
```

This error occurs because `capacity` is a variable. As a variable, its value is computed at *run* time, whereas the compiler needs the value at *compile* time.

The solution is to convert `capacity` to a constant expression:

```
// before C++11
const int capacity = JSON_OBJECT_SIZE(2);

// since C++11
constexpr int capacity = JSON_OBJECT_SIZE(2);
```

Let's compare the two keywords:

- `const`: the value is a constant (i.e., not a variable).
- `constexpr`: the value is computed at compile time.

In this context, both work but `constexpr` is more suited.

7.5 Asking for help

If none of the directions provided in this chapter helps, you should visit [ArduinoJson's FAQ](#), it covers more questions than this book and is frequently updated.

If you cannot find the answer in the FAQ, I recommend opening a new [issue on GitHub](#). You may search existing issues first, but the FAQ already covers most topics.

When you write your issue, please take the time to write a good description. Providing the right amount of information is essential. On the one hand, if you give too little (for example, just an error message without any context), I'll have to ask for more. On the other hand, if you provide too much information, I'll not be able to extract the signal from the noise.

The perfect description is composed of the following:

1. A Minimal, Complete, and Verifiable Example (MCVE)
2. The expected outcome
3. The actual (buggy) outcome

As the name suggests, an MCVE should be a minimalist program that demonstrates the issue. It should have less than 50 lines. It's a good idea to test the MCVE on [wandbox.org](#) and share the link in the description of the issue. The process of writing an MCVE seems cumbersome, but it guarantees that the recipient (me, most likely) understands the problem quickly. Moreover, we often find a solution to our problem when we write the MCVE.

This advice works for any open-source project and, in a sense, in any human relationship. Carefully writing a good question shows that you care about the person receiving the request. Nobody wants to read a gigantic code sample with dozens of suspicious dependencies. If you respect the time of others, they'll respect yours and give you a quick answer.

GitHub issues for ArduinoJson usually get an answer in less than 24 hours. A very high priority is given to actual bugs, but a very low priority is given to frequently asked questions.



MCVE on `wandbox.org`

If you want to write an MCVE on `wandbox.org`, it's easier to start from an existing example; you can find links here:

- [ArduinoJson's home page](#)
- [GitHub releases page](#)

The screenshot shows the GitHub repository page for `bbblanchon / ArduinoJson`. A red arrow points to the 'Try online' section, which lists three examples: `JsonGeneratorExample`, `JsonParserExample`, and `MsgPackParserExample`. These examples are highlighted with a red box.

7.6 Summary

In this chapter, we saw how to diagnose and solve the most common problems that you may have with ArduinoJson.

Here are the key points remember:

- If you found a bug, it's more likely to be in your program than in the library.
- Undefined behaviors are latent bugs: your program may work for a while and then fail for obscure reasons.
- Common coding mistakes include:
 - Undefined behaviors with null pointers
 - Use after free
 - Return of stack variable address
 - Buffer overflow
 - Stack overflow
- If you use an ESP8266 or similar, you can use `EspExceptionDecoder`.
- “`DynamicJsonDocument` was not declared in this scope” means that the library is not correctly included, or that it's another version of the library.
- “Invalid conversion from `const char*` to `char*`” means you forgot a `const` somewhere.
- “Invalid conversion from `const char*` to `int`” means you used a `JSONArray` like a `JsonObject`.
- “No match for `operator[]`” means you used a `JsonObject` like a `JSONArray`.
- You can solve most “ambiguous” errors by calling `as<T>()`
- “`capacity` is not usable in a constant expression” means you forgot a `constexpr` in the declaration of `capacity`
- If you need assistance with the library, open an issue on GitHub and make sure you provide the right information.

In the next chapter, we'll study several projects and discover the common techniques you can use with ArduinoJson.

Chapter 8

Case Studies

”

I'm not a great programmer; I'm just a good programmer with great habits.

– Kent Beck

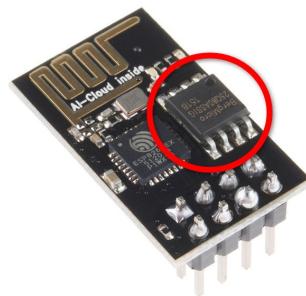
8.1 Configuration in SPIFFS

8.1.1 Presentation

For our first case study, we'll consider a program that stores its configuration in a file. The program uses a global configuration object that it loads at startup and saves after each modification.

In our example, we'll consider the file-system SPIFFS, but you can quickly adapt the code to any other file-system. SPIFFS allows storing files in a Flash memory connected to an SPI bus. Such a memory chip is attached to every ESP8266.

The code for this case study is in the SpiffsConfig folder of the zip file.



8.1.2 The JSON document

Here is the layout of the configuration file:

```
{  
  "access_points": [  
    {  
      "ssid": "SSID1",  
      "passphrase": "PASSPHRASE1"  
    },  
    {  
      "ssid": "SSID2",  
      "passphrase": "PASSPHRASE2"  
    }  
],  
  "server": {  
    "host": "www.example.com",  
    "path": "/resource",  
    "username": "admin",  

```

```
    "password": "secret"  
}  
}
```

This document contains two main parts:

1. a list of configurations for access points,
2. a configuration for a web service.

We assume the following constraints:

- there can be up to 4 access points,
- an AP SSID has up to 31 characters,
- an AP passphrase has up to 63 characters,
- each server parameter has up to 31 characters.

8.1.3 The configuration class

To store the configuration in memory during the execution of the program, we need to create a structure that contains all the information.

We'll mirror the hierarchy of the JSON document in the configuration classes. While it is not mandatory to have the same hierarchy, it dramatically simplifies the serialization code.

```
struct ApConfig {  
    char ssid[32];  
    char passphrase[64];  
};  
  
struct ServerConfig {  
    char host[32];  
    char path[32];  
    char username[32];  
    char password[32];  
};
```

```
struct Config {  
    static const int maxAccessPoints = 4;  
    ApConfig accessPoint[maxAccessPoints];  
    int accessPoints = 0;  
  
    ServerConfig server;  
};
```

The `Config` class stores the complete configuration of the program. It contains an array of four `ApConfig`s to store the configuration of the access points, and a `ServerConfig` to store the configuration of the web service.

8.1.4 `load()` and `save()` members

To make sure that the JSON serialization is always in-sync with the content of each structure, we'll perform the mappings in member functions. We need two functions: one to load from a JSON document and another to save to a JSON document.

```
struct ApConfig {  
    char ssid[32];  
    char passphrase[64];  
  
    void load(JsonObjectConst);  
    void save(JsonObject) const;  
};  
  
// other structures also get their own load() and save()
```

As you see, `load()` takes a `JsonObjectConst` because we want a read-only access to the JSON object. We could use a classic `JsonObject`, but it's clearer and safer to use a `JsonObjectConst`.

On the other hand, `save()` needs to alter the configuration object, so it takes a `JsonObject`. However, because `save()` doesn't modify the `ApConfig`, it can (and should) be declared `const`.

To simplify the code of `load()` and `save()`, we only pass the part of the JSON document that is necessary. In the case of the `ApConfig` above, we only need the object for one access point:

```
{  
    "ssid": "SSID1",  
    "passphrase": "PASSPHRASE1"  
}
```

There is a direct mapping from an `ApConfig` to a `JsonObject` because we decided to mirror the layout of the JSON document in the structures. The advantage is that this pattern keeps the related parts together; the drawback is that it couples the program with the JSON document: when one changes, you need to change the other.

8.1.5 Saving an ApConfig into a JsonObject

Let's zoom into the `save()` function:

```
void ApConfig::save(JsonObject obj) const {  
    obj["ssid"] = ssid;  
    obj["passphrase"] = passphrase;  
}
```

The role of this function is to save the content of the `ApConfig` into a `JsonObject`. As we said, it is marked a `const` because it doesn't modify the `ApConfig`. The code is very straightforward: it solely maps the members of the struct to the values in the `JsonObject`.

8.1.6 Loading an ApConfig from a JsonObject

Now, let's see the other side:

```
void ApConfig::load(JsonObjectConst obj) {  
    strlcpy(ssid, obj["ssid"] | "", sizeof(ssid));  
    strlcpy(passphrase, obj["passphrase"] | "", sizeof(passphrase));  
}
```

As you can see, `load()` reverses the operations of `save()`. This time, the function is not `const` because it modifies the `ApConfig`. However, we use a `JsonObjectConst` instead of a `JsonObject`, because we don't need to modify the JSON object.

`load()` is a bit more complicated than `save()` because we can't just copy the pointers returned by the `JsonObjectConst`; instead, we need to duplicate the strings.

8.1.7 Copying strings safely

Your intuition was probably to write something like:

```
ssid = obj["ssid"];
```

However, the member `ssid` is a `char[]`, so we cannot assign it from a `const char*`. Instead, we need to copy the string returned by `ArduinoJson`. The C Standard library provides a function to copy a string to another: `strcpy()`.

```
strcpy(ssid, obj["ssid"]);
```

This code compiles and works most of the time, but it presents a security risk because `strcpy()` doesn't check the capacity of the destination buffer. If the source is longer than expected, `strcpy()` blindly copies to the destination, causing a buffer overrun. The solution is to use `strlcpy()` instead, and to specify the size of the destination:

```
strlcpy(ssid, obj["ssid"], sizeof(ssid));
```

Again, this code compiles and works as long as the JSON document contains a `ssid` key. However, if `ssid` is missing, `obj["ssid"]` returns null, which is not supported by `strlcpy()`. The solution is to provide a default value that is not null, using the `|` syntax of `ArduinoJson`:

```
strlcpy(ssid, obj["ssid"] | "", sizeof(ssid));
```

I used an empty string as the default value, but we could use something else:

```
strlcpy(ssid, obj["ssid"] | "default ssid", sizeof(ssid));
```

8.1.8 Saving a Config to a JSON object

We saw how to map an `ApConfig` to a JSON object, and the process is the same for the `ServerConfig`.

It's time to see how to build the complete JSON document described at the beginning. It is the role of the `save()` function of `Config`, the top-level configuration structure:

```
void Config::save(JsonObject obj) const {
    // Add "server" object
    server.save(obj.createNestedObject("server"));

    // Add "acces_points" array
    JSONArray aps = obj.createNestedArray("access_points");

    // Add each access point in the array
    for(int i=0; i<accessPoints; i++)
        accessPoint[i].save(aps.createNestedObject());
}
```

As you can see, the `Config` structure delegates to its children the responsibility of saving themselves:

1. it calls `ServerConfig::save()` once to fill the `server` object,
2. it calls `ApConfig::save()` multiple times to save the configuration of each access point.

8.1.9 Loading a Config from a JSON object

Let's see the `load()` side:

```
void Config::load(JsonObjectConst obj) {
    // Read "server" object
    server.load(obj["server"]);

    // Get a reference to the "access_points" array
    JSONArrayConst aps = obj["access_points"];
```

```
// Extract each access points
accessPoints = 0;
for (JsonObjectConst ap : aps) {
    // Load the AP
    accessPoint[accessPoints].load(ap);

    // Increment AP count
    accessPoints++;

    // Is array full?
    if (accessPoints >= maxAccessPoints) break;
}
}
```

Like the function `save()`, the function `load()` delegates to the children the responsibility of loading themselves. However, this function has a little extra work because it needs to count the number of access points and make sure it doesn't exceed the capacity of the array.

Notice that we had to use a `JSONArrayConst` instead of a `JSONArray` because `JsonObjectConst` returns read-only references.

8.1.10 Saving the configuration to a file

Up to now, we only wrote the code to map the configuration structures into `JsonObject`s and `JsonArrays`, but we didn't create any JSON document. Here is the function that produces the JSON document from the `Config` instance:

```
bool serializeConfig(const Config &config, Print& dst) {
    DynamicJsonDocument doc(512);

    // Create an object at the root
    JsonObject root = doc.to<JsonObject>();

    // Fill the object
    config.save(root);

    // Serialize JSON to file
}
```

```
    return serializeJson(doc, dst) > 0;
}
```

The function above is responsible for the serialization, but it doesn't create the file on disk. The following function creates the file and calls `serializeConfig()`:

```
bool saveFile(const char *filename, const Config &config) {
    // Open file for writing
    File file = SPIFFS.open(filename, "w");
    if (!file) {
        Serial.println(F("Failed to create configuration file"));
        return false;
    }

    // Serialize JSON to file
    bool success = serializeConfig(config, file);
    if (!success) {
        Serial.println(F("Failed to serialize configuration"));
        return false;
    }

    return true;
}
```

I decided to split the serialization and the file handling into different functions because it keeps `serializeConfig()` independent from the underlying file-system. So, if you need to use another file system, you only need to modify `saveFile()`.

8.1.11 Reading the configuration from a file

We load the file in a similar way: one function is responsible for opening the file, and another does the deserialization.

```
bool deserializeConfig(Stream &src, Config &config) {
    DynamicJsonDocument doc(1024);

    // Parse the JSON object in the file
```

```
DeserializationError err = deserializeJson(doc, src);
if (err) return false;

config.load(doc.as<JsonObject>());
return true;
}

bool loadFile(const char *filename, Config &config) {
// Open file for reading
File file = SPIFFS.open(filename, "r");

// This may fail if the file is missing
if (!file) {
    Serial.println(F("Failed to open config file"));
    return false;
}

// Parse the JSON object in the file
bool success = deserializeConfig(file, config);

// This may fail if the JSON is invalid
if (!success) {
    Serial.println(F("Failed to deserialize configuration"));
    return false;
}

return true;
}
```

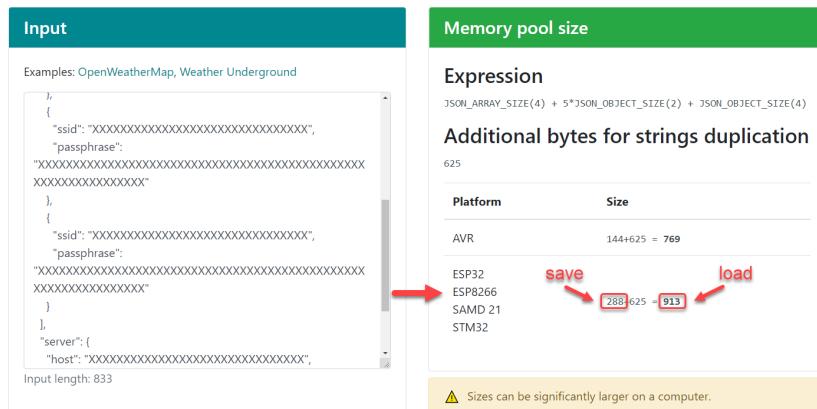
8.1.12 Choosing the JsonDocument

In both cases, I used a `DynamicJsonDocument`, but I could have used a `StaticJsonDocument` because the capacity (1024 bytes) is significantly lower than the limit (4096 bytes on the ESP8266).

The size of the `JsonDocument` for the deserialization is a lot bigger than the one for the serialization because it needs to copy the strings contained in the input stream. Remember that the file is a stream, each byte is read one by one, and it's the `JsonDocument`

that is responsible for assembling the result.

I used the ArduinoJson Assistant to compute the capacities and rounded the results to the closest powers of two (512 and 1024). I created a sample JSON document with the maximum allowed: four access points and all string filled to the max.



8.1.13 Conclusion

There are many ways to write this program; other developers would do it differently. One could argue that the `save()` and `load()` functions should not be members of the configuration structures. Another could say that the configuration classes should not expose `JsonObject` in their public interface.

All these remarks make sense. However, the perfect code doesn't exist, so we need to choose something that is practical, and that makes sense in our context. I decided to present this implementation as a case study because:

1. It's scalable. You can add more configuration members and continue to apply the same `load()`/`save()` pattern to each child. As these functions are concise, the overall complexity will remain constant.
2. It's easy to maintain. The pattern groups the things that must change together in the same class. For example, if you add a member in `ServerConfig`, it's easy to figure out that `ServerConfig::load()` and `ServerConfig::save()` must change accordingly.



Avoiding the custom data structure

Creating the `Config` data structure represents a significant work, so you might be tempted to use `JsonDocument` as your configuration container. If you made it so far in this book, you should know why I discourage you from doing that, but let's recap anyway:

1. The monotonic allocator is the secret behind the performance of Arduino; unfortunately, it leaks memory when we replace or remove values from a `JsonDocument`.
2. A `JsonDocument` contains objects that can contain several types of value (variants). This flexibility isn't free. Compared to a custom data structure, it adds overhead in speed, memory consumption, and program size.
3. A custom data structure contains a known list of members with fixed types. This rigidity simplifies and strengthens the rest of the program because you don't have to worry about missing members or incorrect types.
4. By restricting the use of `ArduinoJson` to the serialization functions, you decouple the rest of your code from the library. First, it allows you to switch gears at any time: you could decide to use another library or another serialization format. Second, it allows you to split the compilation units and possibly reduce the build times.

8.2 OpenWeatherMap on mkr1000

8.2.1 Presentation

For our second case study, we'll use a Genuino mkr1000 to collect weather forecast information from OpenWeatherMap, an online service that provides weather data. The program connects to a WiFi network, sends an HTTP request to openweathermap.org, and extracts the weather forecast from the response.



Because the response from OpenWeatherMap is too large to fit in the RAM, we'll reduce the size of the JSON document using the filtering technique.

In this section, I assume that you already read the [GitHub](#) and [Adafruit IO](#) examples; we'll use the same technique to perform the HTTP request.

You'll find the source code of this case study in the `OpenWeatherMap` directory of the zip file. To run this program, you need to create a free account on OpenWeatherMap and create an API key.

8.2.2 OpenWeatherMap's API

OpenWeatherMap offers several services; in this example, we'll use the 5-day forecast. As the name suggests, it returns the weather information for the next 5 days. Each day is divided into periods of 3 hours (8 per day), so the response contains 40 forecasts.

To download the 5-day forecast, we need to send the following HTTP request:

```
GET /data/2.5/forecast?q=London&units=metric&appid=APIKEY HTTP/1.0
Host: api.openweathermap.org
Connection: close
```

In the URL, `London` is just an example; you can replace it with another city. Then, `APIKEY` is a placeholder; you must replace it with your API key.

Note that we use `HTTP/1.0` instead of `HTTP/1.1` to disable “chunked transfer encoding.”

The response to this request has the following shape:

```
HTTP/1.0 200 OK
```

```
Date: Thu, 30 Nov 2017 10:19:59 GMT
```

```
Content-Type: application/json; charset=utf-8
```

```
Content-Length: 14754
```

```
{"cod": "200", "message": 0.005, "cnt": 40, "list": [{"dt": 1512043200, "main": {...
```

8.2.3 The JSON response

The body of the response is a minified JSON document of 14 KB. This document looks like so:

```
{
  "cod": "200",
  "message": 0.005,
  "cnt": 40,
  "list": [
    {
      "dt": 1512043200,
      "main": {
        "temp": 3.33,
        "temp_min": 1.49,
        "temp_max": 3.33,
        "pressure": 1015.46,
        "sea_level": 1023.3,
        "grnd_level": 1015.46,
        "humidity": 79,
        "temp_kf": 1.84
      },
      "weather": [
        {
          "id": 600,
          "main": "Snow",
          "description": "light snow",
          "icon": "13d"
        }
      ],
    }
  ],
}
```

```
"clouds": {  
    "all": 36  
},  
"wind": {  
    "speed": 6.44,  
    "deg": 324.504  
},  
"snow": {  
    "3h": 0.036  
},  
"sys": {  
    "pod": "d"  
},  
"dt_txt": "2017-11-30 12:00:00"  
},  
...  
],  
"city": {  
    "id": 2643743,  
    "name": "London",  
    "coord": {  
        "lat": 51.5085,  
        "lon": -0.1258  
    },  
    "country": "GB"  
}  
}
```

In the sample above, I preserved the hierarchical structure of the document, but I kept only one forecast. In reality, the `list` array contains 40 objects.

8.2.4 Reducing the size of the document

According to the ArduinoJson Assistant, we need a `JsonDocument` of nearly 32 KB to deserialize this response. In practice, it's impossible to do that on a Genuino mkr1000 because it has a total of 32 KB of RAM.

To reduce memory consumption, we'll filter the document to keep only the fields that are relevant to our project. In our case, we're only interested in the following fields:

- "dt", which contains the timestamp for the forecast,
- "temp", which contains the temperature in degrees,
- "description", which contains a textual description of the weather condition.

After applying the filter, the JSON document should look like so:

```
{  
  "list": [  
    {  
      "dt": 1511978400,  
      "main": {  
        "temp": 3.95  
      },  
      "weather": [  
        {  
          "description": "light rain"  
        }  
      ]  
    },  
    {  
      "dt": 1511989200,  
      "main": {  
        "temp": 3.2  
      },  
      "weather": [  
        {  
          "description": "light rain"  
        }  
      ]  
    },  
    ...  
  ]  
}
```

Again, this isn't the complete object, I reproduced only two of the forty forecast objects.

According to the ArduinoJson Assistant, this smaller document requires less than 8 KB to deserialize, which is definitely doable on the mkr1000.

8.2.5 The filter document

Now let's look at the filter itself. As we saw in the chapter "Advanced Techniques," to apply a filter, we need to create a second `JsonDocument` that acts as a template for the final document. The filter document only contains the fields we want to keep. Instead of actual values, it contains the value `true` as a placeholder.

When the filter document contains an array, only the first element is considered. This element acts as a filter for all elements of the array of the original document.

With this information, we can write the filter document:

```
{  
    "list": [  
        {  
            "dt": true,  
            "main": {  
                "temp": true  
            },  
            "weather": [  
                {  
                    "description": true  
                }  
            ]  
        }  
    ]  
}
```

We are now ready to create the corresponding `JsonDocument`:

```
StaticJsonDocument<128> filter;  
filter["list"][0]["dt"] = true;  
filter["list"][0]["main"]["temp"] = true;  
filter["list"][0]["weather"][0]["description"] = true;
```

We'll now wrap this document in a `DeserializationOption::Filter` before passing it to `deserializeJson()`:

```
deserializeJson(doc, client, DeserializationOption::Filter(filter));
```

8.2.6 The code

Here is a simplified version of the program without error checking. As I said, you'll find the source code in the `OpenWeatherMap` directory, and you'll need a key for the API of OpenWeatherMap.

```
// Connect to WLAN
WiFi.begin(SSID, PASSPHRASE);

// Connect to server
WiFiClient client;
client.connect("api.openweathermap.org", 80);

// Send HTTP request
client.println("GET /data/2.5/weather?q=" QUERY
                "&units=metric&appid=" API_KEY
                " HTTP/1.0");
client.println("Host: api.openweathermap.org");
client.println("Connection: close");
client.println();

// Skip response headers
client.find("\r\n\r\n");

// Deserialize the response
DynamicJsonDocument doc(8192);
deserializeJson(doc, client, DeserializationOption::Filter(filter));

// Close the connection to the server
client.stop();

// Extract the list of forecasts
JSONArray forecasts = doc["list"];

// Loop through all the forecast objects:
for (JsonObject forecast : forecasts) {
    Serial.print(forecast["dt"].as<long>());
    Serial.print(" : ");
    Serial.print(forecast["weather"][0]["description"].as<char *>());
```

```
Serial.print(" , ");
Serial.print(forecast["main"]["temp"].as<float>());
Serial.println(" °C");
}
```

8.2.7 Summary

This second case study was the opportunity to show a new technique the memory consumption when the input contains many irrelevant fields. Here are the key points to remember:

- Use `HTTP/1.0` to prevent chunked transfer encoding.
- Create a second `JsonDocument` that will act as a filter.
- Insert the value `true` as a placeholder for every field you want to keep.
- When filtering an array, only the first element of the filter matters.

In the next section, we'll use another technique to reduce the memory consumption: we'll deserialize the input in chunks.

8.3 Reddit on ESP8266

8.3.1 Presentation

We'll create a program that interacts with Reddit, a social network where users share links with other members of the network. Reddit has been around since 2005; it has thousands of communities organized in "subreddits." There are subreddits for everything, including Arduino. Not only can users share links, but they can also vote for links shared by others, and they can post comments as well.

For our case study, we'll print the current topics in /r/arduino, the subreddit about Arduino. Of course, this is just an example; in a real project, you would probably use an LCD instead of the serial port.



To give you an idea, here is the result:

The image shows a comparison between a web browser and the Arduino Serial Monitor. The browser window displays the /r/arduino subreddit with several posts listed. The serial monitor window shows the raw text being received from the browser, with red arrows indicating the mapping between the posts and the printed text in the monitor. This demonstrates how the Arduino code is reading and printing the content from the subreddit.

On the left, you see Reddit in my browser; on the right, you see the Arduino Serial Monitor.

In this case study, we'll use another technique to reduce memory consumption. Instead of filtering the input, we'll read it in chunks, a technique presented in chapter 5.

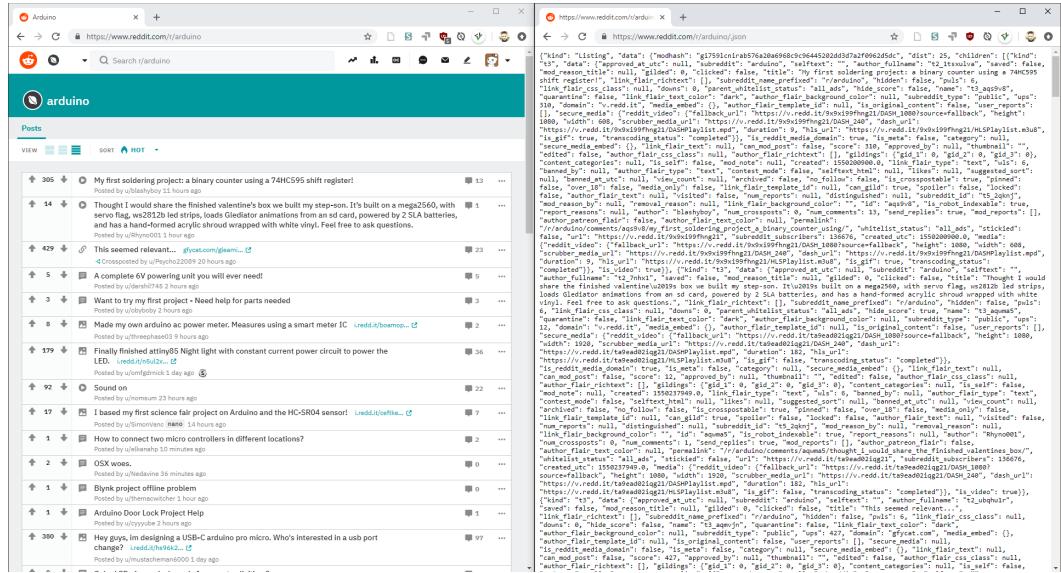
We'll use an ESP8266, like in the [GitHub example](#). This time, however, we'll not use the library `ESP8266HTTPClient`; instead, we'll work directly with `WiFiClientSecure`. We'll not check the SSL certificate of the server because we don't transmit sensitive information.

8.3.2 Reddit's API

Reddit has a complete API that allows sharing links, casting votes, posting comments, etc. However, to keep things simple, we'll only use a tiny part of the API: the one that allows us to download a Reddit page as a JSON document.

Reddit has a very interesting feature: you can append `.json` to nearly any URL, and you'll get the page as a JSON document. For example, if you go to www.reddit.com/r/arduino/, you'll see the Reddit website; but if you go to www.reddit.com/r/arduino/.json, you'll see the content of the page as a JSON document.

You can see the two versions in the picture below:



On the left, it's the HTML version. On the right, it's the JSON version.

I'm not sure we can really call that an "API," but that's perfect for our little project.

8.3.3 The response

As you can see from the picture above, the JSON document is very large. Here is the skeleton of the document:

```
{  
  "kind": "Listing",  
  "data": {  
    "modhash": "ra74f8579udc04d9b06c5308bf1aed96501a6ae8265dae6a14",  
    "dist": 25,  
    "children": [  
      {  
        "kind": "t3",  
        "data": {  
          "title": "When you have no friends, u need to practice highfives  
          ↳ in secret. Introducing cardboard hand with LSR",  
          "score": 350,  
          "author": "Crazi12345",  
          "num_comments": 15,  
          ...  
        }  
      },  
      {  
        "kind": "t3",  
        "data": {  
          "title": "Hey guys, im designing a USB-C arduino pro micro. Who's  
          ↳ interested in a usb port change?",  
          "score": 81,  
          "author": "mustacheman6000",  
          "num_comments": 20,  
          ...  
        }  
      },  
      ...  
    ]  
  }  
}
```

Again, the JSON document is huge, so I had to remove a lot of stuff to make it fit on this page. First, I reduced the number of posts from 25 to 2. Then I reduced the

number of values in each post from about 100 to just 4.

As you can see, after isolating the relevant parts, this JSON document is simpler than it initially looked. For our project, we are only interested in the `data.children` array. This array contains 25 objects: one per post. Each post object has around one hundred members, but we'll only use "title" and "score."

8.3.4 The main loop

We cannot deserialize this giant JSON document in one shot, because it would never fit in the memory of the microcontroller. We could filter the input as we did for OpenWeatherMap, but we'll try another approach this time. Instead of calling `deserializeJson()` once, we'll call it several times: once for each post object.

First, we'll make the HTTP request and check the status code. Then, we'll apply the technique as presented in chapter 5: we'll jump to the "children" array by calling `Stream::find()`.

```
client.find("\"children\"");
client.find("[");
```

As you can see, I made two calls to `Stream::find()`. One call would be sufficient if the document were minified, but it's not the case here. Therefore, we need to allow one or more spaces the colon (:).

Once we are in the array, we can call `deserializeJson()` to deserialize one "post" object. When the function returns, the next character should be either a comma (,) or a closing bracket (]). If it's a comma, we need to call `deserializeJson()` again to deserialize the next post. If it's a closing bracket, it means that we reached the end of the array.

As explained in chapter 5, we can use `Stream::findUntil()` to skip the comma or the closing bracket. This function takes two arguments; it returns true if it finds the first parameter, and returns false otherwise. Therefore, we can use the return value as a stop condition for the loop.

Here is the code of the loop:

```
DynamicJsonDocument doc(16 * 1024);
do {
    deserializeJson(doc, client);
```

```
// ...extract values from the document...
} while (client.findUntil(",", "["));
```

Notice that, this time, I declared the `JsonDocument` out of the loop, because it avoids calling `malloc()` and `free()` repeatedly. We don't have to clear the `JsonDocument`, because `deserializeJson()` does that for us.

8.3.5 Escaped Unicode characters

Unlike most HTTP APIs, Reddit doesn't encode the JSON document with UTF-8. Instead, it uses only the ASCII charset and encodes special characters with an escape sequence, for example, it writes my first name as "`Beno\u00E9t`" instead of "`Benoît`".

ArduinoJson can decode these escape sequences but, because this feature increases the program size significantly, it's turned off by default. So if we try to deserialize the response without changing the library settings, we'll probably get a `NotSupportedException`.

To turn on the support for Unicode escape sequence, we need to define the macro `ARDUINOJSON_DECODE_UNICODE` to 1 before including the library:

```
#define ARDUINOJSON_DECODE_UNICODE 1
#include <ArduinoJson.h>
```

When this option is activated, `deserializeJson()` translates the escape sequence to UTF-8.

8.3.6 Sending the request

Unlike OpenWeatherMap, Reddit forbids plain HTTP requests; it only accepts HTTPS requests. So, instead of using `WiFiClient` and port 80, we must use `WiFiClientSecure` and port 443.

```
WiFiClientSecure client;
client.connect("www.reddit.com", 443);
```

Appart from that, the rest of the program is similar to what we saw:

```
// Send the request
client.println("GET /r/arduino.json HTTP/1.0");
client.println("Host: www.reddit.com");
client.println("User-Agent: Arduino");
client.println("Connection: close");
client.println();
```

As before, we use HTTP version 1.0 to avoid chunked transfer encoding.

8.3.7 Assembling the puzzle

I think we covered all the important features of this case study, let's put everything together:

```
#define ARDUINOJSON_DECODE_UNICODE 1

#include <ArduinoJson.h>
#include <ESP8266WiFi.h>
#include <WiFiClientSecure.h>

void setup() {
    Serial.begin(115200);

    // Connect to the WLAN
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

    // Connect to the server
    WiFiClientSecure client;
    client.connect("www.reddit.com", 443);

    // Send the request
    client.println("GET /r/arduino.json HTTP/1.0");
    client.println("Host: www.reddit.com");
    client.println("User-Agent: Arduino");
    client.println("Connection: close");
    client.println();
```

```
// Jump to the "children" array
client.find("\"children\":");
client.find("[");

// Allocate a huge JsonDocument (16 KB)
DynamicJsonDocument doc(16 * 1024);

do {
    // Deserialize the next post
    deserializeJson(doc, client);

    // Print score and title
    Serial.print(doc["data"]["score"].as<int>());
    Serial.print('\t');
    Serial.println(doc["data"]["title"].as<char*>());
} while (client.findUntil(",", "]"));
}
```

I removed the error checking to make the code more readable. Please check out the complete source code in the `Reddit` directory of the zip file.

8.3.8 Summary

This case study demonstrated the "deserialization in chunks" technique.

Here are the key points to remember:

- Append `.json` to a Reddit page URL to get the page in JSON format.
- For HTTPS, replace `WiFiClient` with `WiFiClientSecure`, and port 80 with 443.
- Call `Stream::find()` to jump to the relevant location.
- Call `Stream::find()` twice instead of relying on the number of spaces.
- Call `Stream::findUntil()` to jump over the comma or the closing bracket.
- Use the return value of `Stream::findUntil()` as a stop condition for the loop.
- Define `ARDUINOJSON_DECODE_UNICODE` to 1 to enable the support for Unicode escape sequences.
- If you forget to do that, `deserializeJson()` returns `NotSupportedException`

I purposely used a small subset of the Reddit API. To go further, you'll need to create an account and request an access token for your program.

In the next case study, we'll see how we can use a `JsonDocument` as a member of a class.

8.4 JSON-RPC with Kodi

8.4.1 Presentation

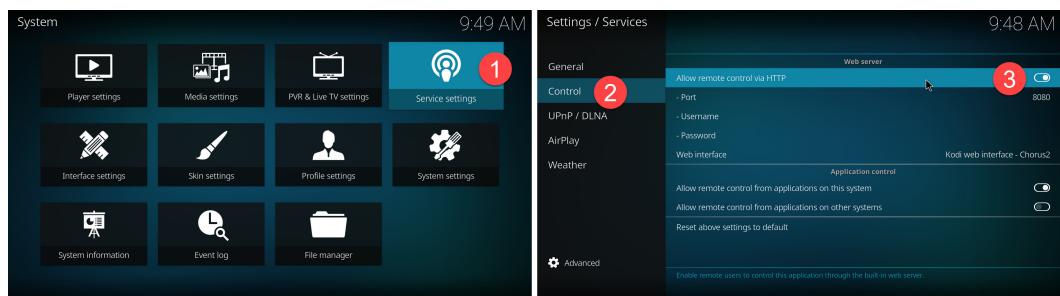
For our fourth case study, we'll create a remote controller for Kodi. Kodi (formerly known as XBMC) is a software media player: it allows playing videos and music on a computer. It has a full-screen GUI designed to be controlled with a TV remote. Kodi is the most popular software to create an HTPC (Home Theater Personal Computer).

Kodi can also be controlled remotely via a network connection; we'll use this feature to control it from an Arduino. It uses the JSON-RPC protocol, a lightweight remote procedure call (RPC) protocol, over HTTP. JSON-RPC is not very widespread, but you occasionally find it in open-source projects (Jeedom, for example), so you can reuse most of the code from this project.



To spice things up, we'll use an Arduino UNO with an Ethernet Shield. This constraint makes the task more difficult because the UNO has only 2KB of RAM, so we need to be careful not to waste it.

The code for this case study is in the `KodiRemote` folder of the zip file that you received with this book. To run this program, you need to run an instance of Kodi and check the box “Allow remote control via HTTP” in the “Service settings,” as shown in the picture below.



8.4.2 JSON-RPC Request

When a client wants to call a remote procedure, it sends an HTTP request to Kodi with a JSON document in the body. The JSON document contains the name of the procedure and the arguments.

Here is an example of a JSON-RPC request:

```
{  
    "jsonrpc": "2.0",  
    "method": "GUI.ShowNotification",  
    "params": {  
        "title": "Title of the notification",  
        "message": "Content of the notification"  
    },  
    "id": 1  
}
```

This request asks for the execution of the procedure `GUI.ShowNotification` with the two parameters `title` and `message`. When Kodi receives this request, it displays a popup message on the top-right corner of the screen.

The object contains two other members, `jsonrpc` and `id`, which are imposed by the JSON-RPC protocol.

8.4.3 JSON-RPC Response

When the server has finished executing the procedure, it returns a JSON document in the HTTP response. This document contains the result of the call.

Here is an example of a JSON-RPC response:

```
{  
    "jsonrpc": "2.0",  
    "result": "OK",  
    "id": 1  
}
```

This document is the expected response from the procedure `GUI.ShowNotification`. In this case, `result` is a simple string, but most of the time, it is a JSON object.

If a call fails, the server removes `result` and adds an `error` object:

```
{  
  "jsonrpc": "2.0",  
  "error": {  
    "code": -32601,  
    "message": "Method not found."  
  },  
  "id": 1  
}
```

8.4.4 A JSON-RPC framework

We'll create a reusable JSON-RPC framework composed of three classes:

- `JsonRpcRequest` represents a JSON-RPC request,
- `JsonRpcResponse` represents a JSON-RPC response,
- `JsonRpcClient` sends `JsonRpcRequests` and receives `JsonRpcResponses`.

As the RAM is scarce in this project, we'll make sure that only the `JsonRpcRequest` or the `JsonRpcResponse` is in memory, but not both at the same time. To implement this, we'll use the following steps:

1. create the request,
2. send the request,
3. destroy the request,
4. receive the response,
5. destroy the response.

This technique requires extra work and discipline but reduces memory consumption in half.

8.4.5 JsonRpcRequest

We start by creating the class `JsonRpcRequest`, which represents a JSON-RPC request. This class owns a `JsonDocument` and exposes a `JsonObject` named “params” to the caller:

```
class JsonRpcRequest {
public:
    JsonRpcRequest(const char *method);

    JsonObject params;

    size_t length() const;
    size_t printTo(Print &client) const;

private:
    StaticJsonDocument<256> _doc;
};
```

I used a `StaticJsonDocument` with a fixed size of 256 bytes because I know it would be unreasonable to ask more from a poor little UNO. Still, you can use a `DynamicJsonDocument` if you’re using a bigger microcontroller.

Embedding the `JsonDocument` inside `JsonRpcRequest` ensures that their lifetimes are identical. We are sure that every byte is released when we destroy the request.

The constructor of `JsonRpcRequest` takes the name of the procedure to call. It is responsible for creating the skeleton of the request:

```
JsonRpcRequest::JsonRpcRequest(const char *method) {
    _doc["method"] = method;
    _doc["jsonrpc"] = "2.0";
    _doc["id"] = 1;
    params = _doc.createNestedObject("params");
}
```

Finally, the class exposes two functions to serialize the request, they will be used by the `JsonRpcClient`:

```
// Computes Content-Length
size_t JsonRpcRequest::length() const {
    return measureJson(_doc);
}

// Serializes the request
size_t JsonRpcRequest::printTo(Print &client) const {
    return serializeJson(_doc, client);
}
```

8.4.6 JsonRpcResponse

We use a very similar pattern for the class `JsonRpcResponse` which represents a JSON-RPC response. This class owns a `JsonDocument` and exposes two `JsonVariants` named “result” and “error”:

```
class JsonRpcResponse {
public:
    JsonVariantConst result;
    JsonVariantConst error;

    bool parse(Stream &stream);

private:
    StaticJsonDocument<256> _doc;
};
```

Notice that I used `JsonVariantConst` instead of `JsonVariant` because the caller doesn’t need to modify the values.

This class has only one function, `parse()`, which is called by the `JsonRpcClient` when the response arrives:

```
bool JsonRpcResponse::parse(Stream &stream) {
    DeserializationError err = deserializeJson(_doc, stream);
    if (err) return false;
    result = _doc["result"];
```

```
    error = _doc["error"];
    return true;
}
```

8.4.7 JsonRpcClient

We can now create the last piece of our JSON-RPC framework: the `JsonRpcClient`. This class is responsible for sending requests and receiving responses over HTTP. It owns an instance of `EthernetClient`, and saves the hostname and port to be able to reconnect at any time:

```
class JsonRpcClient {
public:
    JsonRpcClient(const char *host, short port)
        : _host(host), _port(port) {}

    // Sends a JSON-RPC Request
    bool send(const JsonRpcRequest &req);

    // Receives a JSON-RPC Response
    bool recv(JsonRpcResponse &res);

private:
    EthernetClient _client;
    const char *_host;
    short _port;
};
```

`JsonRpcClient` exposes two functions that the calling program uses to send requests and receive responses. The two functions are separated because we don't want to have the `JsonRpcRequest` and the `JsonRpcResponse` in memory at the same time. That would not be possible with a signature like:

```
bool call(const JsonRpcRequest &req, JsonRpcResponse &res);
```

The `send()` function is responsible for establishing the connection with the server, and for sending the HTTP request:

```
bool JsonRpcClient::send(const JsonRpcRequest &req) {
    // Connect with server
    _client.connect(_host, _port);

    // Send the HTTP headers
    _client.println("POST /jsonrpc HTTP/1.0");
    _client.println("Content-Type: application/json");
    _client.print("Content-Length: ");
    _client.println(req.length());
    _client.println();

    // Send JSON document in body
    req.printTo(_client);

    return true;
}
```

The `recv()` function is responsible for skipping the response's HTTP headers and for extracting the JSON body:

```
bool JsonRpcClient::recv(JsonRpcResponse &res) {
    // Skip HTTP headers
    _client.find("\r\n\r\n");

    // Parse body
    return res.parse(_client);
}
```

I removed the error checking from the two snippets above, please see the source files for the complete code.

8.4.8 Sending a notification to Kodi

To display a popup on Kodi's screen, we need to send the following request:

```
{  
    "jsonrpc": "2.0",  
    "method": "GUI.ShowNotification",  
    "params": {  
        "title": "Title of the notification",  
        "message": "Content of the notification"  
    },  
    "id": 1  
}
```

In return, we expect the following response:

```
{  
    "jsonrpc": "2.0",  
    "result": "OK",  
    "id": 1  
}
```

Let's use our new JSON-RPC framework to do this:

```
// Create the JSON-RPC client  
JsonRpcClient client(host, port);  
  
// This is the scope of the Request object  
{  
    // Create the request, passing the procedure name  
    JsonRpcRequest req("GUI.ShowNotification");  
  
    // Add the two parameters  
    req.params["title"] = title;  
    req.params["message"] = message;  
  
    // Send the request  
    client.send(req);  
}  
  
// This is the scope of the Response object  
{
```

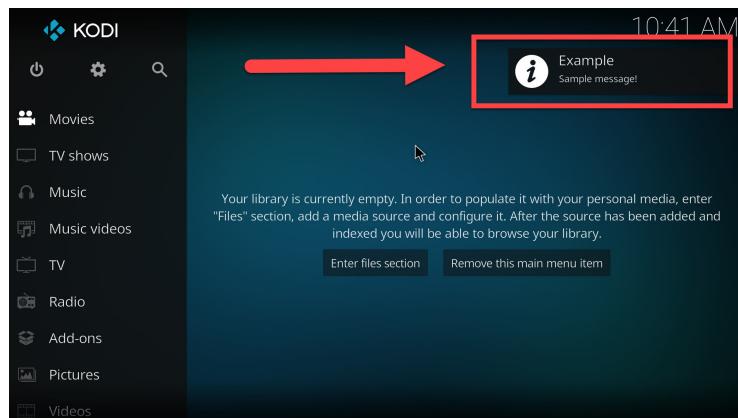
```
// Create empty response
JsonRpcResponse res;

// Read response
client.recv(res);

// Is this the expected result?
if (res.result != "OK") {
    Serial.println(F("ERROR!"));
    // Dump the error object to the Serial
    serializeJsonPretty(res.error, Serial);
}
}
```

As you see, we use a scope to limit the lifetime of the `JsonRpcRequest`. It's not mandatory to wrap the scope of `JsonRpcResponse` between braces, but I did it because it adds symmetry to the code.

See the result of this RPC call on the screen capture below:



8.4.9 Reading properties from Kodi

The procedure `Application.GetProperties` allows retrieving information about Kodi; we'll use it to get the current version of the application. This procedure takes one parameter: an array containing the names for the properties to read. In our case, there are two properties: `name` and `version`.

We need to send the following request:

```
{  
    "jsonrpc": "2.0",  
    "method": "Application.GetProperties",  
    "params": {  
        "properties": [  
            "name",  
            "version"  
        ]  
    },  
    "id": 1  
}
```

Here is the expected response:

```
{  
    "id": 1,  
    "jsonrpc": "2.0",  
    "result": {  
        "name": "Kodi",  
        "version": {  
            "major": 17,  
            "minor": 6,  
            "revision": "20171114-a9a7a20",  
            "tag": "stable"  
        }  
    }  
}
```

Let's use our framework again:

```
// Create the JSON-RPC client  
JsonRpcClient client(host, port);  
  
// This is the scope of the Request object  
{  
    // Create the request
```

```
JsonRpcRequest req("Application.GetProperties");

// Add the "properties" array
req.params["properties"].add("name");
req.params["properties"].add("version");

// Send the request
client.send(req);
}

// This is the scope of the Response object
{
    // Create an empty response
JsonRpcResponse res;

    // Read response
client.recv(res);

    // Print "Kodi 17.6 stable"
Serial.print(res.result["name"].as<char*>());
Serial.print(" ");
Serial.print(res.result["version"]["major"].as<int>());
Serial.print(".");
Serial.print(res.result["version"]["minor"].as<int>());
Serial.print(" ");
Serial.println(res.result["version"]["tag"].as<char*>());
}
```

8.4.10 Summary

The goal of this case study was to teach three things:

1. how to do JSON-RPC with ArduinoJson,
2. how to embed a JsonDocument in a class,
3. how to control the lifetime of objects.

The tricky part was to keep only the request or the response in RAM. Once you understand the pattern, however, the code is simple, and you can implement virtually any remote procedure call with a minimal amount of memory.

If you look at the source files, you'll see that I created a class `KodiClient` that provides the abstraction on top of `JsonRpcClient`. You could easily extend this class to add more procedures, for example, to control the playback.

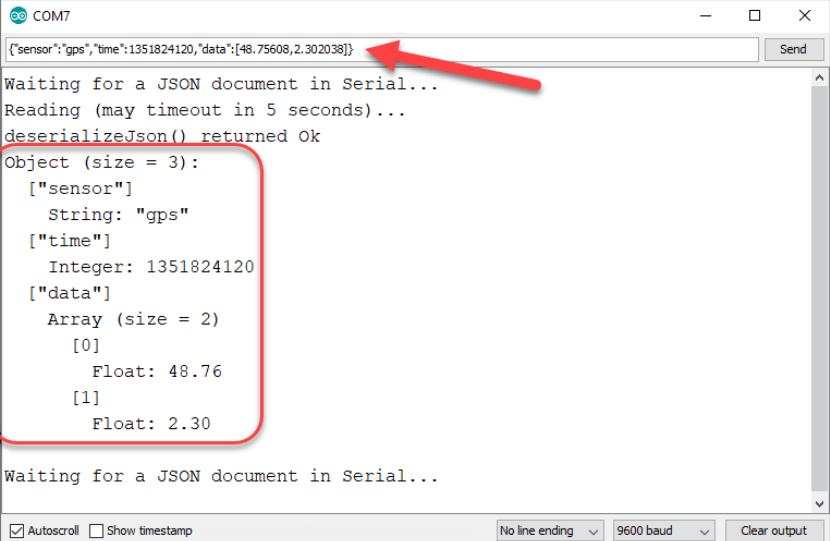
Another difference in the source files is that I added an optimization. The program reuses the TCP connections to the server, using the "keep-alive" mode. This technique considerably improves performance if you need to call several procedures in a row. To do that, the `KodiClient` reuses the same `JsonRpcClient` for each call.

In the next case study, we'll see how to read a JSON document from the serial port and recursively print its content.

8.5 Recursive analyzer

8.5.1 Presentation

For our last case study, we'll create a program that reads a JSON document from the serial port and prints a hierarchical representation of the document. The goal is to demonstrate how to scan a `JsonDocument` recursively. This case study is also an opportunity to see how we can read a JSON document from the serial port.



```
COM
>{"sensor": "gps", "time": 1351824120, "data": [48.75608, 2.302038]}
Waiting for a JSON document in Serial...
Reading (may timeout in 5 seconds)...
deserializeJson() returned OK
Object (size = 3):
  ["sensor"]
    String: "gps"
  ["time"]
    Integer: 1351824120
  ["data"]
    Array (size = 2)
      [0]
        Float: 48.76
      [1]
        Float: 2.30
Waiting for a JSON document in Serial...

Autoscroll  Show timestamp  No line ending  9600 baud  Clear output
```

The source code of this program is in the `Analyzer` folder of the zip file.

8.5.2 Read from the serial port

`Serial` is an instance of the class `HardwareSerial`, which derives from `Stream`. As such, you can directly pass it to `deserializeJson()`:

```
deserializeJson(doc, Serial);
```

However, if we only do that, `deserializeJson()` will wait for a few seconds until it times out and returns `IncompleteInput`. To avoid the timeout, we need to add a loop that waits for incoming characters:

```
// Loop until there is data waiting to be read
while (!Serial.available())
    delay(50);
```

When this loop exits, we can safely call `deserializeJson()`. As we saw in the Reddit case study, it will read the stream and stop when the object (or array) ends.

When `deserializeJson()` returns, some characters may remain in the buffer. For example, the Arduino Serial Monitor may send the characters carriage-return ('\r') and line-feed ('\n') to terminate the line. To remove these trailing characters, we must add a loop that drops remaining spaces from the serial port:

```
// Skip all spaces from the serial port
while (isspace(Serial.peek()))
    Serial.read();
```

As you see, we call `Stream::peek()` to look at the next character without extracting it. Then we use the standard C function `isspace()` to test if this character is a space. This function returns true for the space character but also for tabulation, carriage-return, and line-feed. If that's the case, we call `Stream::read()` to consume this character. We repeat this operation until there are no more space characters in the stream.

8.5.3 Flushing after an error

As we just saw, `deserializeJson()` stops reading at the end of the object (or array), but this statement is true only if the call succeeds. When `deserializeJson()` fails, it stops reading immediately, so the reading cursor might be in the middle of a JSON document. To make sure we can be able to deserialize the next document correctly, we must flush and restart with an empty stream. A simple loop should do the job:

```
// Flush the content of the serial port buffer
while (Serial.available())
    Serial.read();
```

This snippet is simpler than the previous one because we don't need to look at the character; instead, we blindly discard everything until the buffer is empty.

8.5.4 Testing the type of a JsonVariant

We'll now create a function that prints the content of a `JsonVariant`. This function needs to be recursive to be able to print the values that are inside objects and arrays.

Before printing the content of a `JsonVariant`, we need to know its type. We inspect the variant with `JsonVariant::is<T>()`, where `T` is the type we want to test. A `JsonVariant` can hold six types of values:

1. boolean: `bool`
2. integral: `long` (but `int` and others would match too)
3. floating point: `double` (but `float` would match too)
4. string: `const char*`
5. object: `JsonObject`
6. array: `JsonArray`

We'll limit the responsibility of `dump(JsonVariant)` to the detection of the type, and we'll delegate the work of printing the value to overloads. The code is therefore just a sequence of `if` statement:

```
void dump(JsonVariant variant) {  
    // if boolean, then call the overload for boolean  
    if (variant.is<bool>())  
        dump(variant.as<bool>());  
  
    // if integral, then call the overload for integral  
    else if (variant.is<long>())  
        dump(variant.as<long>());  
  
    // if floating point, then call the overload for floating point  
    else if (variant.is<double>())  
        dump(variant.as<double>());  
  
    // if string, then call the overload for string
```

```
else if (variant.is<char *>())
    dump(variant.as<char *>());

// if object, then call the overload for object
else if (variant.is<JsonObject>())
    dump(variant.as<JsonObject>());

// if array, then call the overload for array
else if (variant.is<JsonArray>())
    dump(variant.as<JsonArray>());

// none of the above, then it's null
else
    Serial.println("null");
}
```



Order matters

It's important to test long before double because an integral can always be stored in a floating-point. In other words, `is<long>()` implies `is<double>()`.

8.5.5 Printing values

Printing simple values is straightforward:

```
// For clarity, I omitted the code dedicated to indentation

void dump(bool value) {
    Serial.print("Bool: ");
    Serial.print(value ? "true" : "false");
}

void dump(long value) {
    Serial.print("Integer: ");
    Serial.println(value);
}
```

```
void dump(double value) {
    Serial.print("Float: ");
    Serial.println(value);
}

void dump(const char *str) {
    Serial.print("String: \"");
    Serial.print(str);
    Serial.println("\"");
}
```

Printing objects requires a loop and a recursion:

```
void dump(JsonObject obj) {
    Serial.print("Object: ");

    // Iterate though all key-value pairs
    for (JsonPair kvp : obj) {
        // Print the key (simplified for clarity)
        Serial.println(kvp.key().c_str());

        // Print the value
        dump(kvp.value()); // <- RECURSION
    }
}
```

So does printing an array:

```
void dump(const JSONArray &arr) {
    Serial.print("Array: ");

    int index = 0;
    // Iterate though all elements
    for (JsonVariant value : arr) {
        // Print the index (simplified for clarity)
        Serial.println(index);
```

```
// Print the value
dump(value); // <- RECURSION

    index++;
}
}
```



Prefer range-based for loop

We use the syntax `for(value:arr)` instead of `for(i=0;i<arr.size();++i)` because it's much faster. Indeed, it prevents from calling `arr[i]`, which needs to walk the linked-list (complexity $O(n)$).

Does that sound familiar? Indeed, we talked about that [in the deserialization tutorial](#).

8.5.6 Summary

It was by far the shortest of our case studies, but I'm sure many readers will find it helpful because the recursive part can be tricky if you are not familiar with the technique.

We could have used `JsonVariantConst` instead of `JsonVariant`, but I thought it was better to simplify this already complicated case study.

If you compare the actual source of the project with the snippets above, you'll see that I removed all the code responsible for the formatting, so that the book is easier to read.

One last time, here are the key points to remember:

- You can pass `Serial` directly to `deserializeJson()`.
- To avoid the timeout, add a wait loop before `deserializeJson()`.
- Add another loop after `deserializeJson()` to discard trailing spaces or linebreaks.
- Test the type with `JsonVariant::is<T>()`.
- Always test integral types (`long` in this example) before floating-point types (`double` in this example), because they may both be true at the same time.

That was the last case study; in the next chapter, we'll conclude this book.

Chapter 9

Conclusion

It's already the end of this book. I hope you enjoyed reading it as much as I enjoyed writing it. More importantly, I hope you learn many things about Arduino, C++, and programming in general. Please tell me what you thought of the book at book@arduinojson.org. I'll be happy to hear from you.

I want to thank all the readers that helped me improve and promote this book: Adam Iredale, Avishek Hardin, Bon Shaw, Carl Smith, Craig Feied, Daniel Travis, Darryl Jewiss, Dieter Grientschnig, Doug Petican, Douglas S. Basberg, Ezequiel Pavón, Gayrat Vlasov, Georges Auberger, Jon Freivald, Joseph Chiu, Leonardo Bianchini, Matthias Wilde, Nathan Burnett, Pierre Olivier Théberge, Robert Balderson, Ron VanEtten, Thales Liu, Vasilis Vorrias, Walter Hill, Yann Büchau, and Yannick Corroenne. Thanks to all of you!

Again, thank you very much for buying this book. This act encourages the development of high-quality libraries. By providing a (modest) source of revenue for open-source developers like me, you ensure that the libraries that you rely on are continuously improved and won't be left abandoned after a year.

Sincerely, Benoît Blanchon



Satisfaction survey

Please take a minute to answer a short survey.

Go to arduinojson.survey.fm/book

Index

__FlashStringHelper	52, 92	class	38
Adafruit IO	106	code samples	2
AIO key	123	Comments	13
aJson	19	const	67
analogRead()	111	Content-Length	118, 125
Arduino Library Manager	228	copyArray()	213
Arduino_JSON	18	DefaultAllocator	151
ArduinoJson Assistant	78	delete	43
ARDUINOJSON_DECODE_UNICODE	219, 251, 290	DeserializationError	68, 245
ARDUINOJSON_DEFAULT_NESTING_LIMIT	253	DeserializationOption::Filter ..	135, 221, 283
ARDUINOJSON_ENABLE_COMMENTS	220, 250	DeserializationOption::NestingLimit	217
ARDUINOJSON_ENABLE_INFINITY	220, 225	deserializeJson() ..	67, 144, 161, 215, 283, 289, 306
ARDUINOJSON_ENABLE_NAN	220, 225	deserializeMsgPack()	167
ARDUINOJSON_NAMESPACE	226	design pattern	46, 81
ARDUINOJSON_VERSION	226	DOM	20
ARDUINOJSON_VERSION_MAJOR	226	Douglas Crockford	8
ARDUINOJSON_VERSION_MINOR	226	DynamicJsonDocument	65
ARDUINOJSON_VERSION_REVISION	226	DynamicJsonDocument::shrinkToFit()	148, 152
ArduinoTrace	242	ESP.getMaxAllocHeap()	147
ATmega328	31, 35, 36, 53, 235, 240	ESP.getMaxFreeBlockSize()	147
auto	69	ESP32	244
Base64	12	ESP8266 ..	35, 54, 235, 240, 244, 268
BasicJsonDocument<T>	151, 185	EthernetClient	125
BOM	249	exceptions	71
BSON	12	F()	32, 53
BufferedStream	158	FAT file-system	95
BufferingPrint	157		
CBOR	12		

- fgetc() 163
File 94, 123
finally 44
Flash 52, 57, 92, 129, 257, 268
fopen() 162
fputc() 162
fragmentation 34, 173
fread() 163
free() 43, 184, 236
fwrite() 162
git bisect 242
GitHub's API 63, 95
GraphQL 97
GSMClient 94
HardwareSerial 94, 122
heap_caps_malloc() 152
heap_caps_realloc() 152
IncompleteInput 245, 307
Infinity 9, 220, 225
InvalidInput 220, 246
isNull() 80
isspace() 307
jsmn 19
JSON streaming 144
JSON-RPC 294
json-streaming-parser 20
JSON_ARRAY_SIZE() 77
JSON_OBJECT_SIZE() 66
JsonArray 209
JsonArray::add() 209
JsonArray::begin() / end() 210
JsonArray::clear() 210
JsonArray::createNestedArray() 211
JsonArray::createNestedObject() 112, 211
JsonArray::getElement() 211
JsonArray::getOrAddElement() 212
JsonArray::isNull() 212
JsonArray::operator[] 113
JsonArray::remove() 114, 212
JsonArray::size() 83, 212
JsonArrayConst 274
JsonDocument 65
JsonDocument::add() 112
JsonDocument::capacity() 176, 251
JsonDocument::clear() 176
JsonDocument::garbageCollect() 149
JsonDocument::memoryUsage() 83, 176
JsonDocument::to<JsonArray>() 113
JsonDocument::to<JsonObject>() 109
JsonDocument::to<T>() 176
JSONLines 145
JsonObject 201
JsonObject::begin() / end() 204
JsonObject::clear() 205
JsonObject::containsKey() 75
JsonObject::createNestedArray() 205
JsonObject::createNestedObject() 206
JsonObject::getMember() 203, 206
JsonObject::getOrAddMember() 203, 206
JsonObject::isNull() 206
JsonObject::operator[] 203
JsonObject::remove() 207
JsonObject::size() 83, 207
JsonObjectConst 207, 270
JsonPair 73
JsonRpcClient 299
JsonString 73
JsonVariant 73, 187
JsonVariant::add() 196
JsonVariant::as<T>() 95, 191, 196, 262
JsonVariant::createNestedArray() 196
JsonVariant::createNestedObject() 197
JsonVariant::is<T>() 74, 84, 197
JsonVariant::isNull() 190, 198
JsonVariant::isUndefined() 190
JsonVariant::operator[] 198
JsonVariant::set() 108, 188, 198

JsonVariant::size()	199	SPIFFS	268
JsonVariantConst	200	SPIRAM	153
keep-alive	305	sprintf()	18, 57, 239
LDJSON	145	sprintf_P()	57
LoggingPrint	155	sscanf()	18
LoggingStream	156	StaticJsonDocument	65
malloc()	43, 147, 184	std::istream	94, 160
MCVE	264	std::ostream	121, 160
measureJson()	118, 223	std::shared_ptr	46
measureJsonPretty()	223	std::unique_ptr	46
measureMsgPack()	223	stdio.h	162
MemberProxy	203	strcmp()	55, 235
MessagePack	12, 16, 165	strcpy()	52, 239
mkr1000	279	strcpy_P()	52
NaN	9, 220, 225	Stream	94, 160
NDJSON	145	Stream::available()	145
new	43	Stream::find()	140, 289
NoMemory	250	Stream::findUntil()	142, 289
NotSupported	251, 290	Stream::peek()	307
null	114	Stream::println()	146
null object	81	Stream::setTimeout()	246
null pointer	41	StreamUtils	155, 157
nullptr	41	String	92, 119
OpenWeatherMap	279	String Interning	51
Print	121, 160	strlcpy()	239
Print::println()	262	strncpy()	240
PROGMEM	32	struct	38
RAII	44, 46	TooDeep	252
ReadBufferingStream	157	TwoWire	94, 122
ReadLoggingStream	155	typedef	185
realloc()	148	Unicode	10, 219, 251
Reddit's API	287	using	185
SAX	20	Variable-length array	117
serialized()	114, 131, 187	wandbox.org	16, 230, 264
SerializedValue	187	WiFiClient	94, 290
serializeJson()	116, 146, 222	WiFiClientSecure	94, 290
serializeJsonPretty()	117, 118, 222	XML	8
serializeMsgPack()	166, 222		
shrinkToFit()	182		
SoftwareSerial	94, 122		